

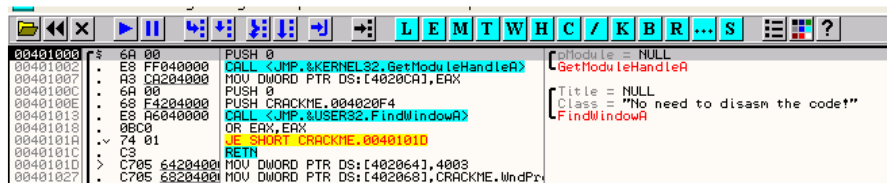
INTRODUCCIÓN AL CRACKING CON OLLYDBG PARTE 16

Comenzaremos en esta parte con los crackmes que a diferencia de los hardcoded como hasta ahora, el serial es variable y depende del nombre que ingresamos, y es calculado a partir de el.

El mecanismo para solucionarlos es muy similar, pero veremos algunos ejemplos para que quede claro.

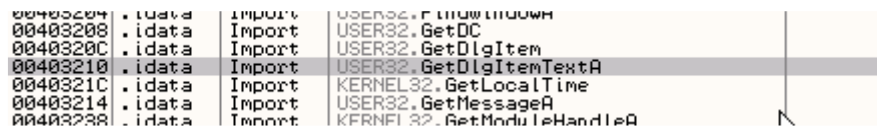
En la parte anterior los anime a que intenten hallar para su propio nombre, el serial correcto para el CRACKME DE CRUEHEAD y bueno, ese será el primero que haremos, así ven como se soluciona y si estaban en el buen camino, y si lo solucionaron pues mis mas gratas felicitaciones.

Abramos el crackme de cruehead en OLLYDBG.



Allí estamos detenidos en el ENTRY POINT

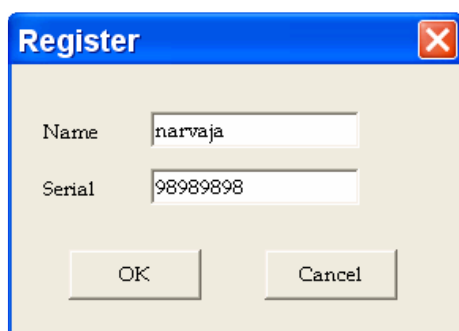
Veamos las apis que utiliza para ingresar el texto del serial que tipeamos



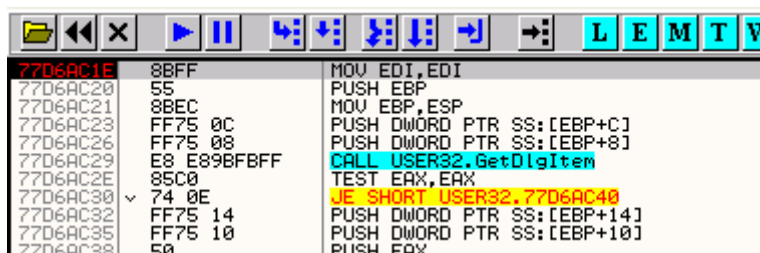
Probamos colocando un BP en dicha api a ver si se detiene cuando ingresa el nombre y serial correcto



Damos RUN



Al apretar OK para en la api y en el stack vemos los parámetros



```

0012F9D0 004012C9 CALL to GetDlgItemTextA from CRACKME.004012C4
0012F9D4 00880A70 hWnd = 00880A70 ('Register',class='#32770')
0012F9D8 000003E8 ControlID = 3E8 (1000.)
0012F9DC 0040218E Buffer = CRACKME.0040218E
0012F9E0 0000000B Count = B (11.)
0012F9E4 0012FA58
0012F9E8 00401253 CRACKME.00401253
0012F9FC 00000000

```

El buffer donde guardara el texto que ingresa esta en 40218E, veamos esa dirección en el dump, hacemos click derecho - FOLLOW IN DUMP en la línea del stack que nos informa del buffer.

Address	Hex dump	ASCII
0040218E	00 00 00 00 00 00 00 00
00402196	00 00 00 00 00 00 00 00
0040219E	00 00 00 00 00 00 00 00
004021A6	00 00 00 00 00 00 00 00
004021AE	00 00 00 00 00 00 00 00
004021B6	00 00 00 00 00 00 00 00

Allí esta, aun vacío porque no se ejecuto la api, así que hagamos DEBUG- EXECUTE TILL RETURN y como estamos en el RET de la api hago F7 para volver al programa.

Address	Hex dump	ASCII
0040218E	6E 61 72 76 61 6A 61 00	narvaja.
00402196	00 00 00 00 00 00 00 00
0040219E	00 00 00 00 00 00 00 00
004021A6	00 00 00 00 00 00 00 00

Vemos que en el buffer guardo mi nombre a partir del cual realizara operaciones y creara la clave correcta para el mismo, si quisiéramos hacer un keygen, deberíamos estudiar a partir de aquí las operaciones que realiza el programa para desde un nombre cualquiera, calcular la clave correcta, pero como este no es el caso, dejaremos que el programa genere la clave correcta y trataremos de ver si compara con el serial que tipeamos damos RUN nuevamente.

```

0012F9D0 004012E9 CALL to GetDlgItemTextA from CRACKME.004012E4
0012F9D4 00880A70 hWnd = 00880A70 ('Register',class='#32770')
0012F9D8 000003E9 ControlID = 3E9 (1001.)
0012F9DC 0040217E Buffer = CRACKME.0040217E
0012F9E0 0000000B Count = B (11.)
0012F9E4 0012FA58
0012F9E8 00401253 CRACKME.00401253
0012F9EC 00000000

```

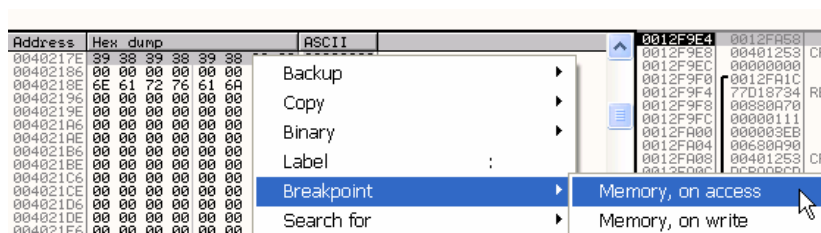
Para por segunda vez en la api, el nuevo BUFFER para la entrada del serial que tipee es 40217E, así que lo busco en el dump.

Address	Hex dump	ASCII
0040217E	00 00 00 00 00 00 00 00
00402186	00 00 00 00 00 00 00 00
0040218E	6E 61 72 76 61 6A 61 00	narvaja.
00402196	00 00 00 00 00 00 00 00
0040219E	00 00 00 00 00 00 00 00
004021A6	00 00 00 00 00 00 00 00

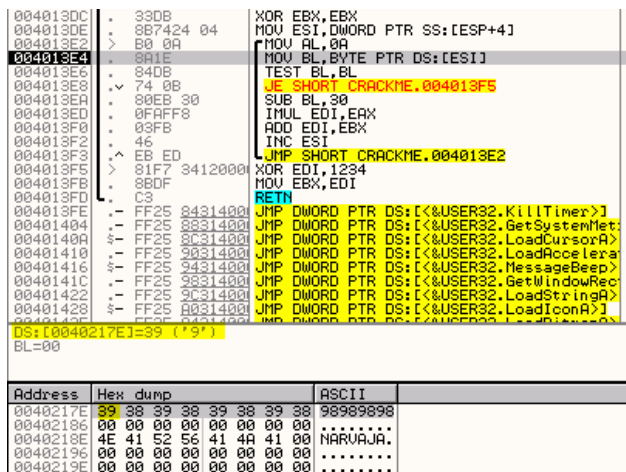
Allí esta justo arriba del otro, ejecuto la api con EXECUTE TILL RETURN y para volver al programa desde el RET apreto F7.

Address	Hex dump	ASCII
0040217E	39 38 39 38 39 38 39 38	98989898
00402186	00 00 00 00 00 00 00 00
0040218E	6E 61 72 76 61 6A 61 00	narvaja.
00402196	00 00 00 00 00 00 00 00
0040219E	00 00 00 00 00 00 00 00
004021A6	00 00 00 00 00 00 00 00

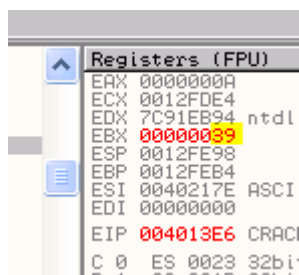
Allí ingreso nuestro serial falso, a partir de este punto el programa comparara el serial falso con el correcto que calculo a partir de mi nombre, así que podemos colocar un BPM ON ACCESS en este serial falso para ver que hace el programa con el.



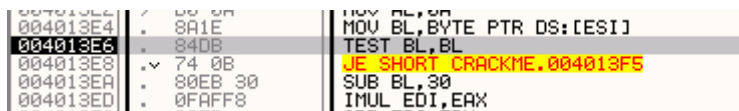
Marco los bytes y hago click derecho – BREAKPOINT –MEMORY ON ACCESS y doy RUN



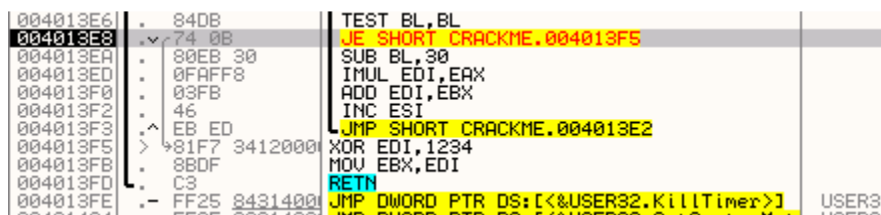
Vemos que para allí cuando el programa lee el primer byte de mi serial falso y lo mueve a BL si apreto F7.



Allí esta en BL el valor 39 de mi primer byte, debemos perseguir lo que el programa realiza con el, y en lo posible ir anotando para no perdernos si realiza operaciones matemáticas con el mismo.



Allí testea si es cero, para salir del loop al encontrar el final de la string, sigamos con f7



Como no es cero no salta y llega a SUB BL,30

Registers (FPU)	
EAX	0000000A
ECX	0012FDE4
EDX	7C91EB94 ntc
EBX	00000009
ESP	0012FE98
EBP	0012FEB4
ESI	0040217E ASC
EDI	00000000
EIP	004013ED CRF
C 0	ES 0023 32t
D 1	CS 001B 32t

Al restarle 30, queda en BL el valor numérico decimal, del primer carácter de nuestro serial falso que es 9.

En la siguiente línea multiplica EDI con EAX.

0040141C	.- FF
EAX=0000000A	
EDI=00000000	

Dichos valores estaban inicializados EAX en 0A justo al inicio de la rutina y EDI antes de entrar al loop se colocó a cero con XOR EDI,EDI

004013DC	. 33DB	XOR EBX,EBX
004013DE	. 8B7424 04	MOV ESI,DWORD PTR SS:[ESP+4]
004013E2	> B0 0A	MOV AL,0A
004013E4	. 8A1E	MOV BL,BYTE PTR DS:[ESI]
004013E6	. 84DB	TEST BL,BL
004013E8	. 74 0B	JE SHORT CRACKME.004013FF
004013EA	. 80EB 30	SUB BL,30
004013ED	. 0FAFF8	IMUL EDI,EAX
004013F0	. 03FB	ADD EDI,EBX
004013F2	. 46	INC ESI
004013F3	. EB ED	JMP SHORT CRACKME.004013E2
004013F5	> 81F7 3412000	XOR EDI,1234
004013FB	. 8BDF	MOV EBX,EDI
004013FD	. C3	RETN

004013D7	. C3	RETN
004013D8	. 33C0	XOR EAX,EAX
004013DA	. 33FF	XOR EDI,EDI
004013DC	. 33DB	XOR EBX,EBX
004013DE	. 8B7424 04	MOV ESI,DWORD PTR SS:[ESP+4]
004013E2	> B0 0A	MOV AL,0A
004013E4	. 8A1E	MOV BL,BYTE PTR DS:[ESI]
004013E6	. 84DB	TEST BL,BL
004013E8	. 74 0B	JE SHORT CRACKME.004013F5
004013EA	. 80EB 30	SUB BL,30
004013ED	. 0FAFF8	IMUL EDI,EAX
004013F0	. 03FB	ADD EDI,EBX
004013F2	. 46	INC ESI
004013F3	. EB ED	JMP SHORT CRACKME.004013E2
004013F5	> 81F7 3412000	XOR EDI,1234
004013FB	. 8BDF	MOV EBX,EDI
004013FD	. C3	RETN
004013FE	.- FF25 8431400	JMP DWORD PTR DS:[<&USER32.KillTim
00401404	.- FF25 8831400	JMP DWORD PTR DS:[<&USER32.GetSyst

Allí está de donde surgen ambos valores, al apretar f7 si recordamos la definición de IMUL con dos operandos, en dicho caso multiplica ambos considerando el signo, y guarda el resultado en el primer operando o sea EDI en este caso.

Registers (FPU)	
EAX	0000000A
ECX	0012FDE4
EDX	7C91EB94 ntc
EBX	00000009
ESP	0012FE98
EBP	0012FEB4
ESI	0040217E ASC
EDI	00000000
EIP	004013F0 CRF
C 0	ES 0023 32t
P 1	CS 001B 32t
a a	SS 0023 32t

Por eso guarda el resultado en EDI el cual queda a cero.

```

004013EA . 80EB 30      SUB BL,30
004013ED . 0FAFF8      IMUL EDI,EAX
004013F0 . 03FB        ADD EDI,EBX
004013F2 . 46          INC ESI
004013F3 ^ EB ED      JMP SHORT CRACKME.004013E2
004013F5 > 81F7 3412000 XOR EDI,1234
004013F8 . 8BDF        MOV EBX,EBX

```

Luego le suma a EDI el valor de EBX

```

Registers (FPU)
EAX 0000000A
ECX 0012FDE4
EDX 7C91EB94 ntdll.K
EBX 00000009
ESP 0012FE98
EBP 0012FEB4
ESI 0040217E ASCII "
EDI 00000009
EIP 004013F2 CRACKME
C 0 ES 0023 32bit 0
P 1 CS 001B 32bit 0

```

Quedando el EDI el valor 9, a no desesperar que esto parece difícil pero ya verán que no lo es en la próxima línea INC ESI, incrementa ESI para retornar al inicio del LOOP con el JMP y poder leer el siguiente byte de mi serial falso.

```

004013D0 . 330B        XOR EBX,EBX
004013D2 . 8B7424 04   MOV ESI,DWORD PTR SS:[ESP+4]
004013E2 > B0 0A      MOV AL,0A
004013E4 . 8A1E        MOV BL,BYTE PTR DS:[ESI]
004013E6 . 84DB        TEST BL,BL
004013E8 ^ 74 0B      JE SHORT CRACKME.004013F5
004013EA . 80EB 30      SUB BL,30
004013ED . 0FAFF8      IMUL EDI,EAX
004013F0 . 03FB        ADD EDI,EBX
004013F2 . 46          INC ESI
004013F3 ^ EB ED      JMP SHORT CRACKME.004013E2
004013F5 > 81F7 3412000 XOR EDI,1234
004013F8 . 8BDF        MOV EBX,EDI
004013FD . C3          RETN
004013FE .- FF25 8431400 JMP DWORD PTR DS:[&USER32.KillTim
00401404 .- FF25 8831400 JMP DWORD PTR DS:[&USER32.GetSyst
0040140A $- FF25 8C31400 JMP DWORD PTR DS:[&USER32.LoadCuz
00401410 $- FF25 9031400 JMP DWORD PTR DS:[&USER32.LoadAcc
00401416 $- FF25 9431400 JMP DWORD PTR DS:[&USER32.Message
0040141C $- FF25 9831400 JMP DWORD PTR DS:[&USER32.GetWinC
00401420 $- FF25 9C31400 JMP DWORD PTR DS:[&USER32.Lo

```

DS:[0040217F]=38 ('8')

BL=09 (TAB)

Address	Hex dump	ASCII
0040217F	38 09 00 00 00 00 00 00	'8' TAB

Luego de volver a colocar en AL el valor 0A, ahora lee el segundo byte de mi serial falso, ahí vemos que moverá a BL el valor 38 al igual que antes, testeara si es cero, le restara 30, y llegamos a la IMUL.

```

004013ED . 0FAFF8      IMUL EDI,EAX
004013F0 . 03FB        ADD EDI,EBX
004013F2 . 46          INC ESI
004013F3 ^ EB ED      JMP SHORT CRACKME.004013E2
004013F5 > 81F7 3412000 XOR EDI,1234
004013F8 . 8BDF        MOV EBX,EDI
004013FD . C3          RETN
004013FE .- FF25 8431400 JMP DWORD PTR DS:[&USER32.KillTim
00401404 .- FF25 8831400 JMP DWORD PTR DS:[&USER32.GetSyst
0040140A $- FF25 8C31400 JMP DWORD PTR DS:[&USER32.Loas
00401410 $- FF25 9031400 JMP DWORD PTR DS:[&USER32.Loas
00401416 $- FF25 9431400 JMP DWORD PTR DS:[&USER32.Mes
0040141C $- FF25 9831400 JMP DWORD PTR DS:[&USER32.Get
00401420 $- FF25 9C31400 JMP DWORD PTR DS:[&USER32.Lo

```

EAX=0000000A

EDI=00000009

Vemos que multiplica EDI que tiene el valor que arrastra de la pasada anterior con 0A y el resultado lo guardara en EDI veamos.

Registers (FPU)	
EAX	0000000A
ECX	0012FDE4
EDX	7C91EB94 ntd
EBX	00000008
ESP	0012FE98
EBP	0012FEB4
ESI	0040217F ASC
EDI	0000005A
EIP	004013F0 CRA
C 0	ES 0023 32b
P 0	CS 001B 32b

EDI queda en 5A en la siguiente línea le suma EBX

Registers (FPU)	
EAX	0000000A
ECX	0012FDE4
EDX	7C91EB94 ntd
EBX	00000008
ESP	0012FE98
EBP	0012FEB4
ESI	0040217F ASC
EDI	00000062
EIP	004013F2 CRA
C 0	ES 0023 32b
P 0	CS 001B 32b

O sea que va realizando operaciones y va juntando el resultado en EDI, para los que son ya asiduos al cracking ya sabrán que hace aquí, lo explicaremos.

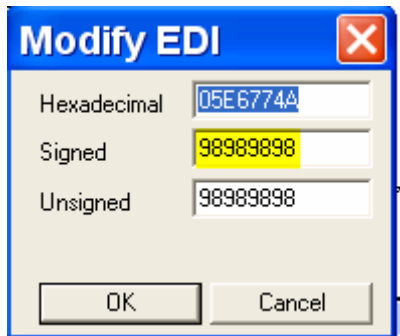
Registers (FPU)	
EAX	0000000A
ECX	0012FDE4
EDX	7C91EB94 ntd
EBX	00000008
ESP	0012FE98
EBP	0012FEB4
ESI	0040217F ASC
EDI	00000062
EIP	004013F2 CRA
C 0	ES 0023 32b
P 0	CS 001B 32b

004013D7	. C3	RETN
004013D8	. 33C0	XOR EAX,EAX
004013DA	. 33FF	XOR EDI,EDI
004013DC	. 33DB	XOR EBX,EBX
004013DE	. 8B7424 04	MOV ESI,DWORD PTR SS:[ESP+4]
004013E2	> B0 0A	MOV AL,0A
004013E4	. 8A1E	MOV BL,BYTE PTR DS:[ESI]
004013E6	. 84DB	TEST BL,BL
004013E8	. 74 0B	JE SHORT CRACKME.004013F5
004013EA	. 80EB 30	SUB BL,30
004013ED	. 0FAFF8	IMUL EDI,EAX
004013F0	. 03FB	ADD EDI,EBX
004013F2	. 46	INC ESI
004013F3	. EB ED	JMP SHORT CRACKME.004013E2
004013F5	> 31F7 3412000	XOR EDI,1234
004013FB	. 8BDF	MOV EBX,EDI
004013FD	. C3	RETN

Lo mas sencillo es poner un BP en la salida del loop apretamos F9 y cuando se detiene veo que EDI vale

Registers (FPU)	
EAX	0000000A
ECX	0012FDE4
EDX	7C91EB94 ntd
EBX	00000000
ESP	0012FE98
EBP	0012FEB4
ESI	00402186 CRA
EDI	05E6774A
EIP	004013F5 CRA
C 0	ES 0023 32b
P 1	CS 001B 32b

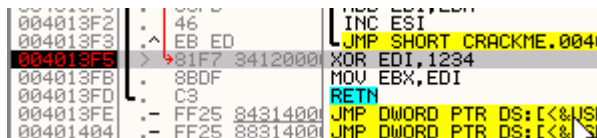
Si hago doble click en EDI



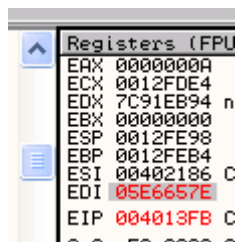
El valor de EDI es un valor hexadecimal, en la segunda línea veo su valor decimal, que corresponde a mi serial falso o sea lo que ha hecho con todo este loop (que es un loop bien conocido para los asiduos al cracking jeje ya lo ven y saben que a la salida estará el valor HEXA del serial falso.

O sea en resumidas cuentas si tipee

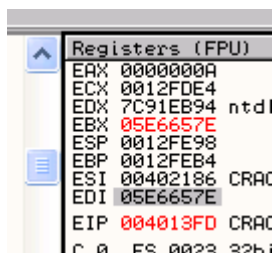
98989898 lo primero transformo esto en el número decimal 98989898 y luego lo transformo a hexa quedando 5E6774A.



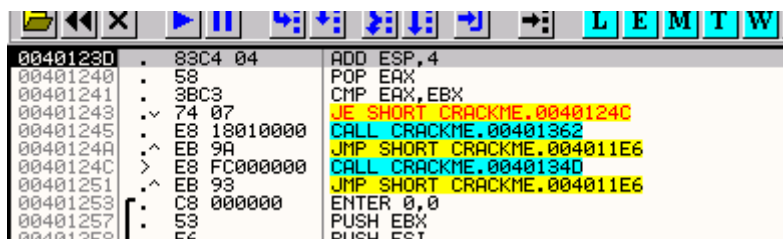
En la siguiente línea se hace EDI XOR con 1234



Queda en mi caso 5E6657E sigamos traceando, lo mueve a EBX y llega al RET.



Vemos que al volver del ret llega a la comparación EAX con EBX, lleguemos allí, ese era el salto que decidía si era correcto o no así que estamos bien,



0040123D	83C4 04	ADD ESP,4
00401240	59	POP EAX
00401241	3BC3	CMPL EAX,EBX
00401243	74 07	JL SHORT CRACKME.0040124C
00401245	E8 18010000	CALL CRACKME.00401262
0040124A	E8 9A	JMP SHORT CRACKME.004011E6
0040124C	E8 FC000000	CALL CRACKME.0040134D
00401251	E8 93	JMP SHORT CRACKME.004011E6
00401253	C8 000000	ENTER 0,0
00401257	53	PUSH EBX
00401258	56	PUSH ESI
00401259	57	PUSH EDI
0040125A	817D 0C 1001	CMPL DWORD PTR SS:[EBP+C],110
00401261	74 34	JL SHORT CRACKME.00401239
00401263	817D 0C 1101	CMPL DWORD PTR SS:[EBP+C],111
0040126A	74 35	JL SHORT CRACKME.004012A1
0040126C	837D 0C 10	CMPL DWORD PTR SS:[EBP+C],10
00401270	0F84 01000000	JL CRACKME.004011F7
00401276	817D 0C 0102	CMPL DWORD PTR SS:[EBP+C],201
0040127D	74 0C	JL SHORT CRACKME.00401288
0040127F	B8 00000000	MOV EAX,0
00401284	5F	POP EDI
00401285	5E	POP ESI
00401286	5B	POP EBX
EBX=05E6657E		
EAX=0000547B		

Al llegar vemos que compara el valor que calculó que esta en EBX con EAX que vale 547B

Como EAX es un valor determinado del programa y EBX es calculado a partir de mi serial falso el cual es incorrecto obviamente, aquí no hay igualdad.

Si EBX fuera igual a EAX el programa saltaría a CORRECTO, por lo tanto debemos analizar como llegar a que allí haya una igualdad.

Veamos nuestras anotaciones

$EBX = (\text{Valor hexadecimal del serial falso}) \text{ XOR } 1234$

Como yo quiero que EBX sea igual que EAX, en dicho caso se verificara la igualdad y estaremos en el caso correcto

Si $EAX = EBX$

Reemplazo EBX por EAX ya que son iguales

$EAX = (\text{Valor hexadecimal del serial correcto}) \text{ XOR } 1234$

Y ya no es más serial falso porque el serial que hace que EBX y EAX sean iguales es el serial correcto.

Despejo

$EAX \text{ XOR } 1234 = (\text{Valor hexadecimal del serial correcto})$

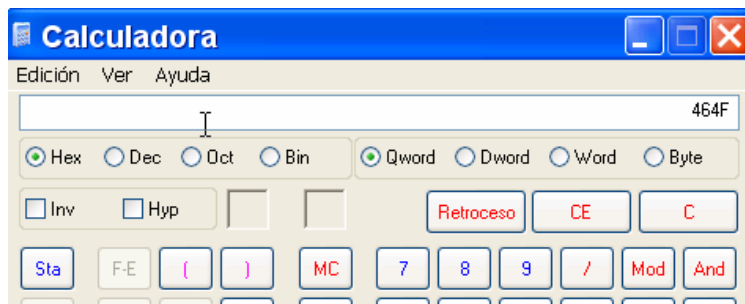
Y EAX es un dato pues lo tengo de la comparación $EAX = 547B$

$547B \text{ XOR } 1234 = (\text{Valor hexadecimal del serial correcto})$

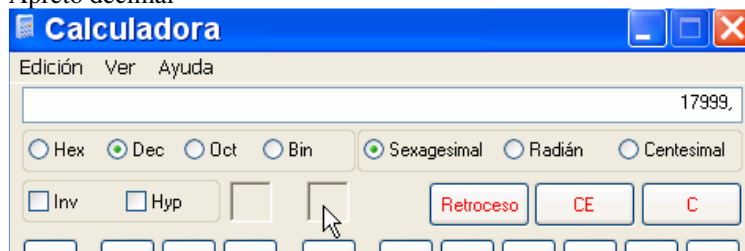
Si soluciono el XOR

$464F = (\text{Valor hexadecimal del serial correcto})$

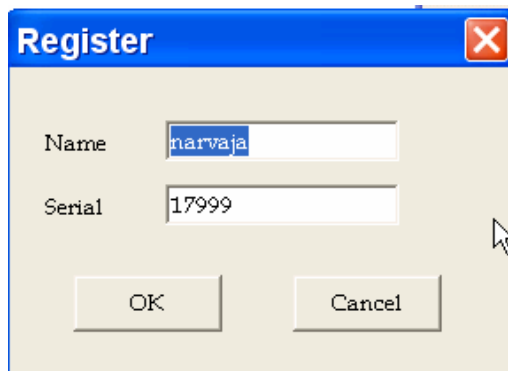
Si 464F es el valor correcto hexadecimal, si lo paso a decimal tendré mi serial para tipear



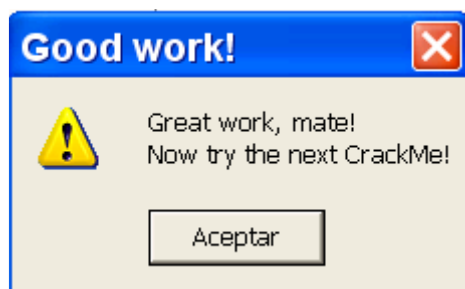
Apreto decimal



Ese es el serial correcto para narvaja quitemos todos los BPM y BPs



Apreto OK



Y obtuve el serial para mi nombre

Esta es una forma de solucionarlo, otra forma de pensarlo que se llega a la misma soluciones es esta

Si a mi serial falso le hace operaciones y lo transforma a esas operaciones las llamamos función F

$F(\text{serial falso}) = \text{EBX}$

O sea que a mi serial falso le realiza ciertas operaciones llamadas F y lo convierte en EBX el cual comparará.

El otro miembro de la comparación que es EAX sigue el mismo parámetro

F (serial verdadero)=EAX

Aplicando las mismas operaciones al serial verdadero en este caso llego al valor de EAX, por lo tanto si tengo EAX ya que es un dato de la comparación y tengo las operaciones realizadas.

Para hallar el serial verdadero deberé a EAX realizarle las operaciones opuestas para hallar el serial verdadero.

Serial verdadero= (funciones opuestas de F) EAX

O sea si fuera que a mi serial falso se le sumo un valor para llegar a la comparación, este se lo deberé restar a EAX y así siempre debo realizarle las mismas operaciones pero opuestas.

En este caso la operación opuesta de XOR es la misma XOR ya que es una operación inversible

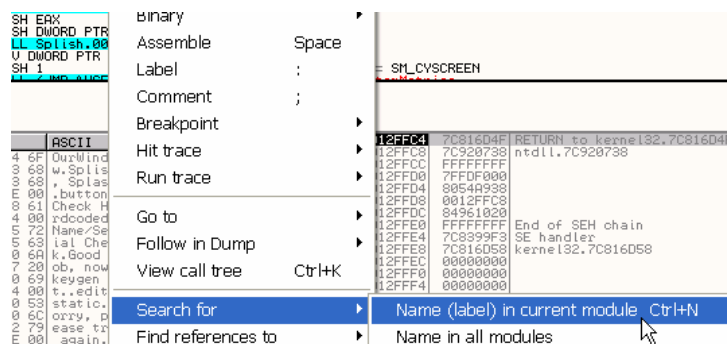
Por lo cual

XOR EAX será realizar la función opuesta de F y me dará el valor hexa de mi serial correcto, a partir del cual, pasándolo a decimal obtendré el serial verdadero.

Esto que parece mucho palabrerío no lo es tanto, de ambas formas despejando o realizando las operaciones opuestas normalmente se puede llegar al serial correcto, salvo que se use una operación que no tiene opuesta, ya veremos esos casos.

Veamos otro caso el del ya visto SPLISH pero en el modo USER-NAME ya que la parte HARDCODED ya la hemos hecho.

Arrancamos el SPLISH en OLLYDBG y estamos en el ENTRY POINT



Veamos las apis que utiliza

00402010	.rdata	Import	KERNEL32.GetModuleHandleA
00402034	.rdata	Import	USER32.GetSystemMetrics
0040200C	.rdata	Import	KERNEL32.GetTickCount
00402024	.rdata	Import	USER32.GetWindowTextA
00402020	.rdata	Import	USER32.LoadBitmapA
00402038	.rdata	Import	USER32.LoadCursorA
00402028	.rdata	Import	USER32.LoadIconA

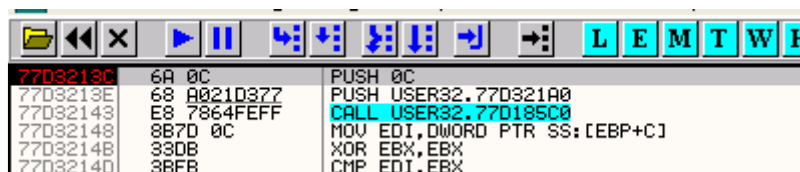
La conocida GetWindowTextA pongamos un BPX en dicha api



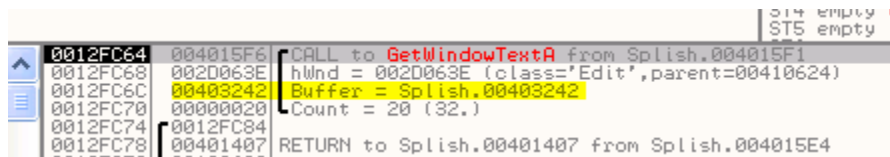
Y demos RUN con F9



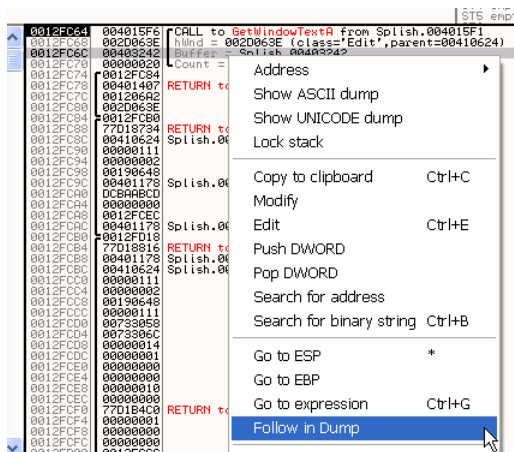
Alli tipeamos el nombre y serial falso y apretamos el boton NAME/SERIAL CHECK



Para en la api veamos el BUFFER



Buscamos el BUFFER en el DUMP

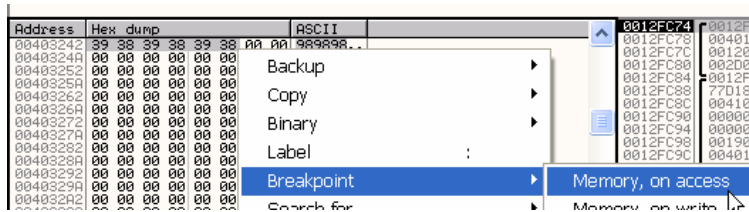


Address	Hex dump	ASCII
00403242	00 00 00 00 00 00 00 00
0040324A	00 00 00 00 00 00 00 00
00403252	00 00 00 00 00 00 00 00
0040325A	00 00 00 00 00 00 00 00
00403262	00 00 00 00 00 00 00 00
0040326A	00 00 00 00 00 00 00 00

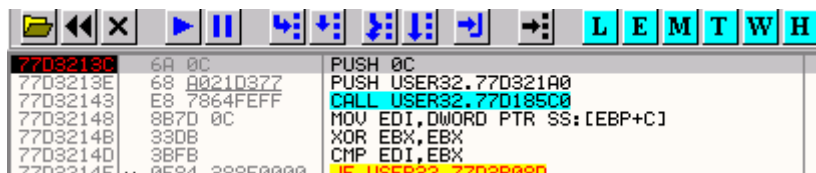
Allí está vacío aun, hagamos EXECUTE TILL RETURN para llegar al RET y luego F7 para volver al programa.

Address	Hex dump	ASCII
00403242	39 38 39 38 39 38 00 00	989898..
00403244	00 00 00 00 00 00 00 00
00403252	00 00 00 00 00 00 00 00
0040325A	00 00 00 00 00 00 00 00
00403262	00 00 00 00 00 00 00 00
0040326A	00 00 00 00 00 00 00 00

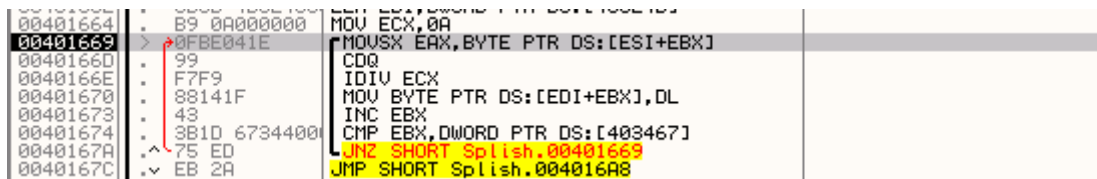
Vemos que en este caso ingreso primero el serial falso, así que marcamos el mismo y ponemos un BPM ON ACCESS.



Damos RUN

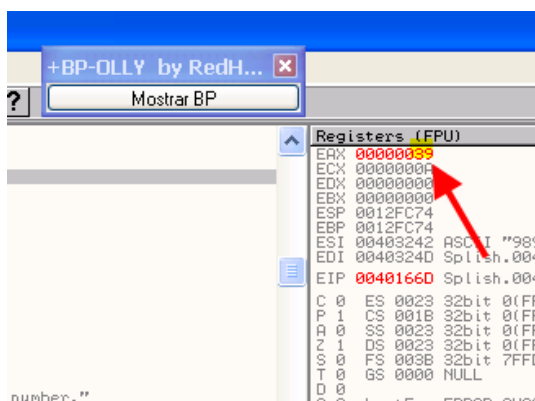


Para de nuevo en la api para ingresar el nombre, como esto por ahora no nos interesa damos RUN nuevamente.



Para en esta rutina

La primera línea mueve nuestro primer byte del serial falso si ejecutamos



Luego tenemos la siguiente instrucción CDQ, ya que no la explicamos veamos que dice nuestro amigo GOOGLE, el puede ayudar tipeamos en el buscador de GOOGLE CDQ ENSAMBLADOR.

```
cdq
idiv esi
```

La instrucción `idiv esi` divide el contenido de EDX:EAX por ESI poniendo el resultado de la división en EAX y el resto en EDX. Para poderse efectuar esta división el contenido de EDX debe ser igual a cero, lo que puede conseguirse mediante `cdq` o `xor edx, edx`. En este caso, no sería necesario poner a cero el registro EDX, pues como el resultado de la multiplicación anterior se coloca en EDX:EAX, con los valores que manejamos en este programa se pone EDX a cero.

En este caso igual no es necesario ya que EDX es cero y esta preparado para obtener el resto de la división pero como es un loop por si acaso siempre mejor antes de una IDIV usar el comando CDQ para preparar los registros EDX y EAX para la misma.

Pues dividirá EDX:EAX por ECX y el resultado va a EAX y el resto a EDX bueno este es todo el truco vemos que el byte nuestro vale 39 y lo dividirá por ECX que es 0A

Registers (FPU)	
EAX	00000039
ECX	0000000A
EDX	00000000
EBX	00000000
ESP	0012FC74
EBP	0012FC74
ESI	00403242 ASCII
EIP	0040166E Split
C 0	ES 0023 32b
P 1	CS 001B 32b

Veamos que ocurre

Registers (FPU)	
EAX	00000005
ECX	0000000A
EDX	00000007
EBX	00000000
ESP	0012FC74
EBP	0012FC74
ESI	00403242 ASCII
EIP	00401670 Split

Allí esta el resultado de la división esta en EAX y es 5 y el resto esta en EDX y es 7

0040165E	. 8D3D 4D324000	LEA EDI,DWORD PTR DS:[40324D]
00401664	. B9 0A000000	MOV ECX,0A
00401669	> 0FBE041E	MOVSX EAX,BYTE PTR DS:[ESI+EBX]
0040166D	. 99	CDQ
0040166E	. F7F9	IDIV ECX
00401670	. 8B141F	MOV BYTE PTR DS:[EDI+EBX],DL
00401673	. 43	INC EBX
00401674	. 3B1D 67344000	CMP EBX,DWORD PTR DS:[403467]
0040167A	. ^ 75 ED	JNZ SHORT Splish.00401669
0040167C	. > EB 2A	JMP SHORT Splish.004016A8
0040167E	. 6A 00	PUSH 0

Vemos que en la próxima línea guarda el resto de la división en 40324D veámoslo en el DUMP

004016A8	> 8D35 4D324000	LEA
004016AF	. 8D3D 4D324000	LEA
DL=07		
DS:[0040324D]=00		
Address	Hex	dump

Address	Hex	dump	ASCII
0040324D	00 00 00 00 00 00 00 00	
00403255	00 00 00 02 08 08 03 05		...00002
0040325D	05 03 00 00 00 00 00 00		..000000
00403265	00 00 00 00 00 00 00 00	

Allí lo guardara ejecuto con f7

Address	Hex dump	ASCII
00403240	07 00 00 00 00 00 00 00
00403255	00 00 00 02 08 08 03 05	...800*+
0040325D	05 03 00 00 00 00 00 00	+*.....
00403265	00 00 00 00 00 00 00 00
0040327D	00 00 00 00 00 00 00 00

Allí esta el 7 guardado

00401670	. 88141F	MOV BYTE PTR DS:[EDI+EBX],DL
00401673	. 43	INC EBX
00401674	. 3B1D 6734400	CMP EBX,DWORD PTR DS:[403467]
0040167A	. ^ 75 ED	JNZ SHORT Splish.00401669
0040167D	. v FR 2A	JMP SHORT Splish.004016A8

004016A8	> 8D35 4D32400
004016AF	0000 0000400
DS:[00403467]=00000006	
EBX=00000001	

Vemos que EBX que era cero es incrementado y comparado con 6 para salir del loop por ahora no son iguales por lo tanto repite.

0040165E	. 8D35 4D32400	LEA EDI,DWORD PTR DS:[403240]
00401664	. B9 0A000000	MOV ECX,0A
00401669	> 0FB8041E	MOVSX EAX, BYTE PTR DS:[ESI+EBX]
0040166D	. 99	CDQ
0040166E	. F7F9	IDIV ECX
00401670	. 88141F	MOV BYTE PTR DS:[EDI+EBX],DL
00401673	. 43	INC EBX
00401674	. 3B1D 6734400	CMP EBX,DWORD PTR DS:[403467]
0040167A	. ^ 75 ED	JNZ SHORT Splish.00401669
0040167C	. v EB 2A	JMP SHORT Splish.004016A8
0040167E	> 6A 00	PUSH 0
00401680	. 68 0A304000	PUSH Splish.0040300A

004016A8	> 8D35 4D3240
004016AF	0000 0000400
DS:[00403243]=38 ('8')	
EAX=00000005	
Jump from 0040167A	
Address	Hex dump

Allí va leer el segundo byte

Registers (FPU)	
EAX	00000038
ECX	0000000A
EDX	00000000
EBX	00000001
ESP	0012FC74
EBP	0012FC74
ESI	00403242 AS
EDI	0040324D Sp
EIP	0040166E Sp
C 1	ES 0023 32

De nuevo lo divide por 0a y guarda el resto

Registers (FPU)	
EAX	00000005
ECX	0000000A
EDX	00000006
EBX	00000001
ESP	0012FC74
EBP	0012FC74
ESI	00403242 ASCII
EDI	0040324D Splish
EIP	00401670 Splish
C 1	ES 0023 32bit
P 0	CS 001B 32bit
D 1	SS 0023 32bit

00401670	8B141F	MOV BYTE PTR DS:[EDI+EBX],DL
00401673	43	INC EBX
00401674	3B1D 67344001	CMP EBX,DWORD PTR DS:[403467]
0040167A	75 ED	JNZ SHORT Splish.00401669
0040167C	EB 2A	JMP SHORT Splish.004016A8
0040167E	6A 00	PUSH 0
00401680	68 0A304000	PUSH Splish.0040300A
00401685	68 A0304000	PUSH Splish.004030A0
0040168A	6A 00	PUSH 0
0040168C	E8 B7000000	CALL <JMP.&USER32.MessageBoxA>
00401691	EB 62	JMP SHORT Splish.004016F5
00401693	6A 00	PUSH 0
00401695	68 0A304000	PUSH Splish.0040300A
0040169A	68 B8304000	PUSH Splish.004030B8
0040169F	6A 00	PUSH 0
004016A1	E8 A2000000	CALL <JMP.&USER32.MessageBoxA>
004016A6	EB 4D	JMP SHORT Splish.004016F5
004016A8	8D35 4D324001	LEA ESI,DWORD PTR DS:[40324D]
004016AE	8D3D 58324001	LEA EDI,DWORD PTR DS:[403258]

DL=06
DS:[0040324E]=00

Address	Hex dump	ASCII
0040324D	07 00 00 00 00 00 00 00
00403255	00 00 00 02 08 08 03 05	...0000*
0040325D	05 03 00 00 00 00 00 00	*.....
00403265	00 00 00 00 00 00 00 00
0040326D	00 00 00 00 00 00 00 00
00403275	00 00 00 00 00 00 00 00

A continuación del otro byte lo guardara

Address	Hex dump	ASCII
0040324D	07 06 00 00 00 00 00 00	..+.....
00403255	00 00 00 02 08 08 03 05	...0000*
0040325D	05 03 00 00 00 00 00 00	*.....
00403265	00 00 00 00 00 00 00 00
0040326D	00 00 00 00 00 00 00 00
00403275	00 00 00 00 00 00 00 00

Bueno vemos que no hay variación realiza la misma operación en todos los bytes.

Address	Hex dump	ASCII
0040324D	07 06 07 06 07 06 00 00	..+..+..
00403255	00 00 00 02 08 08 03 05	...0000*
0040325D	05 03 00 00 00 00 00 00	*.....
00403265	00 00 00 00 00 00 00 00
00403275	00 00 00 00 00 00 00 00

Y sale del loop al no saltar el JNZ que nos retornaba al inicio del mismo.

00401673	43	INC EBX	
00401674	3B1D 67344001	CMP EBX,DWORD PTR DS:[403467]	
0040167A	75 ED	JNZ SHORT Splish.00401669	
0040167C	EB 2A	JMP SHORT Splish.004016A8	
0040167E	6A 00	PUSH 0	Style = MB_OK+MB_APPLMODAL
00401680	68 0A304000	PUSH Splish.0040300A	Title = "Splish, Splash"
00401685	68 A0304000	PUSH Splish.004030A0	Text = "Please enter your name."
0040168A	6A 00	PUSH 0	hOwner = NULL
0040168C	E8 B7000000	CALL <JMP.&USER32.MessageBoxA>	MessageBoxA
00401691	EB 62	JMP SHORT Splish.004016F5	
00401693	6A 00	PUSH 0	Style = MB_OK+MB_APPLMODAL
00401695	68 0A304000	PUSH Splish.0040300A	Title = "Splish, Splash"
0040169A	68 B8304000	PUSH Splish.004030B8	Text = "Please enter your serial number."
0040169F	6A 00	PUSH 0	hOwner = NULL
004016A1	E8 A2000000	CALL <JMP.&USER32.MessageBoxA>	MessageBoxA
004016A6	EB 4D	JMP SHORT Splish.004016F5	
004016A8	8D35 4D324001	LEA ESI,DWORD PTR DS:[40324D]	
004016AE	8D3D 58324001	LEA EDI,DWORD PTR DS:[403258]	

Llegamos allí y vemos los mensajes de correcto o incorrecto por lo cual ya sabemos que estamos cerca.

004016A1	E8 A2000000	CALL <JMP.&USER32.MessageBoxA>	MessageBoxA
004016A6	EB 4D	JMP SHORT Splish.004016F5	
004016A8	8D35 40324000	LEA ESI,DWORD PTR DS:[403240]	
004016AE	8D3D 58324000	LEA EDI,DWORD PTR DS:[403258]	
004016B4	33DB	XOR EBX,EBX	
004016B6	3B1D 63344000	CMP EBX,DWORD PTR DS:[403463]	
004016BC	74 0F	JE SHORT Splish.004016CD	
004016BE	0FB041F	MOVSX EAX,BYTE PTR DS:[EDI+EBX]	
004016C2	0FB0C1E	MOVSX ECX,BYTE PTR DS:[ESI+EBX]	
004016C6	3BC1	CMP EAX,ECX	
004016C8	75 18	JNZ SHORT Splish.004016E2	
004016CA	43	INC EBX	
004016CB	EB E9	JMP SHORT Splish.004016B6	
004016CD	6A 00	PUSH 0	
004016CF	68 0A304000	PUSH Splish.0040300A	
004016D4	68 42304000	PUSH Splish.00403042	
004016D9	6A 00	PUSH 0	
004016DB	E8 68000000	CALL <JMP.&USER32.MessageBoxA>	
004016E0	EB 13	JMP SHORT Splish.004016F5	
004016E2	6A 00	PUSH 0	
004016E4	68 0A304000	PUSH Splish.0040300A	
004016E9	68 67304000	PUSH Splish.00403067	
004016EE	6A 00	PUSH 0	
004016F0	E8 53000000	CALL <JMP.&USER32.MessageBoxA>	
004016F5	C9	LEAVE	
004016F6	C2 0800	RETN 8	

Luego hay dos LEA que mueven direcciones a ESI y EDI

Registers (FPU)	
EAX	00000005
ECX	0000000A
EDX	00000006
EBX	00000006
ESP	0012FC74
EBP	0012FC74
ESI	00403240 Sp
EDI	00403258 Sp
EIP	004016B4 Sp

ESI apunta a donde guardo los restos y EDI apunta a algo interesante jeje?

Address	Hex dump	ASCII
00403240	07 06 07 06 07 06 00 00	..*.*.*.
00403255	00 00 00 02 08 08 03 05	...0000*
0040325D	05 03 00 00 00 00 00 00	*.....
00403265	00 00 00 00 00 00 00 00
0040326D	00 00 00 00 00 00 00 00
00403275	00 00 00 00 00 00 00 00
0040327D	00 00 00 00 00 00 00 00

Sigamos

004016AE	8D3D 58324000	LEA EDI,DWORD PTR DS:[403258]	
004016B4	33DB	XOR EBX,EBX	
004016B6	3B1D 63344000	CMP EBX,DWORD PTR DS:[403463]	
004016BC	74 0F	JE SHORT Splish.004016CD	
004016BE	0FB041F	MOVSX EAX,BYTE PTR DS:[EDI+EBX]	
004016C2	0FB0C1E	MOVSX ECX,BYTE PTR DS:[ESI+EBX]	
004016C6	3BC1	CMP EAX,ECX	
004016C8	75 18	JNZ SHORT Splish.004016E2	
004016CA	43	INC EBX	
004016CB	EB E9	JMP SHORT Splish.004016B6	
004016CD	6A 00	PUSH 0	
004016CF	68 0A304000	PUSH Splish.0040300A	
004016D4	68 42304000	PUSH Splish.00403042	
004016D9	6A 00	PUSH 0	
004016DB	E8 68000000	CALL <JMP.&USER32.MessageBoxA>	
004016E0	EB 13	JMP SHORT Splish.004016F5	
004016E2	6A 00	PUSH 0	
004016E4	68 0A304000	PUSH Splish.0040300A	
004016E9	68 67304000	PUSH Splish.00403067	
004016EE	6A 00	PUSH 0	
004016F0	E8 53000000	CALL <JMP.&USER32.MessageBoxA>	
004016F5	C9	LEAVE	
004016F6	C2 0800	RETN 8	
004016F9	CC	INT3	
004016FA	FF25 70204000	JMP DWORD PTR DS:[<&USER32.BeginPa	

Vemos que EBX vale cero y lo compara con 7 y si es igual


```

004016B1 . 803D 58324001 LEH EDI,UMURD PTR DS:[403258]
004016B3 . 3D0B XOR EBX,EBX
004016B5 . 3B1D 63344001 CMP EBX,DWORD PTR DS:[403463]
004016B7 . 74 0F JE SHORT Splish.004016CD
004016B9 . 0FBEC041F MOVSW EAX,BYTE PTR DS:[EDI+EBX]
004016BB . 0FBEC0C1E MOVSW ECX,BYTE PTR DS:[ESI+EBX]
004016BD . 3BC1 CMP EAX,ECX
004016BF . 75 18 JNZ SHORT Splish.004016E2
004016C1 . 43 INC EBX
004016C3 . EB E9 JMP SHORT Splish.004016B6
004016C5 . 6A 00 PUSH 0
004016C7 . 68 0A304000 PUSH Splish.0040300A
004016C9 . 68 42304000 PUSH Splish.00403042
004016CB . 6A 00 PUSH 0
004016CD . E8 68000000 CALL <JMP.&USER32.MessageBoxA>
004016CF . EB 13 JMP SHORT Splish.004016F5

```

Salta al cartel correcto lo que pasa que hay un loop y en el medio otro JNZ de las comparaciones que si una sola falla nos llevara antes al cartel de incorrecto.

Bueno veamos que compara

A EAX moverá el primer byte que apunta EDI que es 02 y a ECX mueve el primer byte de mis restos o sea 7.

Y si vemos que si el resto del primer byte hubiera sido 02 la comparación sería exitosa.

$$39 = 5.0A + 7$$

$$39 = 5 \times 0A + 7$$

BYTE CORRECTO= 5 x 0A + 2

O sea 34 que es el numero 4 en ASCII

También podemos ver que si al realizar

$$\text{BYTE CORRECTO} = 5 \times 0A + 2$$

Al realizar $5 \times 0A + 2$ se pasa de los números posibles de una cifra (de 30 a 39) en ese caso es porque el resultado de la división no era cinco sino 4 por lo cual disminuimos y intentamos nuevamente con

$$\text{BYTE CORRECTO} = 4 \times 0A + \text{RESTO}$$

Bueno ya calculamos nuestro primer byte es 34 o sea **4** en ASCII

Vemos los otros bytes correctos que va a comparar con mis restos.

Address	Hex dump	ASCII
00403240	07 06 07 06 07 06 00 00	..+..+..
00403255	00 00 00 02 08 08 03 05	...88835
00403250	05 03 00 00 00 00 00 00	..+...
00403265	00 00 00 00 00 00 00 00
00403260	00 00 00 00 00 00 00 00
00403275	00 00 00 00 00 00 00 00

Con 02 ya calculamos

Sigamos con 08

$$\text{BYTE CORRECTO} = 5 \times 0A + 8$$

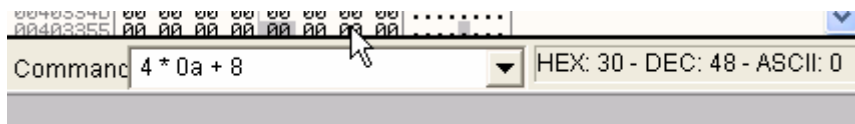
$$\text{BYTE CORRECTO} = 32 + 8$$

En este caso da 40 se pasa del máximo número decimal que es 39 (9 en ASCII) así que disminuimos a

$$\text{BYTE CORRECTO} = 4 \times 0A + 8$$

$$\text{BYTE CORRECTO} = 28 + 8 = 30 \text{ que es 0 en ASCII}$$

Si hacemos la prueba vemos que 30 dividido 0A da 4 con resto 8



Pues ya tenemos el segundo byte correcto que es 30 o sea **0** en ASCII

Si realizamos lo mismo byte a byte

El siguiente es 08, así que se repite lo mismo será 30 o sea **0** en ASCII

El siguiente es 03

$$\text{BYTE CORRECTO} = 5 \times 0A + 3$$

$$\text{BYTE CORRECTO} = 32 + 3 = 35 \text{ o sea el } \mathbf{5} \text{ en ASCII}$$

El siguiente es el 05

$$\text{BYTE CORRECTO} = 5 \times 0A + 5$$

$$\text{BYTE CORRECTO} = 32 + 5 = 37 \text{ o sea el } \mathbf{7} \text{ en ASCII}$$

El siguiente es 5 nuevamente, así que repite el 37 o sea **7** en ASCII

El ultimo es 3 que ya sabíamos era 35 o sea 5 en ASCII

Por lo cual el serial correcto para narvaja seria 4005775 quitando todos los BPM y BPX y dando RUN



Apreto el botón NAME/SERIAL CHECK



Otro adentro jeje

El crackme para practicar por ahora será uno de solo serial hardcoded de a poco iremos aumentando por supuesto con el serial correcto se activa la parte 17, así que a buscar que este es muy fácil, el crackme es el MEXCRK1.ZIP y esta adjunto con la lección.

Que tengan suerte así pueden llegar a la parte 17

Ricardo Narvaja
11 de diciembre de 2005