

## 1.- Introducción

Aprovecharemos la posibilidad de monitorear y proteger nuestro código en user desde RING0.

Para ello me voy a basar en la arquitectura de paginación de windows, la traducción de direcciones virtuales y los registros de control que proporciona Intel.

El proceso se realiza de la siguiente manera:

- Reservaremos 2 espacios en Ring3 y inyectaremos la shellcode en uno de ellos
- Usaremos **DeviceloControl** para comunicarnos con el driver.
- En el driver iremos escalando desde el registro CR3 y la dirección virtual hasta obtener la Page Table Entry y su marco de pagina.
- Des-referenciamos el espacio de memoria "shellcode" asignándole otro pfn a la PTE de su dirección virtual como la de el espacio benigno.

De esta forma mantendremos oculto ese espacio de memoria reservado dentro del proceso que queramos



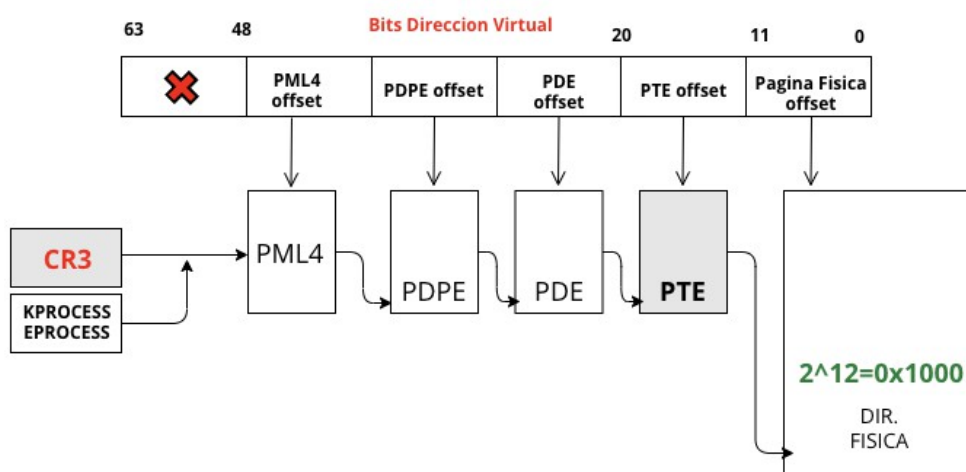
## 2.- Dirección virtuales, físicas, paginación y WinDBG

Todo lo que aremos 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:    0000000000000015203370
Binary:    00000000 00000000 00000000 00000000 00110101 00000110 11111000
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 despiezar 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 pagina correspondiente:



En el diagrama que e hecho se explica un poco el recorrido de la traduccion, de tal forma que cada 9 bits desde el bit 47 se realiza un calculo 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 traduccion:

- **PML4** → bits 47-39 → **2<sup>9</sup>=512** posibles indexaciones
- **PDPE** → bits 38-30 → **2<sup>9</sup>=512** posibles indexaciones
- **PDE** → bits 29-21 → **2<sup>9</sup>=512** posibles indexaciones
- **PTE** → bits 20-12 → **2<sup>9</sup>=512** posibles indexaciones

Con lo que terminaríamos obteniendo la dirección física de la pagina correspondiente la cual en 64bits seria

- **2<sup>12</sup>=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 traduccion, 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 que ocurre, que en este caso será la shellcode que inyectaremos 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 pagina 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";

    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";

    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** (pointing to the VirtualProtect call)

**Sin permisos de escritura, debería crashear** (pointing to the second RtlMoveMemory call)

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 0000000000180000
PXE at FFFFDAED76BB5000  PPE at FFFFDAED76A00000  PDE at FFFFDAED40000000  PTE at FFFFDA8000000C00
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)**

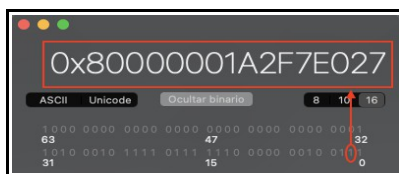
<b>P:</b> Present	<b>G:</b> Global
<b>R/W:</b> Read/Write	<b>AVL:</b> Available
<b>U/S:</b> User/Supervisor	<b>PAT:</b> Page Attribute
<b>PWT:</b> Write-Through	Table
<b>PCD:</b> Cache Disable	<b>M:</b> Maximum
<b>A:</b> Accessed	Physical Address Bit
<b>D:</b> Dirty	<b>PK:</b> Protection Key
<b>PS:</b> 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 **FFFD8000000C00**:



```
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 80000001A2F7E025
pfn 16e800 ---DA--UWEV pfn 76e01 ---DA--UWEV pfn 12e307 ---DA--UWEV pfn 1a2f7e ----A--UR-V

1: kd> ep FFFFDA8000000C00 80000001A2F7E027
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 80000001A2F7E027
pfn 16e800 ---DA--UWEV pfn 76e01 ---DA--UWEV pfn 12e307 ---DA--UWEV pfn 1a2f7e ----A--UR-V
```

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
```

### 3.- SUBVERSION 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**”.

Que es lo que conseguimos con esta técnica? Antes hemos visto que una entrada PTE contiene un marco de pagina llamado **pfn**, que sin entrar en detalles básicamente los PTE obtienen el **pfn** para la siguiente estructura de paginación, por los tanto en un contexto de x64 donde las paginas físicas son de **4096** bytes es decir **0x1000**, y multiplicando ese **pfn** por el tamaño de la pagina 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 FFFFDAAED76BB5000  PPE at FFFFDAAED76A00000  PDE at FFFFDAAED40000000  PTE at FFFFDAA8000000C00
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
```

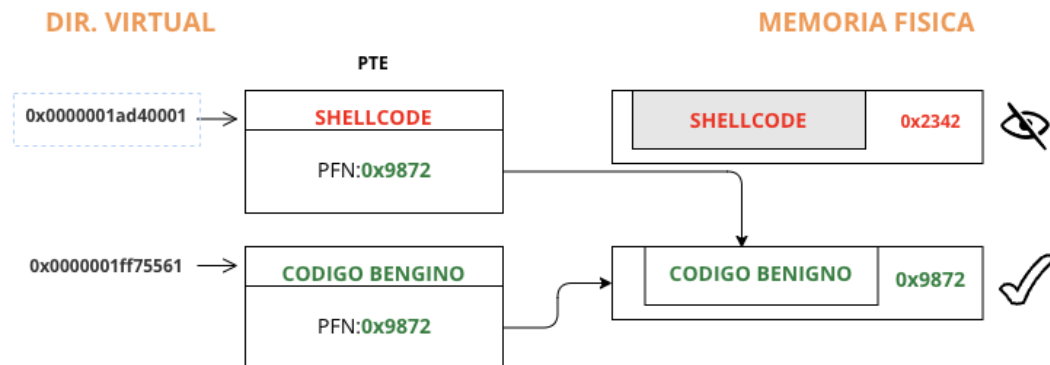
- Extraemos el marco de pagina de PTE y lo multiplicamos por **0x1000**
  - **0x1a2f7e** → **dirección física**
  - **0x18000** → **dirección virtual**

Realizando un dump 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 pagina física de la dirección virtual en tiempo de ejecución, y si aprovechamos para que el marco de pagina de la PTE de 2 direcciones virtuales diferentes apuntaran al mismo **pfn**??:

- Reservamos 2 espacios de memoria en user
- En uno de ellos lo rellenos 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 pagina con el payload por el pfn del código benigno



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

## 5.-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 sleed 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,rax
fffff802`46abadd9 48b80000000080faffff mov     rax,0FFFFFFFA80000000h
fffff802`46abade3 4803c1        add     rax,rcx
fffff802`46abade6 c3           ret

1: kd>

```

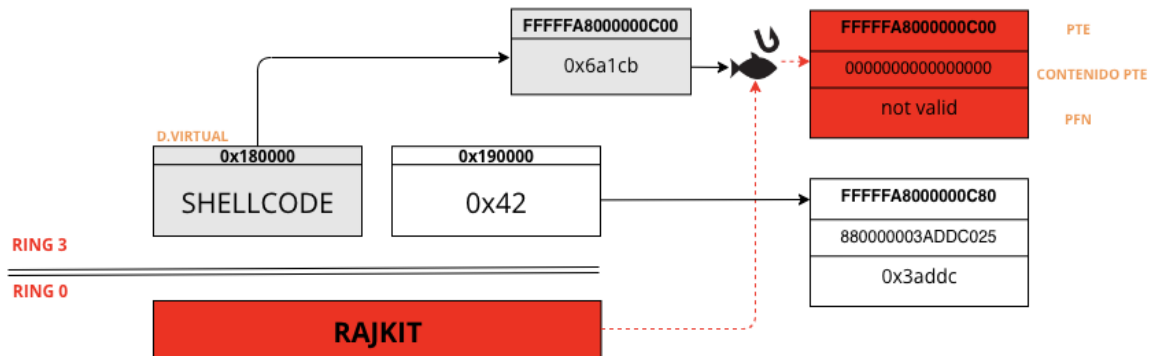
DIR.BASE: PTE

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

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

## 5.1-TECNICA ANTI-FORENSE 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 setamos su PTE a **0000000000000000**, y la dirección de la memoria limpia en **0x19000** con un tamaño también de **0x1000**



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** →  $(0x6a1cb * 0x1000) = \text{DIR.FISICA}$

```
1: kd> !pte 0x18000
PXE at FFFFAFD7EBF5000  PPE at FFFFAFD7EA00000  PDE at FFFFAFD40000000  PTE at FFFFA8000000C00
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 pteaddress FFFFA8000000C00
reference count 0001 used entry count 0000 Cached color 0 Priority 5
restore pte 000000C0 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 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..
1: kd> !vtop 0 0x18000
Amd64VtoP: Virt 0000000000180000, pagedir 00000000405b6000
Amd64VtoP: PML4E 00000000405b6000
Amd64VtoP: PDPE 00000000688c2000
Amd64VtoP: PDE 000000001d5c3000
Amd64VtoP: PTE 00000000504cdc00
Amd64VtoP: zero PTE
Virtual address 180000 translation fails, error 0xD0000147.

1: 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	Col	Tipo	Permisos	Inicial	
0000000000010000	0000000000010000	Usuario			MAP	-RW--	-RW--	
0000000000020000	0000000000010000	Usuario			PRV	ERW--	ERW--	
0000000000030000	0000000000018000	Usuario			MAP	-R---	-R---	
0000000000050000	00000000000FA000	Usuario	Reservado		PRV	-RW-G	-RW--	
0000000000014A000	0000000000006000	Usuario	Stack (8228)		PRV	-R---	-R---	
00000000000150000	0000000000004000	Usuario			MAP	-R---	-R---	
00000000000160000	0000000000001000	Usuario			MAP	-R---	-R---	
00000000000170000	0000000000002000	Usuario			PRV	-RW--	-RW--	
00000000000200000	0000000000017000	Usuario	Reservado		PRV	-RW--	-RW--	
00000000000217000	0000000000007000	Usuario	PEB, TEB (2876), TEB (8228), TEB (1028)		PRV	-RW--	-RW--	
0000000000021E000	000000000001E2000	Usuario	Reservado (0000000000200000)		PRV	-RW--	-RW--	
00000000000400000	00000000000C7000	Usuario	\Device\HarddiskVolume3\windows\System		MAP	-R---	-R---	

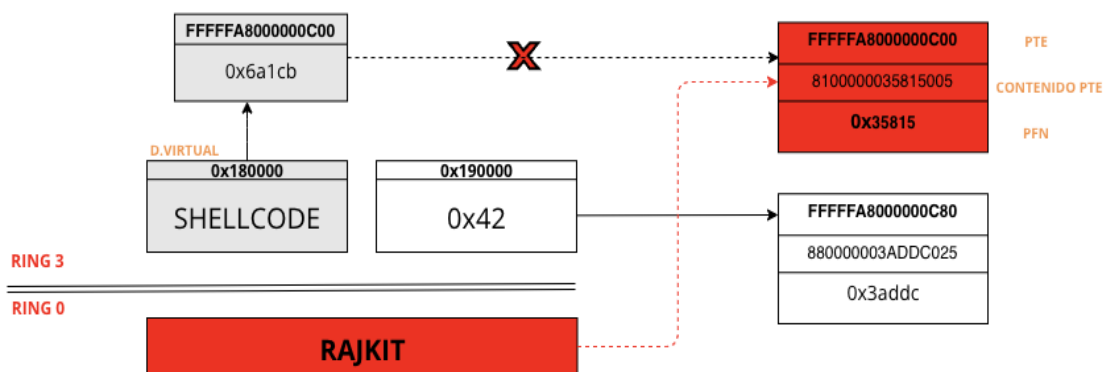
Dirección	Hex	ASCII
0000000000020000	FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51	UH.ãeA...AQAPRQ
0000000000020010	56 48 31 D2 65 48 8B 52 60 48 8B 52 18 48 8B 52	VH10eH.R.H.R.H.R
0000000000020020	20 48 8B 72 50 48 0F B7 4A 4A 4D 31 C9 48 31 C0	H.rPH..JJM1EH1A
0000000000020030	AC 3C 61 7C 02 2C 20 41 C1 C9 0D 41 01 C1 E2 ED	~<a .,AAE.A.Aâf
0000000000020040	52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 8B 80 88	RAQH.R.B<H.D...
0000000000020050	00 00 00 48 85 C0 74 67 48 01 D0 8B 48 18 44	...H.AtqH.DP.H.D
0000000000020060	8B 40 20 49 01 D0 E3 56 48 FF C9 41 8B 34 88 48	.@I.DâVHYEA.4.H
0000000000020070	01 D6 4D 31 C9 48 31 C0 AC 41 C1 C9 0D 41 01 C1	.0M1EH1A-AAE.A.A
0000000000020080	38 E0 75 F1 4C 03 4C 24 08 45 39 D1 75 D8 58 44	8auhL.L\$.E9Nu0XD
0000000000020090	8B 40 24 49 01 D0 66 41 8B 0C 48 44 8B 40 1C 49	..@I.DfA..HD.@.I
00000000000200A0	01 D0 41 8B 04 88 48 01 D0 41 58 41 58 5E 59 5A	.@A...H.DAXAXYZ
00000000000200B0	41 58 41 59 41 5A 48 83 EC 20 41 52 FF E0 58 41	AXAYAZH.1 ARYAXA
00000000000200C0	59 5A 48 8B 12 E9 57 FF FF FF 5D 48 BA 01 00 00	YZH..ewyyy]H°...
00000000000200D0	00 00 00 00 00 48 8D 8D 01 01 00 00 41 BA 31 88	...H.....A°1.
00000000000200E0	6F 87 FF D5 8B F0 B5 A2 56 41 BA A6 95 BD 9D FF	o.y0>0µcVA°!.%y
00000000000200F0	D5 48 83 C4 28 3C 06 7C 0A 80 FB E0 75 05 BB 47	0H.A(< .ûau.»G
0000000000020100	13 72 6F 41 89 DA 48 83 C4 20 C3 63 61 6C 63 2E	.roA.UH.A Aca1c.
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: 000000000020000 -> 000000000020000 (0x00000001 bytes)

## 5.2-TECNICA ANTI-FORENSE FASE 2

En la segunda fase asignamos un PFN a la PTE de la dirección virtual que apunta a la pagina que contiene codigo benigno **0x42** y mantendremos oculta 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 FFFFFFFD7EBF5000 contains 0A000000688C2867 pfn 688c2 ---DA--UWEV
PPE at FFFFFFFD7EA00000 contains 0A0000001D5C3867 pfn 1d5c3 ---DA--UWEV
PDE at FFFFFFFD40000000 contains 0A000000504CD867 pfn 504cd ---DA--UWEV
PTE at FFFFFFFA800000C0 contains 81000000358150B5 pfn 35815 -----UR--
```

[illegible]

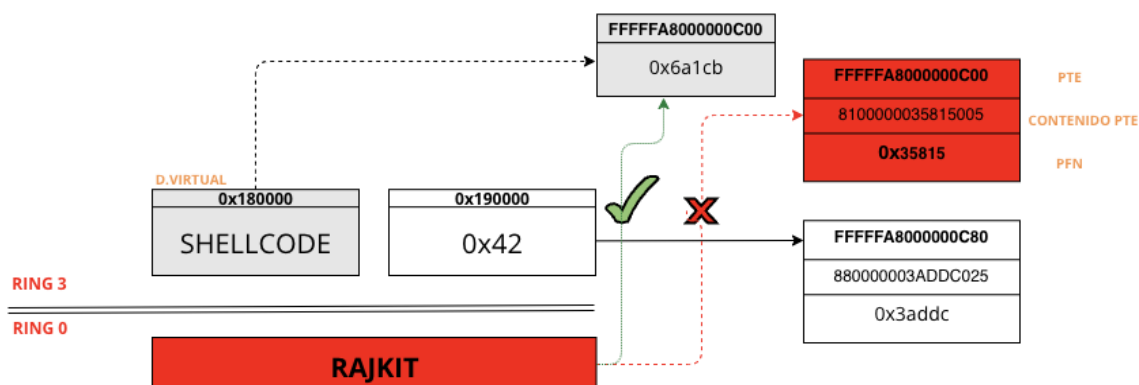
Vemos como volcamos la dirección virtual de la shellcode que si obtenemos su PFN nos devuelve la PTE y si traducimos esa PTE nos devuelve la dirección **0x18000** que a su vez haciendo el volcado en realidad contiene un sleep de **0x42**:

```
0: kd> !db 0x6a1cb000
#6a1cb000: fc 48 83 e4 f0 e8 c0 00-00 00 41 51 41 50 52 51 |.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> !pfno 0x6a1cb
PFN 0006A1CB at address FFFF9C00013E5610
flink 00000001 blink / share count 00000001 pteaddress FFFFFA8000000C00
reference count 0001 used entry count 0000 Cached color 0 Priority 5
restore pte 000000C0 containing page 0504CD Active M
Modified
0: kd> !pte2va_FFFFFA8000000C00
0000000000180000
0: kd> db 0x180000
00000000 00180000 |42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 |BBBBBBBBBBBBBBBB
00000000 00180010 |42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 |BBBBBBBBBBBBBBBB
00000000 00180020 |42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 |BBBBBBBBBBBBBBBB
00000000 00180030 |42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 |BBBBBBBBBBBBBBBB
00000000 00180040 |42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 |BBBBBBBBBBBBBBBB
00000000 00180050 |42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 |BBBBBBBBBBBBBBBB
00000000 00180060 |42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 |BBBBBBBBBBBBBBBB
00000000 00180070 |42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 |BBBBBBBBBBBBBBBB
```

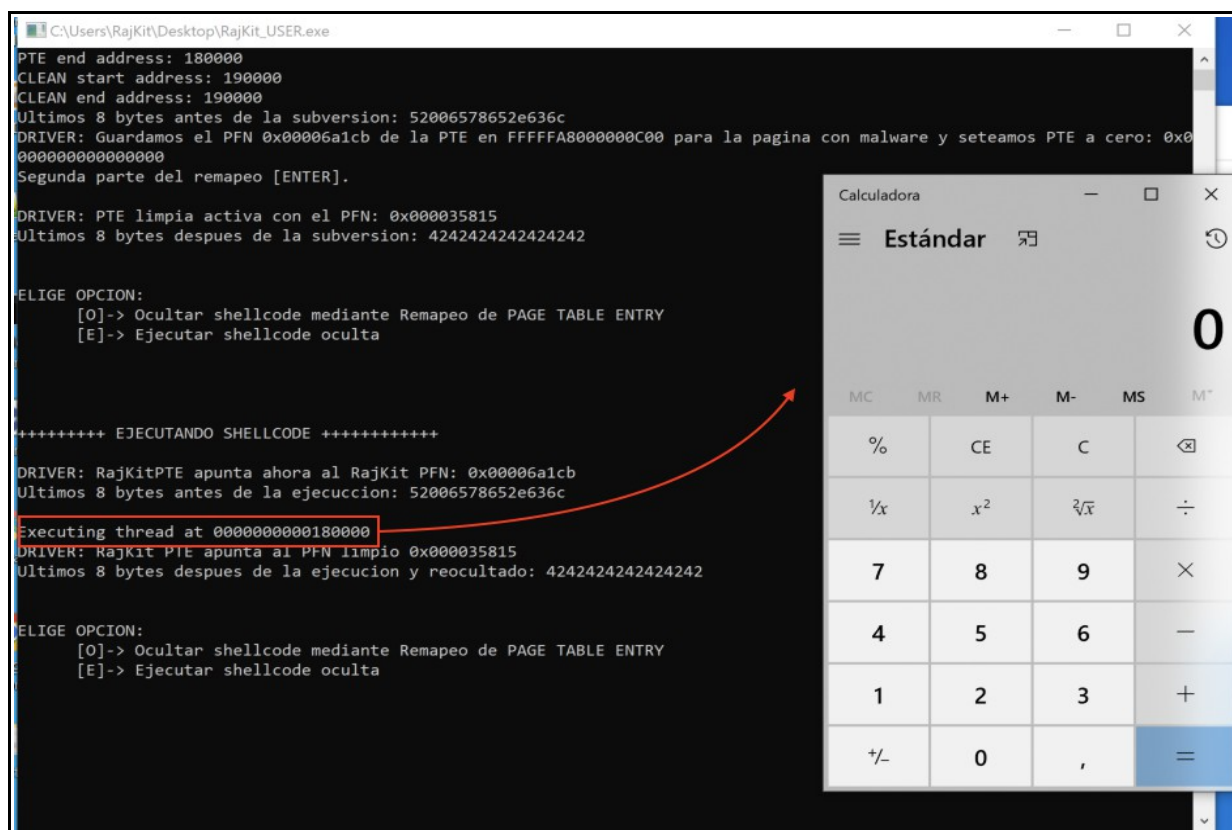
 $\theta: kd>$

## 5.3-TECNICA ANTI-FORENSE FASE 3

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



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



## 6.-FUENTES

- <https://lsi.vc.ehu.eus/pablogn/docencia/manuales/SO/TemasSOuJaen/ADMINISTRACIONDELAMEMORIA/5.1Paginacion.htm>
- <https://www.microsoft.com/en-us/security/blog/2020/07/08/introducing-kernel-data-protection-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://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/kernel-mode-extensions>

