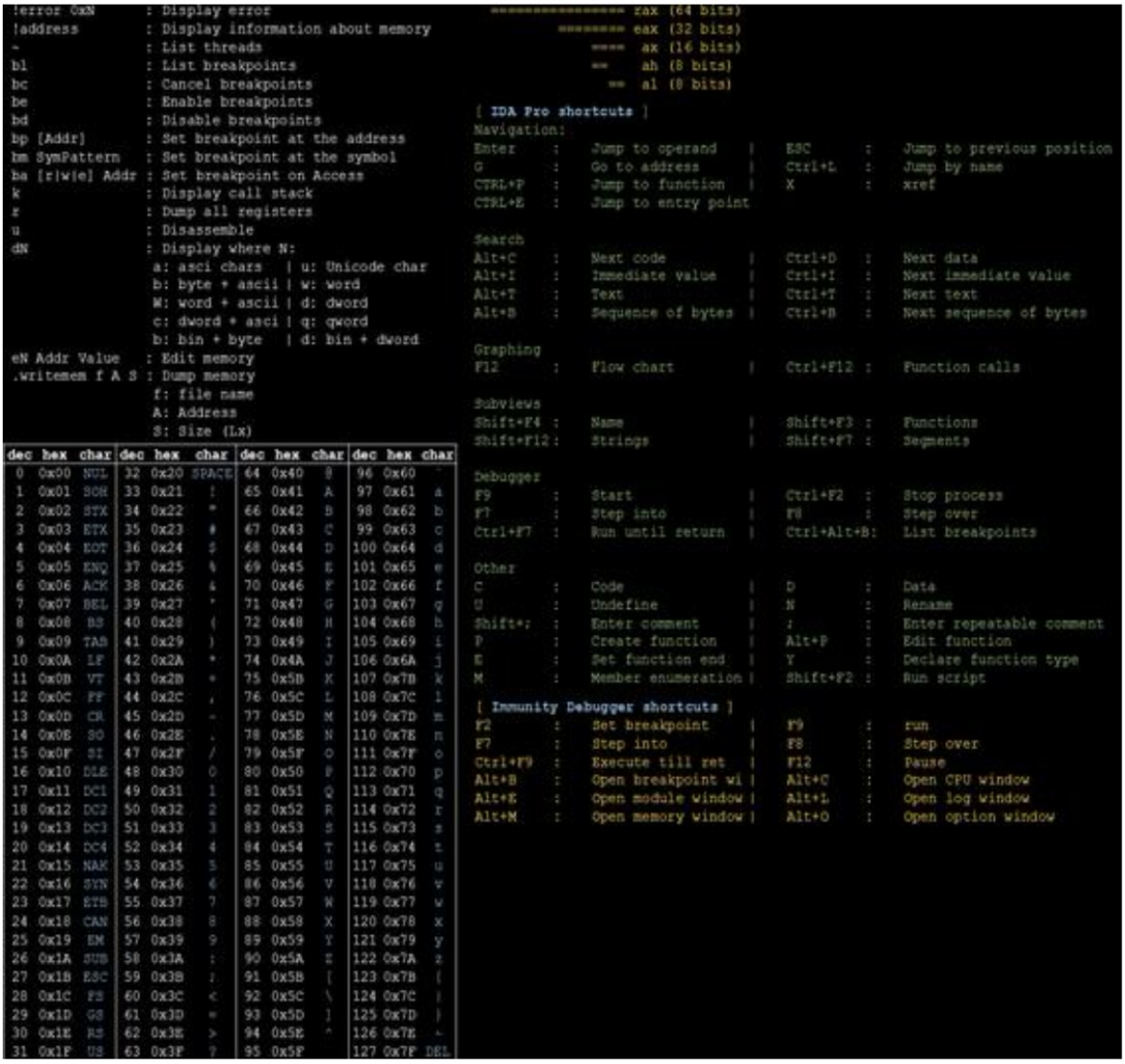


```
!error 0xN        : Display error
!address          : Display information about memory
~                 : List threads
bl                : List breakpoints
bc                : Cancel breakpoints
be                : Enable breakpoints
bd                : Disable breakpoints
bp [Addr]         : Set breakpoint at the address
bm SymPattern     : Set breakpoint at the symbol
ba [r|w|e] Addr   : Set breakpoint on Access
k                 : Display call stack
r                 : Dump all registers
u                 : Disassemble
dN                : Display where N:
                    a: asci chars   | u: Unicode char
                    b: byte + ascii | w: word
                    W: word + ascii | d: dword
                    c: dword + asci | q: qword
                    b: bin + byte   | d: bin + dword
eN Addr Value     : Edit memory
.writemem f A S   : Dump memory
                    f: file name
                    A: Address
                    S: Size (Lx)
```

dec	hex	char	dec	hex	char	dec	hex	char	dec	hex	char
0	0x00	NUL	32	0x20	SPACE	64	0x40	@	96	0x60	`
1	0x01	SOH	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL	39	0x27	'	71	0x47	G	103	0x67	g
8	0x08	BS	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	TAB	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	LF	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	VT	43	0x2B	+	75	0x5B	K	107	0x7B	k
12	0x0C	FF	44	0x2C	,	76	0x5C	L	108	0x7C	l
13	0x0D	CR	45	0x2D	-	77	0x5D	M	109	0x7D	m
14	0x0E	SO	46	0x2E	.	78	0x5E	N	110	0x7E	n
15	0x0F	SI	47	0x2F	/	79	0x5F	O	111	0x7F	o
16	0x10	DLE	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	SUB	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	ESC	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	FS	60	0x3C	<	92	0x5C	\	124	0x7C	|
29	0x1D	GS	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	RS	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	US	63	0x3F	?	95	0x5F	_	127	0x7F	DEL

```
================ rax (64 bits)
                ======== eax (32 bits)
                    ====  ax (16 bits)
                    ==    ah (8 bits)
                      ==  al (8 bits)
```

```
[ IDA Pro shortcuts ]
Navigation:
Enter    :   Jump to operand    |   ESC      :   Jump to previous position
G        :   Go to address      |   Ctrl+L   :   Jump by name
CTRL+P   :   Jump to function   |   X        :   xref
CTRL+E   :   Jump to entry point

Search
Alt+C    :   Next code          |   Ctrl+D   :   Next data
Alt+I    :   Immediate value    |   Ctrl+I   :   Next immediate value
Alt+T    :   Text               |   Ctrl+T   :   Next text
Alt+B    :   Sequence of bytes  |   Ctrl+B   :   Next sequence of bytes

Graphing
F12      :   Flow chart         |   Ctrl+F12 :   Function calls

Subviews
Shift+F4 :   Name               |   Shift+F3 :   Functions
Shift+F12:   Strings            |   Shift+F7 :   Segments

Debugger
F9       :   Start              |   Ctrl+F2  :   Stop process
F7       :   Step into          |   F8       :   Step over
Ctrl+F7  :   Run until return   |   Ctrl+Alt+B:  List breakpoints

Other
C        :   Code               |   D        :   Data
U        :   Undefine           |   N        :   Rename
Shift+;  :   Enter comment      |   ;        :   Enter repeatable comment
P        :   Create function    |   Alt+P    :   Edit function
E        :   Set function end   |   Y        :   Declare function type
M        :   Member enumeration |   Shift+F2 :   Run script

[ Immunity Debugger shortcuts ]
F2       :   Set breakpoint     |   F9       :   run
F7       :   Step into          |   F8       :   Step over
Ctrl+F9  :   Execute till ret   |   F12      :   Pause
Alt+B    :   Open breakpoint wi |   Alt+C    :   Open CPU window
Alt+E    :   Open module window |   Alt+L    :   Open log window
Alt+M    :   Open memory window |   Alt+O    :   Open option window
```

Máster en Análisis de Malware, Reversing y Bug Hunting

ENIIT INNOVA IT BUSINESS SCHOOL

excellence inNova Consultoría y Formación S.L.

UCAM Universidad Católica de Murcia

Campus Internacional CIBERSEGURIDAD

Ramon Gonzalez Gaztelupe

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 traduccion de direcciones virtuales y los registros de control que proporciona Intel.

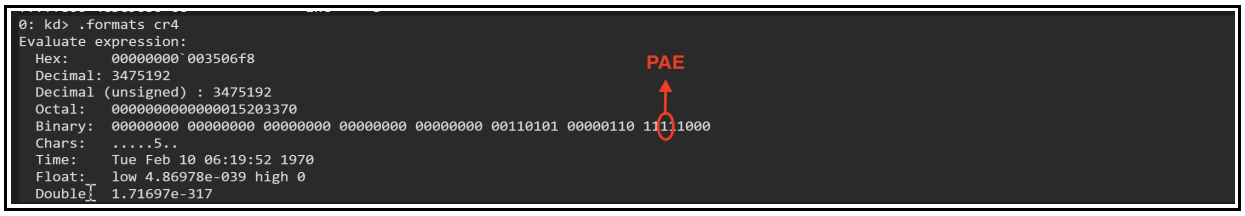
El proceso se realiza de la siguiente manera:

- Reservaremos 2 espacios en Ring3 y inyectaremos la shellcode en uno de ellos
- Usaremos **DeviceIoControl** 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.

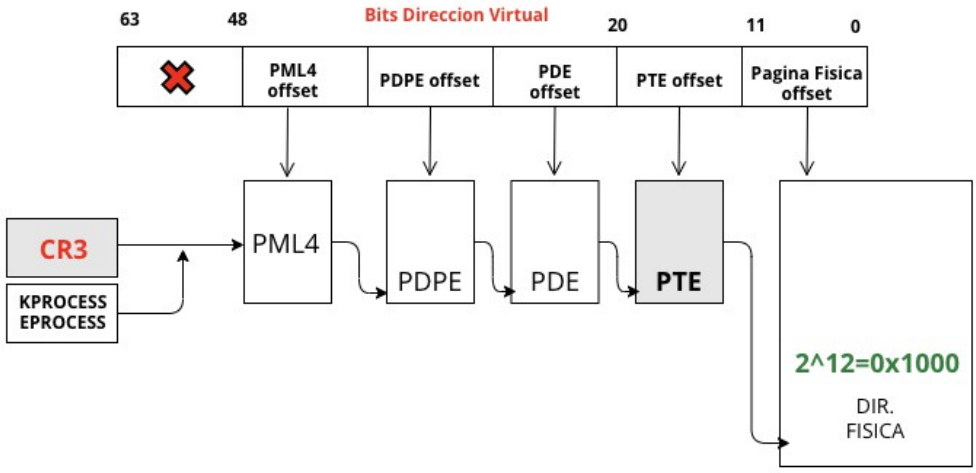
De esta forma mantendremos oculto ese espacio de memoria reservado dentro del proceso que queramos

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:



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 → **2^9=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 pagina correspondiente la cual en 64bits seria

- **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 traduccion, ya que se prescinde de las PTE y se accedería directamente desde PDE

Visto muy por encima el proceso de traduccion 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:

- **VirutalAlloc** → 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";
    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";

    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, deberia 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 ffffdc8fb2be2080
    SessionId: 1  Cid: 0b00    Peb: 002b1000  ParentCid: 1a90
    DirBase: 1456f4000  ObjectTable: ffff830861a89a00  HandleCount:  49.
    Image: RajKit-RING3.exe

1: kd> .process /i ffffdc8fb2be2080
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)

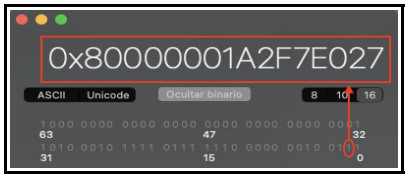
P: Present	**G:** Global
R/W: Read/Write	**AVL:** Available
U/S: User/Supervisor	**PAT:** Page Attribute
PWT: Write-Through	Table
PCD: Cache Disable	**M:** Maximum
A: Accessed	Physical Address Bit
D: Dirty	**PK:** Protection Key
PS: Page Size	**XD:** Execute Disable

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:   10000000000064275760045
  Binary:  10000000 00000000 00000000 00000001 10100010 11110111 11100000 00100101
```

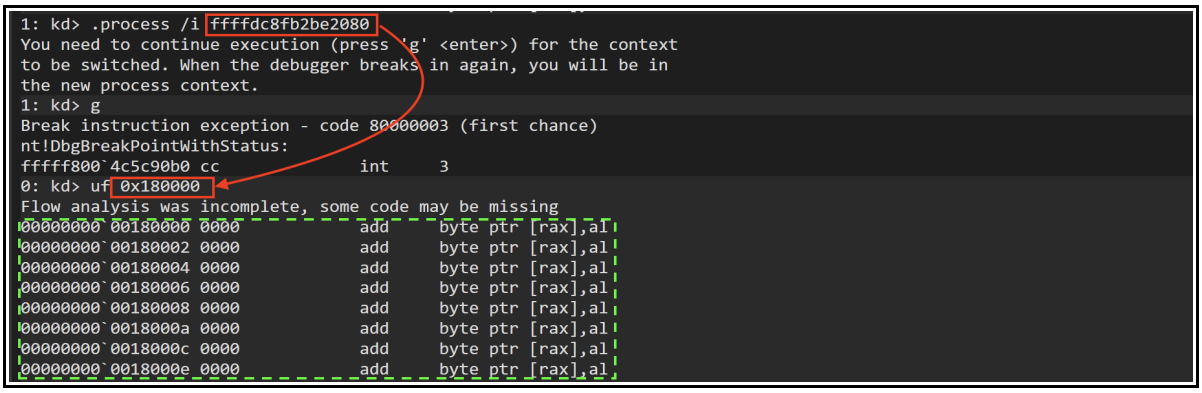
READ

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




```
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--UW-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'**:

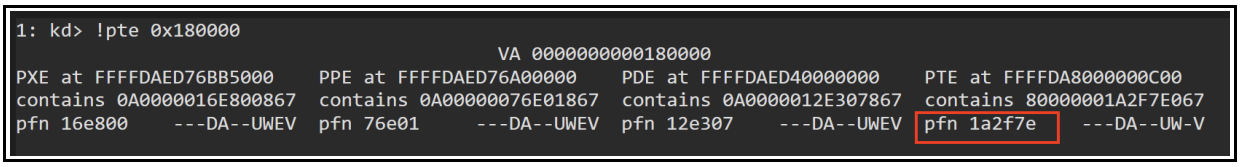


3.- SUBVERSION DE LA MEMORIA

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**".

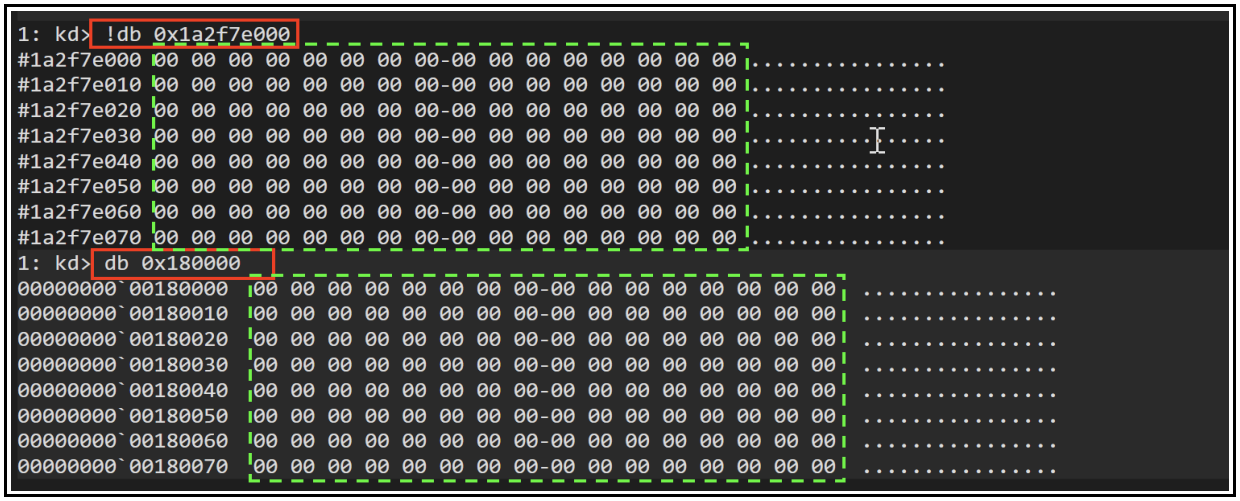
Que es lo que conseguimos con esta técnica? Antes hemos visto que una entrada PTE contiene un marco de pagina llamado **pfn**, que sin entrar en detalles básicamente los PTE obtienen el **pfn** para la siguiente estructura de paginación, por los tanto en un contexto de x64 donde las paginas físicas son 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**:



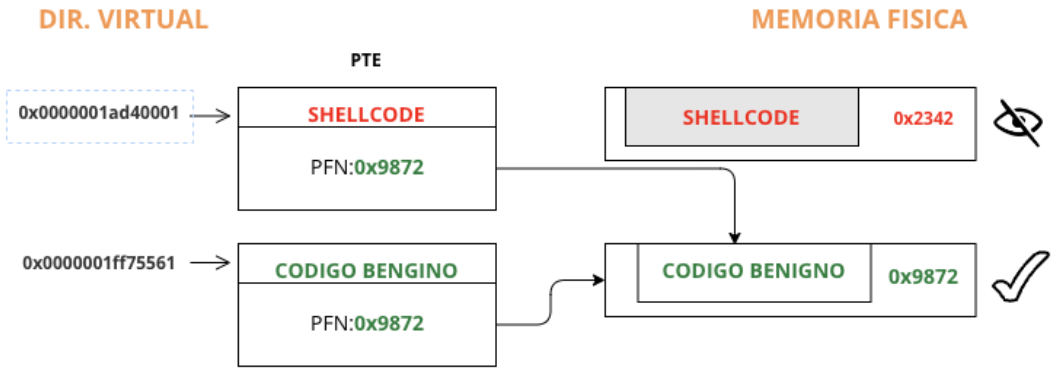
- Extraemos el marco de pagina de PTE y lo multiplicamos por 0x1000
  - **0x1a2f7e → dirección física**
  - **0x18000 → dirección virtual**

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



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 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 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 *nstoskrnl.exe* llamada **nt!MiGetPteAddress** que en el desplazamiento **0x13** contiene la base de los PTE:

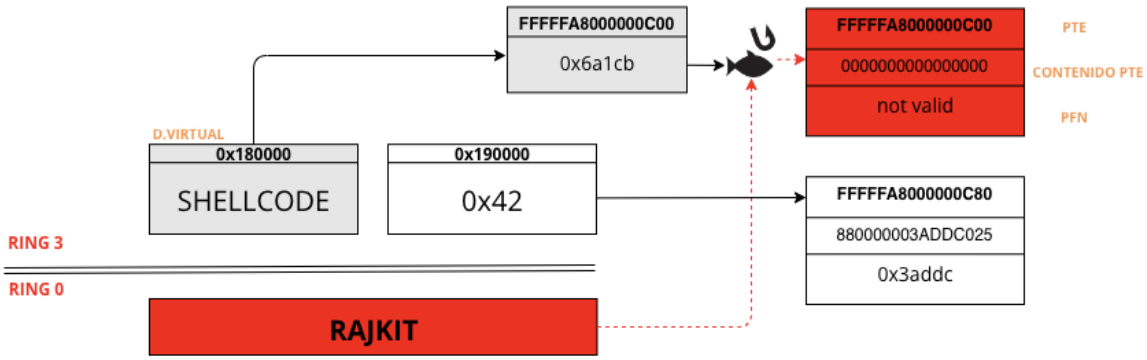
```
1: kd> uf nt!MiGetPteAddress
nt!MiGetPteAddress:
fffff802`46abadc8 48c1e909        shr     rcx,9
fffff802`46abadcc 48b8f8ffffff7f000000 mov rax,7FFFFFFF8h
fffff802`46abadd6 4823c8          and     rcx,rax
fffff802`46abadd9 48b80000000080faffff mov rax,0FFFFFA8000000000h   → DIR.BASE: PTE
fffff802`46abade3 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]**

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 setemos su PTE a **0000000000000000** , y la dirección de la memoria limpia en **0x19000** con un tamaño también de **0x1000**



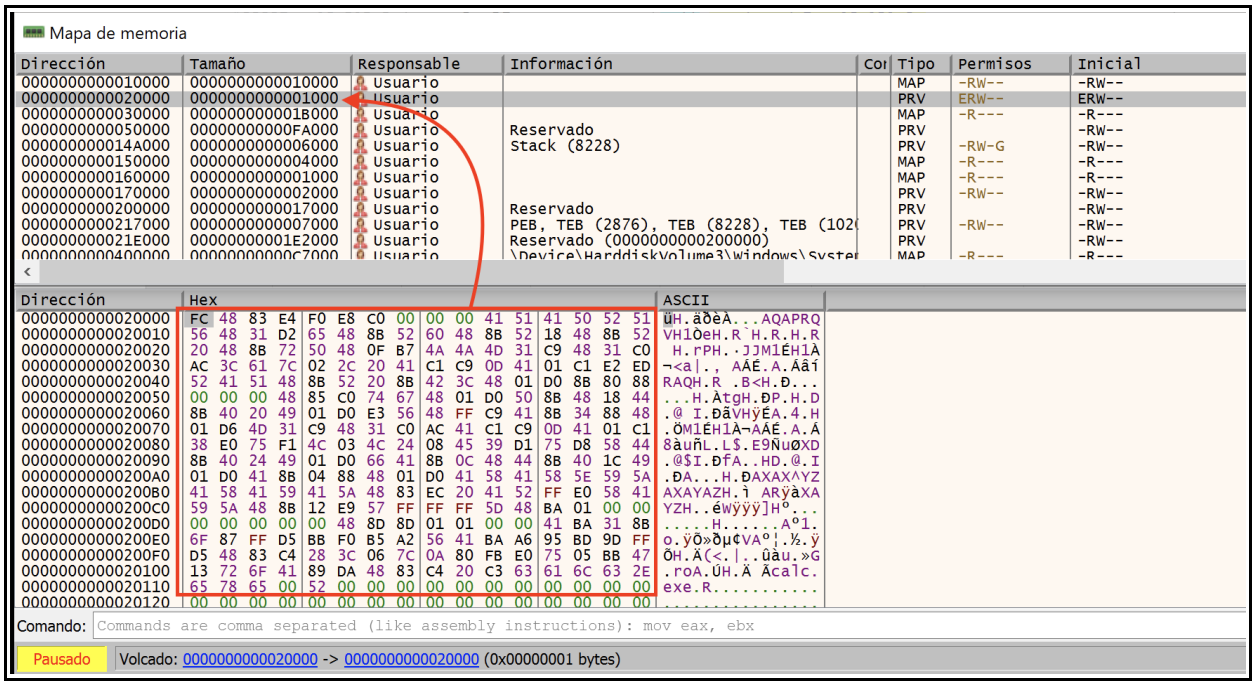
Intento representar en el diagrama la primera fase, recordar que el PFN de la PTE multiplicado por **0x1000** nos devuelve la dirección física real de tal forma que podemos volcar el contenido y mostramos con windbg:

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

```
1: kd> !pte 0x180000
                                           VA 0000000000180000
PXE at FFFFFAFD7EBF5000    PPE at FFFFFAFD7EA00000    PDE at FFFFFAFD40000000    PTE at FFFFFA8000000C00
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 FFFFFA8000000C00
     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 0x180000
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>
```

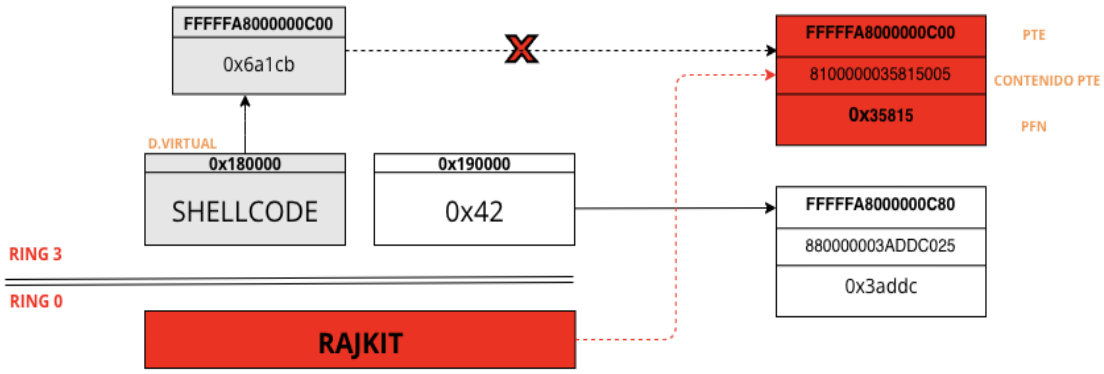
Podemos observar como el volcado de la dirección física **0x6a1cb000** que es la dirección virtual **0x18000** contiene la shellcode descifrada con la clave **RajKit** mediante XOR, lo podemos ver en el debugger en el mapa de memoria:

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

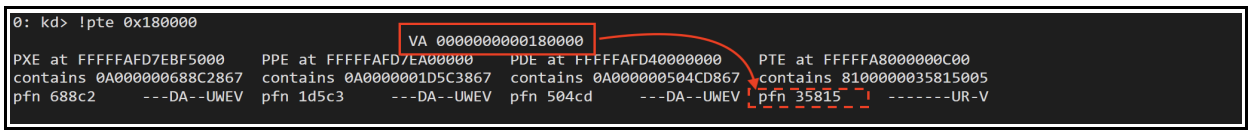


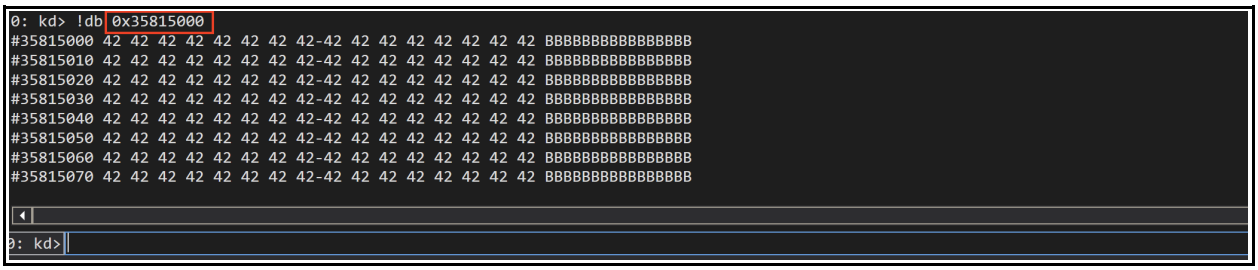
5.2-TECNICA ANTI-FORENSE FASE 2

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:

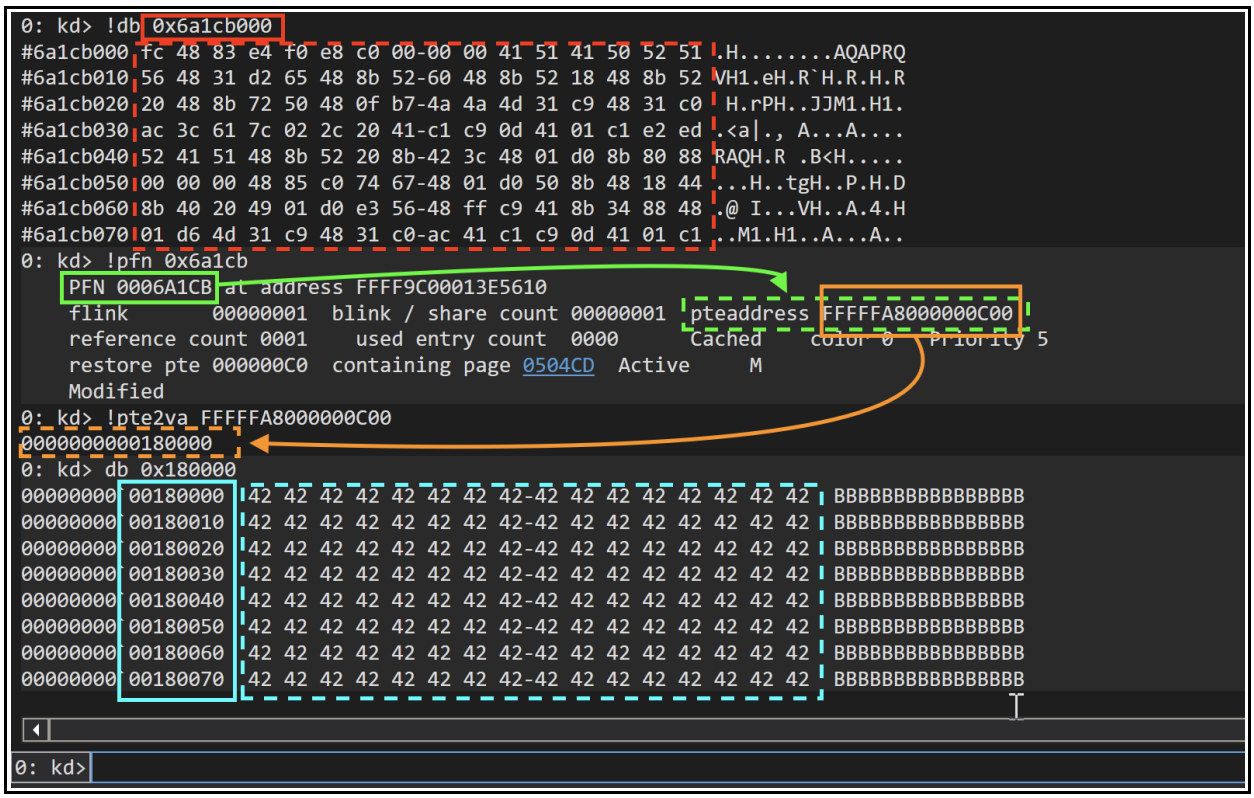


Lo vemos desde el windbg como el volcado del **PFN 0x35815** que en realidad es la dirección física **0x35815000** no contiene la shellcode:



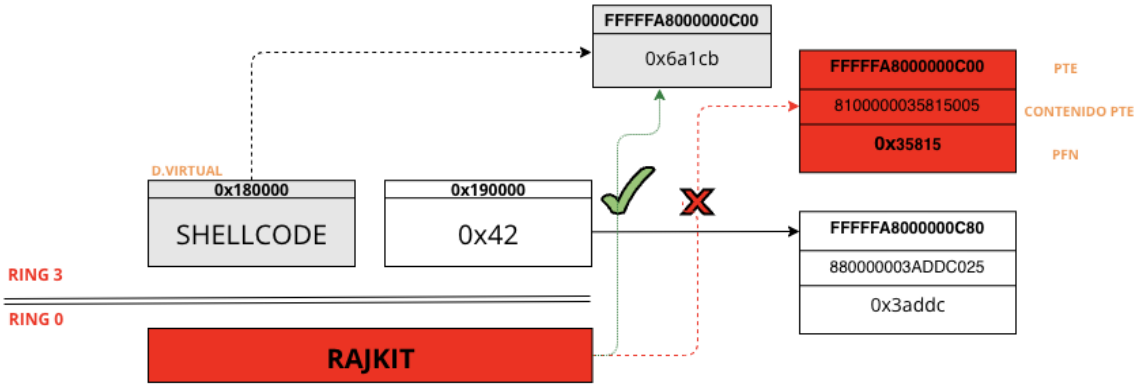


Vemos como volcamos la dirección virtual de la shellcode que si obtenemos su PFN nos devuelve la PTE y si traducimos esa PTE nos devuelve la dirección **0x18000** que a su vez haciendo el volcado en realidad contiene un sleep de "**0x42**":

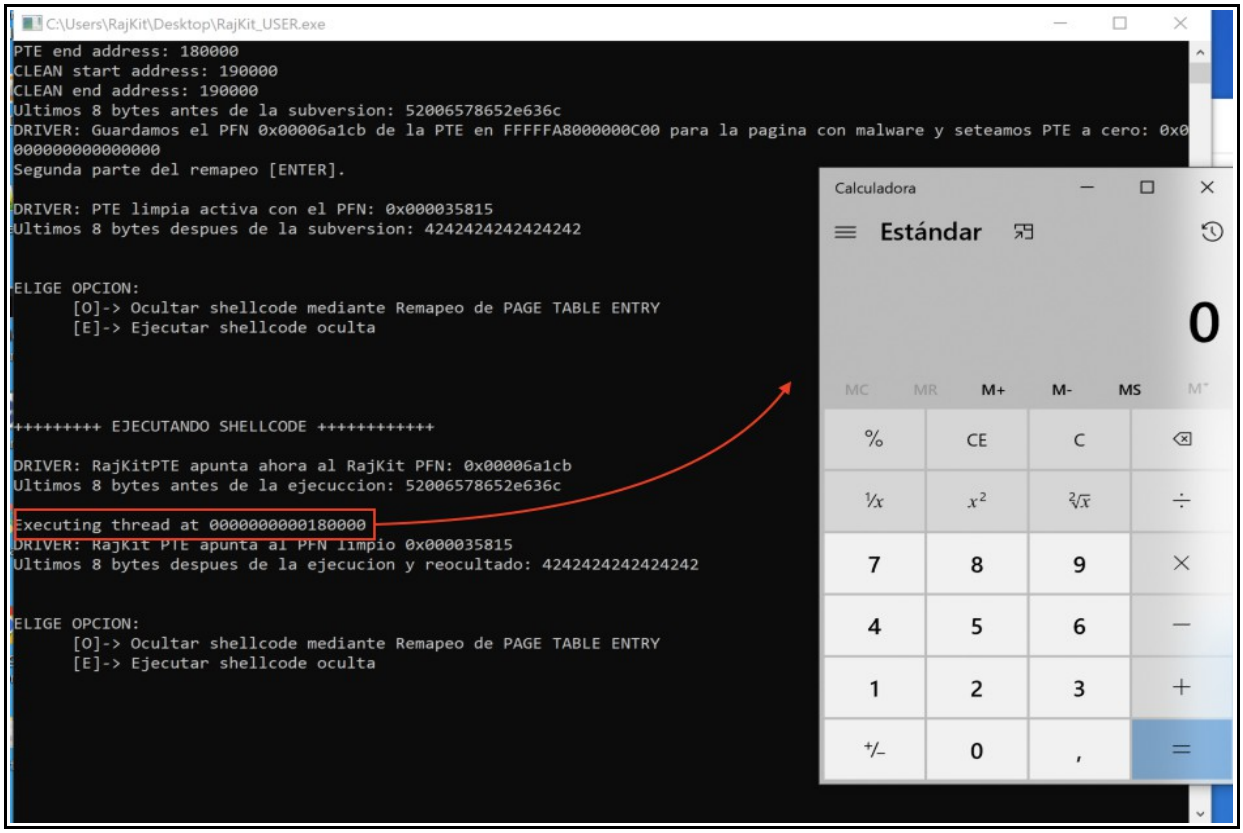


5.3-TECNICA ANTI-FORENSE FASE 3

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:



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