INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 61.

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

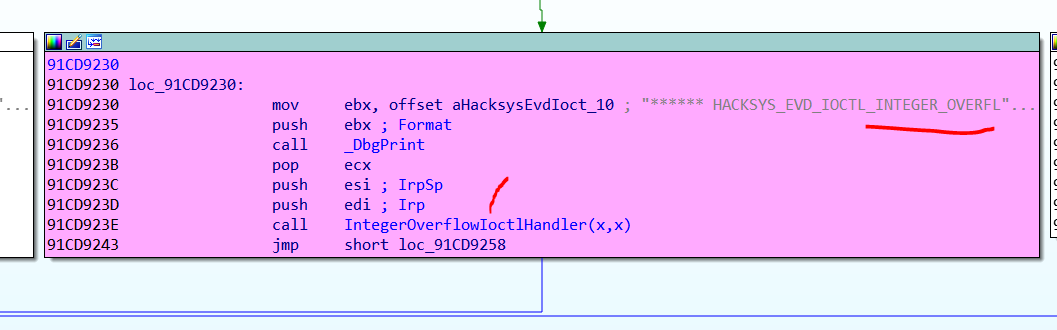
Muchos me preguntan porque no analizarlo directamente en C o C++, el tema es que ya están hechos algunos en c y c++, los metodos son los mismos, por lo tanto portarlos a Python no solo aporta algo nuevo, si no que también nos hace practicar Python y ctypes que es algo importante.

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

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

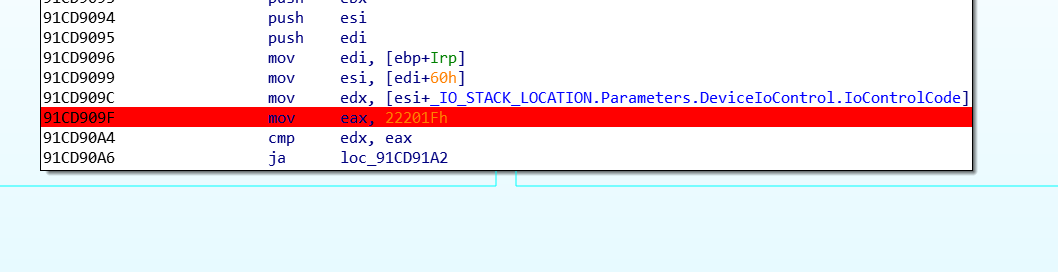
Y alli esta compilado, si quieren intentar alguno pueden debuggear y comparar el resultado que van teniendo en Python con el original, eso ayuda mucho.

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

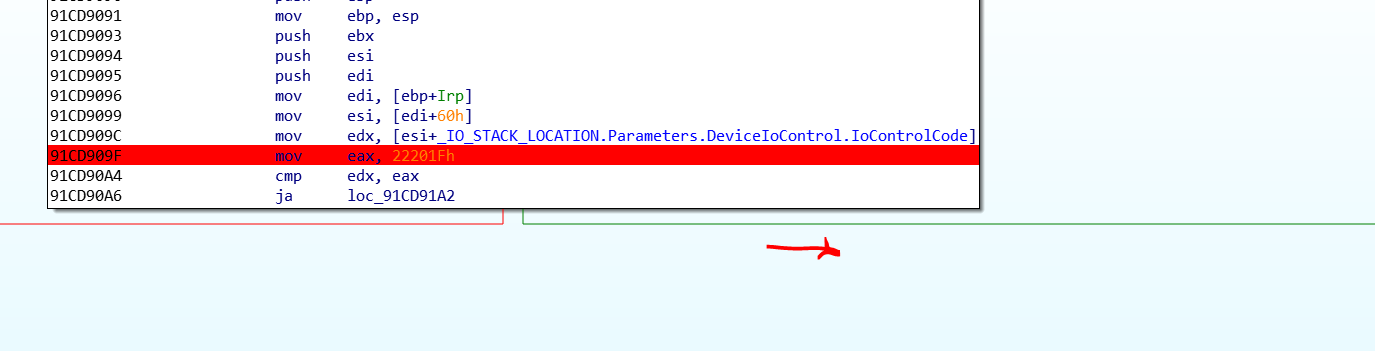


Alli tenemos el bloque que nos llevara el IOCTL que triggerea el integer overflow.

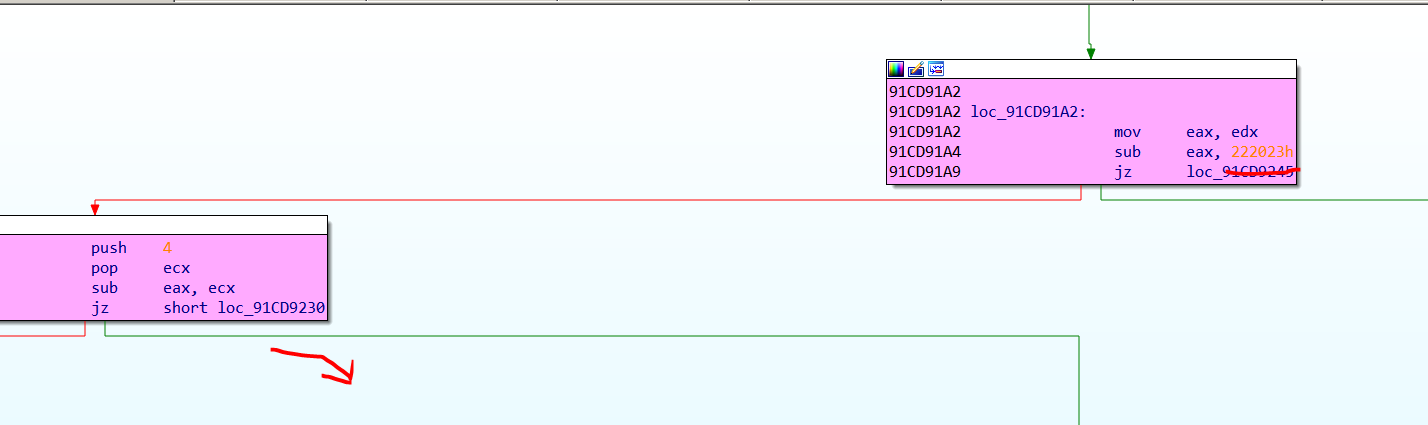
Veamos que IOCTL llega alli, al inicio



EAX es 0x22201f



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.

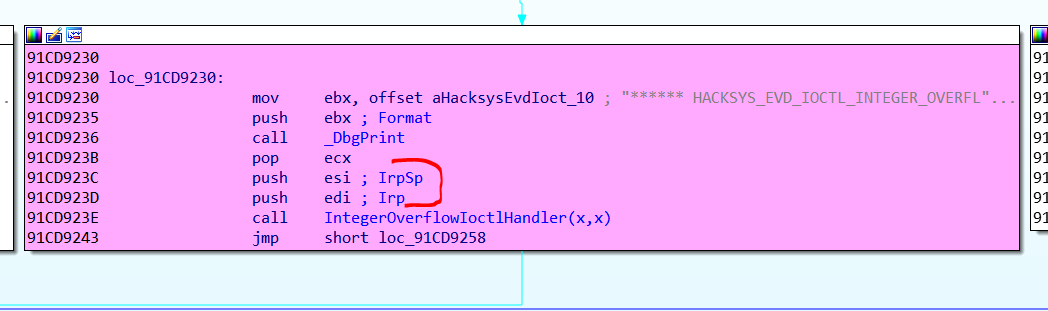
IOCTL-0x222023-0x4=0

IOCTL=0x222023+0x4

Python>hex(0x222023+4)

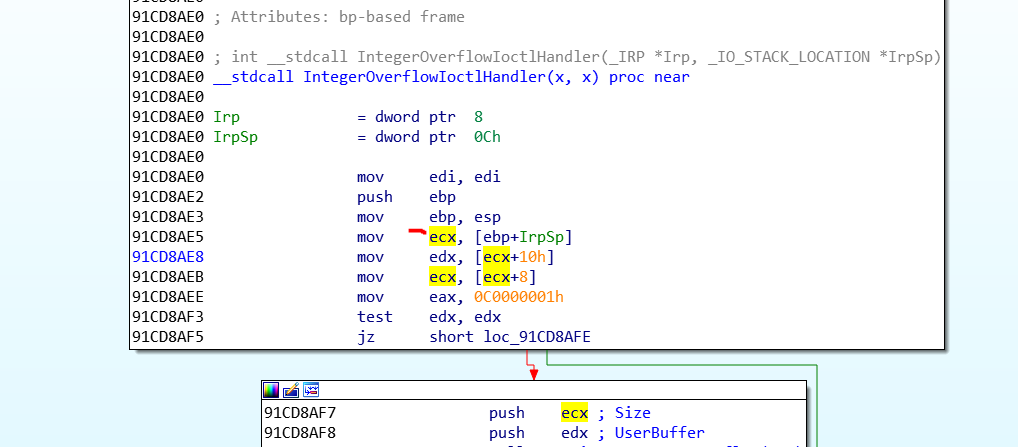
0x222027

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

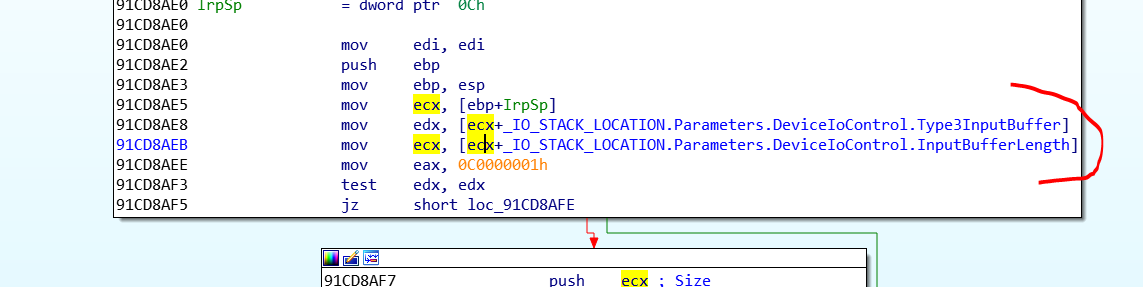


Vemos que al igual que en el caso anterior le pasa dos argumentos a la función, uno la direccion de la estructura IRP y el otro la de \_IO\_STACK\_LOCATION.

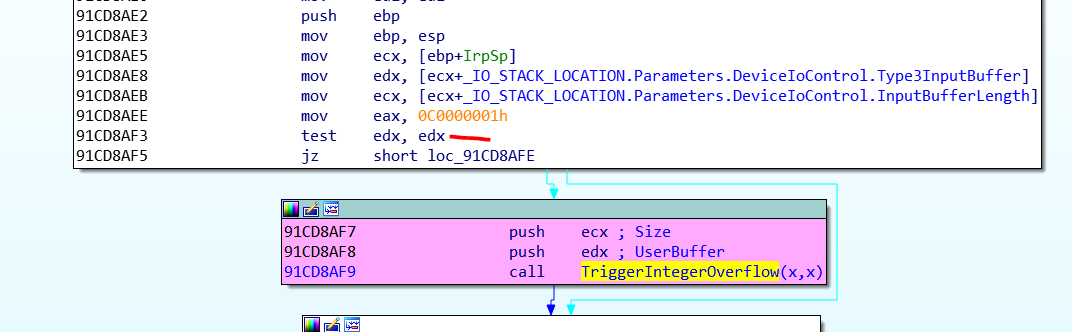
Como ya teníamos importada la estructura \_IO\_STACK\_LOCATION, alli mueve la direccion de inicio de la misma y empieza a trabajar con sus offsets, el campo 0x10, veamos que es apretando T y eligiendo la estructura correspondiente.



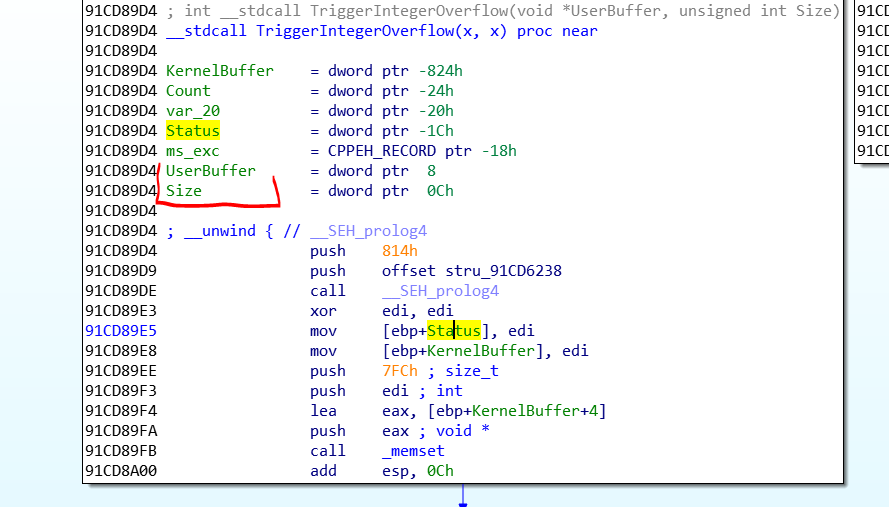
Vemos



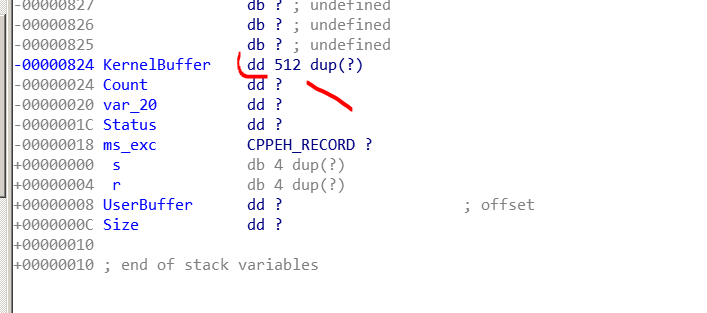
Que son el buffer de entrada y el largo del mismo que le pasamos nosotros, no quiere decir que sea el largo real.



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



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

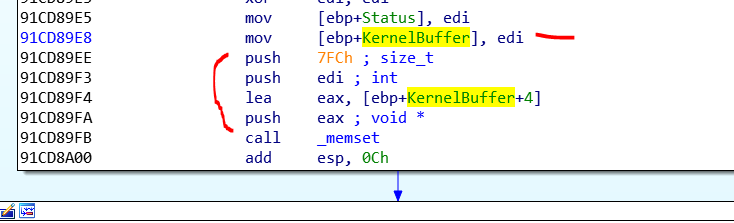


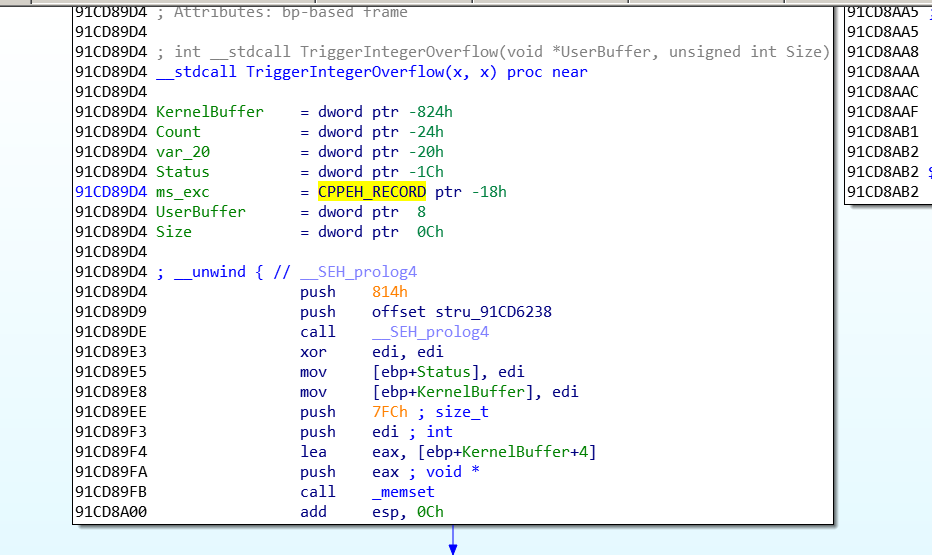
Son 512 decimal por 4 ya que cada componente es un dword (dd) asi que el largo total da.

Python>hex(512 \*4)

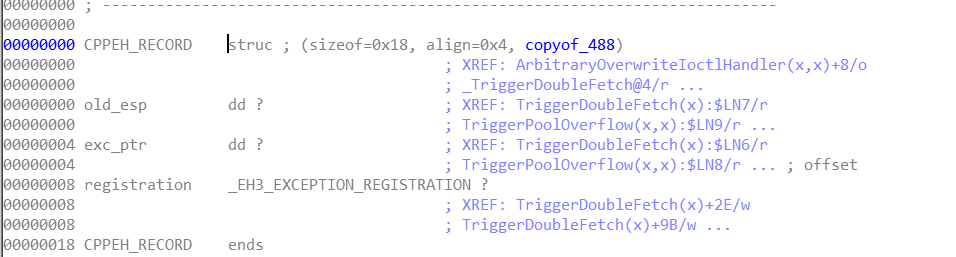
0x800

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.

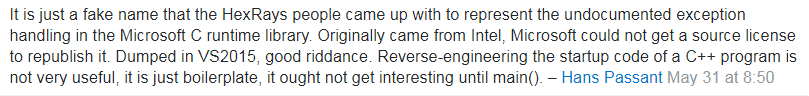




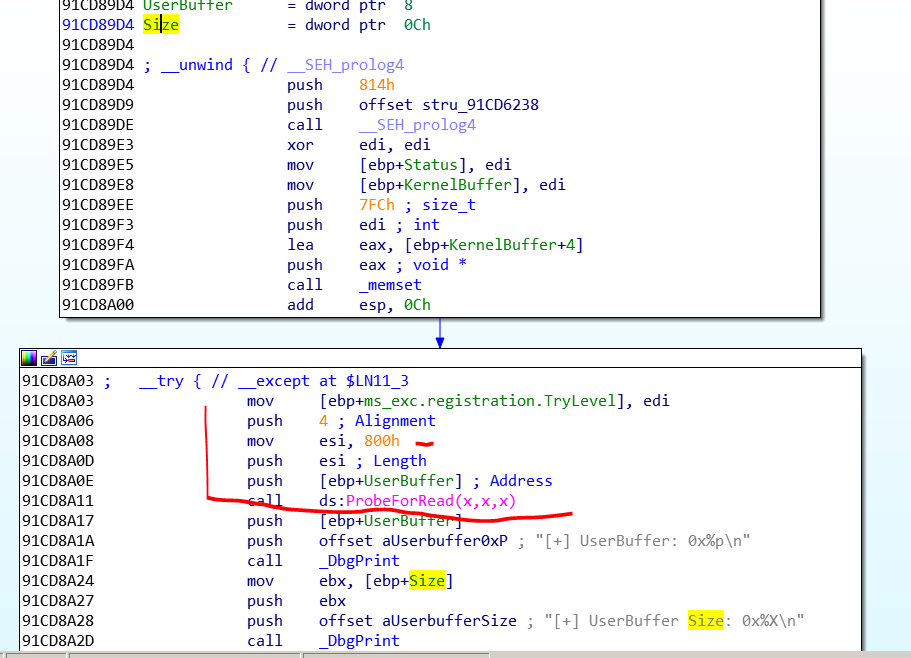
También hay una estructura alli veremos para que sirve, IDA la detecto.



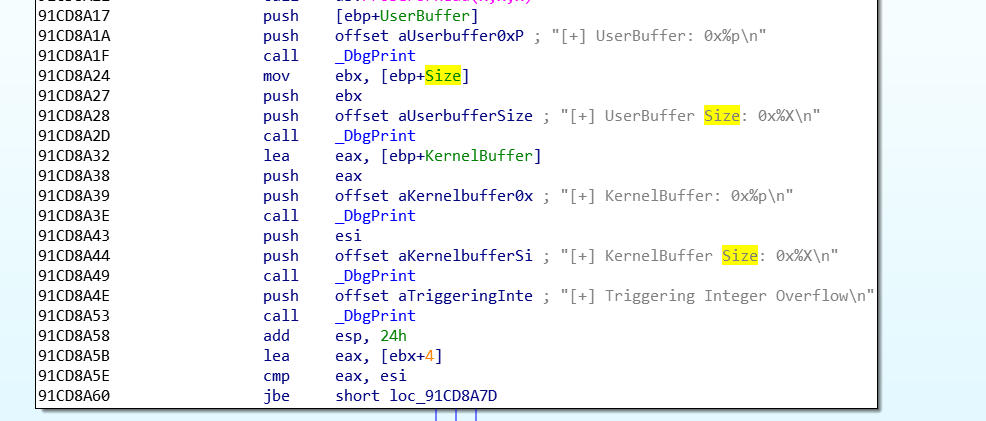
Ya veremos que hace, aquí dice esto.



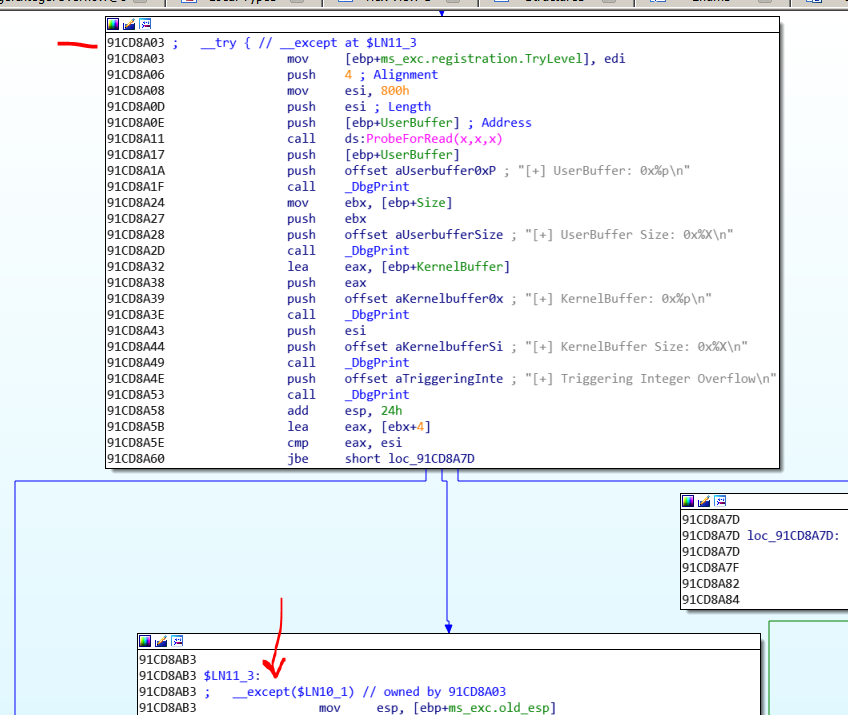
Vemos que cuando chequea el buffer, no usa el valor que pasamos nosotros de size sino 0x800 harcodeado.



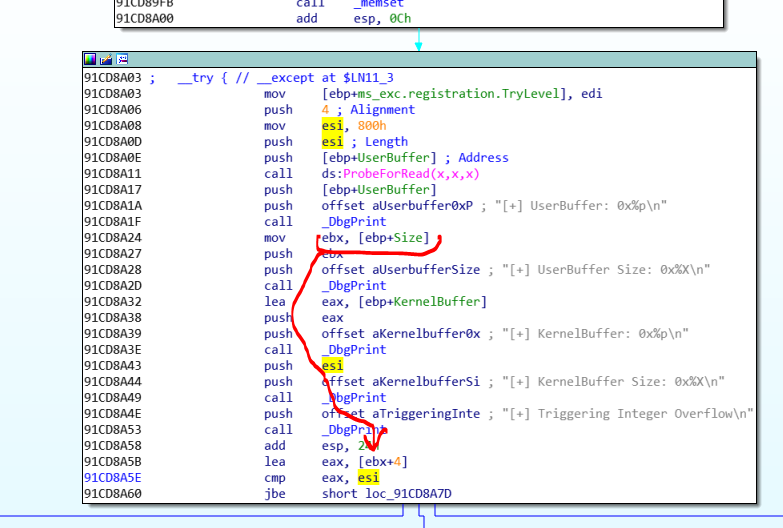
Luego imprime los 4 valores el size que le pasamos del buffer de user, el puntero al buffer de user, la direccion del KernelBuffer y el size del mismo.



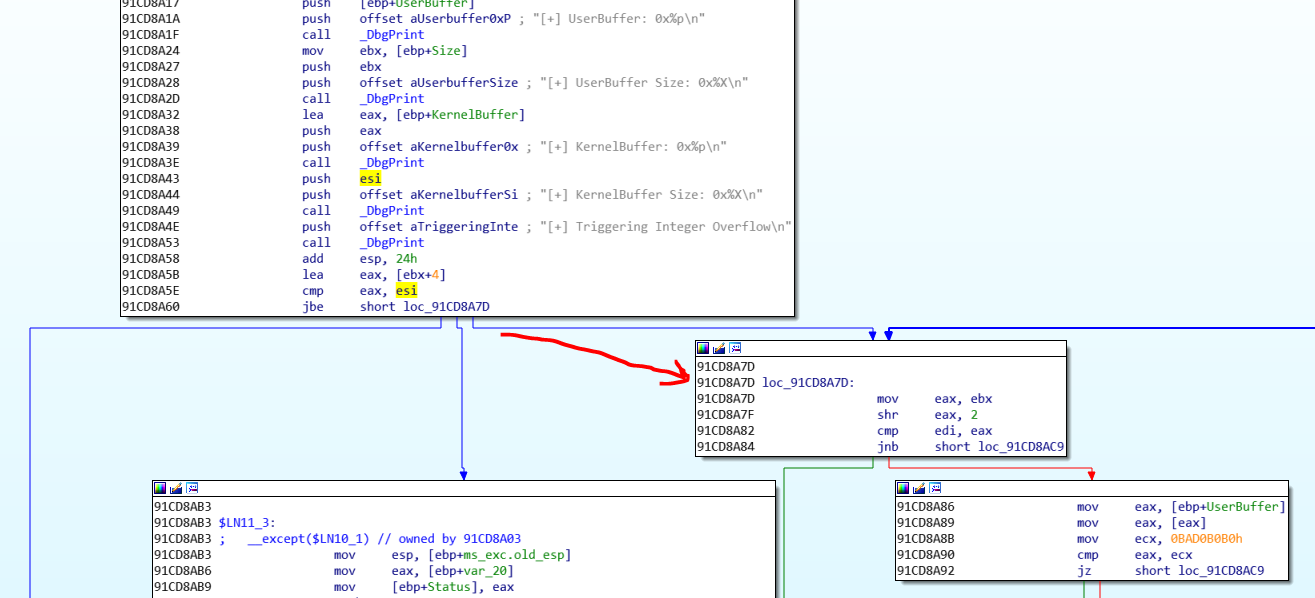
Vemos que IDA nos marca que hay un TRY- EXCEPT o sea que si hay una excepción en ese bloque salta al de abajo, por eso del bloque superior salen tres flechas, dos las normales de la comparación y la otra del try-except.



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



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



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

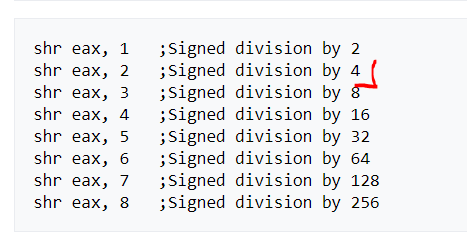
Si lo recortamos a 32 bits como hace el procesador

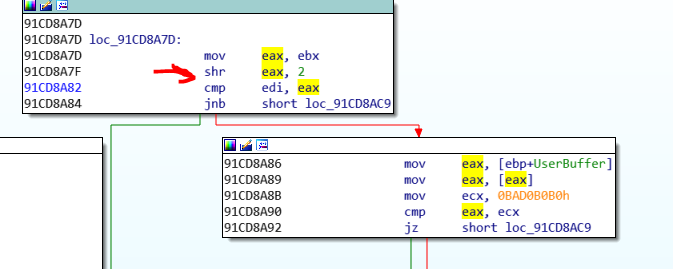
Python>hex((0xffffffff+ 4) & 0xffffffff)

0x3L

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

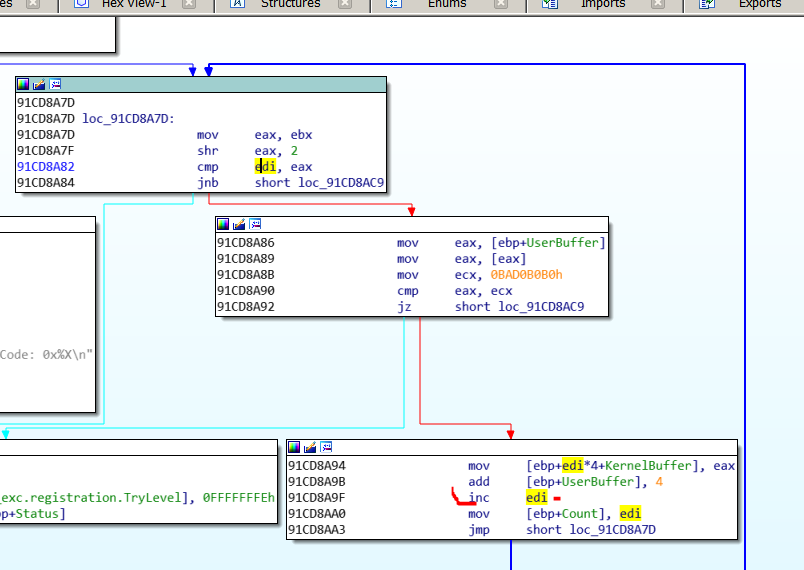
Luego toma el size original y le hace SHR o sea que lo divide por 4 teniendo en cuenta el signo, esto lo realiza porque copiara DWORDS y el indice va de uno en uno, asi el size es el total dividido 4.



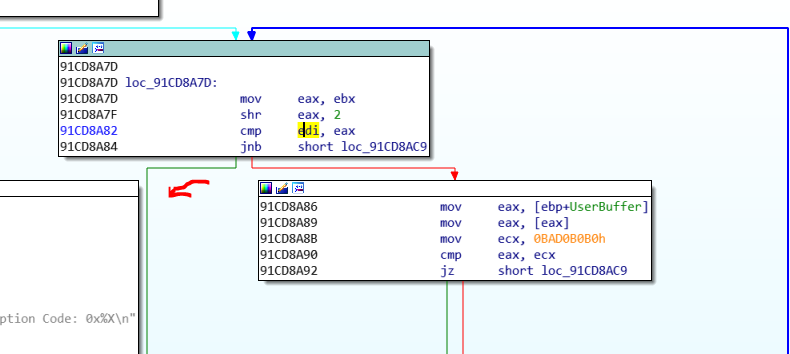


Si nuestro size fuera 0xffffffff al dividirlo por 4 daría 0x3FFFFFFF.

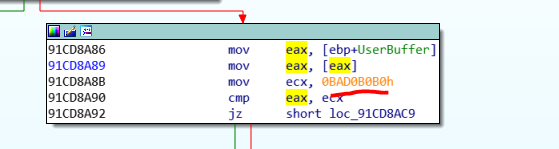
Vemos que es un loop donde EDI es el contador



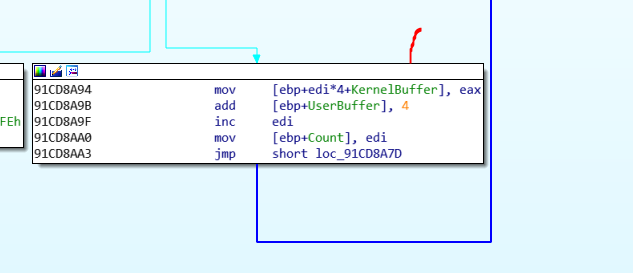
La condición de salida es que EDI no sea mas bajo o sea que sea mas grande o igual para salir, lo cual si empieza de cero y se va incrementando de a uno, dará bastantes vueltas al loop hasta llegar a 0x3fffffff.



Vemos que tiene otra condición de salida muy conveniente



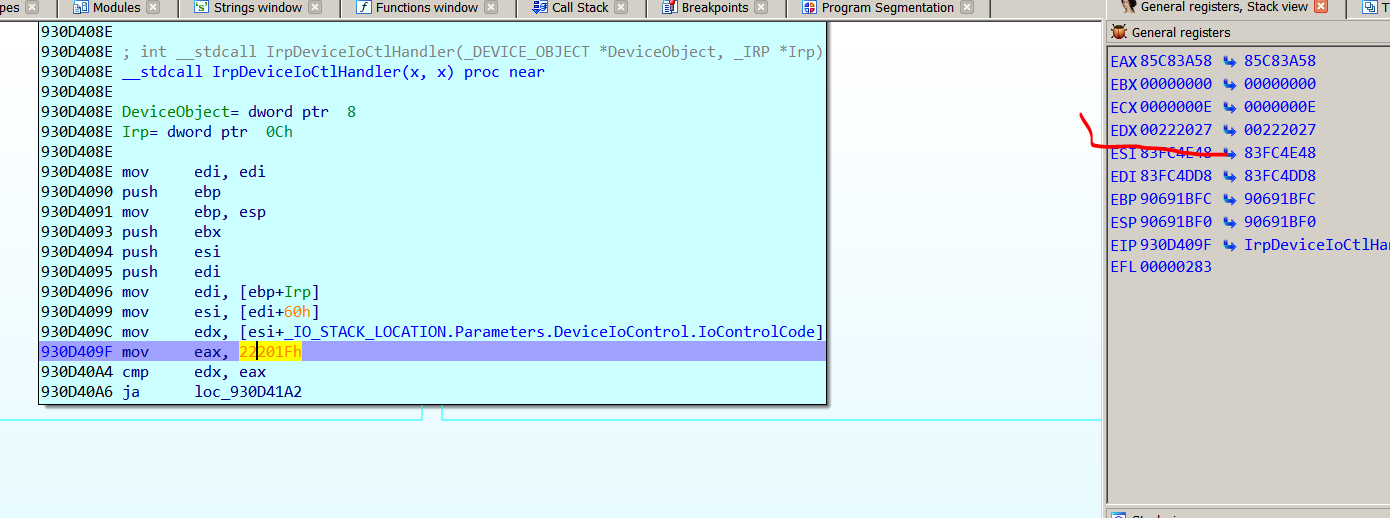
Si lee del buffer de user que le enviamos un valor 0x0BAD0B0B0 saldrá del loop, lo cual hará que no rompamos todo con un size negativo, muy buena gente el programador.



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.

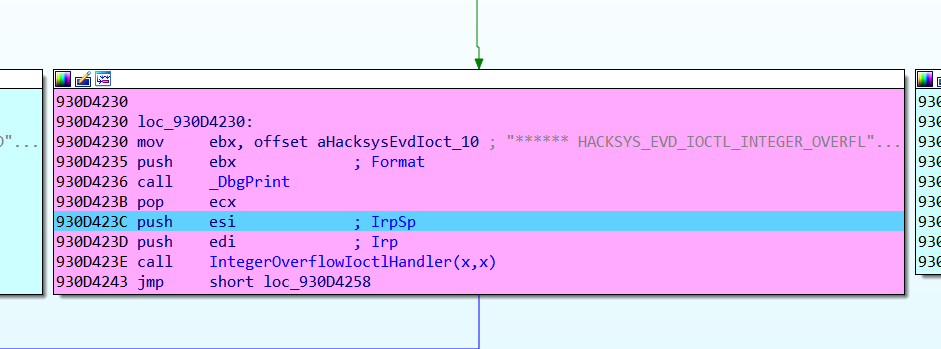
Luego le suma 4 a la direccion del buffer de user, incrementa EDI, lo guarda en Count y listo eso es todo, asi que podemos producir un stack overflow controlado con un size grande, y que incluso podemos salir antes que rompa todo el stack, ya que nos da una forma de salida del loop, manejada por nosotros.

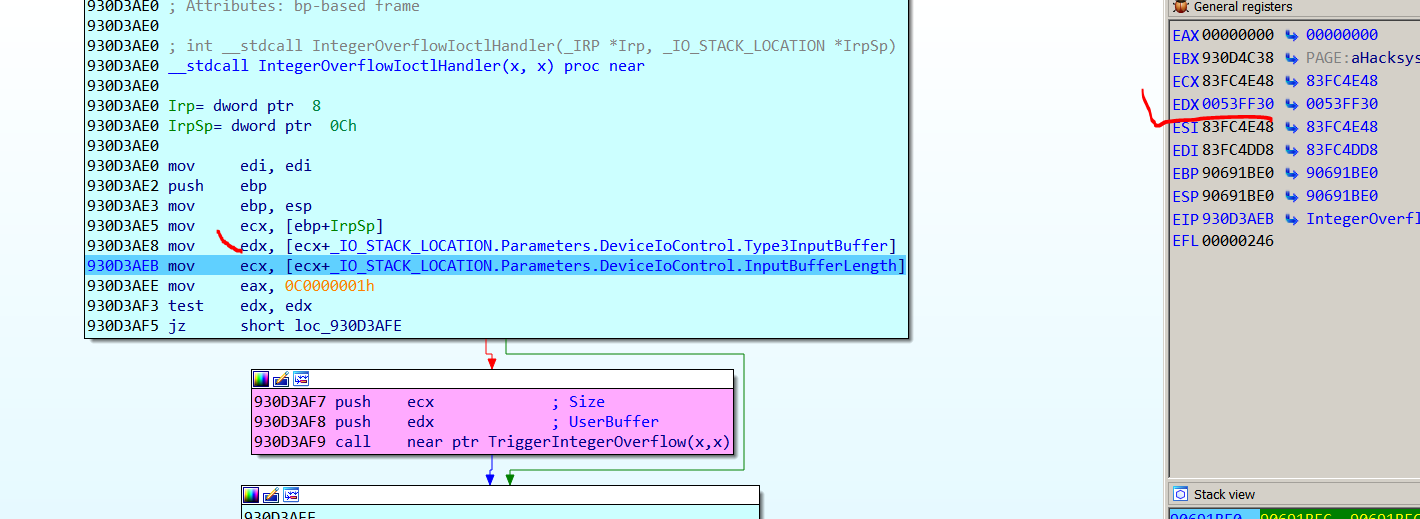
Con eso podemos pisar el return address sin problemas.



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

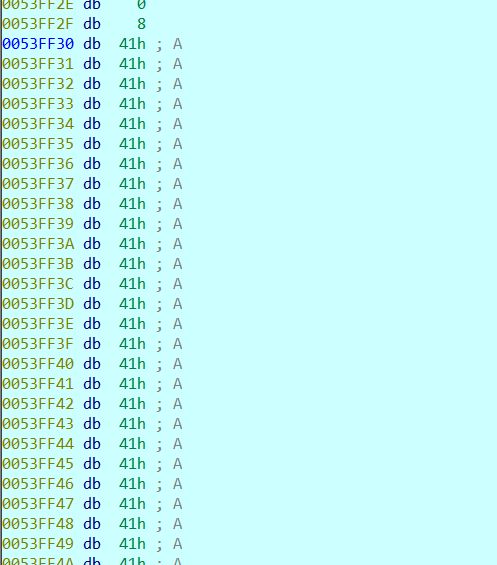
Luego llega al bloque



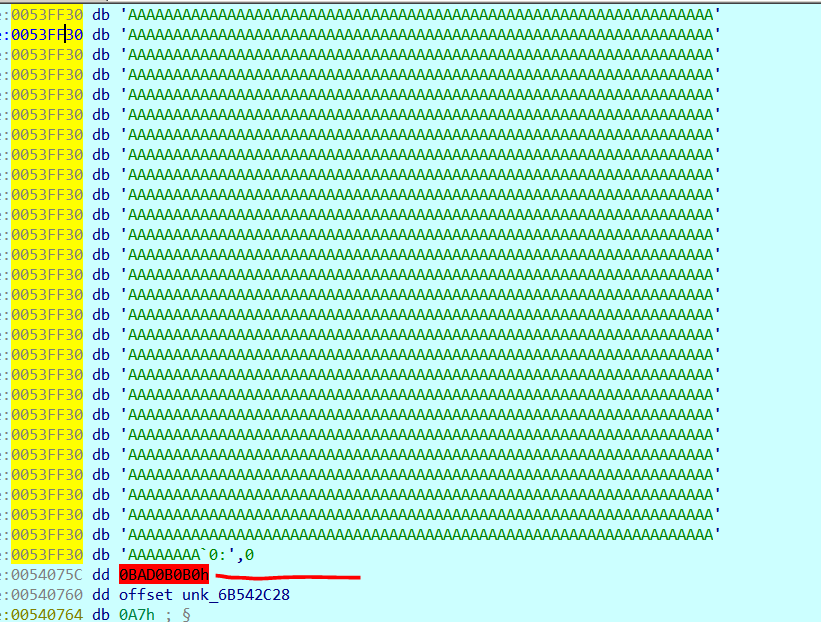


Como vemos alli le pasa el buffer que crea en user, en EDX esta su direccion.

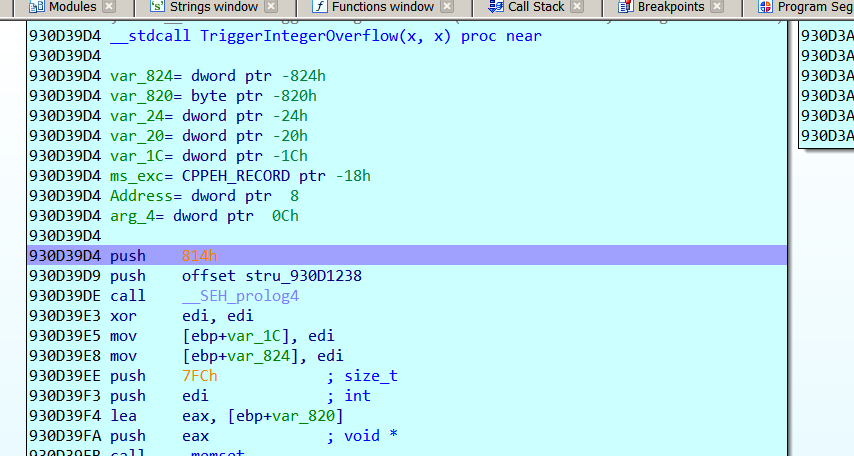
Alli vemos su contenido



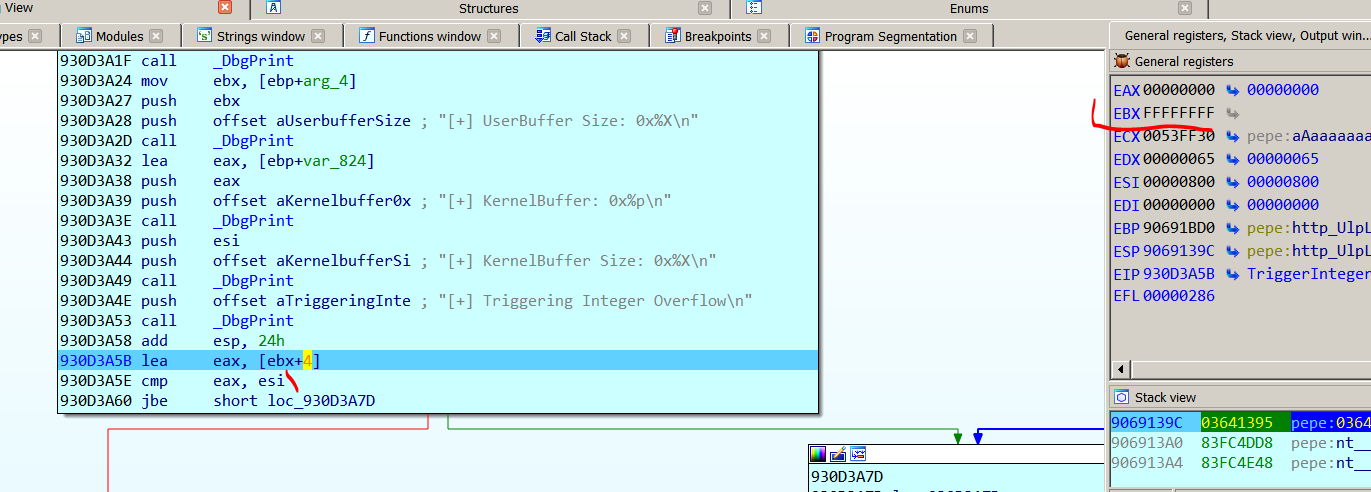
Si creo un segmento ya puedo agrupar las Aes tipeando A



Y veo el DWORD de salida por ahí.

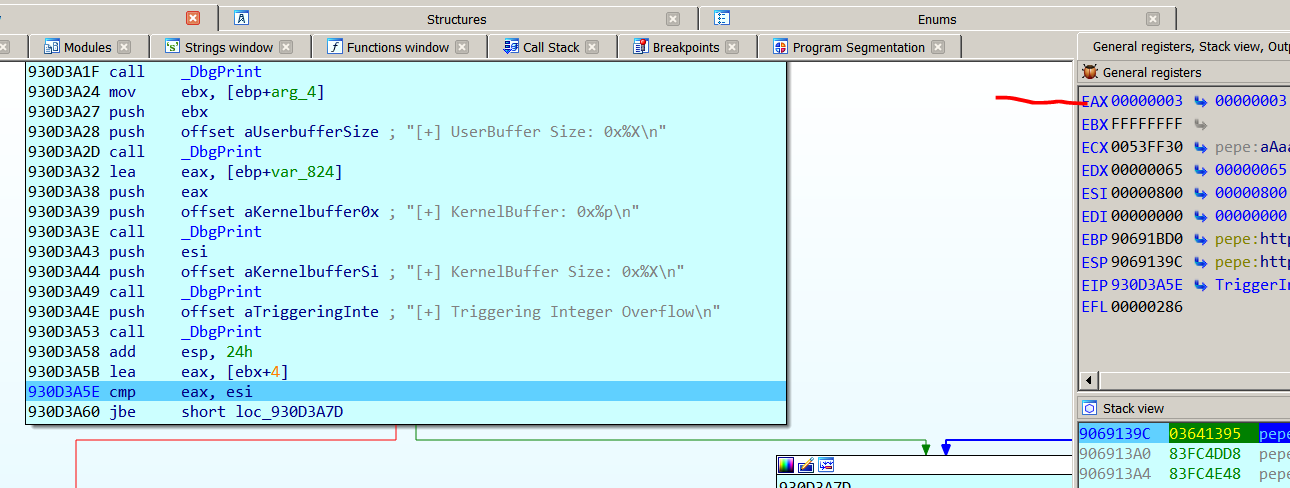


Llego a la función que triggerea el Integer Overflow.

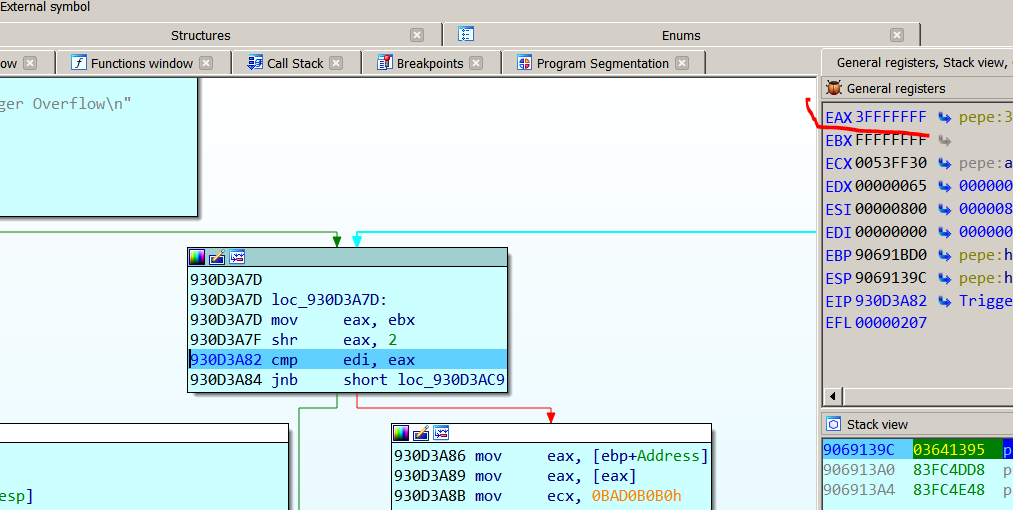


Vemos el size 0xFFFFFFFF que le pasa, o sea que realizo el mismo razonamiento que yo.

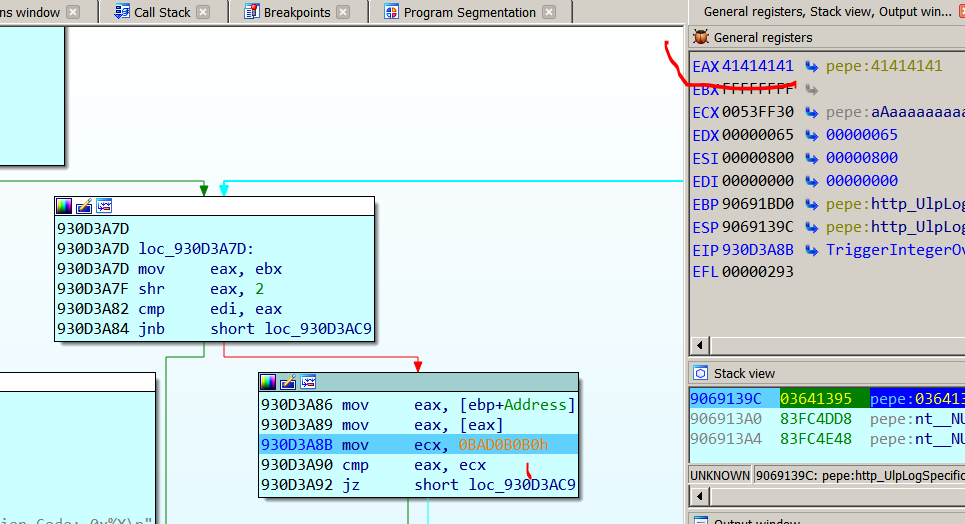
EAX=3 es menor que 0x800.



Despues del SHR

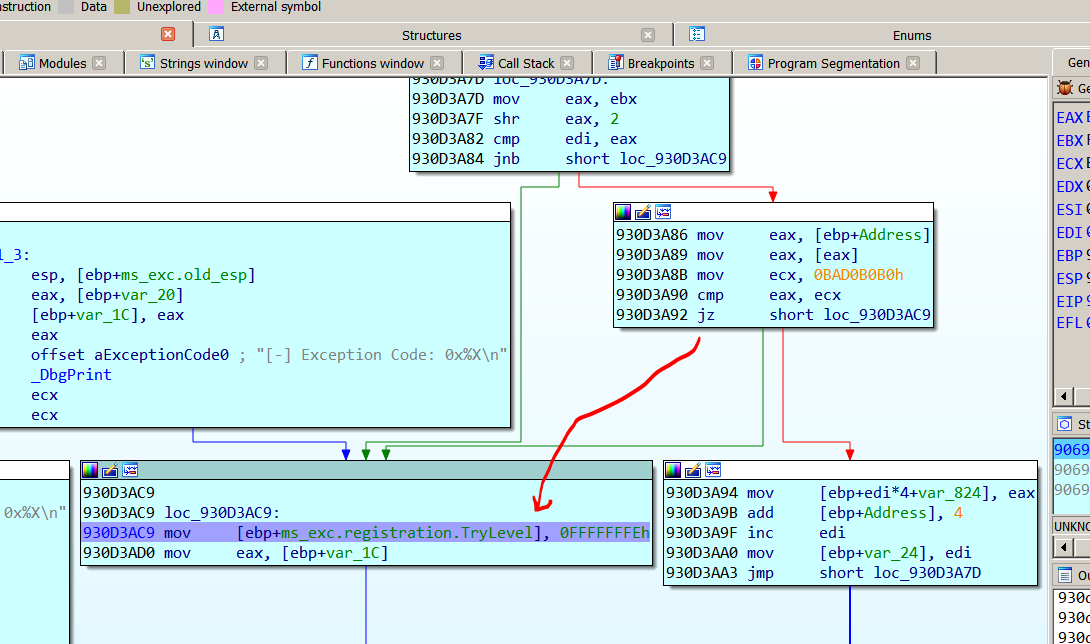


Lee el contenido del buffer de user y es 0x41414141.

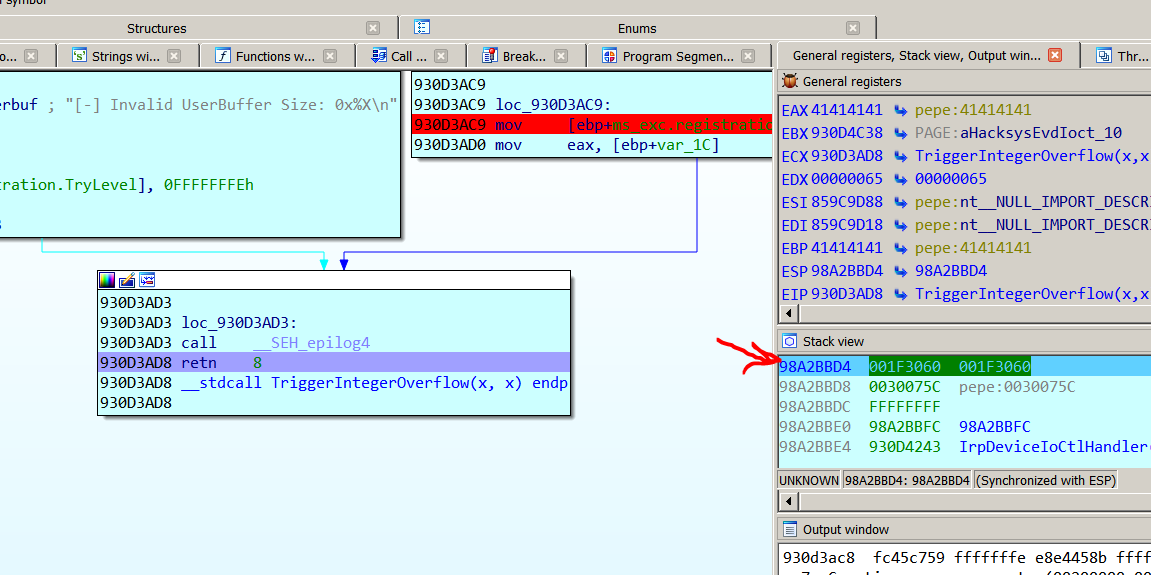


Como no es la constante 0x0BAD0B0B0 de salida lo copia al kernel buffer del stack.

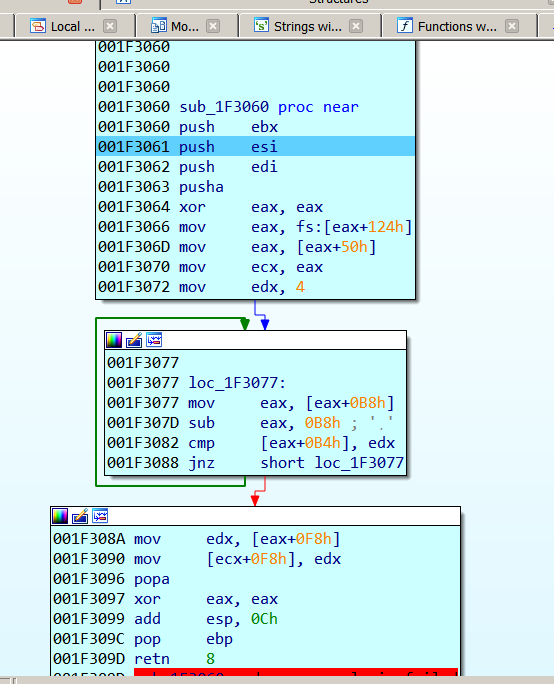
Pongo un breakpoint alli para que pare al terminar de copiar.



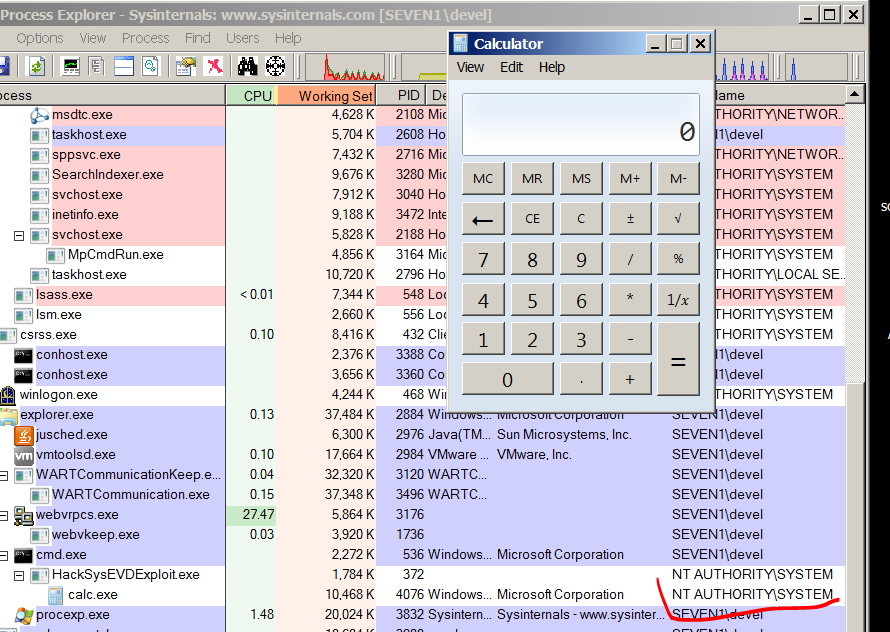
Vemos que cuando llega a pisar el return address le pasa un puntero a otro buffer con el shellcode y como sabemos acá no hay SMEP asi que salta alli a ejecutar el shellcode de steal token.



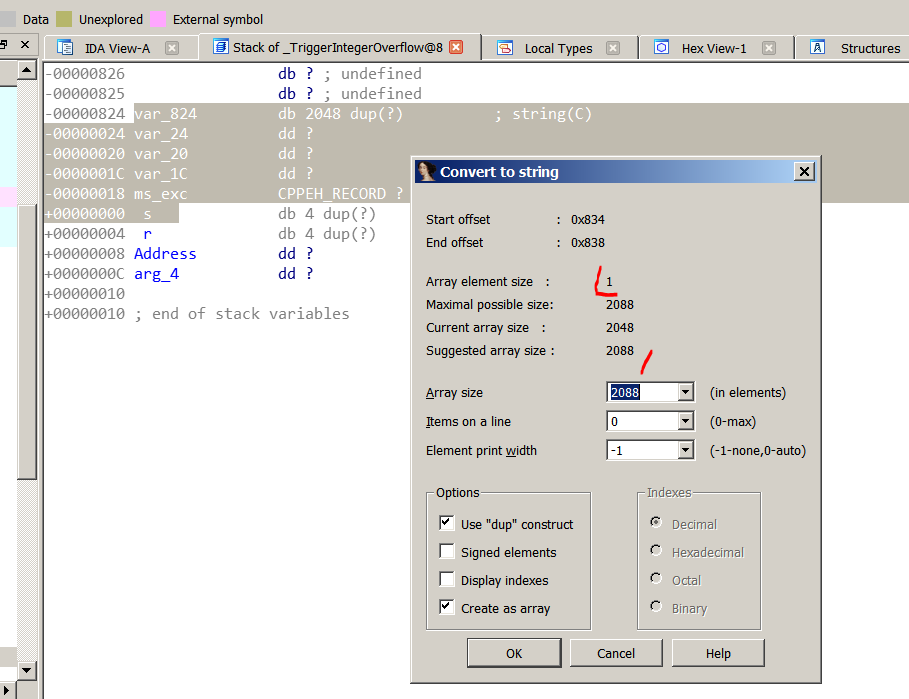
Una vez que creo el segmento lo hago código con la tecla C y creo la función con CREATE FUNCTION y se ve mas lindo.



Alli vemos la calculadora system



Ahora la idea es hacer lo mismo nosotros en Python.



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

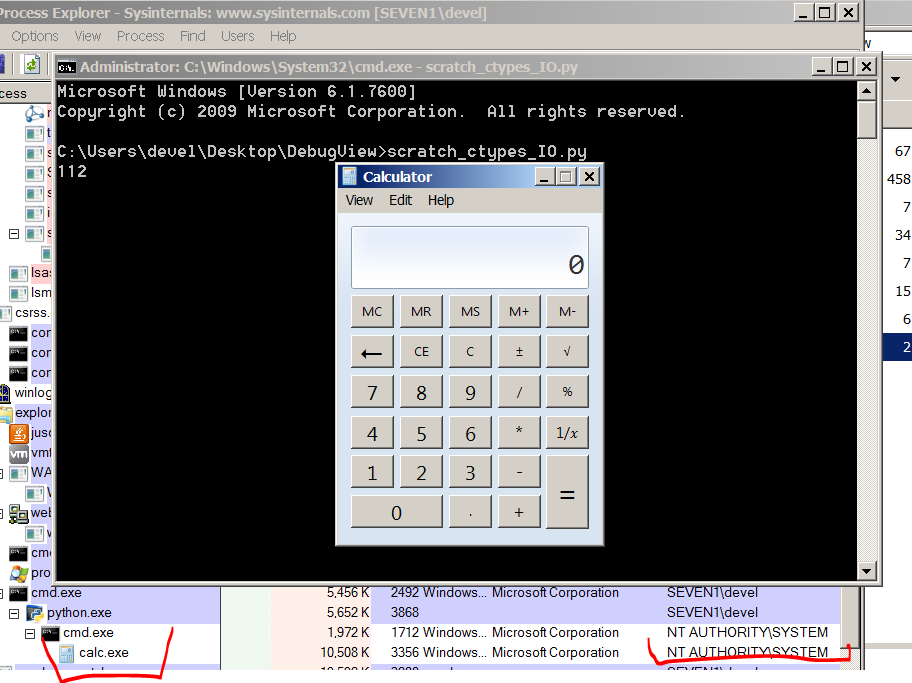


Le cambio el IOCTL; le pongo el size del user buffer igual a -1, le paso el puntero al user buffer ´para que pise el return address, en el exploit en C el realizo dos buffer uno para pisar el return address y otro con el shellcode yo lo metí todo en uno solo.

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, asi que gracias al programador jeje.



Hasta la parte 62

Ricardo Narvaja