EXPLOITING Y REVERSING

USANDO HERRAMIENTAS

GRATUITAS (PARTE 11)

ROPEANDO PASO A PASO

Normalmente hay herramientas que pueden armar un ROP para casos sencillos.

En los casos difíciles generalmente dichas herramientas no los solucionan o solo lo hacen parcialmente dejando que uno complete a mano el trabajo que no pudo hacer la tool.

Que mira un EXPLOIT WRITER para saber si un ROP será dificil o facil?

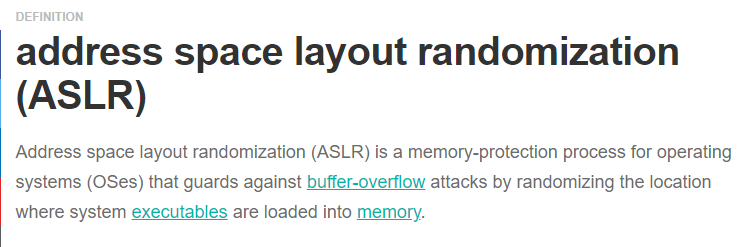
Aquí hay una lista de puntos a observar, cuanto más SI podamos contestar será más fácil, mientras que algunos NO de esta lista complicarán el trabajo, algunos más otros menos. Veamos la lista, luego iremos desgranando las respuestas y sus consecuencias.

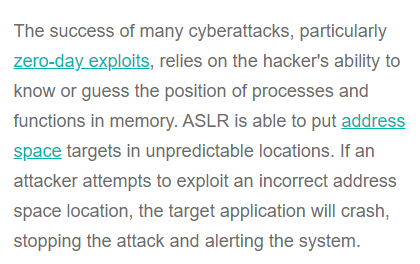
Conteste estas preguntas antes de ROPEAR con SÍ/NO (están ordenadas por orden de importancia).

1. Tiene el proceso módulos que no tengan ASLR?
2. Tiene en algun modulo que no tenga ASLR importada la funcion VIrtualAlloc o VirtualProtect?
3. Están los datos ya ubicados en el stack para comenzar a ropear?
4. Puedo pasar cualquier caracter o sea no hay caracteres inválidos o hay pocos?

Para contestar la primer pregunta veamos primero

QUE ES ASLR?





Lo que nos dice alli que los ejecutables compilados con ASLR no se ubican en la memoria siempre en direcciones fijas, lo cual hace más complicada la explotación pues si todos los ejecutables y dlls de un proceso están compiladas con ASLR, no tendremos direcciones fijas donde saltar y hacer un ROP.

O sea que la primera medida es ver si hay algun modulo sin ASLR, ya que el mismo no se setea por proceso, sino que cada ejecutable al ser compilado, individualmente puede estar compilado con ASLR o no.

Una posibilidad de vencer la protección DEP + ASLR es encontrar algún leak de direcciones de memoria en el sistema o en el mismo proceso que nos devuelva en tiempo de ejecución la dirección de algún módulo ejecutable, lo cual nos permita armar el rop de acuerdo a la dirección obtenida leakeando.

Como vamos paso a paso siempre se empieza por los casos más sencillos primero, veamos si en nuestro proceso hay algun modulo sin ASLR.

Tiene módulos que no tengan ASLR?

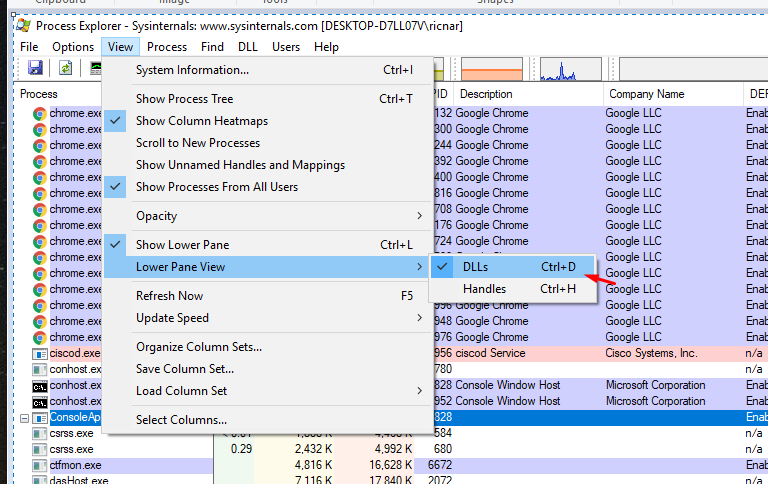
Corremos el ejecutable del ejercicio y miremos nuevamente en el PROCESS EXPLORER.

El process explorer 16.31 que se encuentra en la página de Microsoft tiene un bug que no le funciona mostrar el aslr de cada módulo.

Pongo el link a la versión 16.21 que si le funciona y es la que uso yo:

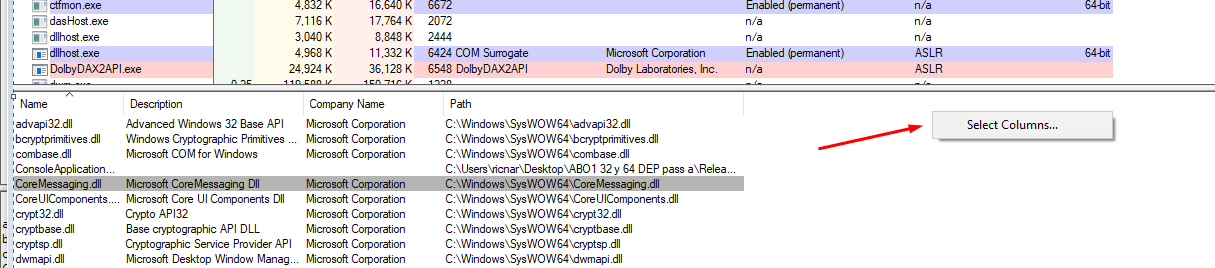
https://drive.google.com/file/d/1cgF49ZS\_GUskxCUJ7ZLDEsoz710mq106/view?usp=drivesdk

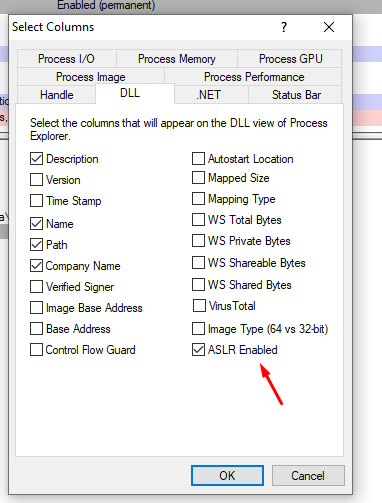




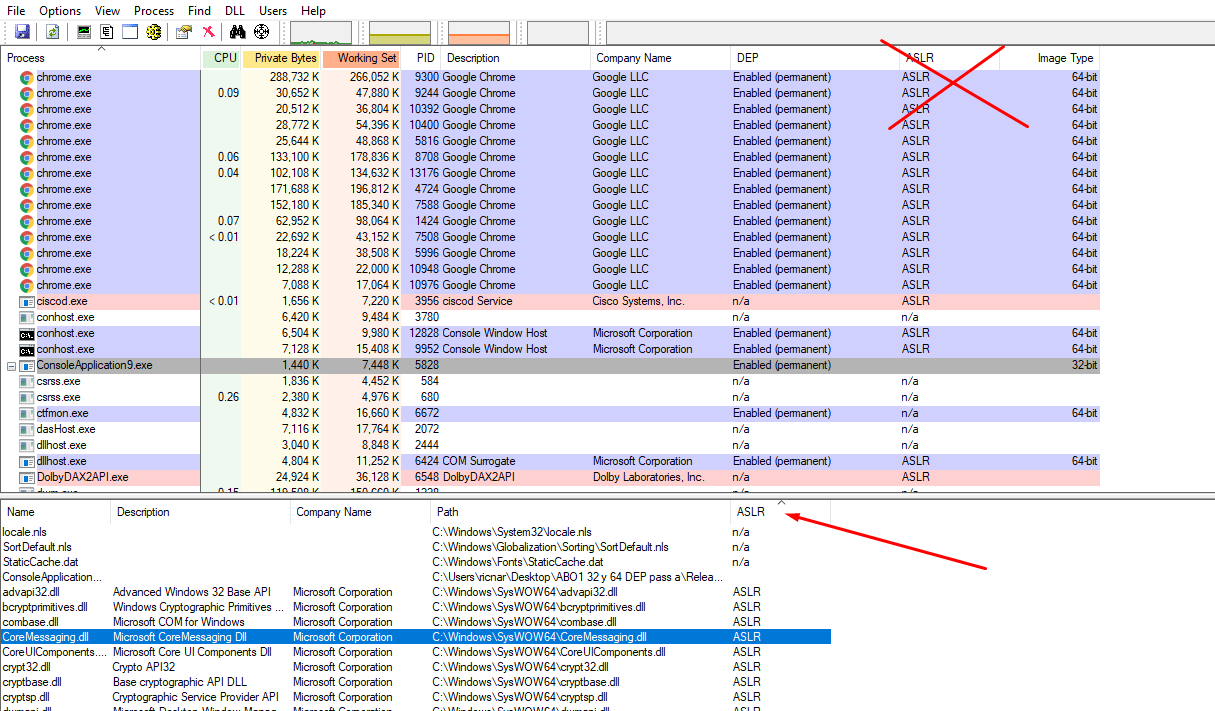
Seteamos para que en la parte inferior se vean los módulos que posee el proceso.

En la parte inferior hacemos click derecho en las columnas y elegimos SELECT COLUMNS.





Elegimos ASLR Enabled y lo marcamos.

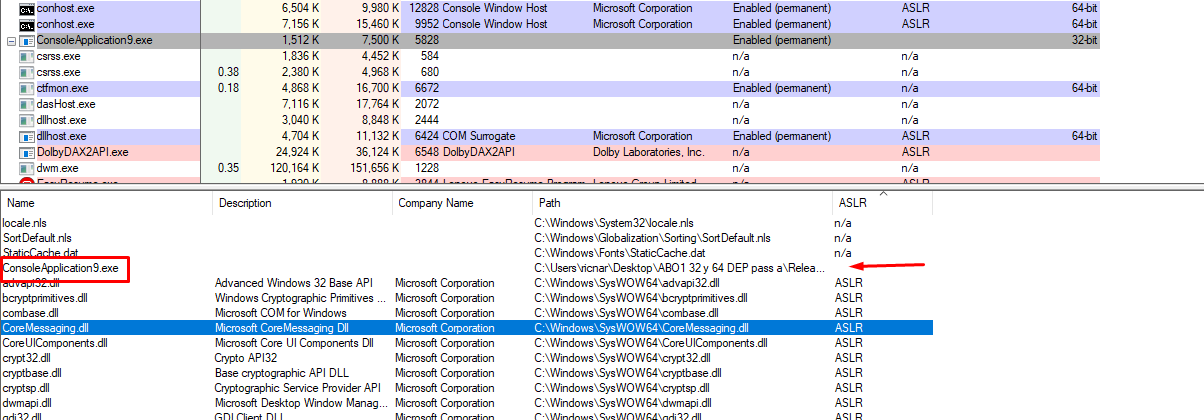


Así que vemos en la parte inferior, la columna importante que nos marca cada módulo si fue compilado con ASLR o no.

La columna de la parte superior, también llamada ASLR no nos servirá pues en un mismo proceso, puede haber módulos con y sin ASLR, así que el valor genérico para todo el proceso no sirve.

Si hago click en la columna ASLR que agregamos para que ordene los módulos por si tiene ASLR o no, vemos que hay alguno que no tiene ASLR, o sea que sus direcciones seran fijas y podremos utilizarlo para ropear.

Cualquier DLL o EXE sin ASLR que se encuentre en la lista inferior de nuestro proceso, servirá.



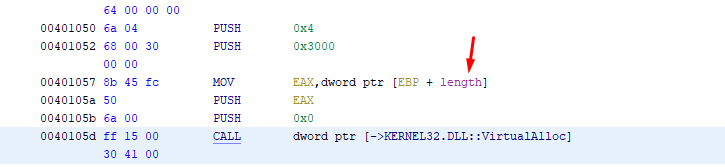
En nuestro caso el mismo ejecutable está compilado sin ASLR.

Por lo cual la primera y más importante pregunta de la lista es un SÍ y sabemos que la máxima dificultad o hasta imposibilidad si no podemos leakear, está salvada.

Continuemos con la segunda pregunta.

Tiene en algun modulo que no tenga ASLR importada la funcion VIrtualAlloc o VirtualProtect?

Si vemos el código que analizamos la respuesta es SÍ, está importada VirtualAlloc, recordemos del análisis de la parte 10, que en nuestro módulo del ejecutable sin ASLR había una llamada a VirtualAlloc.



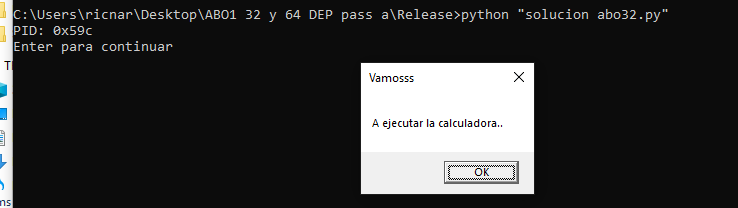
Si ningún módulo sin ASLR o que leakeamos la dirección, tiene importada ni VirtualAlloc ni VirtualProtect, se podrá hacer también, pero el ROP será más largo y complejo pues habrá que resolver ropeando haciendo llamadas a GetModuleHandleA y GetProcAddress por ejemplo, para hallar las dirección de VirtualAlloc o VirtualProtect y se complicará nuestro trabajo.

Así que ya tenemos dos SI, con lo cual vamos bien.

Veamos la tercera pregunta.

Están los datos ya ubicados en el stack para comenzar a ropear?

Corramos la solución parcial que realizamos hasta ahora.



Atacheemos el X64Dbg y lleguemos el RET que salta al primer GADGET de nuestro minirop.

Alli vemos que nuestro ROP está ubicado en el stack listo para ropear y saltar al primer GADGET.

Si no fuera así y nuestro ROP no estuviera ubicado en el stack, y tuviéramos un salto a una sola dirección posible, no podremos encadenar más gadgets porque al no tener el stack controlado por nosotros, al terminar de ejecutar el primer gadget al llegar a su RET, no podriamos encadenar un segundo ya que no tenemos nuestros datos en el stack y alli terminaría todo.

Para eso existe un tipo de gadget especial llamado ROP PIVOT, cuya funcion es acomodar nuestros datos en el stack para poder seguir ropeando normalmente, o sea el código del ROP PIVOT debe mover los datos al stack y al llegar a su propio RET ya que es un GADGET, deben quedar los mismos acomodados para poder seguir ropeando, más adelante veremos ejemplos de uso del ROP PIVOT.

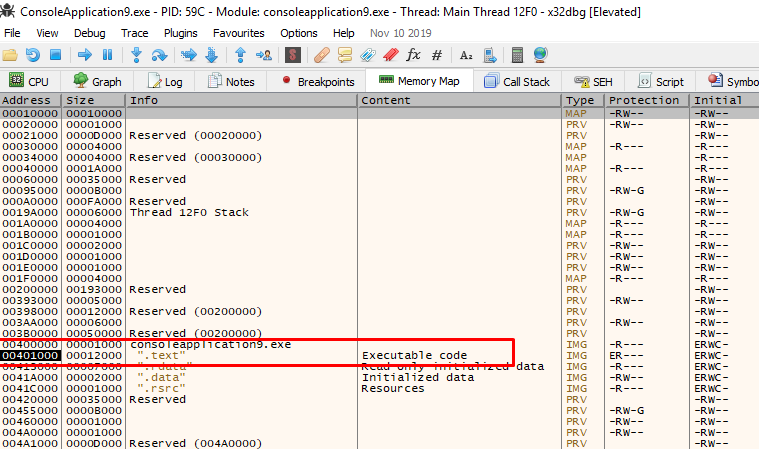
Por ahora la respuesta a la pregunta es SI y por lo tanto en nuestro caso, no es necesario usar un ROP PIVOT para comenzar a ROPEAR.

Así que ya tenemos tres SI, veamos la cuarta pregunta.

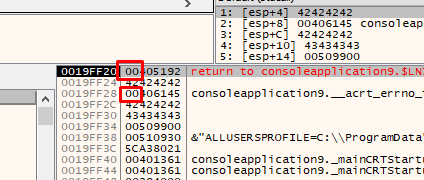
Puedo pasar cualquier caracter o sea no hay caracteres inválidos?

Cuanto más caracteres inválidos tenga en mi proceso a explotar, más se complicará el ROP, hasta a veces pudiendo llegar a imposibilitarlo según el caso.

Obviamente como debemos ubicar direcciones de nuestro módulo sin ASLR en nuestro ROP, si dichas direcciones tienen un caracter que no podemos pasar, la cosa se puede complicar, por ejemplo la sección ejecutable donde podemos hallar gadgets en nuestro ejemplo se ubica.



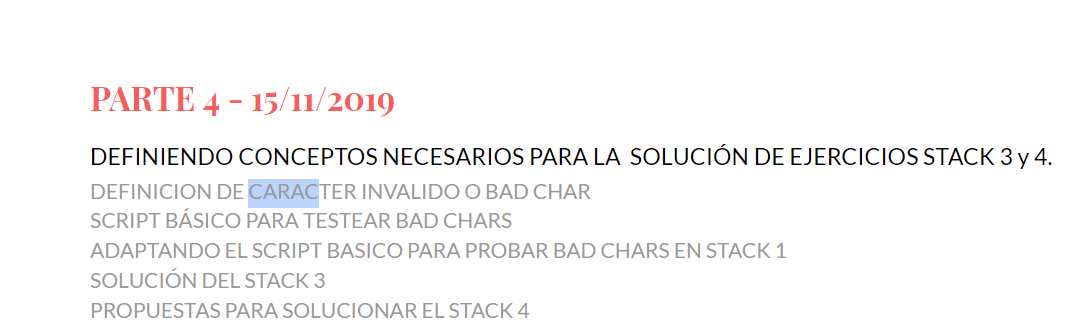
Desde 0x401000 hasta 0x413000, solo alli podemos encontrar GADGETS pues es el único módulo sin ASLR y alli se ubica su sección ejecutable, si por ejemplo 0x00 fuera un caracter invalido nos impediría saltar a GADGETS en esa sección pues el 0x00 es imprescindible para armar la dirección donde saltar.

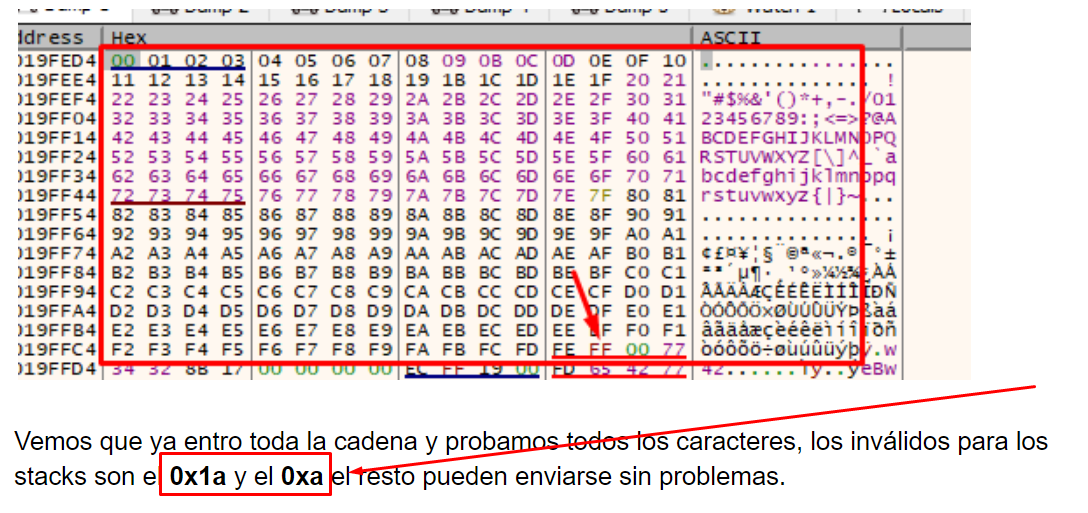


Como vemos en nuestro minirop los ceros fueron necesarios, ya que los gadgets están en direcciones que comienzan con cero.

Si tuviéramos 0x00 como caracter invalido, solo nos quedaría ver si podemos leakear, no podriamos ropear.

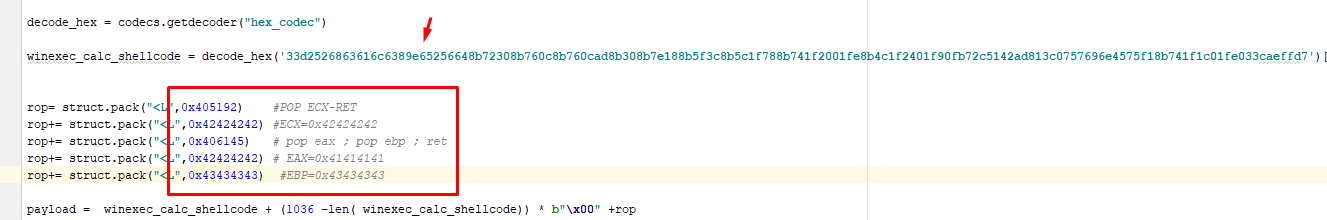
Ahora recordemos nuestro estudio de los caracteres inválidos que realizamos en la parte 4, ya que la funcion para ingresar los datos sigue siendo gets(), al igual que el ejercicio de la parte 4.





Así que tenemos alguna restricción, que es no poder utilizar direcciones que contengan 0xa y 0x1a, pero además tampoco podremos usarlos en los valores que usamos para mover a las registros.

En nuestro minirop movemos a ECX, EAX y EBP los valores 0x41414141, 0x42424242 y 0x43434343 no utilizamos ningún **0x1a** ni **0xa,** pero cuando armemos el ROP que ejecute código debemos tener en cuenta esta restricción, nada de caracteres inválidos en ninguna parte del ROP y si más adelante colocamos un shellcode tampoco podrá usar ningún caracter invalido.



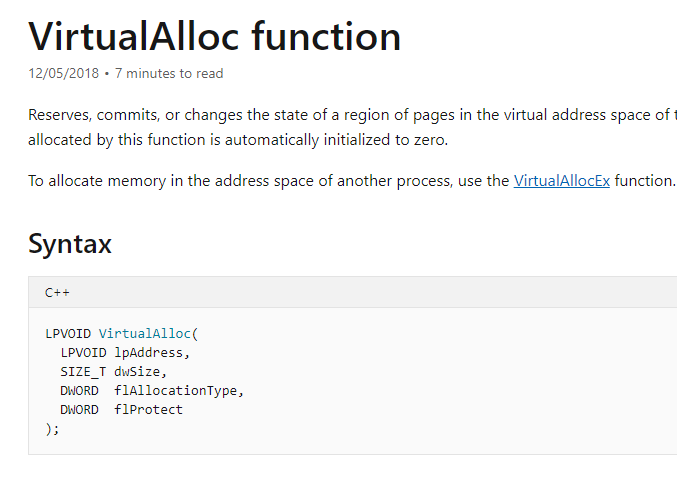
Mirar bien al armar el ROP que no haya en nuestro caso 0x1a ni 0xa en el ROP ni en el SHELLCODE, así que la respuesta a la última pregunta en nuestro caso es un NO, pero es un NO que afecta poco, si los caracteres inválidos hubieran incluido 0x00 sería mucho peor, dentro de todo no son caracteres muy importantes los restringidos.

Así que estamos en uno de los casos más fáciles dentro de todo, más adelante iremos aumentando la dificultad con casos donde haya más NO, jeje.

Buscaremos los gadgets y con FILE OFFSET en el x64dbg buscaremos sus direcciones virtuales

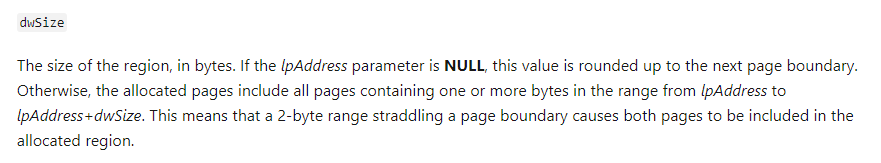
En este caso el problema que tenemos es que el RP++ si hay un CALL termina el gadget no nos muestra lo que sigue a continuación, lo cuál puede ser útil para la mayoría de los casos pero en nuestro caso en particular nos complica un poco.

Veamos la funcion VirtualAlloc en MSDN.



Según esto el primer argumento que debería haber en el stack sería la dirección a desproteger lpAddress.

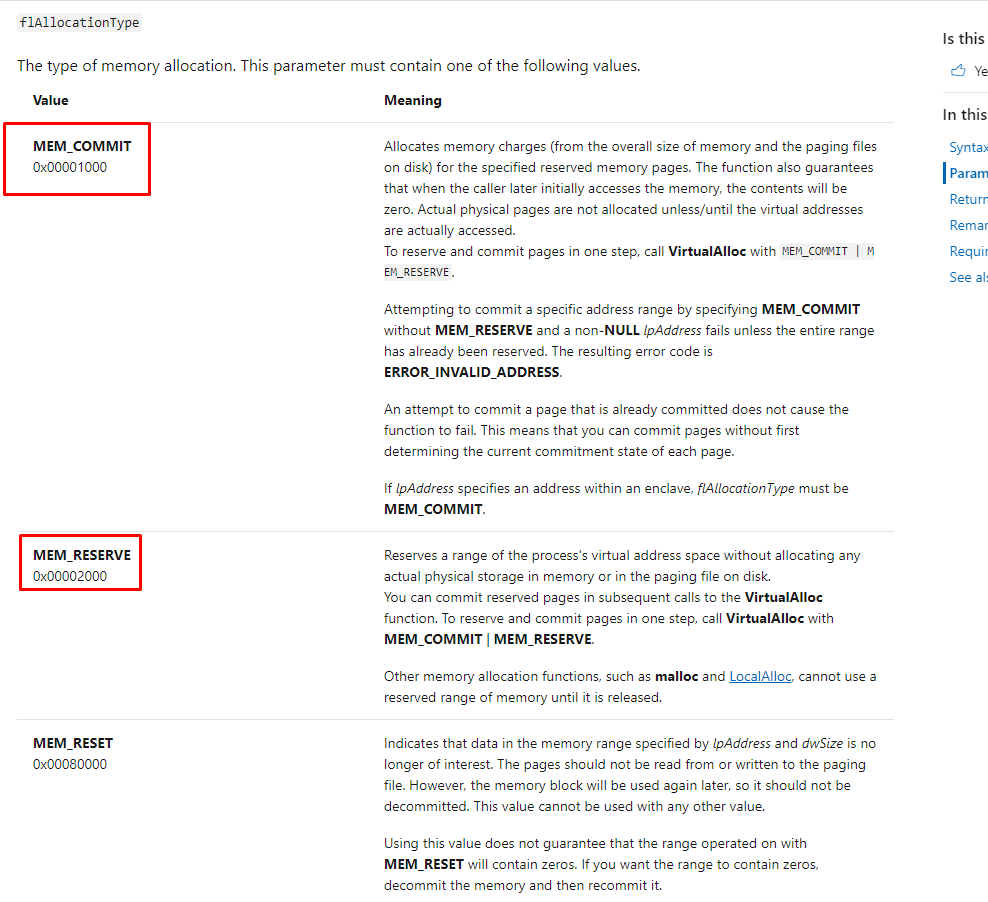
A continuación el segundo argumento es dwSize.

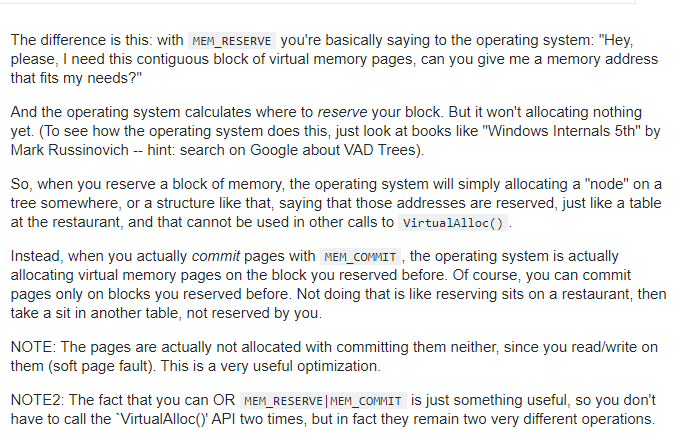


En forma práctica se desprotegerá una página de memoria o más según el size que coloquemos, si por ejemplo ponemos el valor 1, se desprotegerán 0x1000 bytes que generalmente es el tamaño de las páginas de memoria, cualquier valor entre 1 y 0x1000 desprotegerá 0x1000, si pongo un size entre 0x1001 y 0x2000 desprotegerá 0x2000 y así.

De cualquier forma no conviene poner un valor muy grande pues si este valor sumado al lpAddress cae fuera de la sección, la funcion devolverá error, mientras que si pongo un valor menor ese chequeo que hace la funcion en su inicio pasara y funcionará desprotegiendo hasta donde termine la sección.

El tercer argumento es flAllocationType.



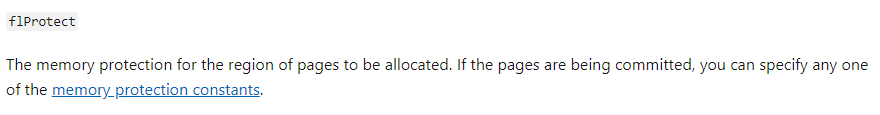


Vemos que si creamos una sección nueva, necesitamos llamar dos veces, la primera usando MEM\_RESERVE (0x2000) para que la reserve y otra usando MEM\_COMMIT (0x1000) para allocar finalmente, lo bueno es que se puede hacer un OR entre los dos valores y hacer las dos operaciones en una.

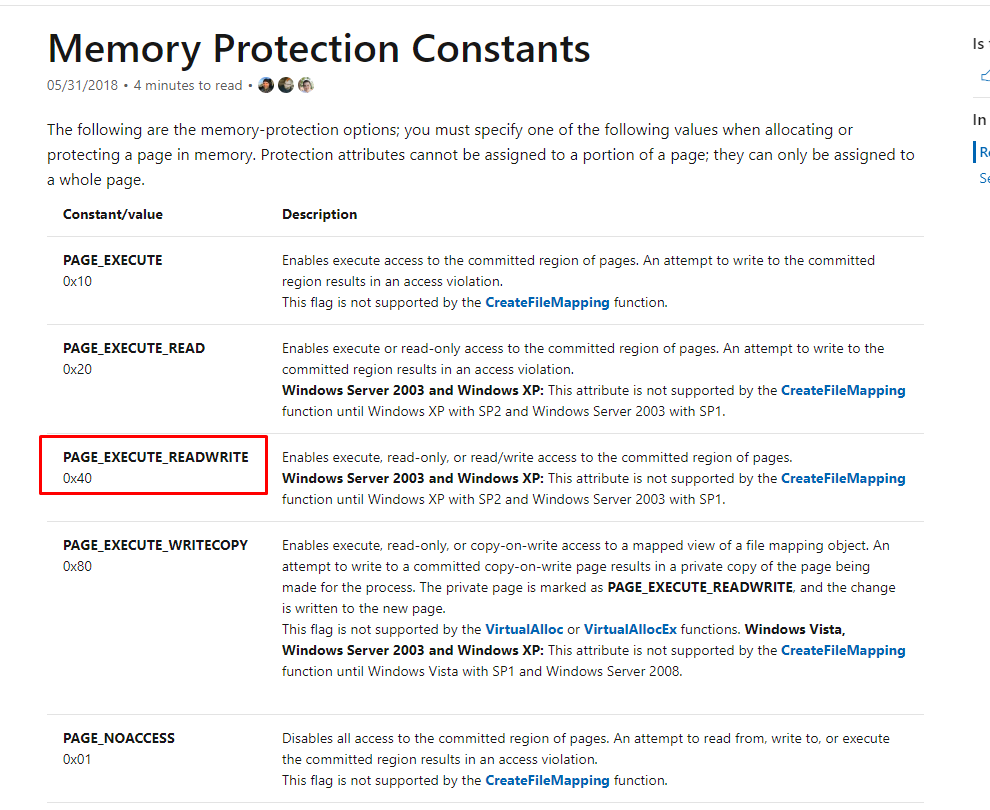


En el caso que utilicemos VirtualAlloc sobre una sección existente, como en el stack deberemos pasarle solo 0x1000, ya que no necesita RESERVAR solo COMITEAR.

El cuarto y último argumento es flProtect.



Como necesitamos que la región sea ejecutable, veamos las constantes para ello.



Está claro que debemos pasar 0x40 para darle permiso RXW.

Vemos que los últimos tres argumentos en nuestro caso pueden venir directamente en nuestros datos, podemos enviarlos y no habrá problemas ya que 0x0 no es un caracter invalido y siempre son fijos.

Los 4 argumentos para VirtualAlloc desprotegiendo el stack serían

lpAddress = ?

dwSize=1

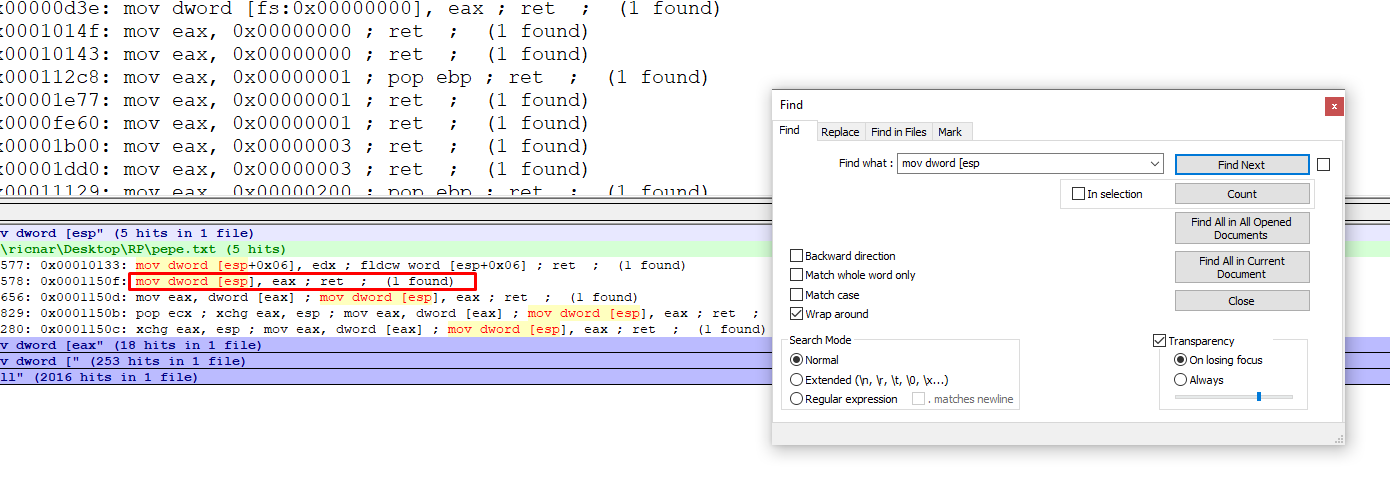
flAllocationType=0x1000

flProtect =0x40

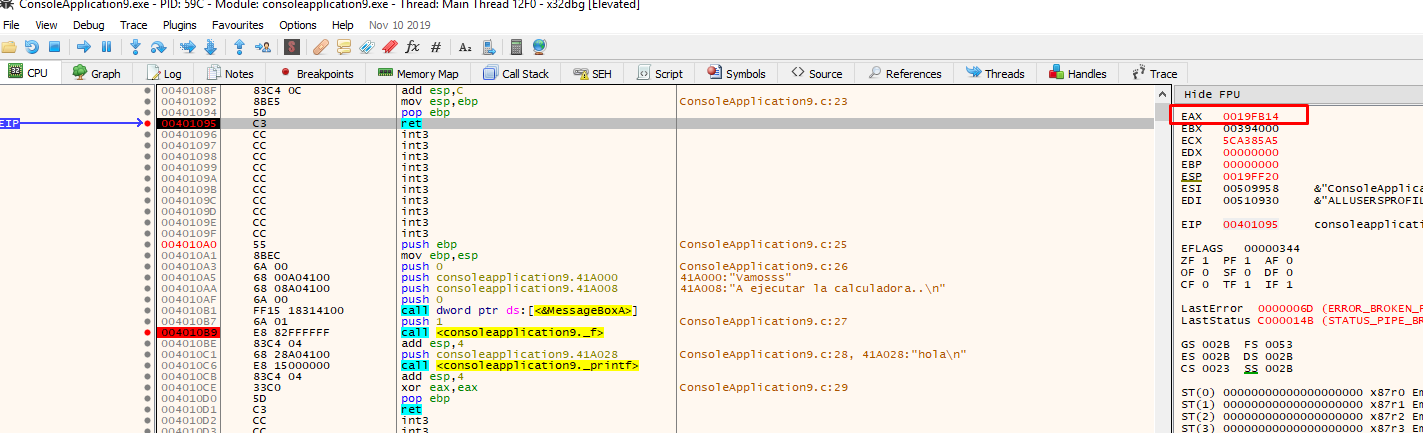
Vemos que el único valor que desconocemos al pasar el exploit es el primero de los argumentos lpAddress, ya que los últimos tres no tenemos problemas para enviarlos tanto el 0x1 como el 0x1000 como el 0x40, son conocidos y posibles dado los caracteres inválidos que existen en nuestro caso, ningún valor tiene 0x1a ni 0xa.

El único problema es poner la dirección a desproteger en el primer lugar, ya que el stack puede moverse y previamente no conocemos su dirección, puede no ser fija.

Como tengo que ubicar en el stack,el argumento lpAddress y quiero que dicho valor sea una dirección del mismo stack para que desproteja el mismo, buscar algún GADGET que escriba en ESP o ESP+XXXX es una buena idea veamos que hay por ahí.



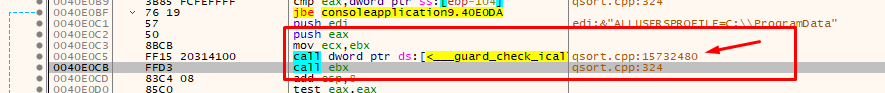
Muy bueno en el contenido de ESP podemos escribir EAX y justo EAX al llegar el RET tiene una dirección del stack, esto podría servir, aunque sería mejor que escribiera en ESP+XXX, un poco más adelante en el stack.

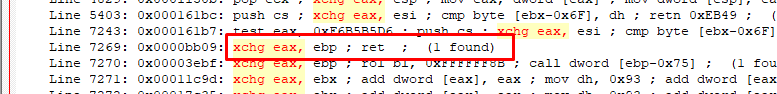


Si colocamos ese valor como lpAddress y abajo los tres argumentos que enviamos nosotros tendríamos los 4 argumentos armados, hay que ver si podemos aprovechar esto.

Como EAX lo usaré para guardar el argumento lpaddress, no lo puedo usar para resolver la llamada a VirtualAlloc.

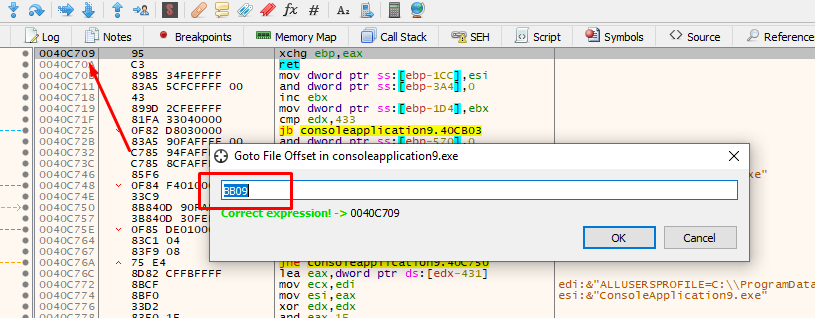
Puedo usar EBX para llamar a la funcion si puedo acomodar en EBX el valor de VirtualAlloc y en EAX el valor que tiene originalmente del stack, usando este gadget que en el medio tiene un call que llama a un ret así que no influye.





Eso nos permitirá guardar el valor original de EAX en EBP, armar todo y luego volver a llamar al mismo GADGET para volverlo a restaurar en EAX y tenerlo como al principio, buenisimo.

Vayamos armando nuestro ROP, lo primero entonces, será preservar el valor de EAX, así que el primer GADGET a llamar está en FILE-OFFSET 0xbb09 cuya dirección virtual es si vamos a GOTO-FILE OFFSET en x64dbg.

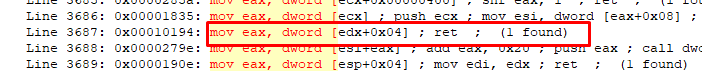


La dirección de nuestro primer gadget es 0x40c709.



Ahora ya tenemos liberado EAX para poder usarlo ya que el valor que nos interesaba está guardado en EBP.

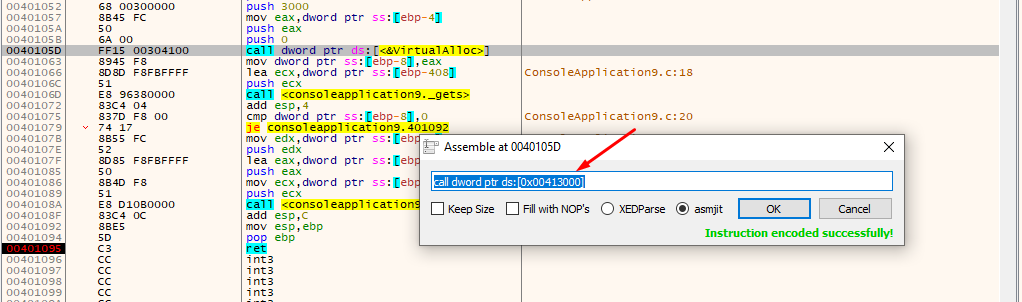
Vamos a usar este GADGET para mover la dirección de VirtualAlloc



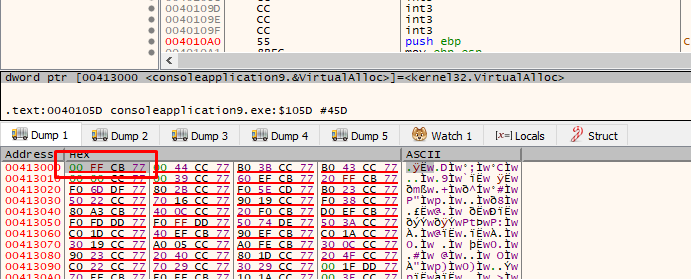
Pero antes debemos setear EDX con la dirección en la IAT de VirtualAlloc y restarle 4 para que termine leyendo la dirección de la funcion y la mueva a EAX

Para ver cuál es la dirección de la IAT de VirtualAlloc donde está guardada la dirección de la funcion, con mirar algun llamado lo veremos facilmente en x64dbg.

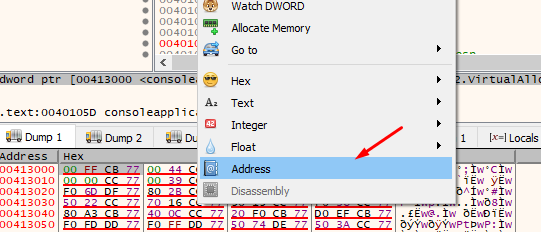
Vemos que cuando salta a VirtualAlloc lee el valor de la dirección de 0x413000.



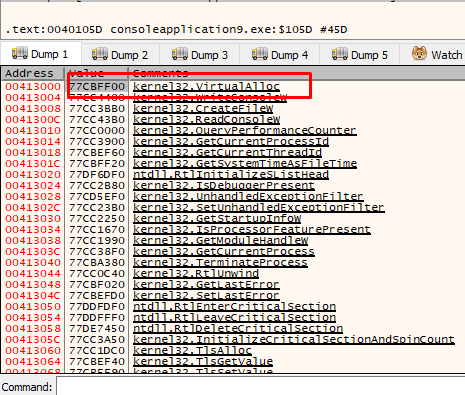
Si lo vemos en el DUMP.



Podemos hacer que muestre el DUMP como direcciones.

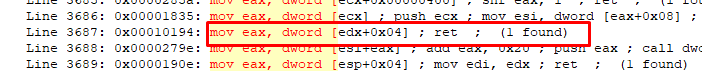


Ahora se verá mejor.



Así que 0x413000 en todas las máquinas, como este modulo no tiene ASLR guardará la dirección de VirtualAlloc, la cuál puede cambiar pero el lugar donde lo guarda no, así que leyendo de 0x413000 tendremos la dirección de la funcion VirtualAlloc para mi proceso, para cualquier maquina.

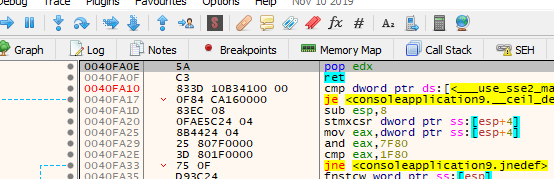
Así que como usaremos EDX tendremos que setear primero EDX a 0x413000-4 porque le suma 4 dentro del GADGET para compensar y que termine leyendo de 0x413000 y mueva a EAX la dirección de la funcion VirtualAlloc.

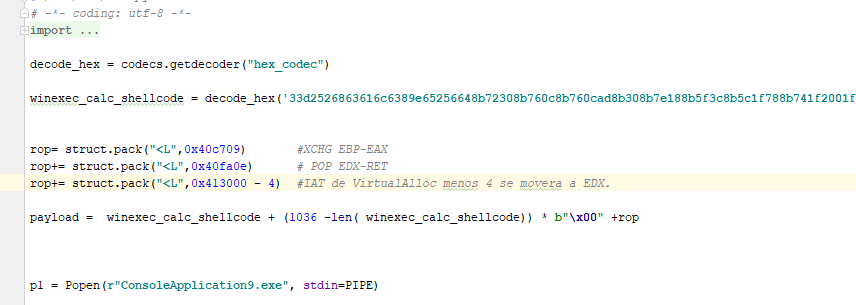


Así que nuestro próximo gadget será



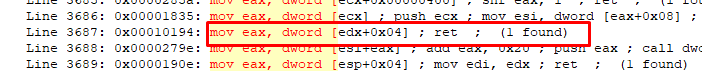
Cuya dirección virtual es 0x40fa0e.



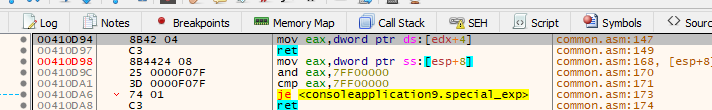


Así que moveremos a EDX el valor de la entrada de la IAT de VirtualAlloc menos 4.

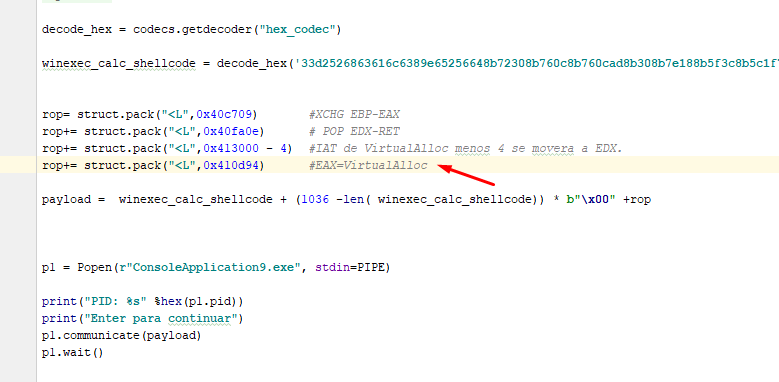
Luego viene el gadget que mueve a EAX la dirección de VirtualAlloc.



Su dirección virtual es 0x410d94

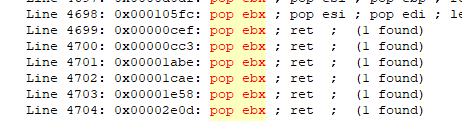


Tenemos 0xD en la dirección pero ya vimos que no es caracter inválido, así que seguimos.



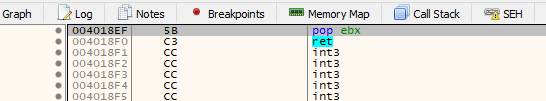
Con esto podemos pasar la dirección de VirtualAlloc a EBX si EBX vale 0.

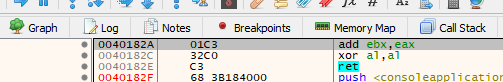




Y con este GADGET podemos poner a cero EBX siempre hay que tratar de evitar los GADGETS con LEAVE ya que se rompe el stack, RP++ no nos los muestra pero si buscamos a mano no hay que usarlos, salvo que podamos acomodar EBP de forma que no se rompa nada.

Así que podemos agregar estos dos gadgets, ya saben como buscar sus direcciones virtuales son **0x4018ef** = POP EBX-RET y **0x40182a**= ADD EBX, EAX -xxx -RET

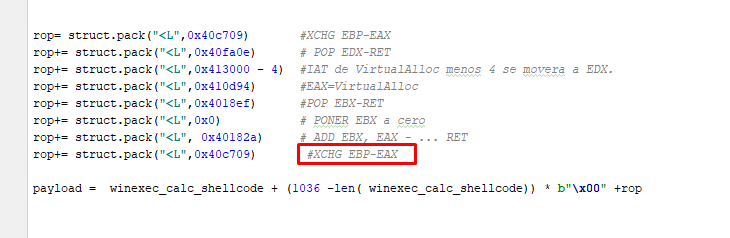




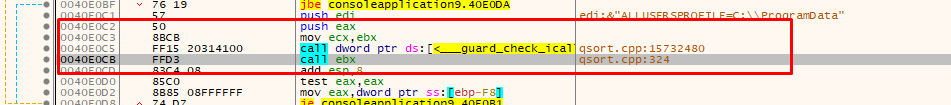


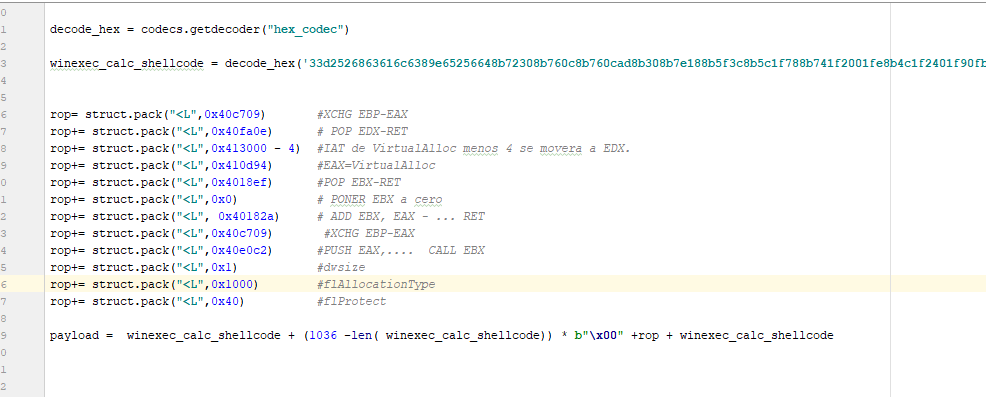
Con eso ya tendríamos en EBX la dirección de VirtualAlloc.

Debemos volver a EAX el valor que tenia originalmente con otro XCHG EAX, EBX.



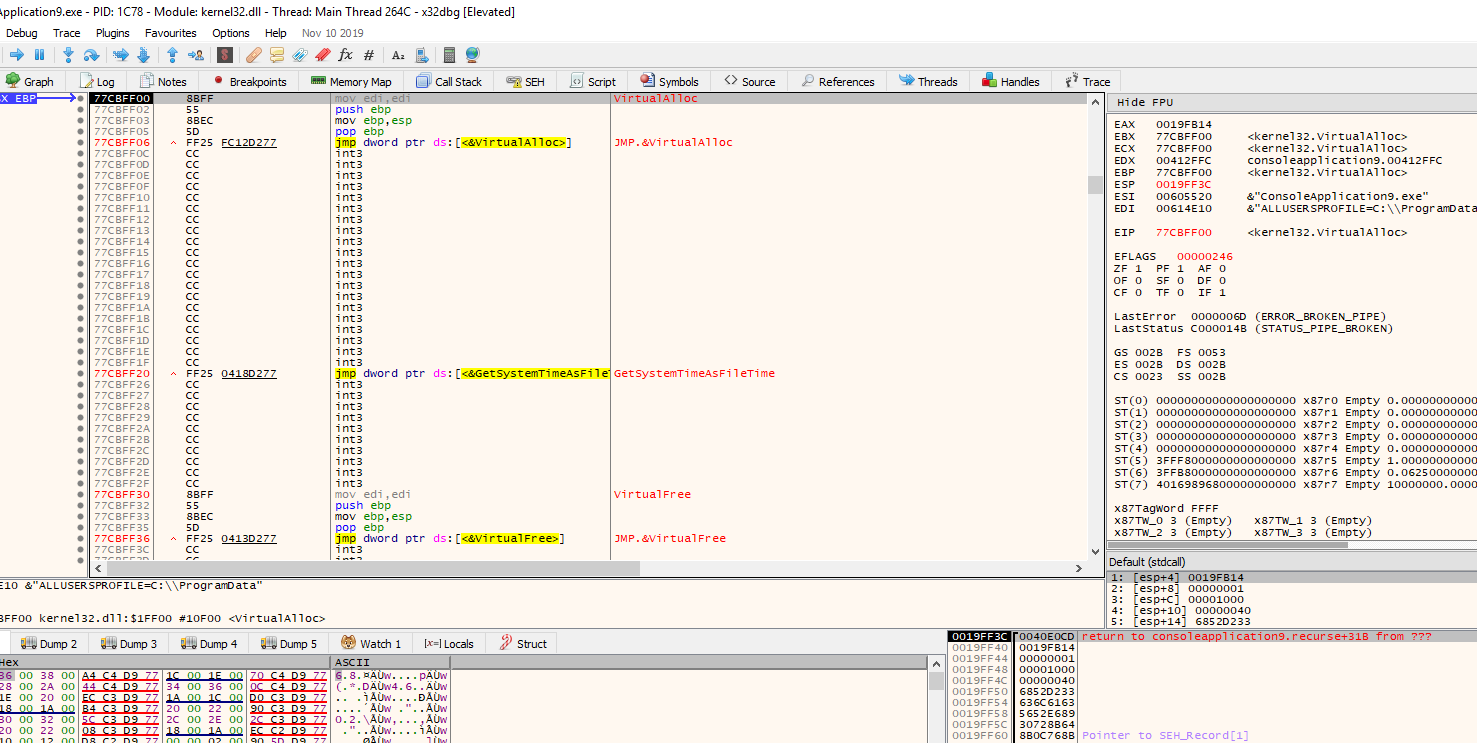
Y luego vendria el llamado a la funcion.





Vemos que si ejecuto el script así, aún hay un problema.

Llega al RET y voy traceando con F7 hasta llegar a VirtualAlloc.



Están los argumentos correctos

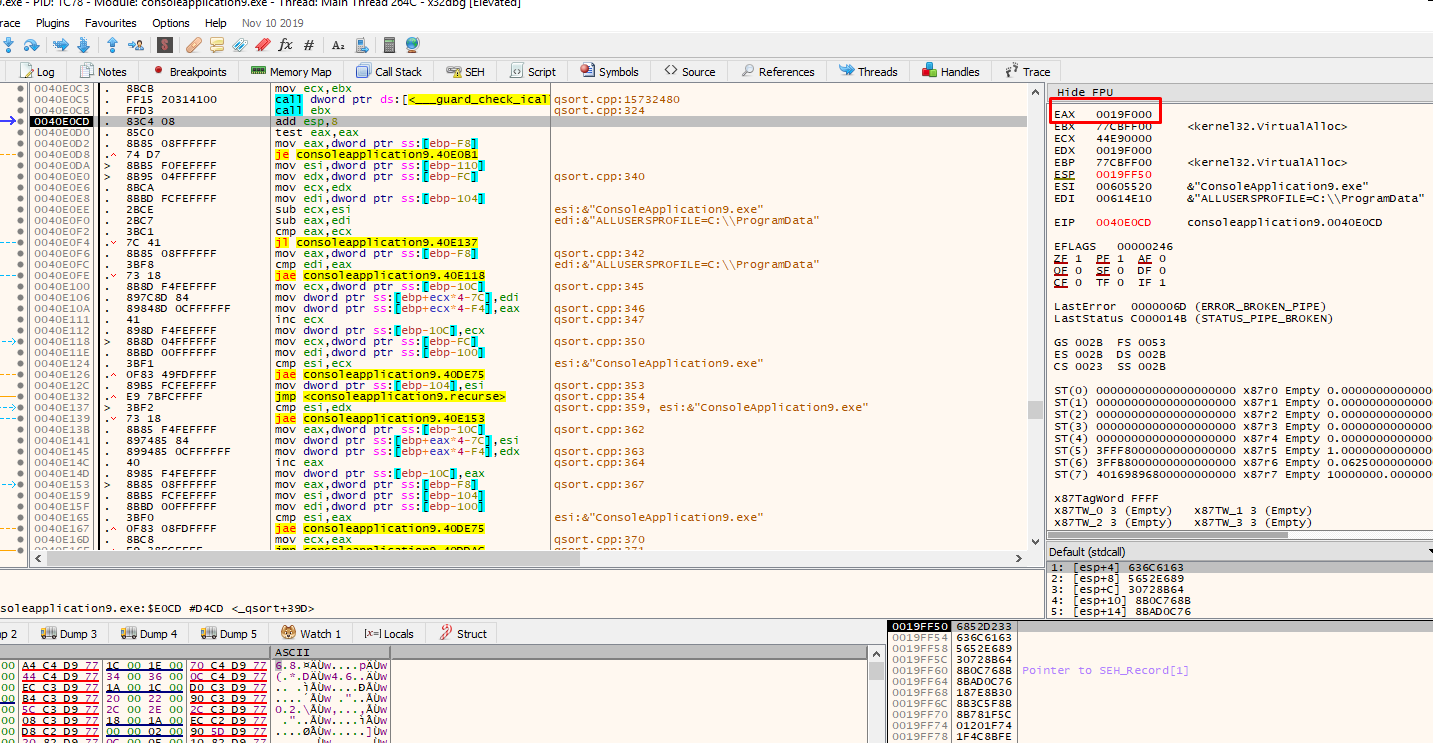
0019FF40 0019FB14 lpAddress

0019FF44 00000001 dwsize

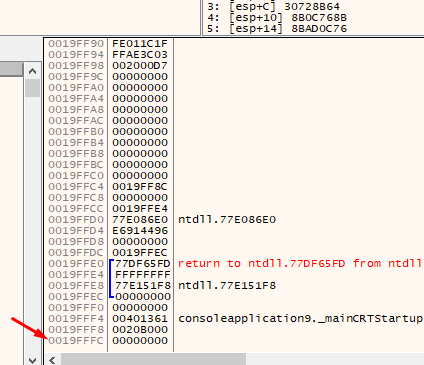
0019FF48 00001000 flAllocationType

0019FF4C 00000040 flProtect

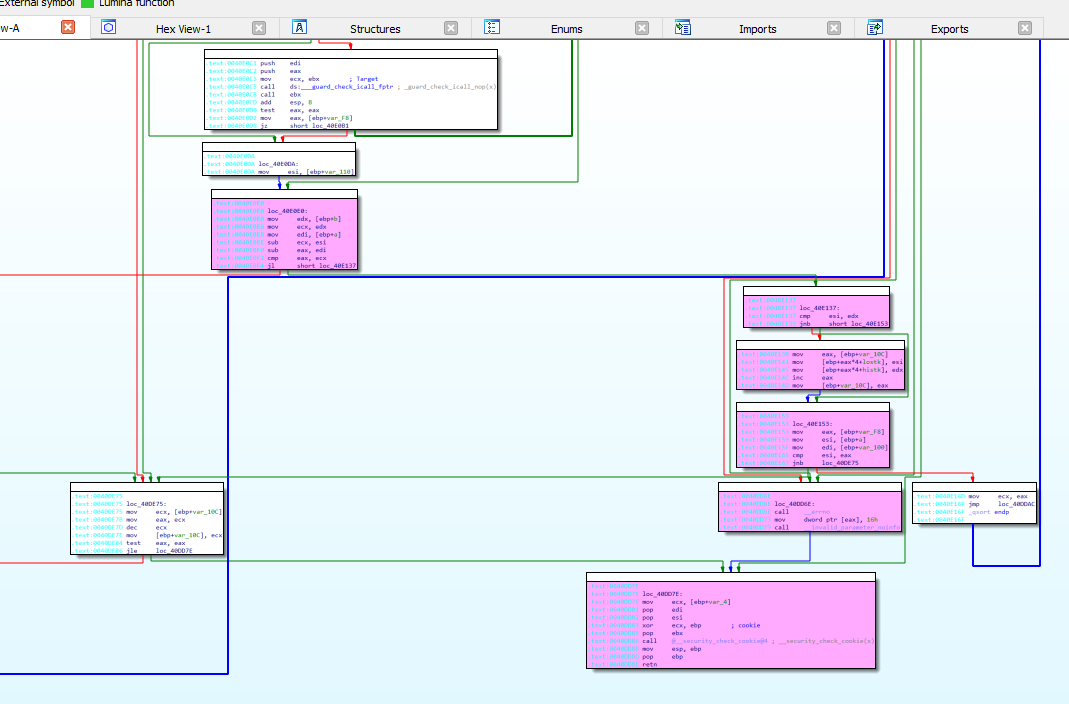
Si pasamos traceando hasta volver de la funcion VirtualAlloc, vemos que el valor de retorno es el correcto.



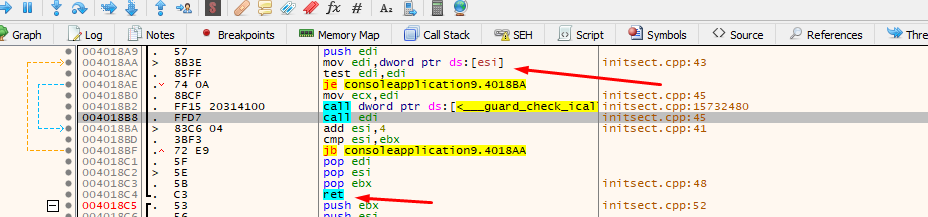
Devolviendo el inicio de la página de la sección que desprotegio, desde 0x19f000 hasta el final del stack.



Ya tenemos stack ejecutable pero aún hay un problema, podemos volver de esta funcion sin que se rompa para ejecutar el shellcode?



Vemos en el IDA el camino para llegar al RET y vemos en las direcciones letras celestes que nos indican que es código de las dlls embebido por eso le agrega la cookie de seguridad de las dlls antes del RET, que nos impide llegar al mismo, deberemos cambiar la estrategia.

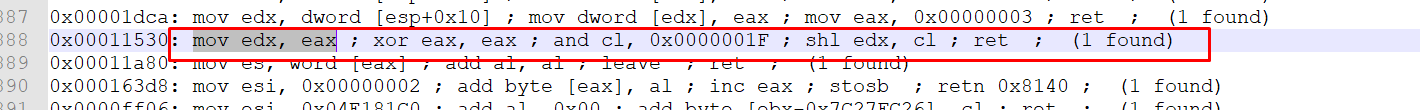


Vemos otro GADGET para saltar a VirtualAlloc.

EDI debe tener al saltar la dirección de VirtualAlloc y como EDI sale del contenido de ESI nos resuelve facilmente poniendo antes en ESI la dirección de la entrada de la IAT 0x413000.

En EDI al inicio debe estar la dirección del stack que tiene EAX originalmente, así que habrá que moverla de EAX a EDI.

Acá vemos dos gadgets el primero si ECX vale cero mueve EAX a EDX (0x412130)

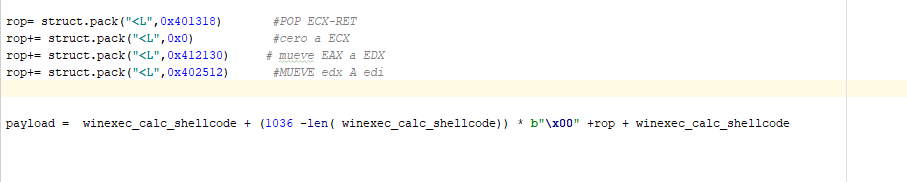


y movemos EDX a EDI (0x402512)



Con esos dos si ECX vale 0 al inicio ya tenemos EDI con el valor del stack, vayamos armando.

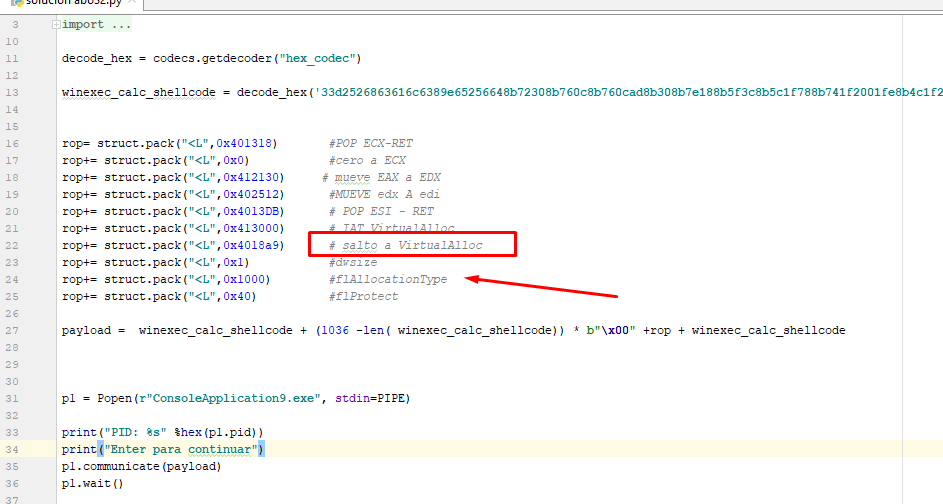
Con este ponemos ECX a cero (0x401318)



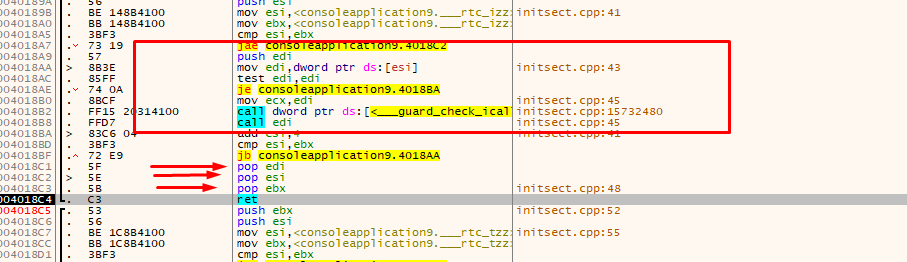
Con eso ya tenemos en EDI el valor del stack que va a pushear solo nos falta poner en ESI la dirección de la entrada de la IAT de VirtualAlloc, está es fácil. (0x4013DB)



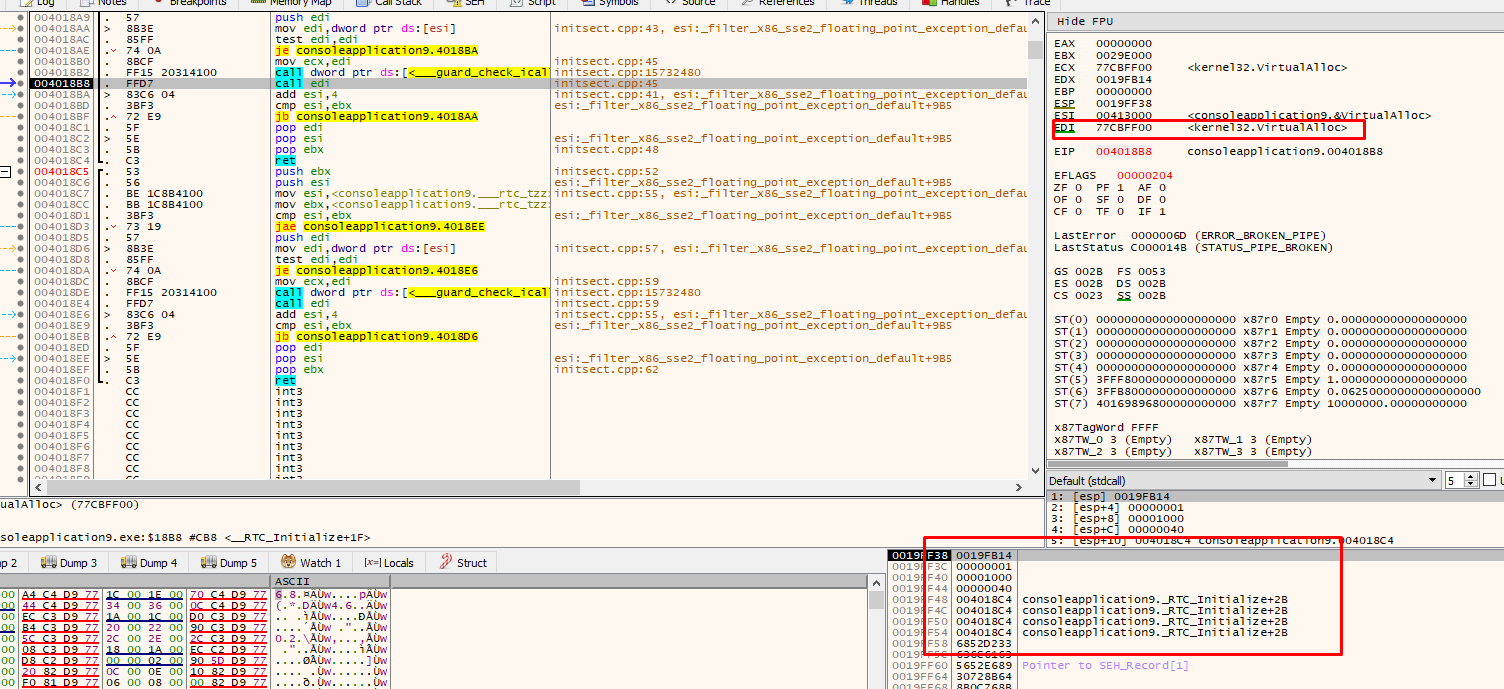
Y saltar a ejecutar la funcion, luego poner los argumentos que faltaban debajo.



Y me queda acomodar el retorno ya que al volver del llamado hace tres POPs.

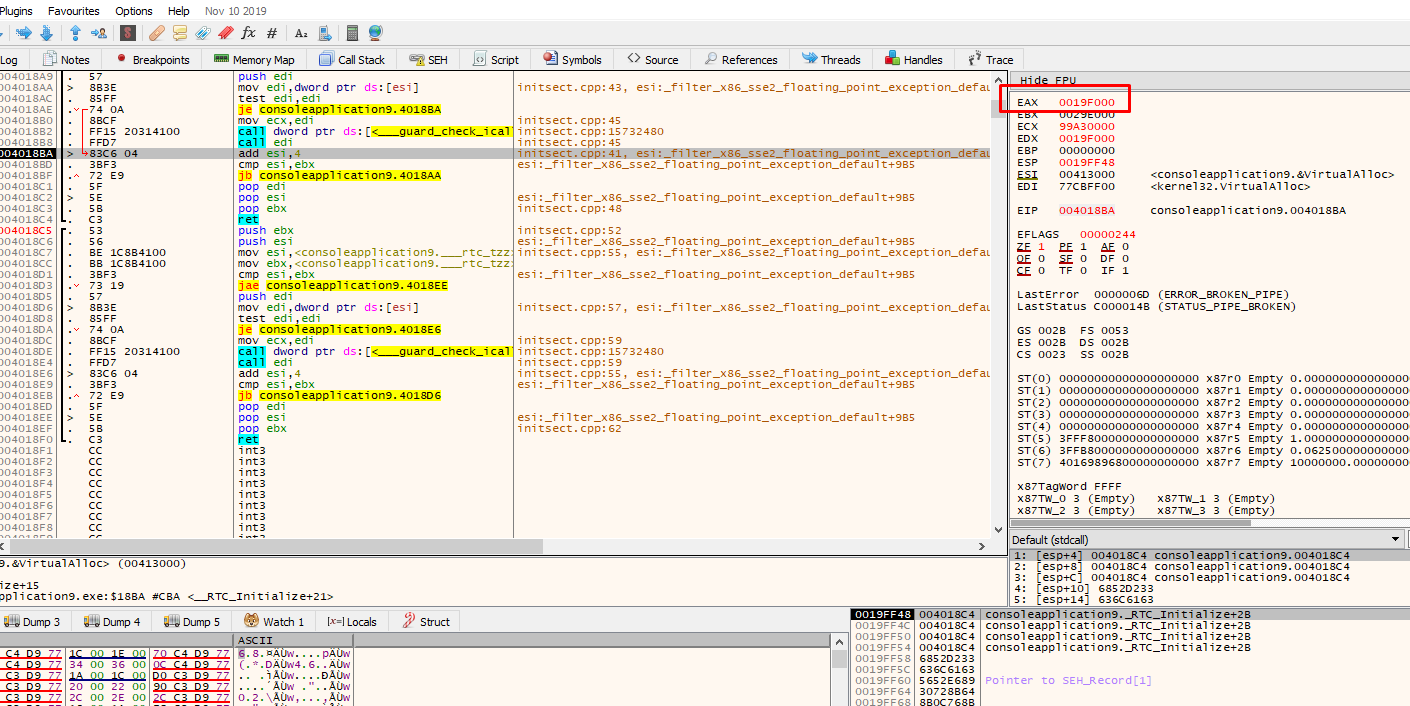


Así que pondré tres punteros RET que son similares a NOPs mientras estamos ROPEANDO no hacen nada, en la imagen anterior puedo ver un RET en 0x4018c4.

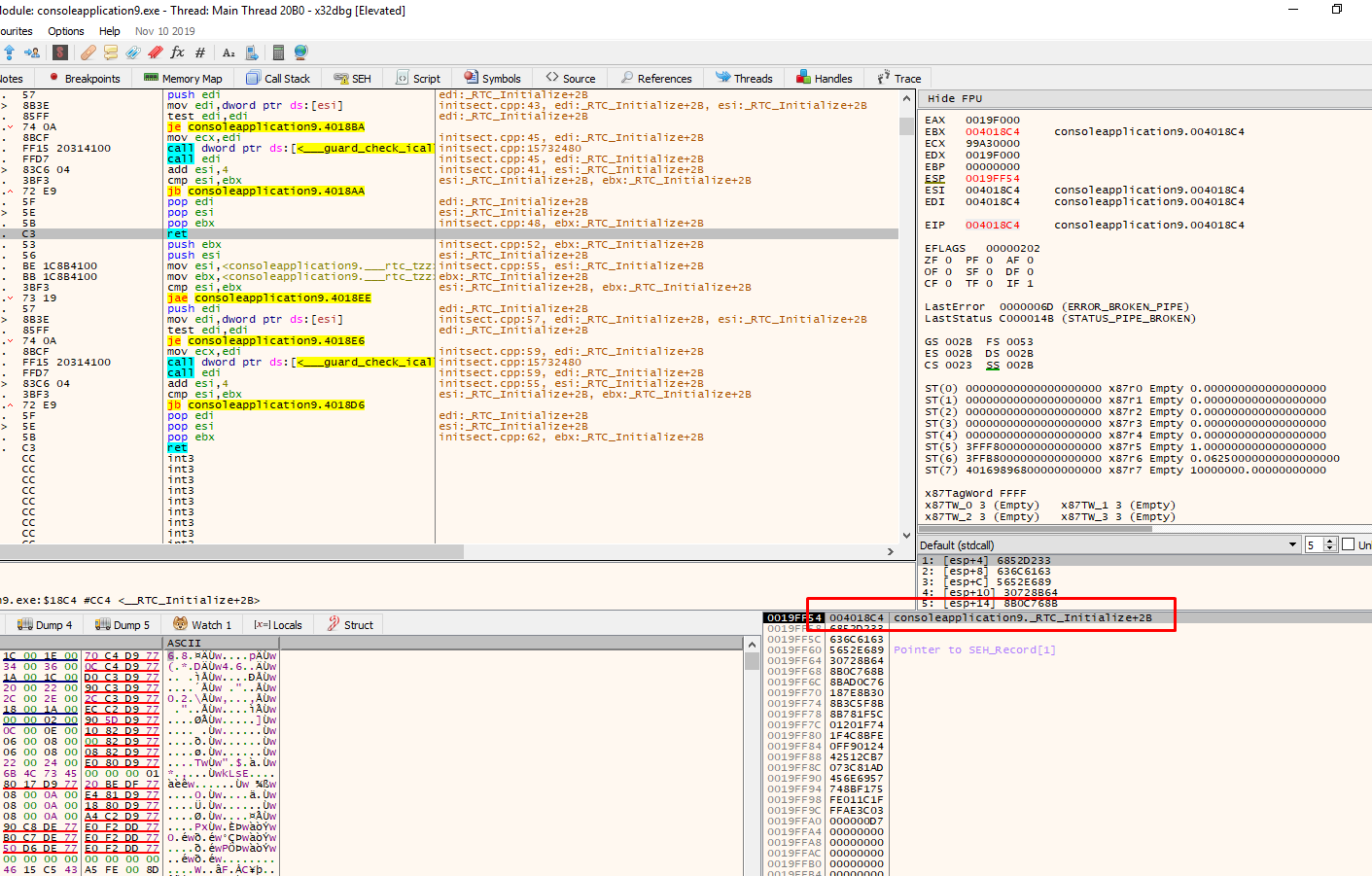


Ahora llegamos con todos los argumentos correctos y con toda la posibilidad de manejar el retorno.

Vemos que al igual que la vez anterior me devolvió la dirección de la página a la cual le dimos permiso de ejecución.

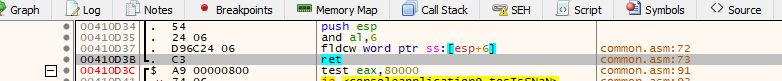


Si sigo veré que los pops están bien solo me falta un último gadget para saltar a ejecutar el stack dónde está mi shellcode.

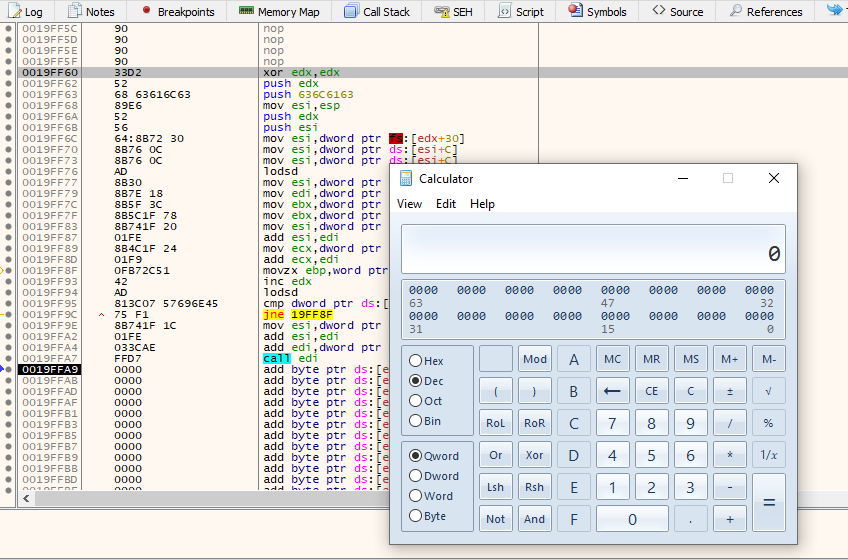


Justo nos falta luego del último RET un CALL ESP o algo similar para saltar a ejecutar al stack.

Usaremos un PUSH ESP-RET con un poco de basura que no hace nada en medio.



Con esto ya saltamos a nuestro shellcode y podemos ejecutar sin problemas.



Hemos vencido al DEP, en un caso sencillo igualmente, jeje.

No quise borrar la parte que hicimos y no termino funcionando porque todo es practica hasta lo que no funciona, y con eso aprendemos también equivocandonos, como nos pasa cuando trabajamos dia a dia.

Seguiremos en la parte 12 con el siguiente ejercicio, que está compilado en 64 bits y habrá que hacer el ROP tal cual hicimos en este.

Luego más adelante veremos casos de ROPs más complejos y iremos avanzando lentamente.

Hasta la parte 12

Ricardo Narvaja (exiliado en casita por el coronavirus)

15/03/2020