

INTRODUCCION AL CRACKING CON OLLYDBG PARTE 4

INSTRUCCIONES

Como ya venimos diciendo en las anteriores partes, la idea de esta introducción es ir aprendiendo a la vez que indigestándonos con la dura teoría, practicándola en OLLYDBG para ir conociendo el mismo, y de paso ir tomado confianza e ir mirando las posibilidades que tiene.

Así como con los flags vimos en OLLYDBG que instrucciones los activaban, ahora veremos que ocurre al ejecutar cada instrucción en el mismo OLLYDBG lo cual da una idea mas cercana a la realidad y se hace mas llevadero.

Veremos las instrucciones mas importantes si al mirar un listado uno encuentra alguna que no se definimos aquí, siempre se puede consultar un manual de ASM para ver que función cumple.

NOP (NO OPERATION)

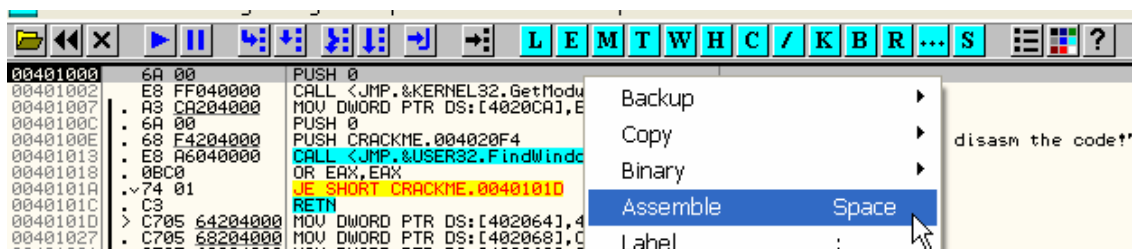
Es la instrucción que al ejecutarla no hace nada ni provoca ningún cambio en registros, stack o memoria, por eso en ingles es NO OPERATION o sea que no hay operación alguna, y para que se usa dirán, muchas veces al cambiar una instrucción por otra, queda un hueco ya que la instrucción que escribimos es mas corta, y hay que rellenar con NOPS para que el procesador no encuentre basura y de error.

También sirve para anular cualquier instrucción, si la reemplazo por nops el programa ejecutara los mismos en vez de la instrucción original y no hará nada lo que es comúnmente conocido como NOPEAR.

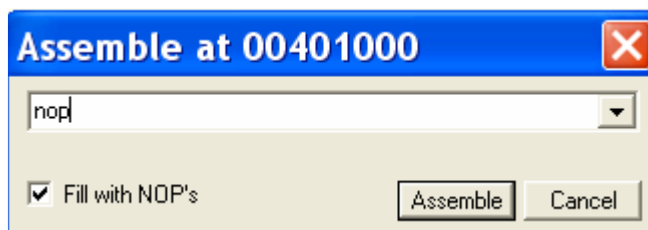
Si abrimos nuevamente el crackme de cruehead.



Allí vemos el código original en el inicio, lo que haremos será nopear la primera instrucción PUSH 0, que es un comando de 2 bytes de extensión, para ello marcamos la línea con el mouse, y luego apretamos la barra espaciadora o bien hacemos CLICK DERECHO –ASSEMBLE.



Allí vemos que en el mismo menú nos confirma que es lo mismo apretar la barra espaciadora, esto nos abrirá una ventana para escribir la instrucción que queramos.



Escribo NOP y apreto ASSEMBLE.



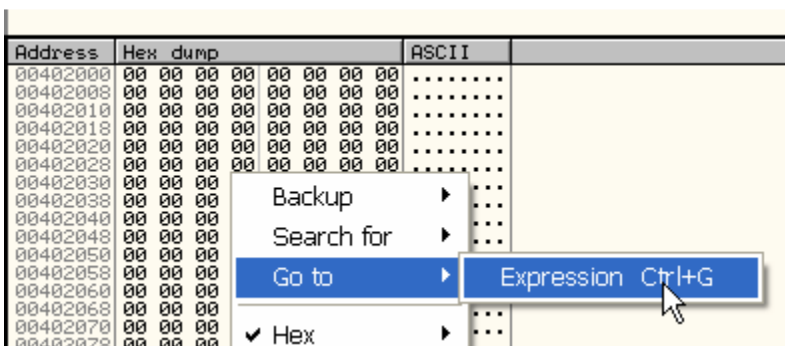
Vemos que OLLY además de escribirnos un NOP, como es bastante inteligente, reconoce que el PUSH que había antes es un comando de DOS BYTES por eso para no desarmar el código siguiente nos agrega otro NOP para completar con el NOPEADO del PUSH.

Si comparamos ambas imágenes vemos que donde estaba el PUSH 0 ahora hay dos NOPS que al ejecutarlos no hacen nada, esto lo podemos corroborar, apretando dos veces F7 hasta llegar al CALL siguiente, si miramos al apretar si cambia algún registro o algo en el stack, o un FLAG, vemos que todo esta igual, lo único que cambio fue EIP pues como sabemos apunta a la instrucción que se va a ejecutar y en este caso ahora es el CALL, pero no hay cambios en el resto de los registros ni stack, ni memoria.

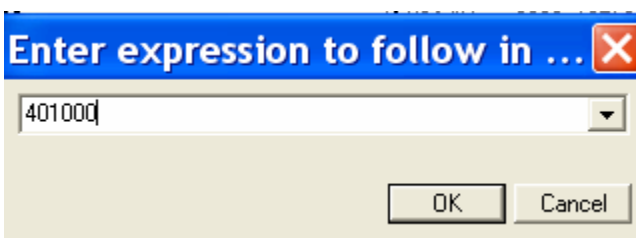
Ahora veremos esos mismos 2 bytes que reemplazamos en el DUMP, para hallarlos allí, debemos buscarlos por la dirección de memoria, vemos que la misma es 401000 y 401001



Voy a la ventana del DUMP y hago CLICK DERECHO - GOTO EXPRESSION y pongo la dirección de memoria a partir de la cual quiero que muestre los bytes que contiene.



Por supuesto tipo 401000



Los que vemos es

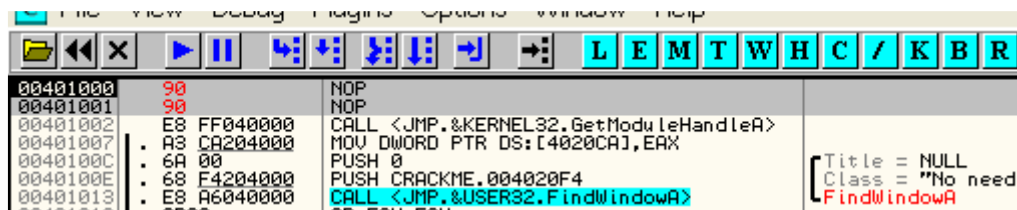
Address	Hex dump	ASCII
00401000	90 90 E8 FF 04 00 00 A3	EE ß ♦...ü
00401008	CA 20 40 00 6A 00 68 F4	± @.j.h¶
00401010	20 40 00 E8 A6 04 00 00	@.þë♦...
00401018	0B C0 74 01 C3 C7 05 64	ø'tøt±d
00401020	20 40 00 03 40 00 00 C7	@.ø@..ä
00401028	05 68 20 40 00 28 11 40	#h @.(4@
00401030	00 C7 05 6C 20 40 00 00	.ä! @..
00401038	00 00 00 C7 05 70 20 40	...äþ @
00401040	00 00 00 00 00 A1 CA 20i±
00401048	40 00 A3 74 20 40 00 6A	@.üt @.j
00401050	64 50 E8 D1 03 00 00 A3	dPßø♦...ü
00401058	78 20 40 00 68 00 7F 00	x @.h.ð.
00401060	00 6A 00 E8 A2 03 00 00	.j.þø♦...
00401068	A3 7C 20 40 00 C7 05 80	ü! @.äþç

El rojo es original puesto POR OLLY ya que cuando cambiamos bytes, OLLY nos los recuerda mostrandolos en color rojo, allí vemos los dos 90 que pusimos y luego E8, FF y los siguientes bytes que ya pertenecen a la siguiente instrucción que es un CALL.

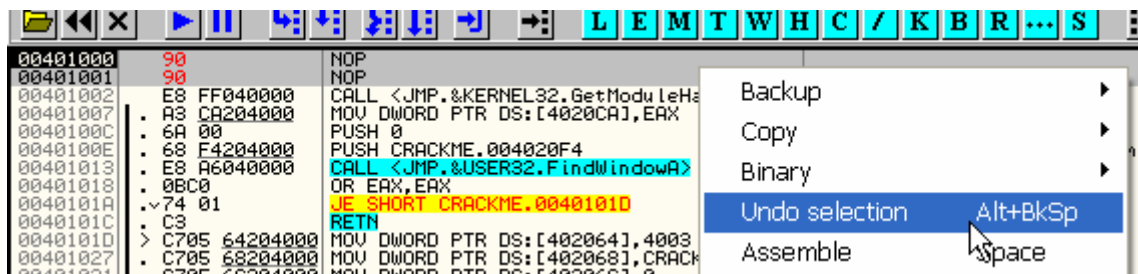
Podemos en OLLY quitar lo que hemos escrito y volver al código original?

Jeje, si podemos.

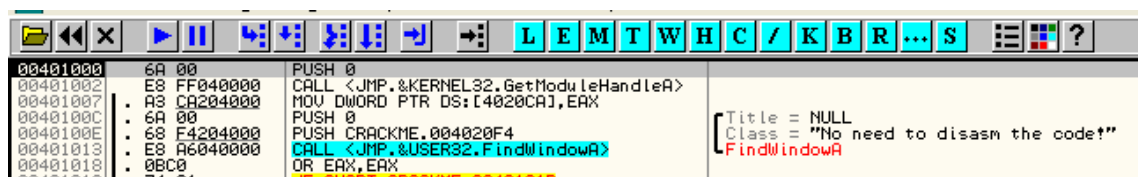
En cualquiera de los dos, sea en el DUMP o en el listado, marcamos los dos bytes que escribí.



Ahora hago CLICK DERECHO-UNDO SELECTION



Y aquí no ha pasado nada, vuelve a aparecer el PUSH original



Lo mismo si miramos el DUMP, vemos que ahora están los bytes originales.

Address	Hex dump	ASCII
00401000	6A 00 E8 FF 04 00 00 A3	j.þ ♦...ü
00401008	CA 20 40 00 6A 00 68 F4	± @.j.h¶
00401010	20 40 00 E8 A6 04 00 00	@.þë♦...
00401018	0B C0 74 01 C3 C7 05 64	ø'tøt±d
00401020	20 40 00 03 40 00 00 C7	@.ø@..ä
00401028	05 68 20 40 00 28 11 40	#h @.(4@
00401030	00 C7 05 6C 20 40 00 00	.ä! @..

Bueno eso es todo en cuanto a la instrucción NOP sigamos adelante.

INSTRUCCIONES DE STACK

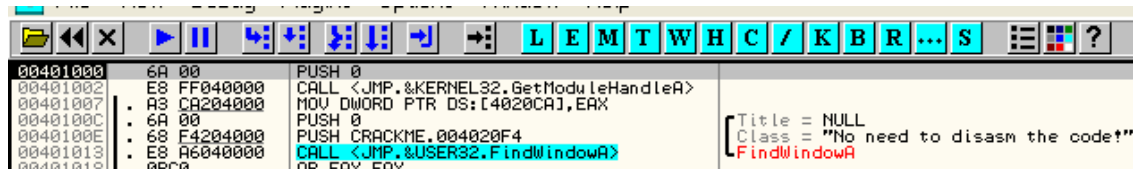
Bueno habíamos dicho que el stack era como un mazo de cartas que se le agregaban o quitaban cartas por arriba del mismo.

Por supuesto hay instrucciones para agregarle y quitarle cartas.

PUSH

La instrucción PUSH es la típica instrucción que agrega una carta o valor al stack

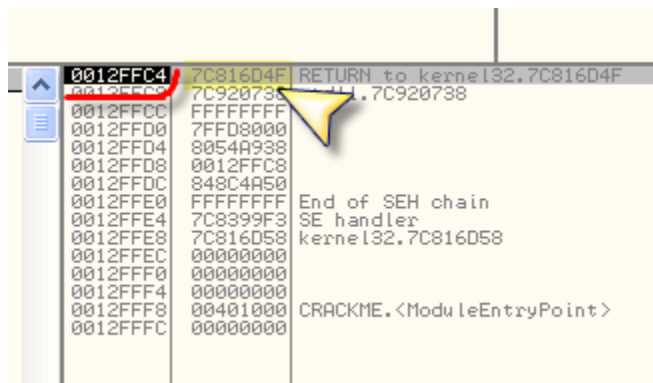
Veámoslo en OLLYDBG la primera instrucción del programa original del crackme de cruehead era un PUSH.



```
00401000 6A 00 PUSH 0
00401002 E8 FF040000 CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007 A3 C8204000 MOV DWORD PTR DS:[4020CA],EAX
0040100C 6A 00 PUSH 0
0040100E 68 F4204000 PUSH CRACKME.004020F4
00401013 E8 A6040000 CALL <JMP.&USER32.FindWindowA>
00401018 90 NOP
```

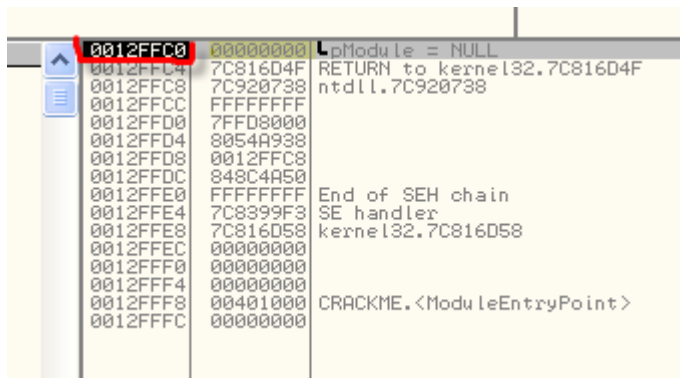
En este caso es un PUSH 0, al ejecutarla lo que hará en este caso, será colocar el 0 en la posición superior del stack, sin sobrescribir lo que se encontraba antes lo cual quedara debajo.

Veamos como esta el stack antes de ejecutar el PUSH, en mi maquina esta en esta dirección, en la de ustedes puede variar aunque el efecto será el mismo.



```
0012FFC4 7C816D4F RETURN to kernel32.7C816D4F
0012FFC8 7C920738 ntdll.7C920738
0012FFCC FFFFFFFF
0012FFD0 7FFD8000
0012FFD4 8054A938
0012FFD8 0012FFC8
0012FFDC 848C4A50
0012FFE0 FFFFFFFF End of SEH chain
0012FFE4 7C8399F3 SE handler
0012FFE8 7C816D58 kernel32.7C816D58
0012FFEC 00000000
0012FFF0 00000000
0012FFF4 00000000
0012FFF8 00401000 CRACKME.<ModuleEntryPoint>
0012FFFC 00000000
```

Este es el stack en mi maquina, la dirección 12FFc4 puede variar en su máquina, ya que el stack se puede acomodar en otra dirección en cada caso, y el contenido inicial a veces también puede variar o sea que ustedes pueden tener otro valor que 7c816d4f, pero no importa al ejecutar F7, el cero pasara a la parte superior del stack y el resto quedara abajo, veamos apreto F7.

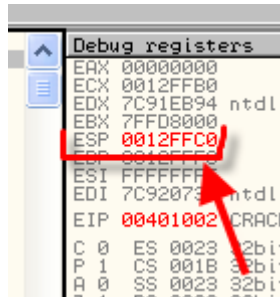


```
0012FFC0 00000000 LpModule = NULL
0012FFC4 7C816D4F RETURN to kernel32.7C816D4F
0012FFC8 7C920738 ntdll.7C920738
0012FFCC FFFFFFFF
0012FFD0 7FFD8000
0012FFD4 8054A938
0012FFD8 0012FFC8
0012FFDC 848C4A50
0012FFE0 FFFFFFFF End of SEH chain
0012FFE4 7C8399F3 SE handler
0012FFE8 7C816D58 kernel32.7C816D58
0012FFEC 00000000
0012FFF0 00000000
0012FFF4 00000000
0012FFF8 00401000 CRACKME.<ModuleEntryPoint>
0012FFFC 00000000
```

Vemos que al ejecutar F7 fue realmente como si se agregara encima el cero que vemos resaltado allí, abajo en 12ffc4 sigue estando 7c816d4f, y no vario nada todos los mismos valores están en las mismas direcciones del stack.

La única diferencia que ahora el valor superior del stack es 12ffc0 y allí esta el cero que pusimos con el PUSH, fue realmente como agregar una carta a un mazo, se colocó arriba y dejó el resto que estaba antes sin cambiar nada debajo.

Vemos también que el puntero ESP que muestra la dirección del valor superior del stack ahora marca 12FFc0.



Por supuesto el comando PUSH tiene variantes no solo puedo agregar números, si hago:

PUSH EAX, agregare el valor de EAX en lugar del cero, podemos PUSHear cualquier registro, número etc.

Podemos PUSHear también el contenido de una dirección de memoria

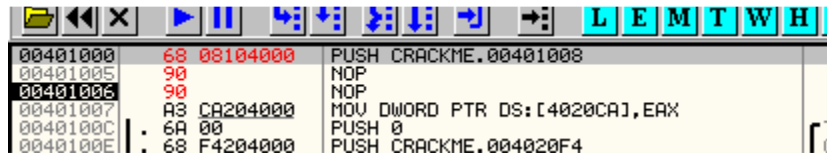
PUSH [401008]

Veamos que se debe interpretar bien la diferencia con

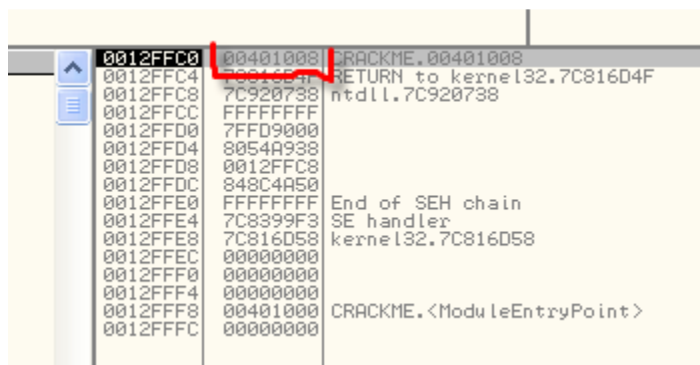
PUSH 401008

Sin corchetes

Si hago PUSH 401008 lo que hará será colocar el número 401008 en el stack



Al ejecutarlo quedará así



En cambio si fuera PUSH [401008]

Los corchetes significan el contenido de la memoria en 401008 o sea que debemos ir en el DUMP a ver esa direcci3n y ver que hay alli.

Con GOTO EXPRESSION 401008 vemos

Address	Hex dump	ASCII
00401008	CA 20 40 00 6A 00 68 F4	z @.j.h
0040100E	40 00 00 00 A6 04 00 00	@.p@..
00401018	0B C0 74 01 C3 C7 05 64	!t@t\$#d
00401020	20 40 00 03 40 00 00 C7	@.@..z
00401028	05 68 20 40 00 28 11 40	#h @.(
00401030	00 C7 05 6C 20 40 00 00	.A!l @..
00401038	00 00 00 C7 05 70 20 40	...z#p @
00401040	00 00 00 00 00 A1 CA 20i
00401048	40 00 A3 74 20 40 00 6A	@.ut @.j
00401050	24 00 00 00 00 00 00 00	DPK.D

Que los 4 bytes que hay all3 son CA 20 40 00, ejecutemos con F7 el PUSH

Vemos que mando al stack el valor que ley3 pero los bytes est3n al rev3s o sea nosotros vemos en el dump CA 20 40 00 y los puso al rev3s o sea 00 40 20 CA.

Bueno esta es una propiedad del procesador, al leer o escribir contenidos de memoria siempre los bytes se toman al rev3s, y bueno quejas al inventor del procesador jeje.

Pero la idea que debe quedar grabada es que sin corchetes el valor es simplemente un n3mero, y con corchetes el valor refiere al contenido de una direcci3n de memoria.

Ahora vemos que el OLLY cuando escribimos PUSH [401000] interpreto y escribi3

PUSH DWORD PTR DS:[401008]

Porque ocurri3 eso

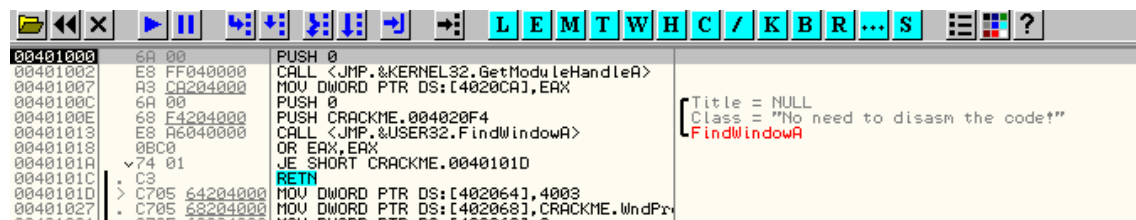
Es que si uno no aclara el OLLY interpreta que uno quiere leer los 4 bytes de esa posición de memoria, eso es DWORD leer los 4 bytes completos ya veremos las otras variantes en otras instrucciones.

POP

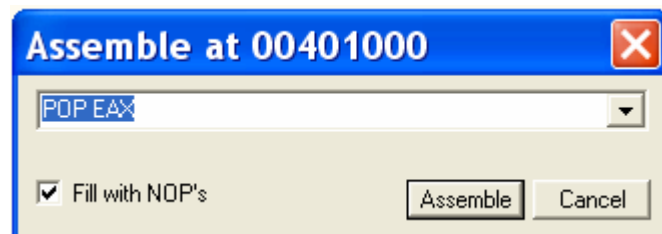
La instrucción POP es la inversa de PUSH lo que hace es quitar la primera carta o el primera valor del stack y lo coloca en la posición que indicamos a continuación del POP, por ejemplo,

POP EAX tomara el primer valor del stack y lo quitara moviéndolo a EAX, y será como si quitamos una carta, ya que el valor que estaba debajo quedara como primero.

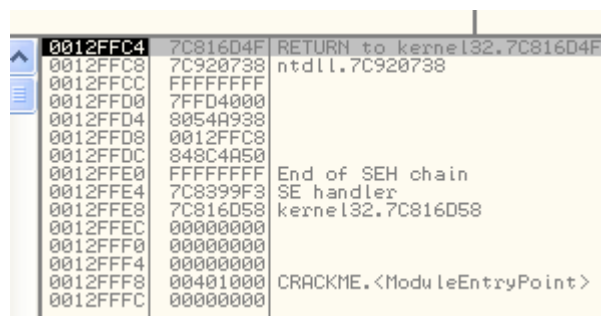
Vemos arranquemos de nuevo el crackme de cruehead y en el inicio esta



Cambiamos dicha instrucción por POP EAX, marcamos la primera línea, apretamos la barra espaciadora y tipeamos.



Ahí esta el stack antes de ejecutar la instrucción esta



Y ESP apunta a 12FFc4 que es el valor superior del stack.

Debug registers		
EAX	00000000	
ECX	0012FFB0	
EDX	7C91EB94	ntdll.
EBX	7FFD4000	
ESP	0012FFC4	
EBP	0012FFC0	
ESI	FFFFFFFF	
EDI	7C920738	ntdll.
EIP	00401000	CRACKME
C 0	ES 0023	32bit
P 1	CS 001B	32bit
A 0	SS 0023	32bit

Y vemos que EAX esta a cero antes de ejecutar la línea en mi caso.

Apreto F7

0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD4000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	848C4A50	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	CRACKME.<ModuleEntryPoint>
0012FFFC	00000000	

Vemos que en el stack desapareció nuestra primera carta, ahora el primer lugar lo ocupa la que antes estaba 2da y ESP apunta a 12ffc8.

Debug registers		
EAX	7C816D4F	kernel32.
ECX	0012FFB0	
EDX	7C91EB94	ntdll.
EBX	7FFD4000	
ESP	0012FFC8	
EBP	0012FFC0	
ESI	FFFFFFFF	
EDI	7C920738	ntdll.
EIP	00401001	CRACKME
C 0	ES 0023	32bit
P 1	CS 001B	32bit
A 0	SS 0023	32bit

Pero donde fue a parar nuestra carta se perdió?, noo como era un POP EAX fue a parar a EAX vemos en la imagen que EAX ahora vale 7c816d4f en mi caso y en el suyo tendrá el valor que antes estaba superior en el stack en vuestra maquina.

Lo mismo si hubiera sido POP ECX el valor superior hubiera ido a ECX o al registro que eligiéramos.

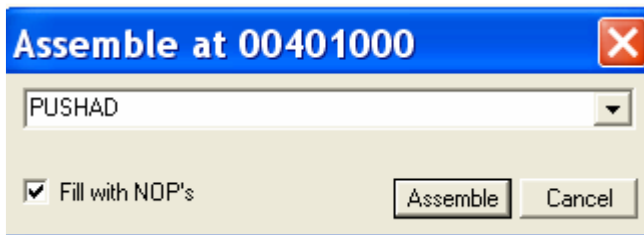
Bueno ya vimos las instrucciones que ponen o quitan una carta al stack ahora tenemos.

PUSHAD

PUSHAD guarda el contenido de los registros en la pila en un orden determinado. Así pues, pushad equivale a: push EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

Veamos si es cierto lo que nos dice nuestro amigo CAOS REPTANTE en su tute de asm, jeje.

Abrimos de nuevo el crackme de cruehead y ya sabemos que apretamos la barra para escribir y allí tipeamos PUSHAD.



0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD5000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	848C4A50	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	CRACKME.<ModuleEntryPoint>
0012FFFC	00000000	

Allí esta mi stack inicial y los registros antes de ejecutar el pushad son

Debug registers	
EAX	00000000
ECX	0012FFB0
EDX	7C91EB94 ntdll
EBX	7FFD5000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll
EIP	00401000 CRACK

Apreto F7 y veamos que paso en el stack

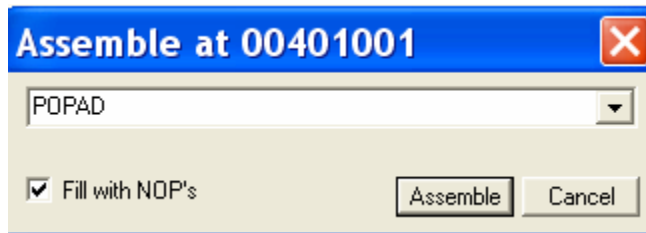
0012FFA4	7C920738	ntdll.7C920738
0012FFA8	FFFFFFFF	
0012FFAC	0012FFF0	
0012FFB0	0012FFC4	
0012FFB4	7FFD5000	
0012FFB8	7C91EB94	ntdll.KiFastSystemCallRet
0012FFBC	0012FFB0	
0012FFC0	00000000	
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD5000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	848C4A50	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	CRACKME.<ModuleEntryPoint>
0012FFFC	00000000	

Vemos que hizo un PUSH a cada uno de los registros, el primero que agrego esta arriba de 12ffc4 que era el valor superior del stack antes de ejecutar, ahora hay un cero arriba que corresponde a PUSH EAX, luego hizo PUSH ECX y mando el 12ffb0 que estaba en ECX, luego envió consecutivamente los valores de los registros uno a uno al stack hasta el ultimo que fue PUSH EDI.

[POPAD](#)

La inversa de PUSHAD es POPAD en este caso toma los valores del stack y los manda a los registros como si fuera un POP a cada uno. Así popad equivale a: pop EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX.

Allí mismo donde quedo del ejemplo anterior escribamos un POPAD



00401000	60	PUSHAD
00401001	61	POPAD
00401002	E8 FF040000	CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007	A3 C8040000	MOV DWORD PTR DS:[4020CA],EAX
0040100C	6A 00	PUSH 0
0040100E	68 F4204000	PUSH CRACKME.004020F4
00401012	EB 06040000	CALL <JMP.&USER32.FindWindow>

Allí esta para ejecutarse el POPAD los registros están guardados en el stack y al apretar F7 vuelven del stack a sus lugares originales.

0012FFC4	7C816D4F	lpModule = "PbT_ eee \$631(63
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD5000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	848C4A50	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	CRACKME.<ModuleEntryPoint>
0012FFFC	00000000	

Allí esta el stack como era antes del pushad y los registros recuperan sus valores

Debug registers	
EAX	00000000
ECX	0012FFB0
EDX	7C91EB94 ntdll
EBX	7FFD5000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll
EIP	00401002 CRACKME
C 0	ES 0023 32t
P 1	CS 001B 32t

La dupla PUSHAD-POPAD se usa mucho cuando se quiere guardar el estado de registros en un punto, realizar muchas operaciones que cambian registros y el stack, y luego con POPAD restaurar los registros y el stack al estado original.

Existen algunas variantes como

[PUSHA](#) equivale a: push AX, CX, DX, BX, SP, BP, SI, DI.

[POPA](#) equivale a: pop DI, SI, BP, SP, BX, DX, CX, AX (los valores recuperados correspondientes a ESP y SP, no se colocan en los registros sino que se descartan).

En el caso de PUSHA y POPA es similar a sus hermanas PUSHAD y POPAD salvo que utilizan los registros de 16 bits como vemos allí en la explicación de CAOS.

Como ejercicio podrían escribir un PUSHA ejecutarlo con F7 y interpretar los resultados y lo mismo con POPA.

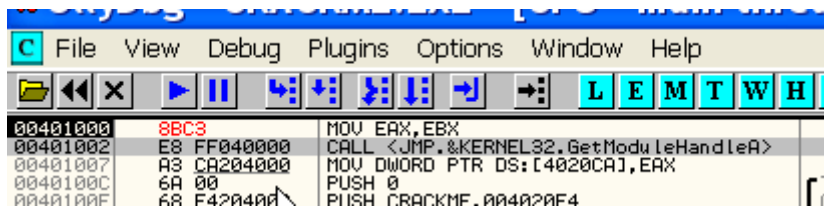
INSTRUCCIONES PARA MOVER DATOS

MOV

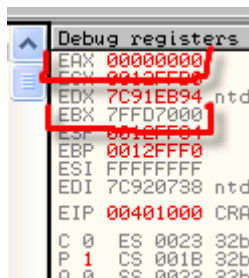
Esta instrucción es lo que comúnmente llamaríamos MOVER, mueve el segundo operando al primero por ejemplo.

MOV EAX, EBX

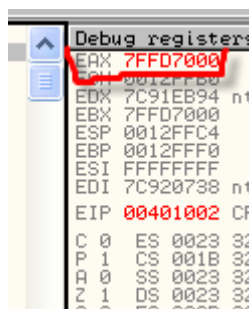
Lo que hace es mover el valor de EBX a EAX, miremos en OLLY y nuestro bendito crackme de cruehead.



Ya no repetiré como se escribe una instrucción ya lo hemos visto, veamos los registros antes de ejecutar



En mi maquina EAX es 0 y ECX es 7c91eb94, como son valores iniciales en la suya pueden ser diferentes pero al apretar F7 lo importante es que el valor de EBX lo moverá a EAX, veamos apretemos F7.



Viola esta claro no?

MOV tiene variantes por ejemplo

MOV AL, CL

Esto movería el valor de CL a AL veamos reinicio OLLYDBG y escribo

Address	Disassembly	Comment
00401000	8AC1	MOV AL,CL
00401002	E8 FF040000	CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007	A3 CA204000	MOV DWORD PTR DS:[4020CA],EAX
0040100C	6A 00	PUSH 0
0040100E	68 F4204000	PUSH CRACKME.004020F4
00401013	E8 A6040000	CALL <JMP.&USER32.FindWindowA>
00401018	0BC0	OR EAX,EAX
0040101A	74 01	JE SHORT CRACKME.0040101D
0040101C	C3	RETN
0040101D	> C705 64204000	MOV DWORD PTR DS:[402064],4003

Los registros

Debug registers	
EAX	00000000
ECX	0012FFB0
EDX	7C91EB94 ntdll
EBX	7FFDE000
ESP	0012FFC4
EBP	0012FFFF
ESI	FFFFFFFF
EDI	7C920738 ntdll
EIP	00401000 CRACK
C 0	ES 0023 32bit
P 1	CS 001B 32bit
A 0	SS 0023 32bit
Z 1	DS 0023 32bit
S 0	FS 003B 32bit
T 0	GS 0000 NULL

Recordamos lo que vimos ya que AL son las ultimas dos cifras de EAX y CL las dos ultimas cifras de ECX, ejecutemos con F7

Debug registers	
EAX	000000B0
ECX	0012FFB0
EDX	7C91EB94 ntdll.K
EBX	7FFDE000
ESP	0012FFC4
EBP	0012FFFF
ESI	FFFFFFFF
EDI	7C920738 ntdll.7
EIP	00401002 CRACKME
C 0	ES 0023 32bit 0
P 1	CS 001B 32bit 0
A 0	SS 0023 32bit 0

Vemos que sin tocar el resto de EAX y ECX el B0 se copio a AL, o sea las ultimas dos cifras de EAX.

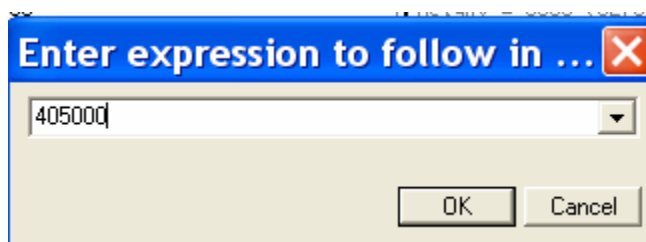
También podemos mover a algún registro el contenido de una posición de memoria o al revés.

Address	Disassembly	Comment
00401000	A1 00504000	MOV EAX,DWORD PTR DS:[405000]
00401005	90	NOP
00401006	90	NOP
00401007	A3 CA204000	MOV DWORD PTR DS:[4020CA],EAX
0040100C	6A 00	PUSH 0
0040100F	68 F4204000	PUSH CRACKME.004020F4

En esta caso moveremos el contenido de 405000 a EAX y como dice DWORD serán los cuatro bytes lo que moveremos.

Esta instrucción puede dar error si la dirección de memoria no existe, lo cual podemos ver fácilmente en el OLLY.

Vamos al DUMP y hacemos GOTO EXPRESSION 405000



Address	Hex dump	ASCII
00405000	00 10 00 00 DC 00 00 00
00405008	08 30 0F 30 1F 30 33 30	0*0030
00405010	3D 30 46 30 4B 30 58 30	=0F0K0X0
00405018	69 30 6F 30 79 30 7D 30	i0o0y0j0
00405020	83 30 87 30 8C 30 99 30	30c0i000
00405028	B8 30 BD 30 C9 30 D1 30	00c0f000
00405030	DC 30 F8 30 08 31 12 31	0°01+1
00405038	1F 31 29 30 2D 30 F0 31	1)0-0-1
00405040	0C 32 F8 31 FE 31 14 32	.2°1=1¶2
00405048	1A 32 29 32 34 32 B8 32	+2)24202
00405050	D8 32 50 33 55 33 6C 33	i2P3U3i3
00405058	71 33 B0 33 B5 33 00 34	q333A3.4
00405060	06 34 0C 34 12 34 18 34	4.44444

Vemos que existe y su contenido es 00 10 00 00 o sea al moverlo a EAX, dado que trabajamos con el contenido de una dirección de memoria se moverá al revés o sea 00 00 10 00 apretemos F7 a ver que pasa

Debug registers	
EAX	00001000
ECX	0012FFB0
EDX	7C91EB94 nt
EBX	7FFD4000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 nt
EIP	00401005 CF
C 0	ES 0023 32
P 1	CS 001B 32

Allí esta el 1000 que leyó de dicha dirección de memoria, ahora si quisiera escribir un valor en dicha dirección seria

MOV DWORD PTR DS:[400500],EAX

Reinicio el OLLYDBG y la escribo

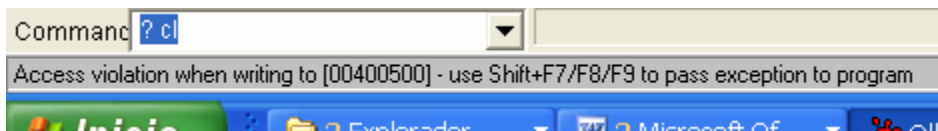
00401000	A3 00054000	MOV DWORD PTR DS:[400500],EAX
00401005	90	NOP
00401006	90	NOP
00401007	A3 CA204000	MOV DWORD PTR DS:[4020CA],EAX

En 405000 veo en el DUMP

DS:[00400500]=00000000

Address	Hex dump	ASCII
00405000	00 10 00 00 DC 00 00 00
00405008	08 30 0F 30 1F 30 33 30	0*0030
00405010	3D 30 46 30 4B 30 58 30	=0F0K0X0
00405018	69 30 6F 30 79 30 7D 30	i0o0y0j0
00405020	83 30 87 30 8C 30 99 30	30c0i000
00405028	B8 30 BD 30 C9 30 D1 30	00c0f000
00405030	DC 30 F8 30 08 31 12 31	0°01+1
00405038	1F 31 29 30 2D 30 F0 31	1)0-0-1
00405040	0C 32 F8 31 FE 31 14 32	.2°1=1¶2
00405048	1A 32 29 32 34 32 B8 32	+2)24202

Al apretar F7 oops



Me da una excepción y eso es porque la sección donde vamos a escribir no tiene permiso de escritura, lo cual impide cambiar bytes ejecutando instrucciones.

Bueno ya veremos como cambiar permisos de secciones mas adelante lo importante es que ya conocen la instrucción.

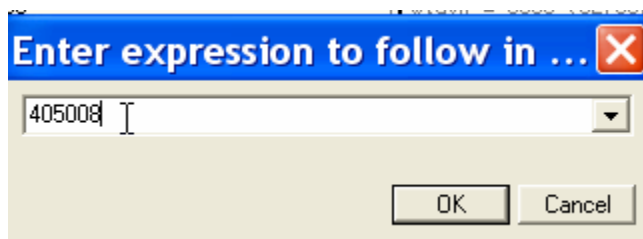
Obviamente como podemos mover 4 bytes especificando la palabra DWORD si usamos WORD moverá 2 y si usamos BYTE moverá 1 .

Veamos

MOV AX,WORD PTR DS:[405008]

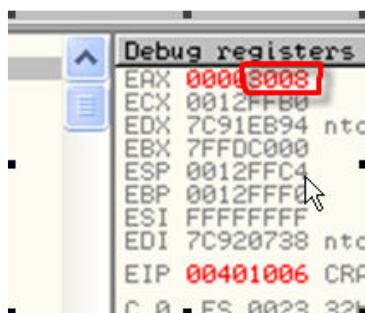
En este caso moverá dos bytes del contenido de 405000 a AX, en este caso no podemos escribir EAX ya que como son solo 2 bytes los que movemos debemos usar el registro de 16 bits AX.

Veamos que hay en el DUMP en 405008



Address	Hex dump	ASCII
00405008	08 30 0F 30 1F 30 33 30	00*00030
00405010	3D 30 46 30 48 30 58 30	=0F0K0X0
00405018	69 30 6F 30 79 30 7D 30	i0o0y0j0
00405020	83 30 87 30 8C 30 99 30	30c0i000
00405028	B8 30 BD 30 C9 30 D1 30	@0c0f000
00405030	DC 30 F8 30 08 31 12 31	-0°01+1
00405038	1F 31 29 30 2D 30 F0 31	1)0-0-1
00405040	0C 32 F8 31 FE 31 14 32	.2°11112
00405048	1A 32 29 32 34 32 B8 32	+2124202

Al apretar F7 debería mover solo esos 2 bytes a AX, veamos

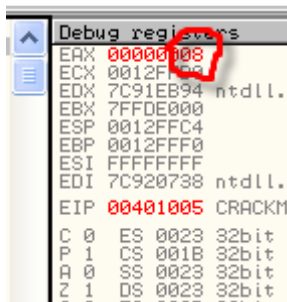


Allí esta en AX, al revés como corresponde a leer de contenidos de memoria, el resto de EAX no ha sido cambiado solo lo correspondiente a AX.

Lo mismo seria si usáramos BYTE

MOV AL, BYTE PTR DS:[405008]

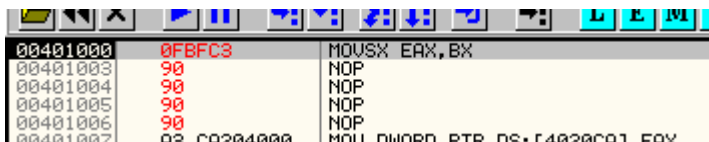
En este caso movería a AL el ultimo byte solamente o sea el 08.



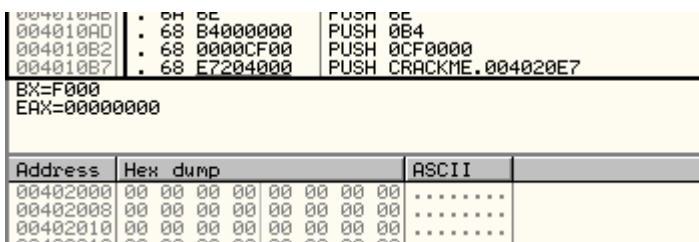
MOVSX (Move with Sign-Extension)

Copia el contenido del segundo operando, que puede ser un registro o una posición de memoria, en el primero (de doble longitud que el segundo), rellenándose los bits sobrantes por la izquierda con el valor del bit más significativo del segundo operando. Aquí tenemos un par de ejemplos:

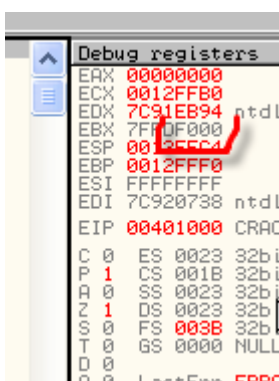
La definición la sacamos del tute de CAOS ahora veamos un ejemplo en OLLYDBG para aclarar y usemos a nuestro amigo CRUEHEAD.



Como aun no se los dije porque soy muy malo y quería que siempre buscaran a mano, los valores de los operandos jeeje, OLLYDBG tiene una ventana de aclaraciones que esta justo debajo de el listado y arriba del DUMP.



Allí vemos que la ventana de aclaraciones nos muestra el valor de los operándooos de nuestra instrucción en mi caso BX vale F000 eso lo puedo corroborar en los registros



Y allí mismo veo que EAX vale cero, así que siempre el OLLYDBG nos ayuda a interpretar los operandos de la instrucción a ejecutar. (que malo soy jeje pero quise que fijen el concepto de donde buscar cada cosa antes de la comodidad jeje)

Al apretar F7

Debug registers	
EAX	FFFF000
ECX	0012FFB0
EDX	7C91EB94 ntd
EBX	7FFDF000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntd
EIP	00401003 CRA

Vemos que se copia el BX que era F000 a AX y que se rellena con FFFF ya que el numero es un negativo de 16 bits , si hubiera sido BX 1234 entonces quedaría EAX=00001234 ya que rellena con ceros al ser BX positivo.

El tema de los positivos y negativos de 16 bits es similar a 32 bites se divide por la mitad el espacio 0000 a FFFF

de 0000 hasta 7FFF son positivos y de 7FFF a FFFF son negativos vemos que si modificamos BX a 7FFF y ponemos EAX a cero y volvemos a ejecutar la instrucción

Debug registers	
EAX	00007FFF
ECX	0012FFB0
EDX	7C91EB94 ntdll.KiF
EBX	00007FFF
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll.7C9
EIP	00401003 CRACKME.00
C 0	ES 0023 32bit 0(F
P 1	CS 001B 32bit 0(F
A 0	SS 0023 32bit 0(F

Copia 7FFF a AX pero rellena con ceros ya que 7FFF es positivo si repetimos pero con BX=8000 que es negativo,

Debug registers	
EAX	00000000
ECX	0012FFB0
EDX	7C91EB94 ntdl
EBX	00008000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdl
EIP	00401003 CRACK
C 0	ES 0023 32bit

Ejecuto nuevamente con F7 y

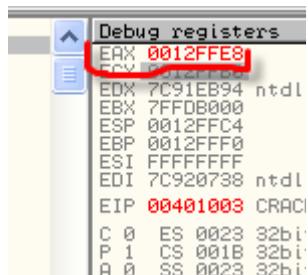
Debug registers	
EAX	FFFF8000
ECX	0012FFB0
EDX	7C91EB94 ntdl
EBX	00008000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdl
EIP	00401003 CRACK
C 0	ES 0023 32bit
P 1	CS 001B 32bit

Copia BX a AX y rellena con FFFF ya que 8000 es negativo

[MOVZX \(Move with Zero-Extend\)](#)

LEA (Load Effective Address)

Esto nos dice nuestro amigo CAOS y significa que en este caso por ejemplo reinicio OLLY.



Dicha dirección se mueve a EAX, hay que tener cuidado porque los corchetes nos llevan a pensar que deberíamos mover el contenido de la dirección 12ffe8 que debemos buscar en el dump como en el caso de la instrucción MOV, pero LEA solo mueve la dirección al primer operando no su contenido.

XCHG (Exchange Register/Memory with Register)

Esta instrucción intercambia los contenidos de los dos operandos.

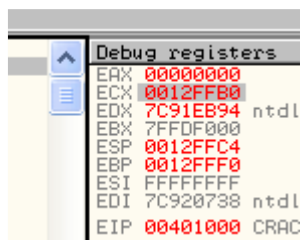
En este caso intercambia los valores si escribimos

XCHG EAX,ECX

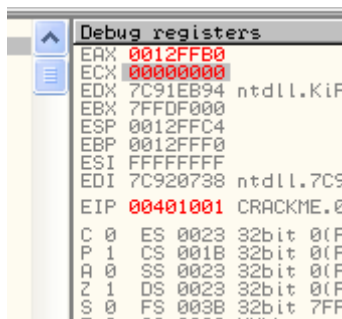
El valor de EAX pasara a ECX y viceversa comprobémoslo en OLLY



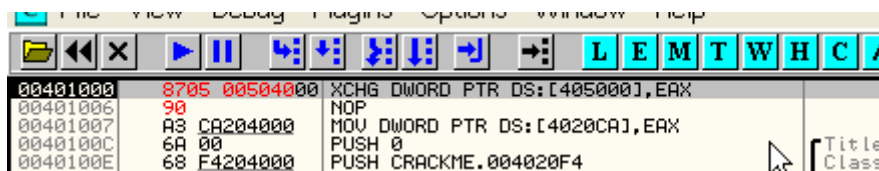
Antes de ejecutar en mi maquina EAX vale cero y ECX vale 12FFb0



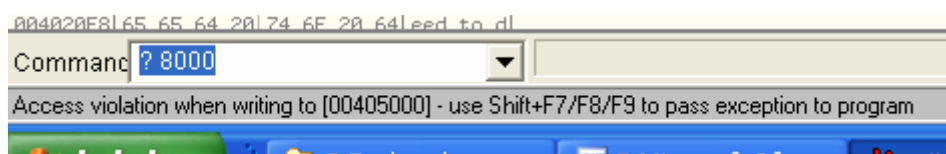
Al apretar F7 vemos que intercambian sus valores



También se puede usar para intercambiar con una posición de memoria siempre que tenga permiso de escritura dicha sección



Al ejecutar con F7



Nos pasa lo mismo que cuando quisimos hacer MOV a dicha dirección al no tener permiso de escritura nos genera una excepción.

Bueno creo que como primera parte de las instrucciones ya tienen para divertirse y practicar, creo que los ejemplos si los van haciendo mientras leen aclaran bastante la cosa, en la siguiente parte seguiremos con mas instrucciones hasta terminar con las mas importantes y tratar de terminar esto que es lo mas duro.

Hasta la parte 5

Ricardo Narvaja

13 de noviembre de 2005