EXPLOITING Y REVERSING

USANDO HERRAMIENTAS

GRATUITAS (PARTE 13)

RESOLUCIÓN DEL EJERCICIO DE 64 BITS

Estamos en cuarentena por el tema del coronavirus y encima es fin de semana largo así que hay 4 días muy aburridos que no se puede salir, por lo tanto ayer hice la parte 12 y hoy comienzo la parte 13, agregarle el shellcode y analizar y adaptar un resolver para 64 bits.

El shellcode no es mio aclaro lo voy a explicar pero es bastante público, lo adapte para el caso.

SCRIPT CON SOLUCIÓN COMPLETA

Acá está la solución completa que ejecuta el NOTEPAD para variar un poco.

Vemos a explicar como funciona.



Alli vemos las partes que agregue a la solución parcial que había hasta ahora.

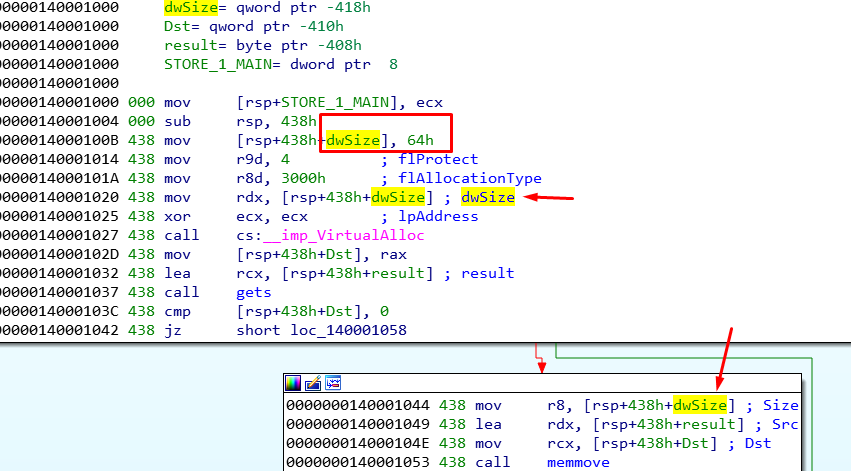
Arriba vemos el shellcode con el resolver que luego explicaremos.

Hay un gadget que cambie

rop+=struct.pack(**"<Q"**,0x1400060b7) *# ADD RAX, 0x20 # CHANGED to have more space*

Este gadget le sumaba 0x28 a RAX antes de saltar a ejecutar lo cambie por uno similar pero que suma 0x20 para no desperdiciar porque hay poco espacio.

Recordemos que el memcpy no copia todo lo que enviamos al espacio reservado en el heap con VirtualAlloc.



Vemos que solo alloca y copia 0x64 bytes así que conviene evitar desperdiciar espacio.

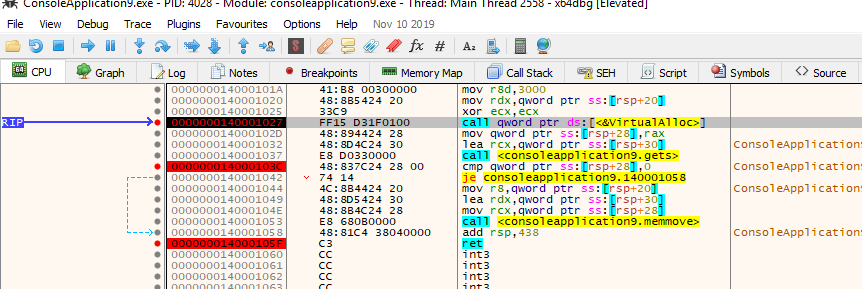
Vemos un código hecho por mi que el memmove va a copiar en la zona reservada creada en el heap y es el que va a preparar todo para ejecutar el shellcode final, ya lo veremos al ejecutar.

Y el shellcode está en el stack abajo del ROP sin permiso de ejecución, me podrían decir porque no lo pones al shellcode en el heap, en lugar de ese código directamente y lo ejecutas.

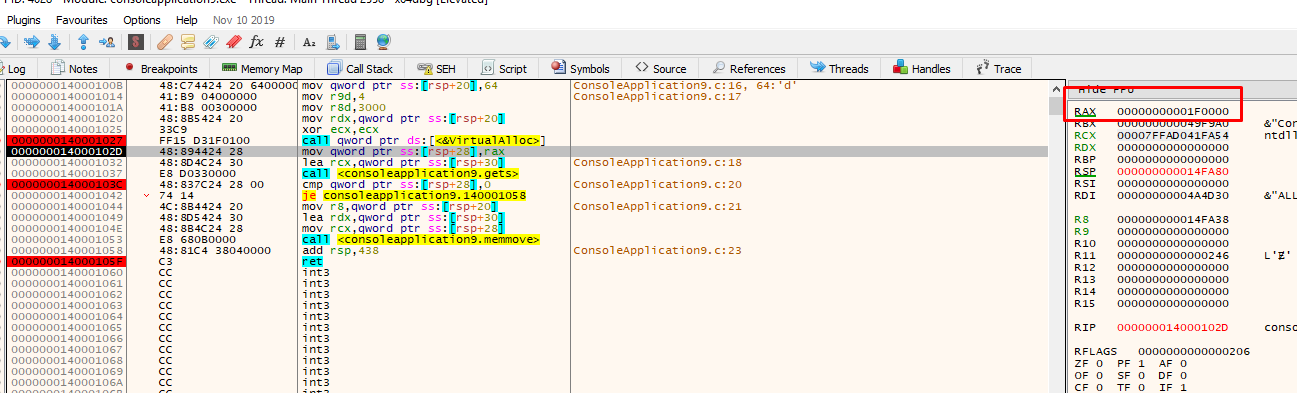
La respuesta es porque no cabe, el espacio en el heap ya vimos que solo reserva 0x64 bytes y encima perdemos los primeros 0x20 al saltar, nos quedan 0x44 bytes y no entra en ese poco espacio.

Ejecutemoslo y vayamos explicando.

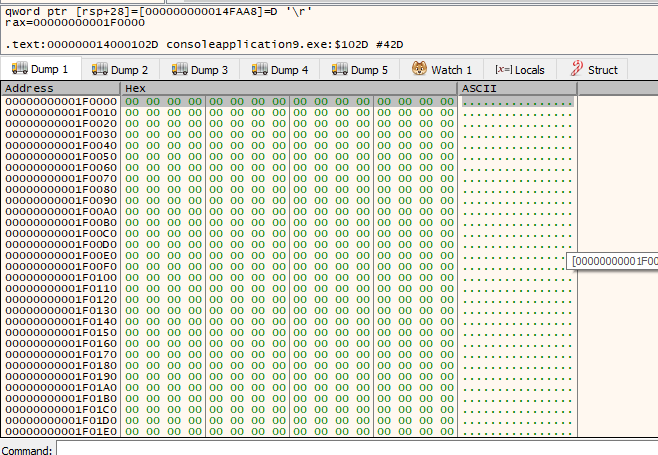
Corro el script, para en los breakpoints que puse.

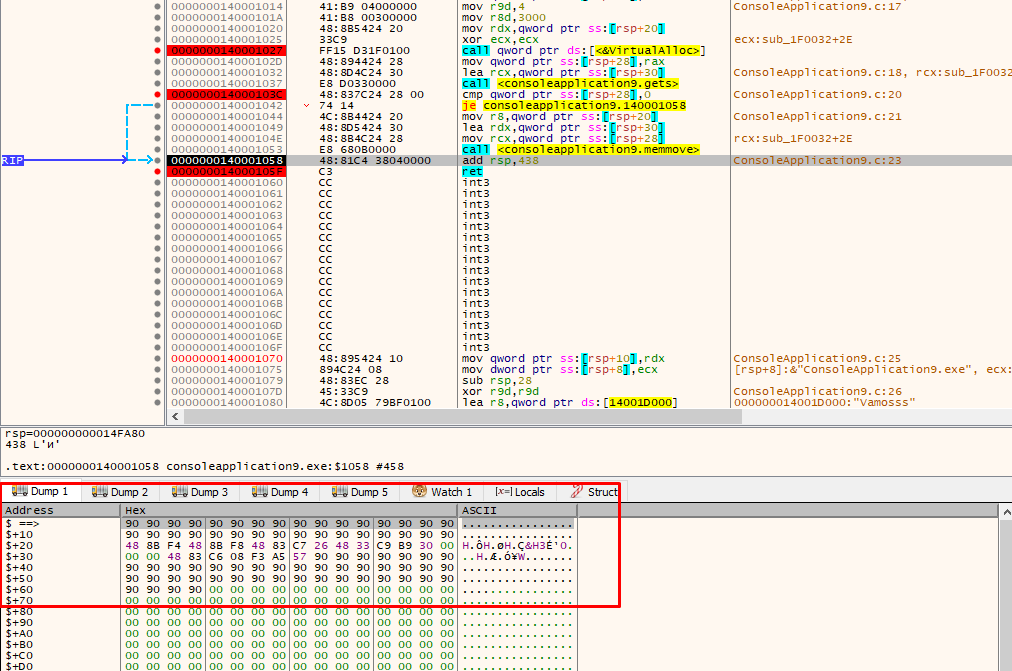


Alli alloco los 0x64 bytes.



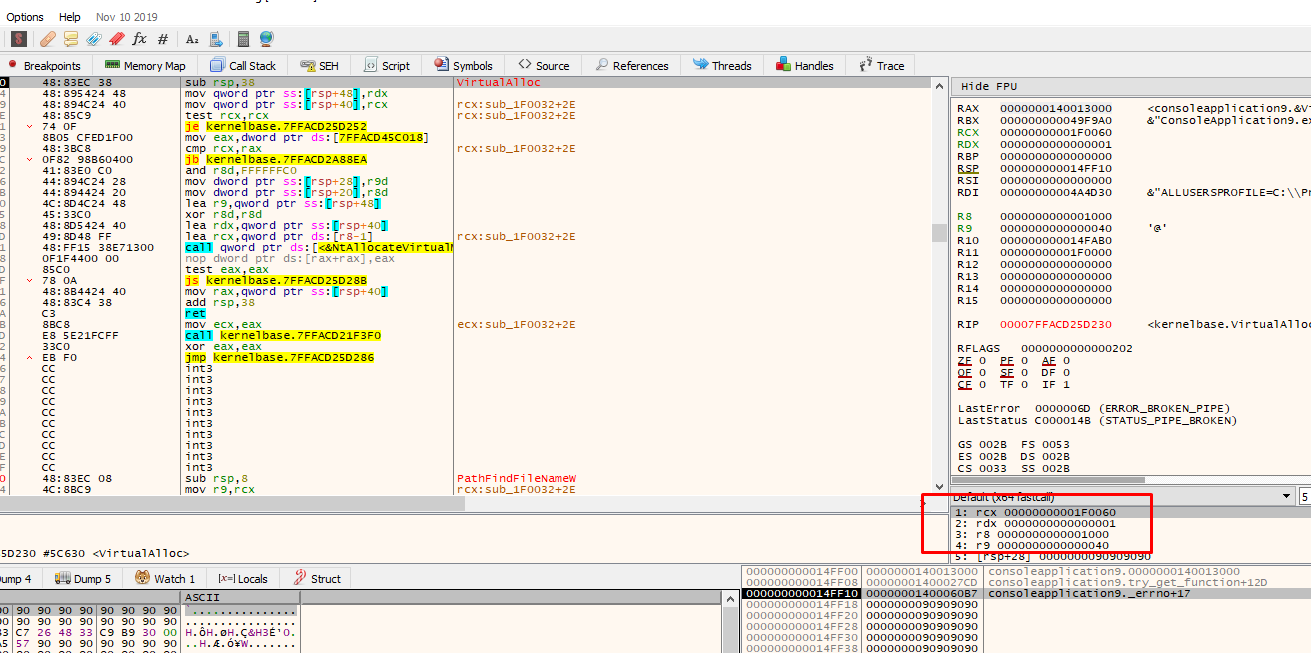
Vemos que en realidad alloca más de 0x64 el problema es que estrictamente copia solo 0x64 bytes.





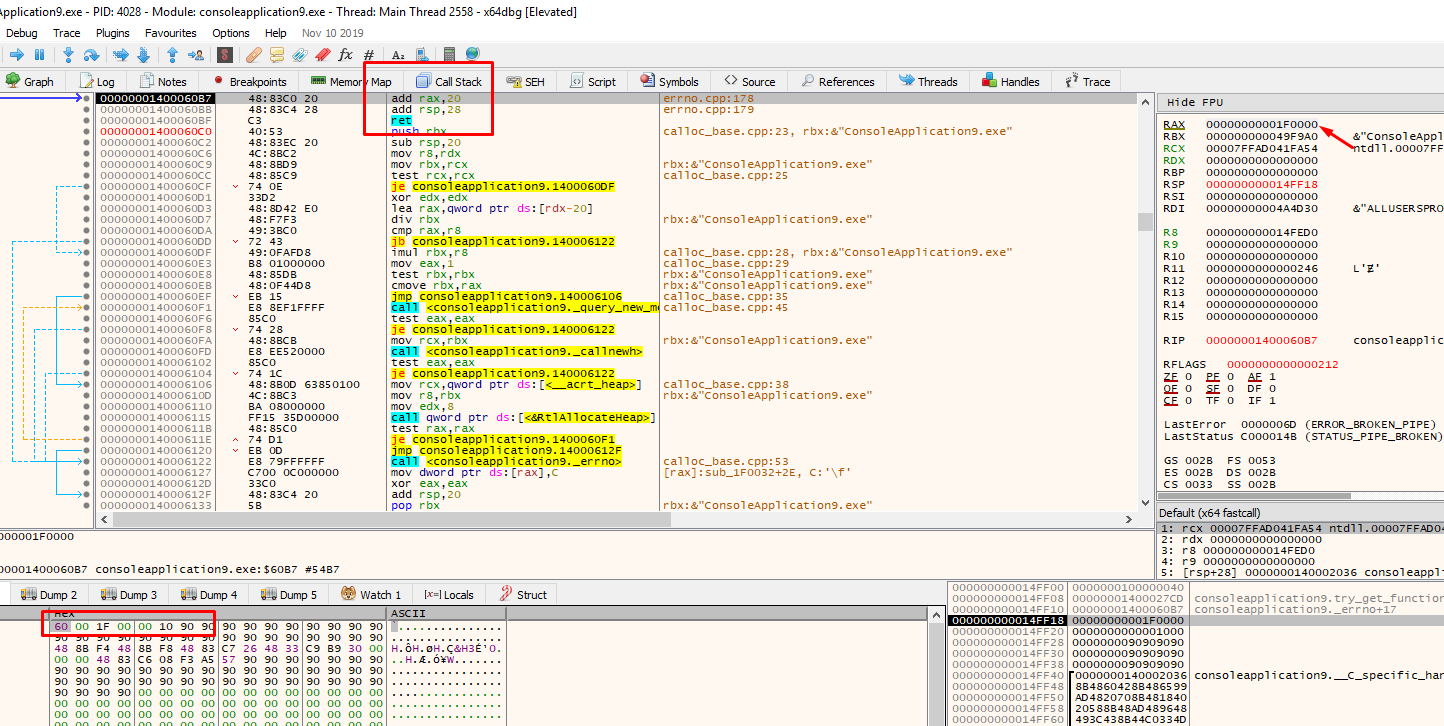
Alli copio, no nos olvidemos que el rop, rompe unos bytes del inicio y para saltar tengo que evitarlos y lo unico que encontre es el gadget add rax, 20, lo cual me deja poco espacio para el shellcode.

Traceo el shellcode hasta llegar a VirtualAlloc.



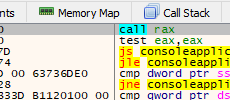
Alli llego con los argumentos correctos, puedo ir al RET, poner el cursor ahí y apretar F4 para no tracear tanto.

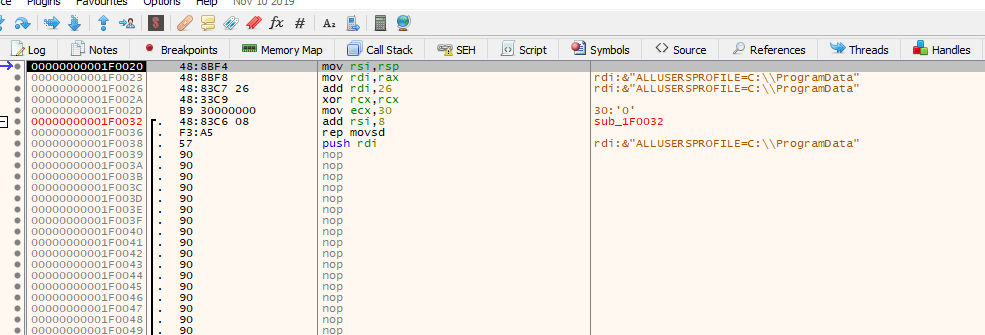




Este es el GADGET que le suma 0x20 a RAX para saltar a ejecutar, evitando los bytes que se rompieron al inicio.

Luego saltamos a ejecutar.





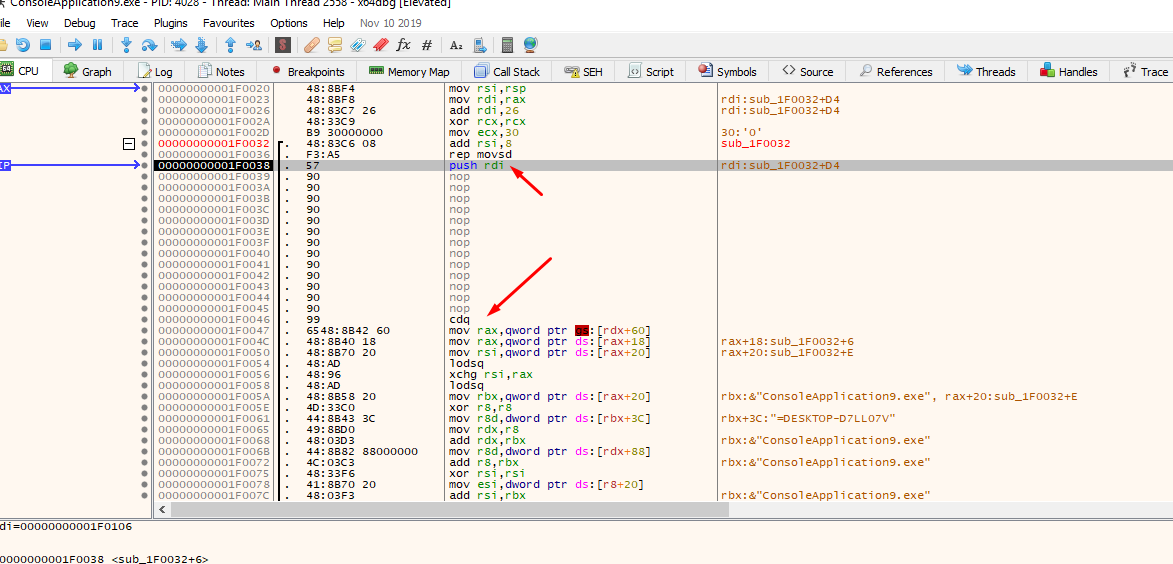
Alli vemos el codigo que hice, lo que hace es buscar el shellcode en el stack, y copiarlo aquí debajo ya que lugar hay, el memmove no lo copiaba completo por el limite de size=0x64 al copiar, pero yo sí puedo copiarlo completo.

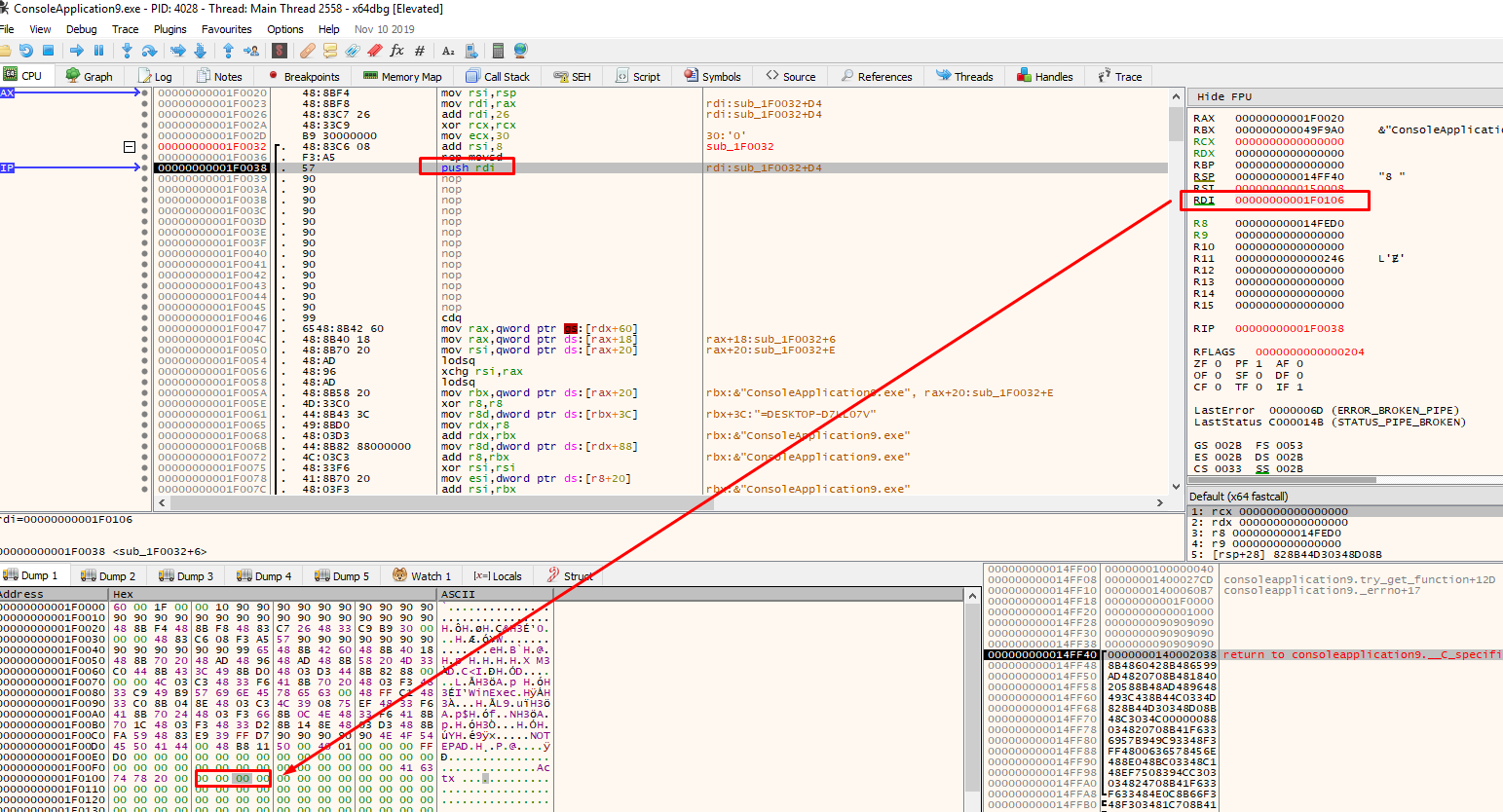
RSI tendrá el source

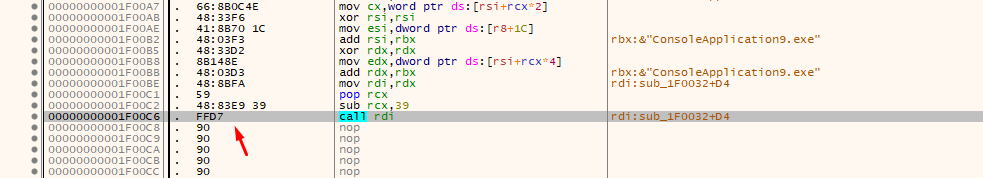
RDI el destino

RCX el size a copiar en dwords

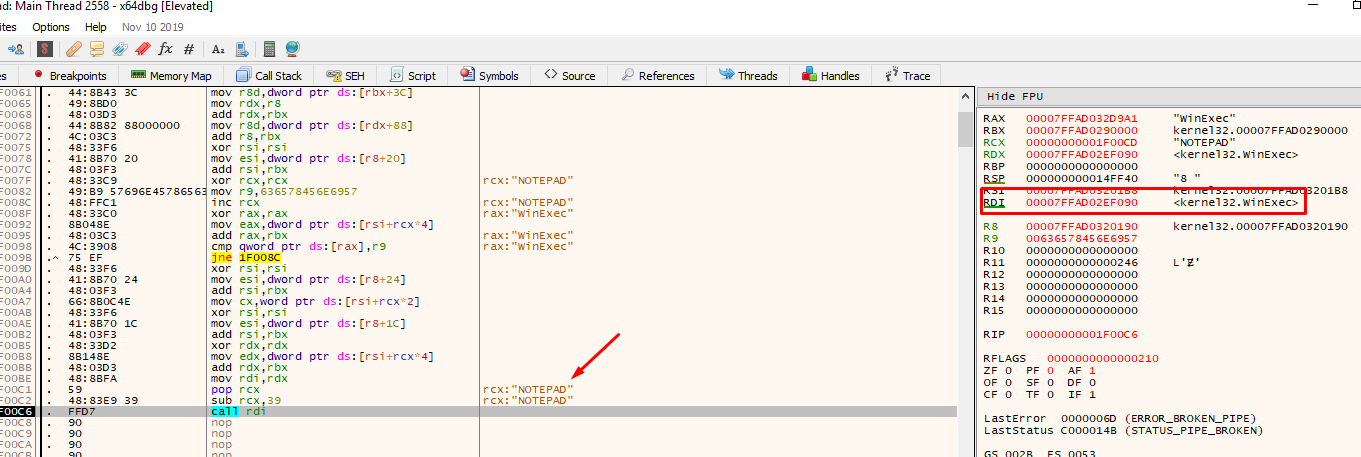
Al llegar al REP MOVSD copia el shellcode y hago un PUSH RDI para guardar la dirección del destino donde copio, que quedó en RDI.





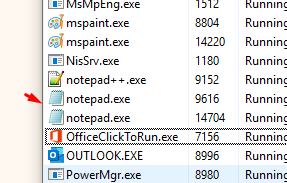


Puedo poner el cursor allí y apretar F4.

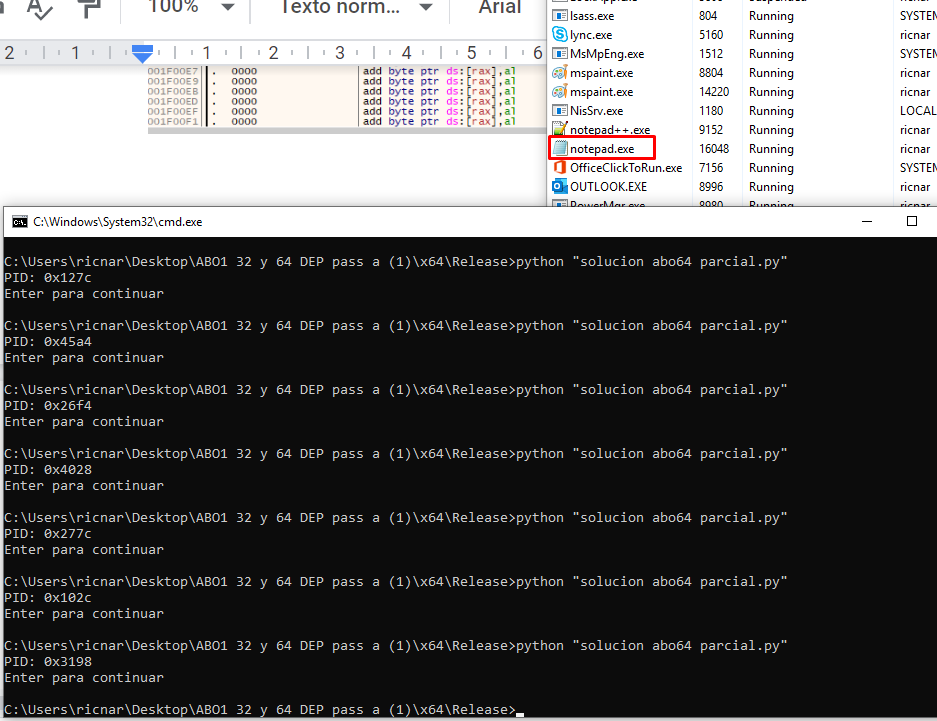


Vemos que va a llamar a WinExec con la string NOTEPAD como argumento, lo cual ejecutará el NOTEPAD.

Bueno ya tengo el shellcode para ejecutar solo me queda explicarlo, lo cual no está muy bueno en x64dbg, así que correré el script y me atacheare con Windbg así puedo ver las estructuras y los símbolos necesarios, igual si antes le doy RUN verifico que corre un NOTEPAD.

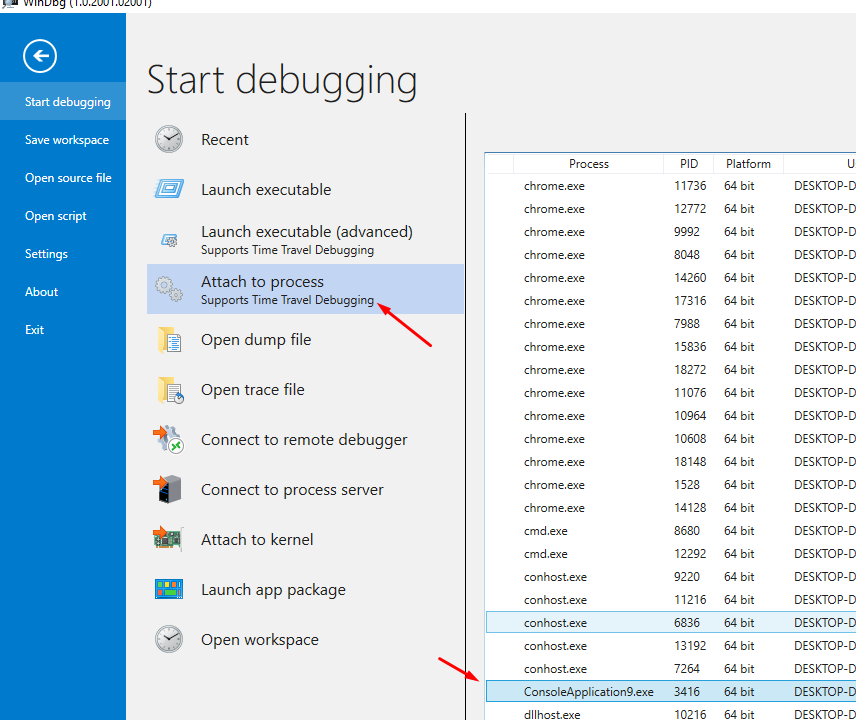


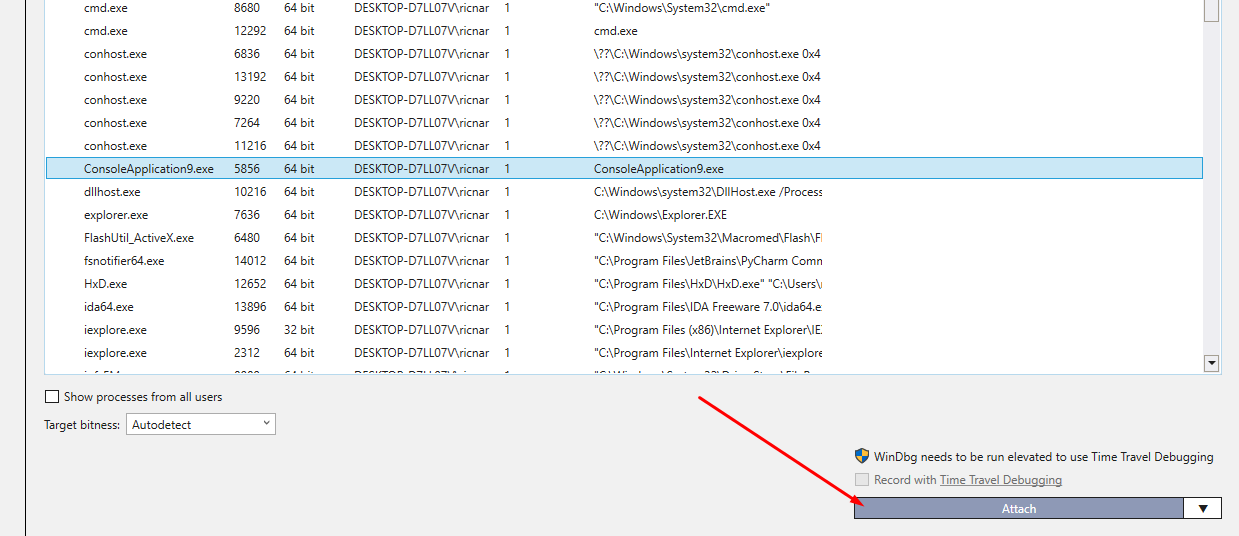
Vemos que corre el NOTEPAD y luego va a ExitProcess y se cierra.



Si lo ejecuto sin debuggear veo que corre el NOTEPAD y se cierra correctamente.

Bueno lo voy a atachear con el Windbg para tracear el RESOLVER.



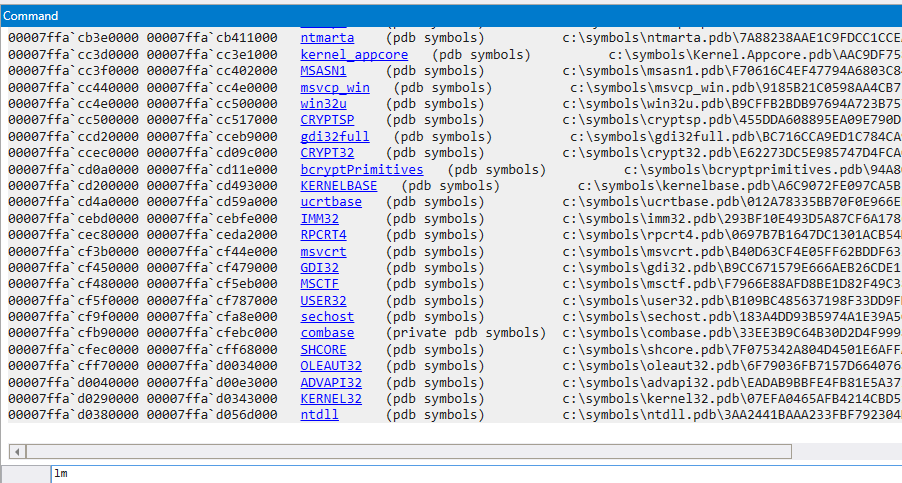


Después de que para

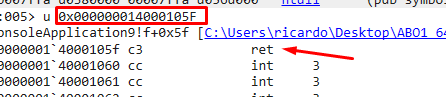
.reload /f

y luego que termina de bajar todos los símbolos.

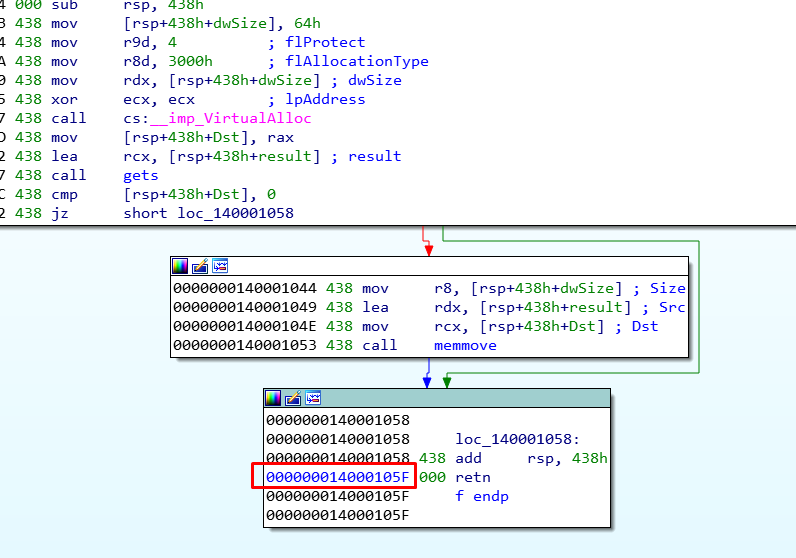
lm

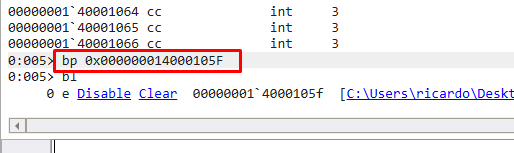


Bueno tengo los símbolos.

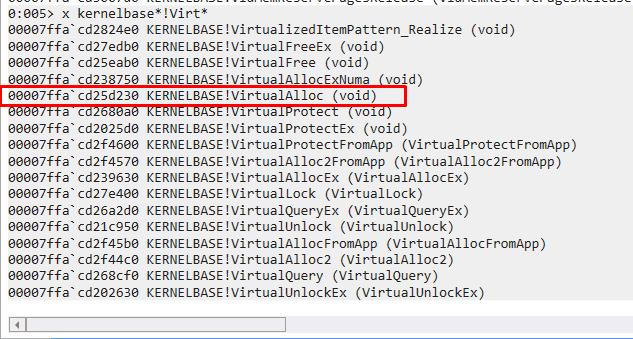


Pongo un Breakpoint alli.



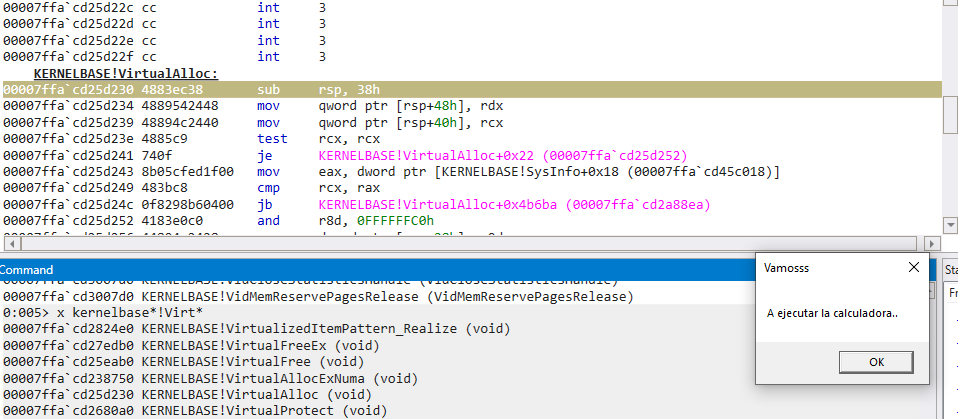


Bueno ahora a tracear hasta llegar al shellcode, puedo poner un Breakpoint en VirtualAlloc



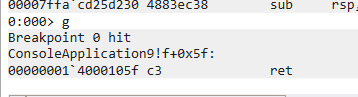
bp KERNELBASE!VirtualAlloc

Doy RUN con G y acepto el MessageBoxA y para cuando llama a VirtualAlloc el programa al inicio, sigo con G.

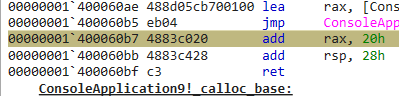


Para otra vez an VirtualAlloc pero debemos parar primero en el RET y la que vale es la siguiente parada en VirtualAlloc seguimos,

Ahi paro en el RET

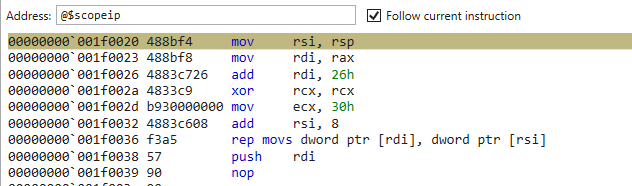


Apreto G nuevamente y para ahora si en VirtualAlloc, SHIFT MÁS f11 es STEP OUT para salir de la funcion justo después del RET.



Traceo con f11.

Llegamos a mi código.

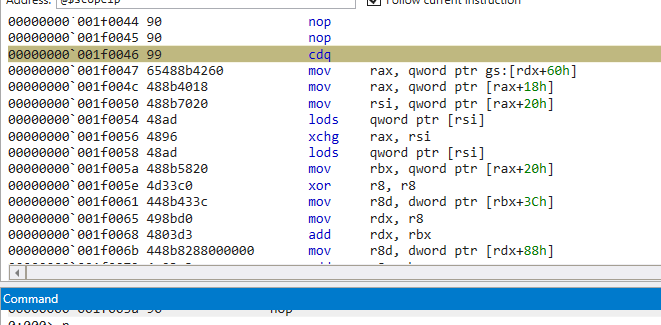


sigo traceando con F10 para pasar el REP MOVS y que no se repita.

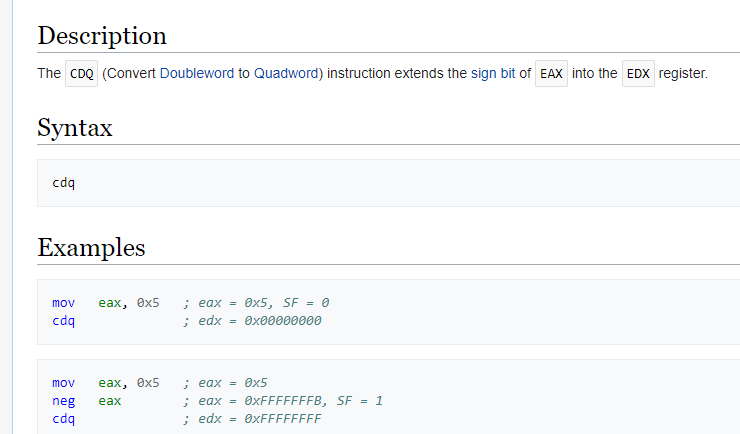
Aquí empieza el RESOLVER del SHELLCODE.

RESOLVER 64 BITS

HALLANDO LA IMAGEBASE DE KERNEL32



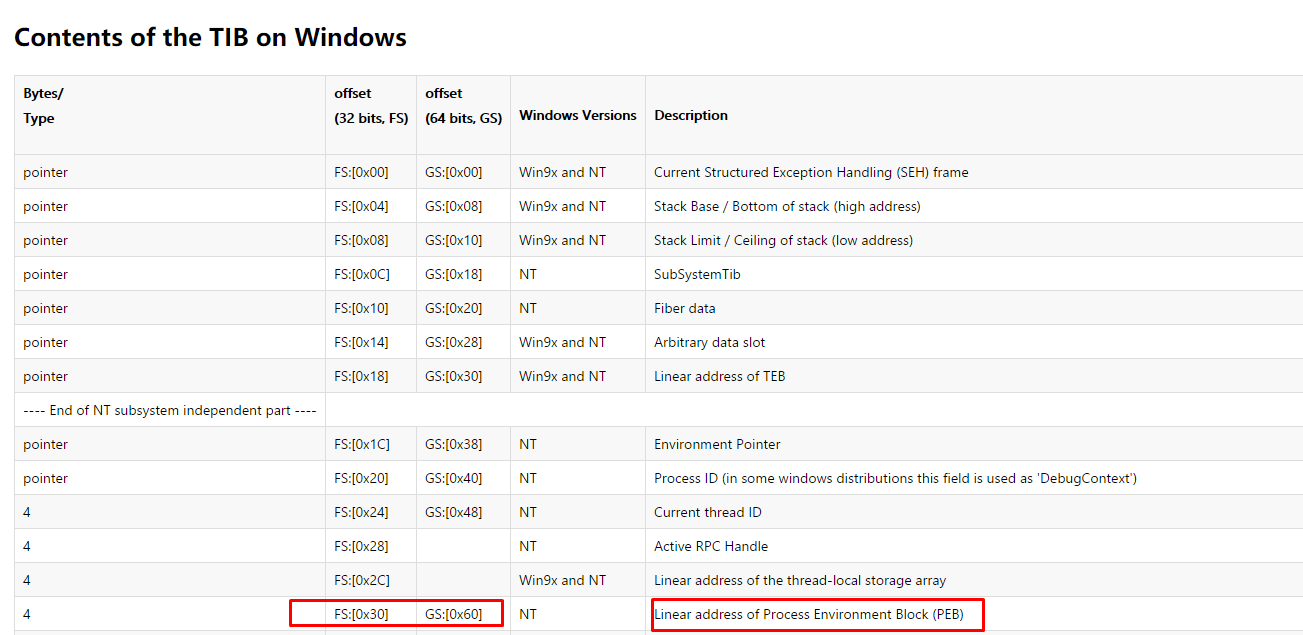
CDQ si el flag de signo SF es cero pone a cero RDX, podría haber sido un XOR RDX, RDX



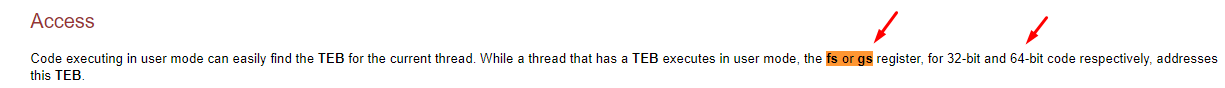


Es cero así que pone RDX=0.

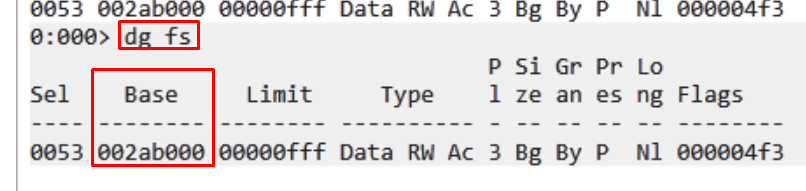
Recordemos que en 32 bits, la TEB o TIB estaba apuntada por FS.



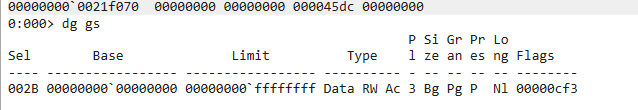
En 64 bits se utiliza el registro GS para la TEB.



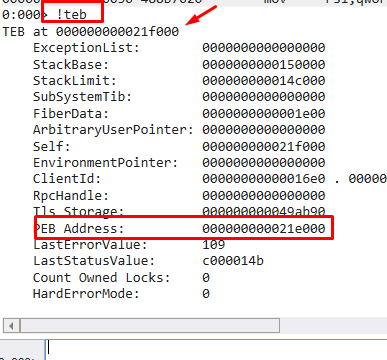
En 32 bits pudimos usar el comando **dg fs** para ver el valor de FS.



No funciona con GS.



Igual tenemos más trucos jeje existe el comando !teb



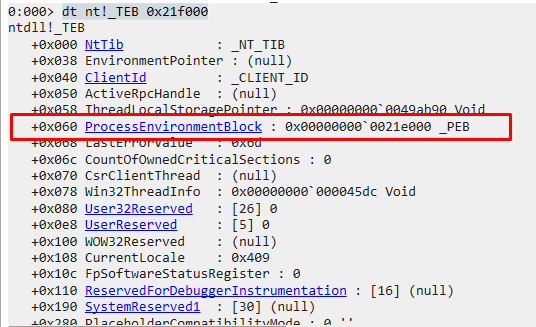
Alli vemos el contenido de la TEB y su base en mi maquina 0x21f000 y la dirección de la PEB en mi máquina 0x21e000.

Si traceo la primera instrucción está leyendo la PEB del campo 0x60.

**mov rax, qword ptr gs:[rdx+60h]**

Como conozco la dirección de la TEB en mi máquina que es 0x21f000 puedo usar el comando dt y verlo mejor.

**dt nt!\_TEB 0x21f000**

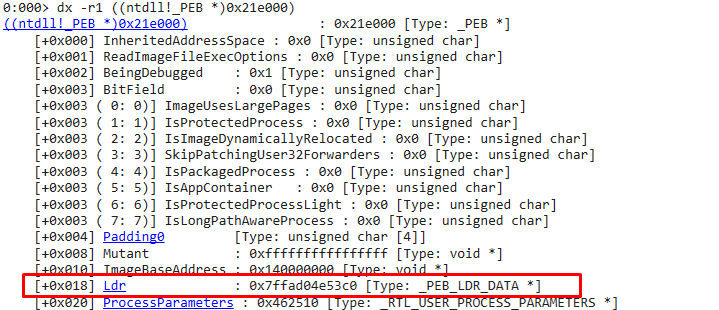


Además tengo un link para mostrar la PEB que me servirá.

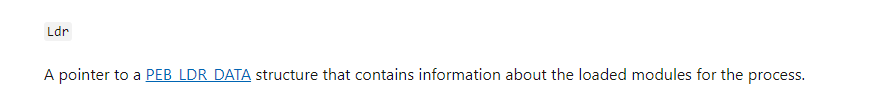
La segunda instrucción es:



Lee el campo que está en el offset 0x18 de la PEB, como tengo el link puedo hacer click y ver el listado de la PEB.

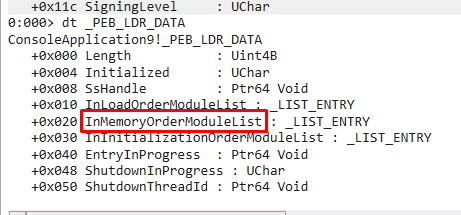


Lee PEB->Ldr



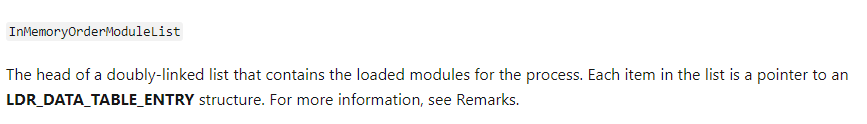
Bueno sigamos.

Podemos hacer click en LDR o listar \_PEB\_LDR\_DATA



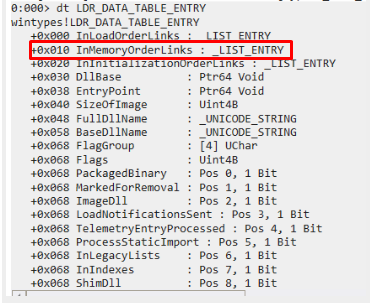
Vemos que en el offset 0x20 carga InMemoryOrderModuleList

Microsoft dice

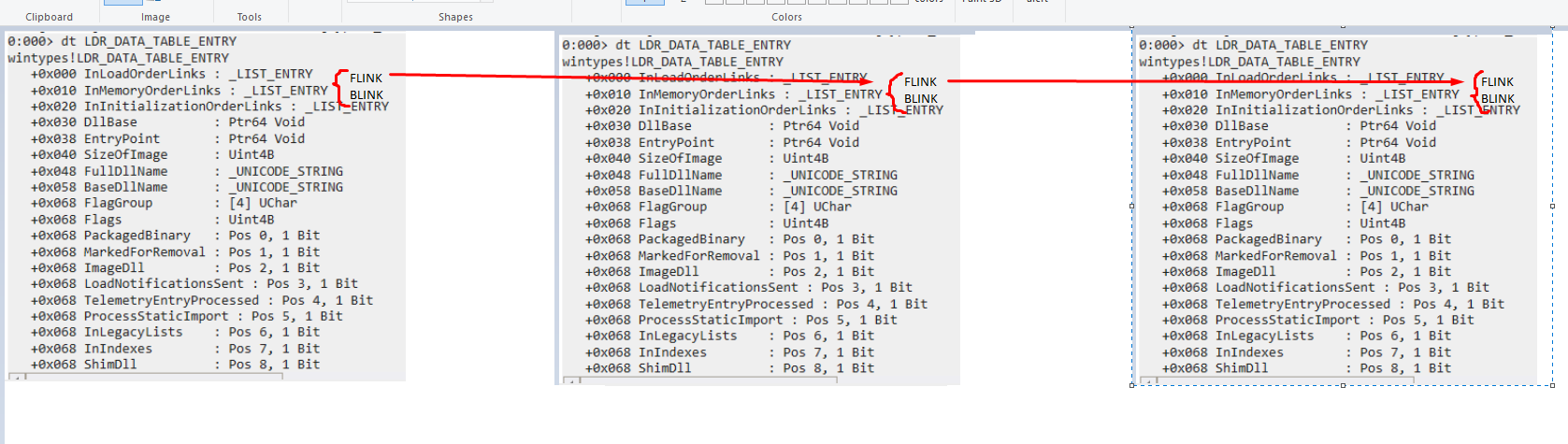


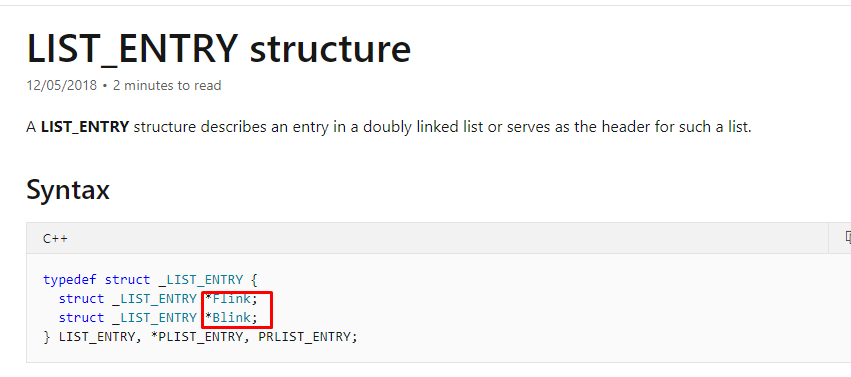
En algunas páginas web (y cuando hicimos la parte del RESOLVER de 32 bits) se le llama a LDR\_DATA\_TABLE\_ENTRY también LDR\_MODULE que es más corto pero es lo mismo.

Igual conviene llamarlo LDR\_DATA\_TABLE\_ENTRY ya que de esa forma se lista en el Windbg.



Como vemos el primer campo es del tipo \_LIST\_ENTRY y como dice la documentación este tiene su FLINK que apunta a una estructura similar que corresponde al siguiente módulo, siendo una lista enlazada.



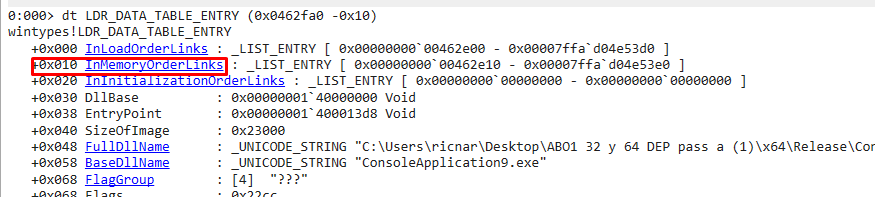


Así que como vemos en la imagen, las estructuras se conectan entre sí, por medio de los FLINK y BLINK, como FLINK es un puntero a la siguiente estructura, solamente hallando el contenido del mismo, tendremos el FLINK de la estructura siguiente.

**mov rsi, [rax + 0x20]**

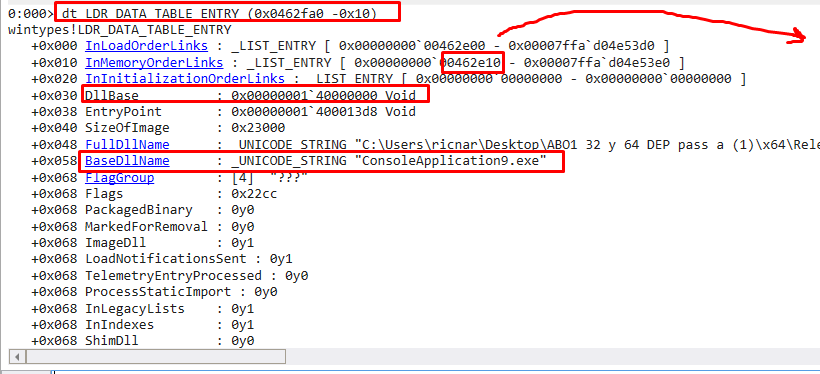
Así que en esta instrucción carga en RSI=InMemoryOrderModuleList que como vimos es el inicio de la lista enlazada y a su vez pertenece al primer modulo de la cadena de LDR\_DATA\_TABLE\_ENTRY.

Los que siguieron el tutorial del resolver en 32 bits recordaran que alli se usaba el primer campo InLoadOrderLinks, ambos sirven son dos listas con la misma información sobre los módulos, sólo cambiará el orden en que están ubicados y que en este caso en vez de estar en el OFFSET 0x0 de la estructura como estaba InLoadOrderLinks, estamos con nuestro FLINK siempre en el offset 0x10.



RSI está en el offset 0x10 del primer LDR\_DATA\_TABLE\_ENTRY , en mi máquina vale 0x0462fa0.

Puedo listar en el Windbg.



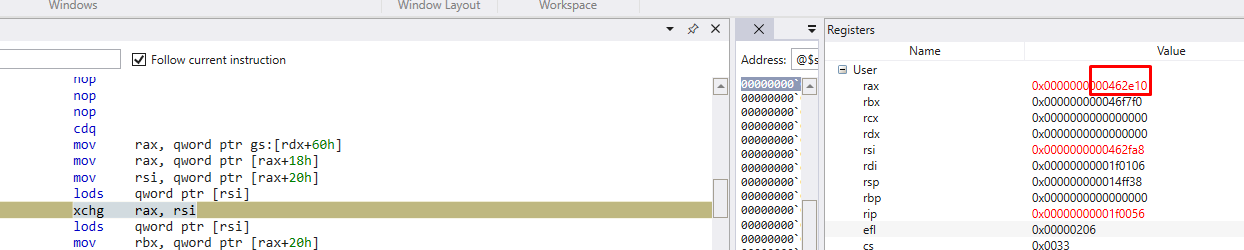
Alli vemos que estábamos en el offset 0x10 por eso tuvimos que restar 0x10 para listar la estructura

dt LDR\_DATA\_TABLE\_ENTRY (0x0462fa0 -0x10)

Vemos que corresponde al módulo ejecutable, que siempre es el primero de la cadena, alli vemos su Imagebase y su nombre, también vemos el FLINK a la estructura del segundo módulo.

Eso programáticamente se hace hallando el contenido de ESI ya que la instrucción LODS lee el contenido de ESI y lo mueve a EAX.

lods qword ptr [rsi] ds:0x00462fa0=**0x0462e10**

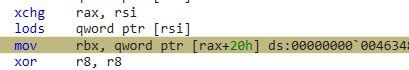


EAX está nuevamente en el offset 0x10 de la segunda estructura podemos mirar a ver a qué módulo corresponde.

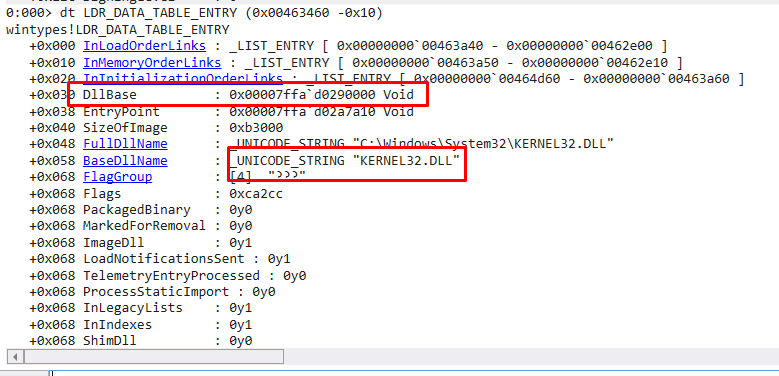


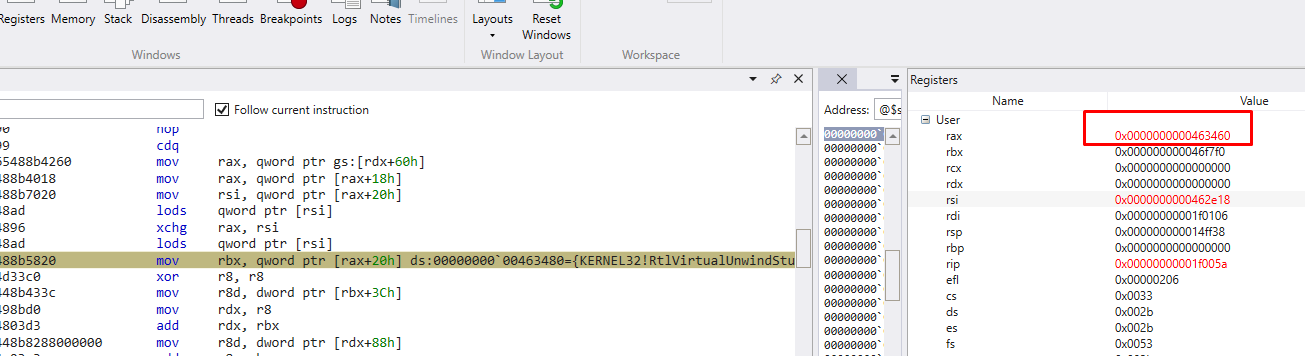
Vemos que el segundo LDR\_DATA\_TABLE\_ENTRY corresponde a ntdll.dll y que el tercero está apuntado por el FLINK será 0x463460.

Luego lo mueve EAX a ESI usando XCHG.

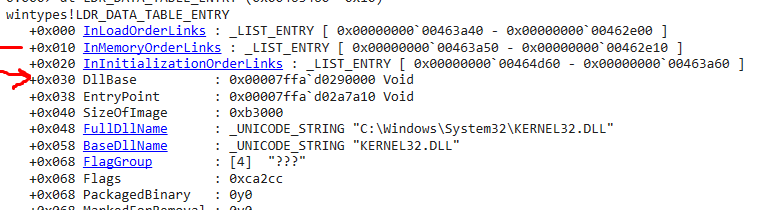


Y luego halla el contenido nuevamente usando LODS y por supuesto coincide será 0x463460.

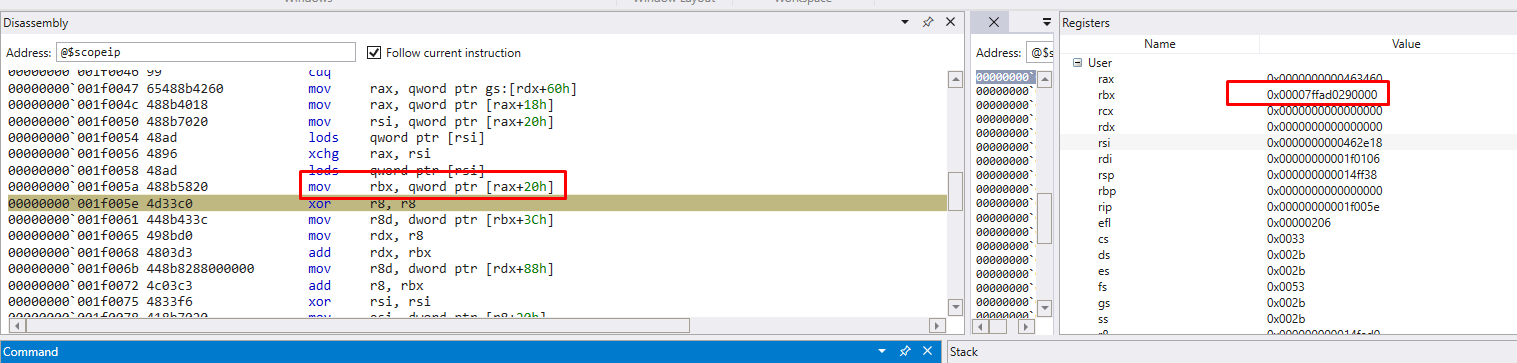




Corresponde a kernel32.dll y como EAX está posicionado en el offset 0x10, para leer la base de kernel32.dll hay que sumarle 0x20 para llegar a 0x30 donde está la misma.

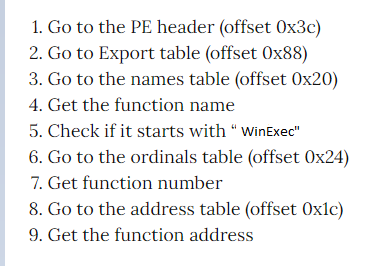


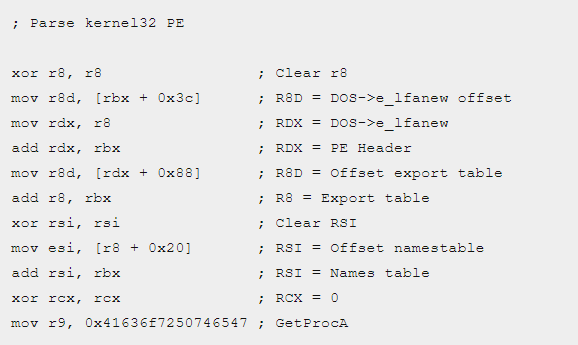
Con esto ya encontró la base de kernel32.dll que era el primer objetivo a buscar.



HALLANDO LA DIRECCIÓN DE WINEXEC

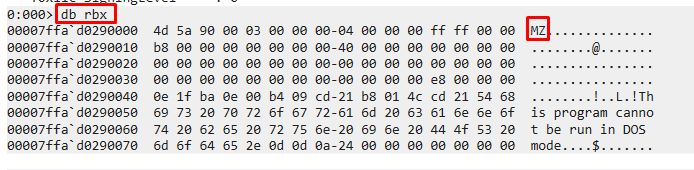
Una vez hallada la base de Kernel32.dll los pasos para hallar WinExec o la funcion que queramos dentro de kernel32 son los siguientes.



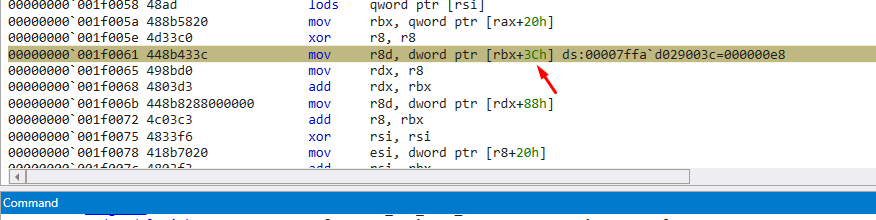


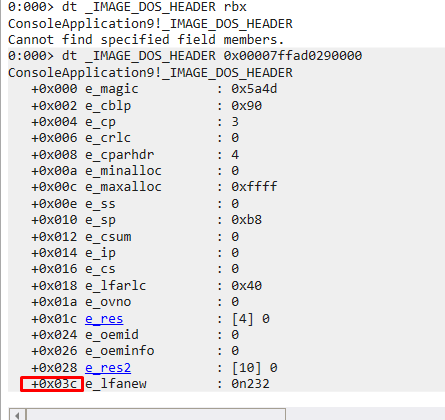
Eso es es parte del código vamos a tracear para chequear que todo corresponda.

La estructura donde comienza el header, que por supuesto está en la dirección de la imagebase de kernel32.dll que hallamos, se llama **\_IMAGE\_DOS\_HEADER** alli vemos el característico MZ los dos bytes que están al comienzo en los ejecutables DOS.



Vemos que en el shellcode se lee el campo del offset 0x3c

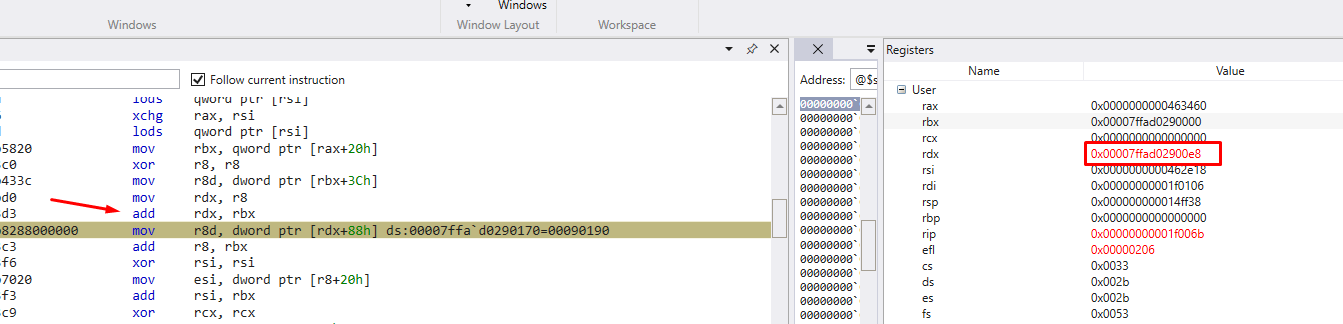




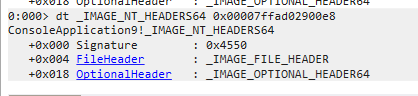
Que vale 232 decimal o sea 0xe8 es el offset de **\_IMAGE\_NT\_HEADERS64**



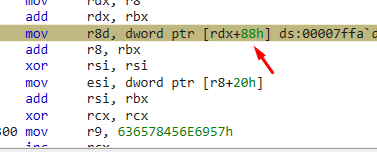
Le suma la imagebase para obtener la dirección.

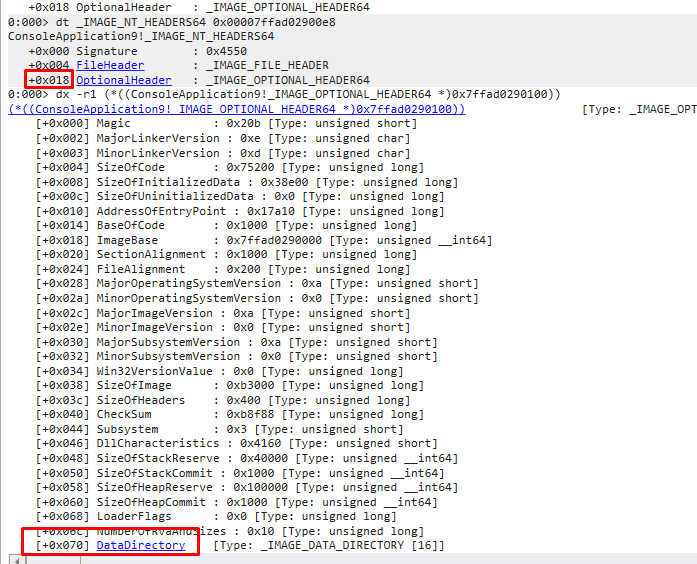


En RDX tenemos la dirección de \_IMAGE\_NT\_HEADERS64



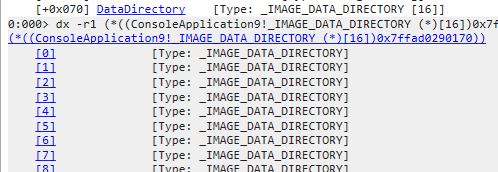
Luego busca el campo 0x88 vemos que está dentro de OptionalHeader que está en 0x18.

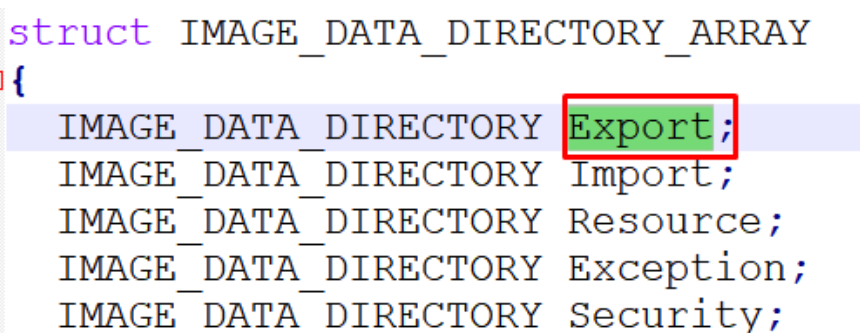




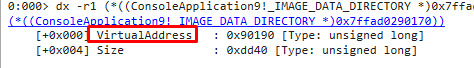
En 0x70 está \_IMAGE\_DATA\_DIRECTORY64 más el 0x18 de \_IMAGE\_OPTIONAL\_HEADER64, estamos en 0x88 como lee el shellcode.

Eso es un ARRAY de \_IMAGE\_DATA\_DIRECTORY

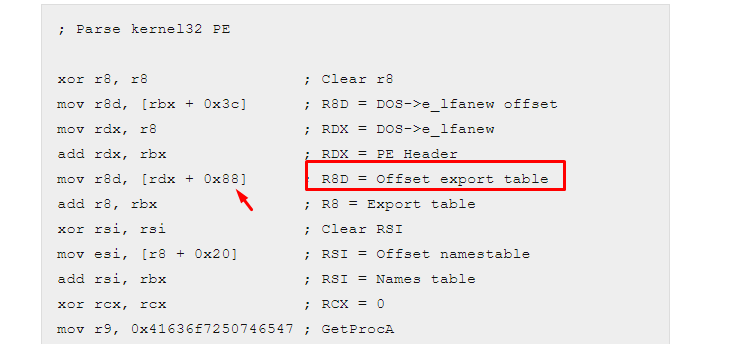




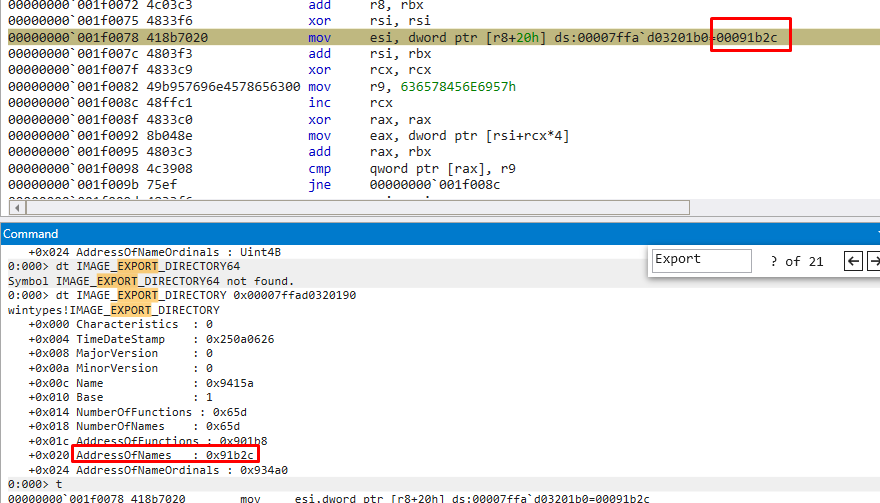
El primero es el offset a la EXPORT TABLE su primer campo es el offset a la dirección.



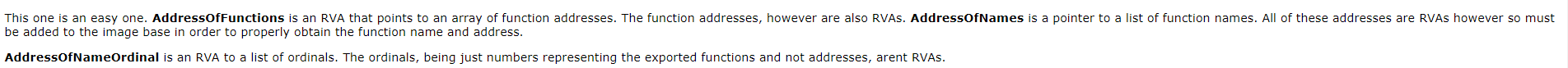
Luego le suma la base para obtener la dirección de la EXPORT TABLE.



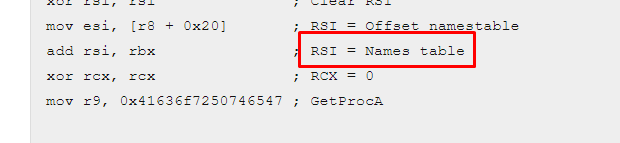




Lee el AddressofNames de offset 0x20.

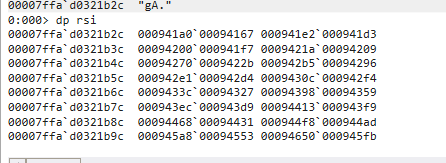


Así que hay tres Arrays AddressOfFunctions en 0x1c, AddressOfNames en 0x20 y AddressOfNameOrdinals en 0x24.



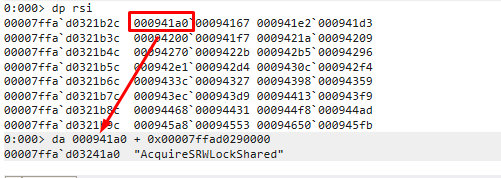


Así que RSI tiene la tabla o Array de los nombres.

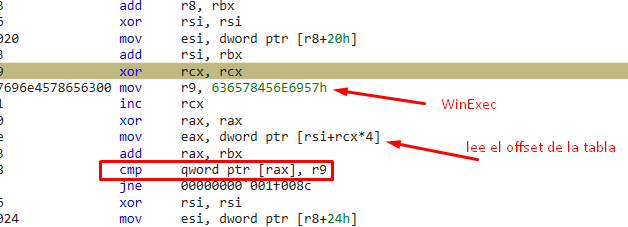


A cada una de las entradas de la tabla, para hallar la string con el nombre hay que sumarle la base y eso es lo que hace va recorriendo la tablita leyendo el offset y le suma la base y compara con WinExec (si queremos otra funcion hay que cambiar aquí por el nombre de la que queremos hallar)

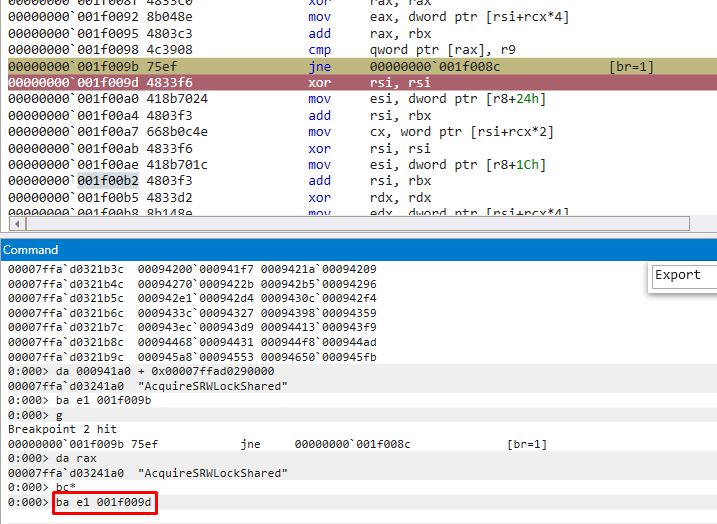
Veamos a qué apunta el primero de la tablita.



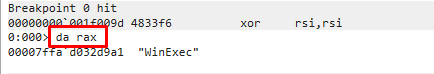
Y así cada uno de estos offset más la base apunta al nombre de una funcion exportada, por eso hace un loop recorriendo la tabla, comparando cada string con WinExec.



Podemos poner un BREAKPOINT luego del JNE y apretar RUN para detenernos cuando halle el nombre.



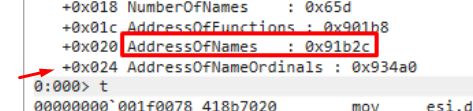
Cuando se detiene.

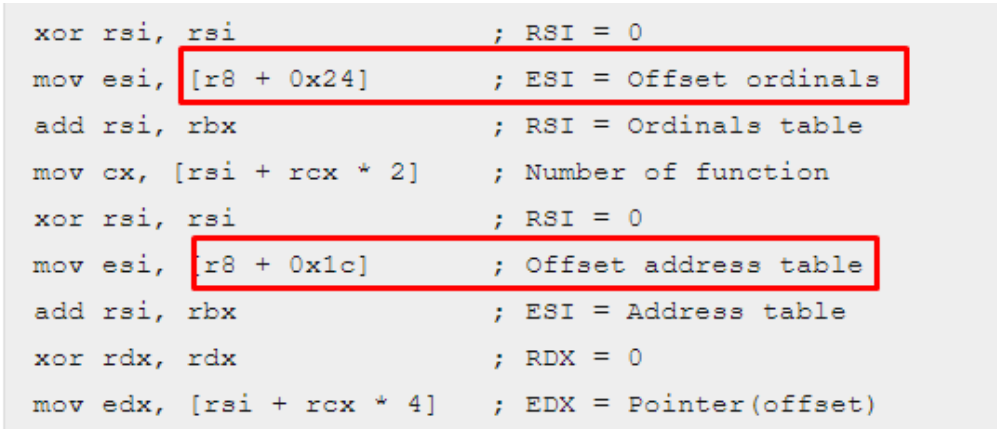


Vemos que fue incrementando RCX que era el índice de la tabla, así que la posición en la tabla para WinExec es RCX=0x60e.

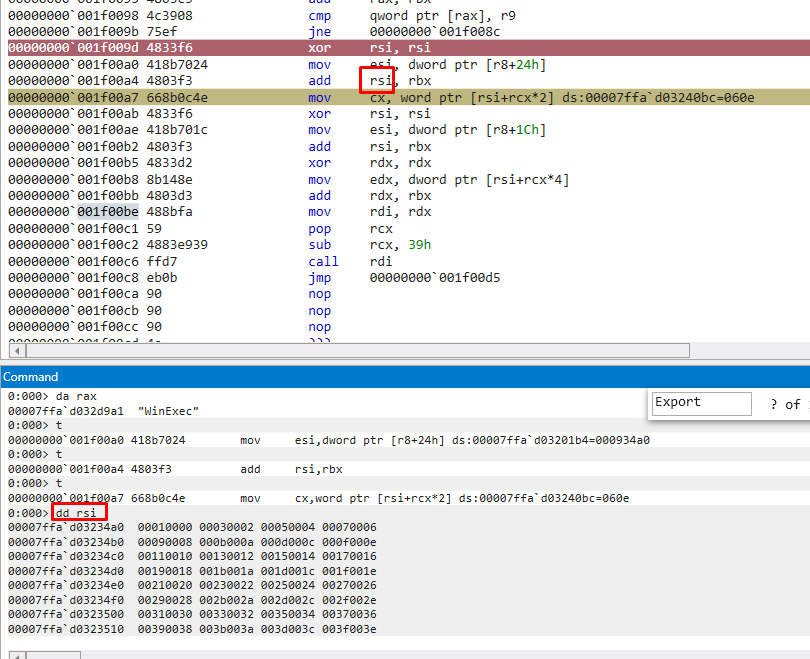


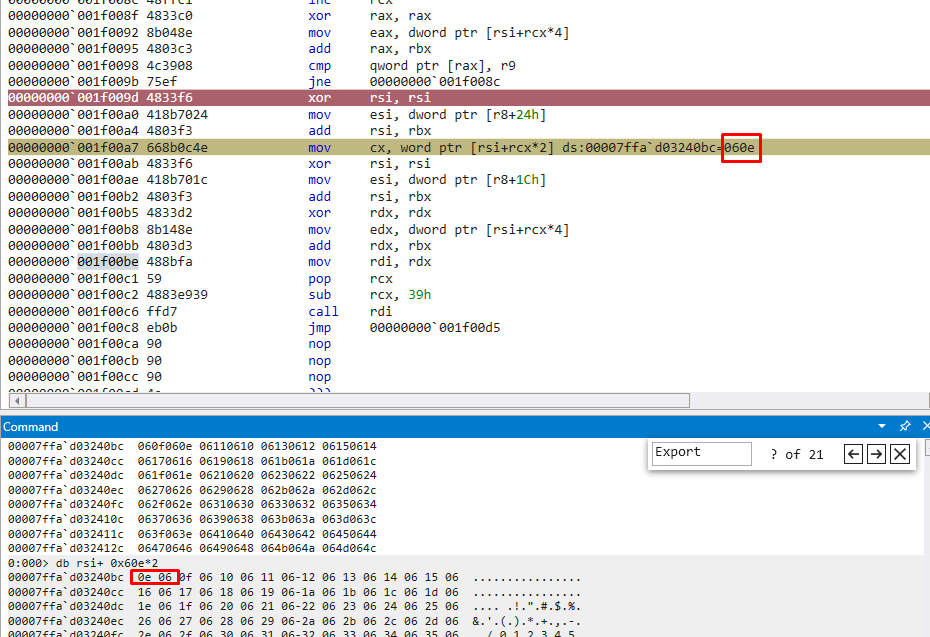
Recordamos que sumamos 0x20 a r8 y luego la base para hallar la tabla de nombres, si le sumamos 0x24 y luego la base hallamos la tabla de ordinales.



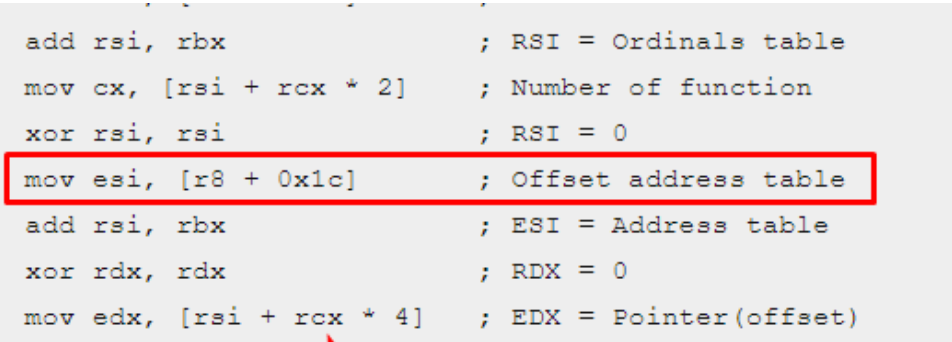


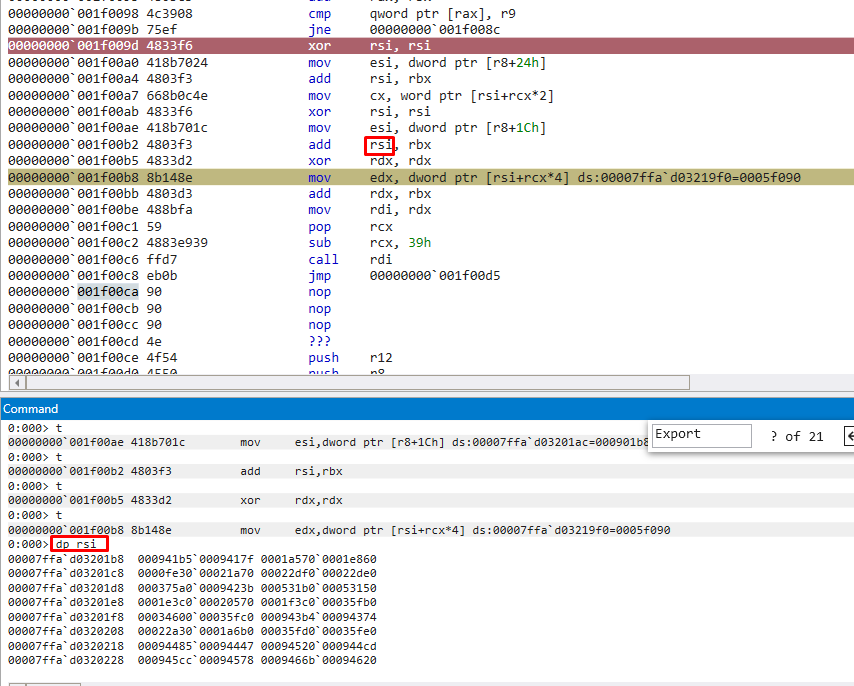
Con ello en la tabla de ordinales sin loopear usando el valor índice que está en RCX de la posición de la tabla de nombres, leemos el numero de la funcion de esa tablita.



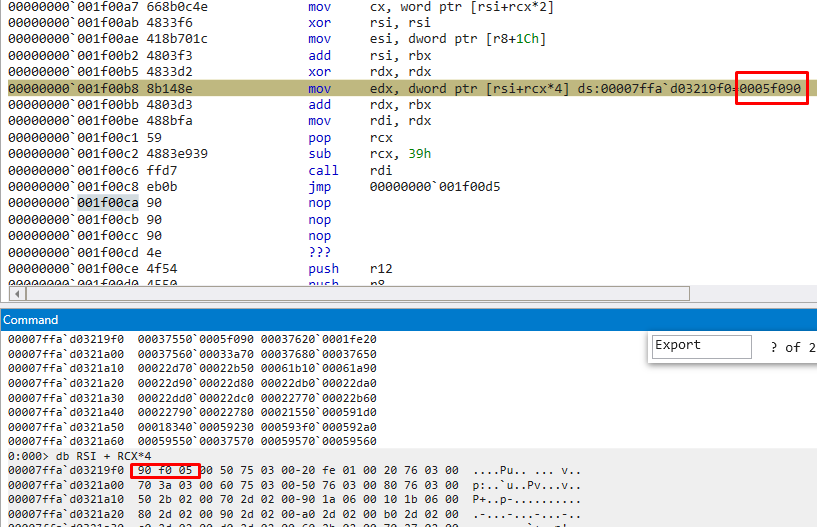


También es 0x60e, eso lo usa en la última tabla para hallar el offset de la funcion WinExec.



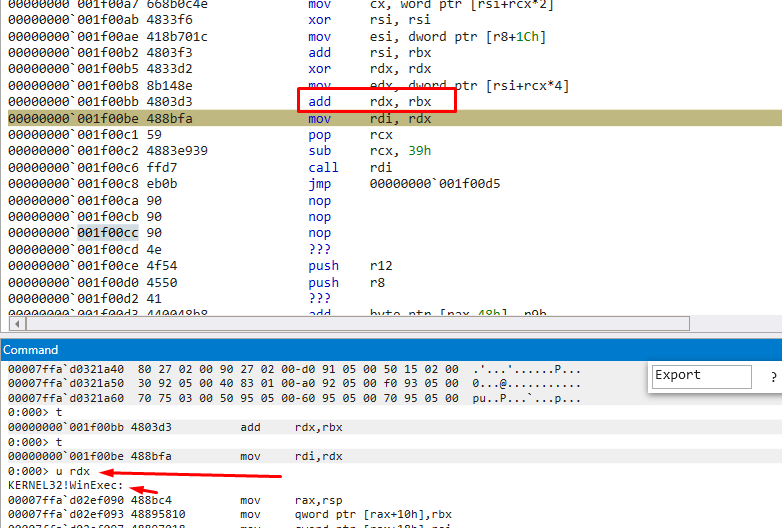


Así que RSI + RCX\*4 nos da el offset de WinExec

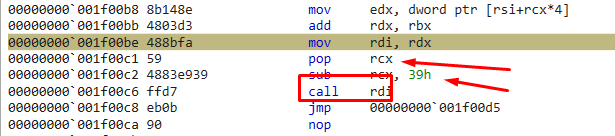


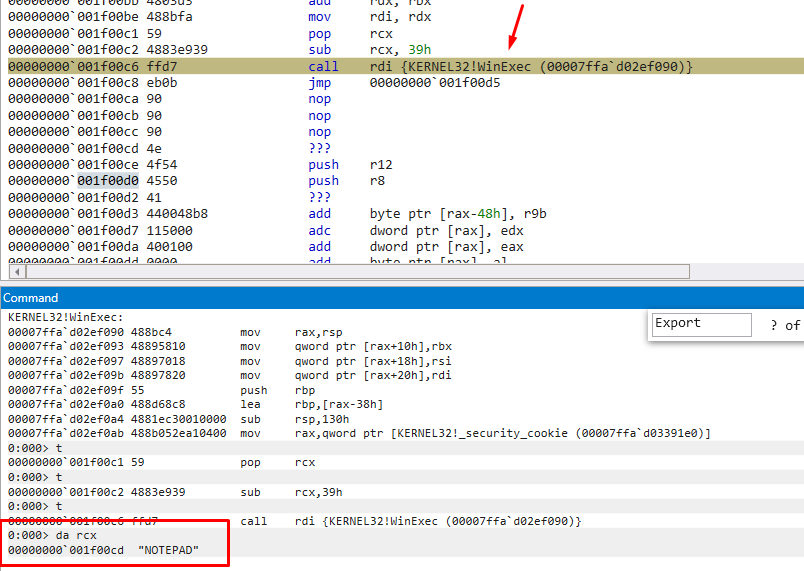
Al cuál le sumamos la base y tenemos la dirección virtual de WinExec.

LLAMANDO A WINEXEC

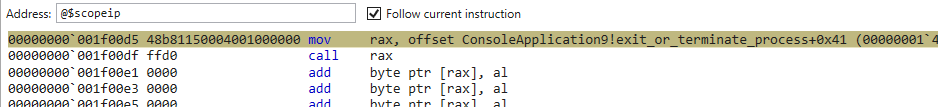


Y eso es todo, luego acomoda una string NOTEPAD que pase y salta a ejecutar WinExec con el argumento “NOTEPAD”.

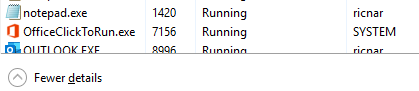




Y luego si pasamos con f10 el call y seguimos llama a ExitProcess para cerrarlo.



Alli quedó el NOTEPAD ejecutandose.



Bueno con esto terminamos el RESOLVER de 64 bits.

Hasta la parte 14.

Ricardo Narvaja

23/03/2020