Ingeniería inversa de software para novatos en win32

- **Índice:**

Presentación

En primer lugar me presento, soy \0x90\ y he escrito este documento para http://www.enye-sec.org.
Bienvenidos a este tutorial de ingeniería inversa de software, espero que todo el mundo que lo lea lo entienda, no usaré demasiados tecnicismos y si los uso intentaré explicar a priori o posteriori los mismos.

Antes de nada quiero pedir disculpas por si meto la pata en algún punto y agradecería que mandaran un e-mail a 0x90@fr33project.org con sugerencias, fallos o lo que quieran.

Este documento no es más que una pequeña iniciación a este campo, espero que cuando hayan terminado de leer este documento se animen a seguir. La ingeniería inversa es una de esas cosas que nunca se acaban de aprender ya que cada día salen protecciones nuevas y vulnerabilidades de las mismas.
Sería conveniente que supieran algo de ASM (lenguaje ensamblador) y algún otro lenguaje como por ejemplo C para empezar a hacer nuestros keygens (generadores de claves, se verá más adelante que es esto) aunque solo sabiendo ASM se podría hacer, pero puede que nos lleve más trabajo o tiempo.

Este documento viene acompañado de un keygen.c que es el código fuente de un keygen de un ejemplo que veremos más adelante, un keygen.exe que es lo que hay en el keygen.c compilado y enlazado (convertir un código en un programa ejecutable), un Crackme1.exe (programa para ser crackeado) que es un programa con el que se practicará, también hay un minicrackme.c que es el código fuente de otro programa que usaremos al final del documento y su ejecutable minicrackme.exe. El resto de archivos que necesitarán los tendrán en http://www.fr33project.org o en la web de cada herramienta que citaré más adelante.

*Nota: En el tutorial usaremos el * para multiplicar y el ^ para elevar.

Un poco de historia

Siempre es bueno tener un poco de cultura, así que voy a comentar por encima como ha evolucionado Intel, ya que es el ensamblador de su microprocesador el que vamos a ver con la sintaxis de win32:

La empresa Intel fue fundada el 18 de Julio de 1968, por ingenieros que antes estaban en la Fairchild.
Sacaron su primer microprocesador en 1971, este era el 4004 y tenía una velocidad de 0,108 MHz, este microprocesador de 4 bits seguía la arquitectura hardvard.
Con el microprocesador 8080 empezaron a hacer PCs (Ordenadores Personales).
El 8086 fue el primer microprocesador de 16 bits, este microprocesador fue clave, ya que todos los microprocesadores posteriores: 80186, 80286 y el 80386 se basaron en él.

El 80386 fue el primero microprocesador con arquitectura IA32, IA32 viene de Intel Architecture 32, (32 bits). Los microprocesadores siguientes hasta el Pentium IV mantienen su compatibilidad con el 80386 y este a la vez con el 8086.

Lista de algunos microprocesadores con su año y velocidad:

En 1971 salió el 404 a 0,108 MHz.
En 1972 salió el 8008 a 0,2 MHz.
En 1974 salió el 8080 a 2 MHz.
En el periodo de 1977 a 19779 salieron los microprocesadores: 8085, 8086 y 8088 con una velocidad de 3 a 10 MHz.
En 1980 salió el 186 PC con una velocidad de 8 a 10 MHz.
En 1982 salió el microprocesador 286 PC con una velocidad de 6 a 25 MHz.
En el periodo de 1985 a 1988 salieron dos microprocesadores: 386DX y 386SX, con una velocidad de 12 a 33 MHz.
En 1989 salió el microprocesador 484-DX con una velocidad de 20 a 50 MHz.
En 1991 salieron los microprocesadores 486SX y 486SX2 con una velocidad de 16 a 66 MHz.
En 1992 salió el 486-DX2 con una velocidad de 40 a 66 MHz.
En el periodo de 1993 a 1994 salió el mircroprocesador 486-DX4 con una velocidad de 75 a 100 MHz, salió también el primer Pentium con una velocidad de 60 a 120 MHz.
En el periodo del 1994 a 1996 salió otro Pentium más potente con una velocidad de 60 a 200 MHz.
En 1995 salió el Pentium PRO con una velocidad de 150 a 200 MHz.
En 1997 salieron tes microprocesadores: el Pentium MMX para portátiles con una velocidad de 150 a 200 MHz y el Pentium MMX normal con una velocidad de 166 a 233 MHz, y salió el Pentium II, con una velocidad de 233 a 450 MHz.
En 1998 salieron cuatro microprocesadores: Pentium II Overdrive con una velocidad de 333 MHz, salió el Pentium II Celeron y el Pentium II Celeron-A con una velocidad de 233 a 450 MHz la diferencia es que el primero no tiene caché y el segundo sí, también salió el Pentium II XEON con una velocidad de 400 a 450 MHz.
En el periodo de 1999 a el 2000 salieron 6 microprocesadores.

1999:
Pentium II PE, para portátiles, con una velocidad de 266 a 400 MHz.
Pentium II Celeron-A, con una velocidad de 300 a 500 MHz.
Pentium III con una velocidad de 450 a 600 MHz.
Pentium III XEON, con una velocidad de 500 a 550 MHz.

1999 al 2000:
Pentium III E, con una velocidad de 400 a 1000 MHz.

2000:
Pentium 4, con una velocidad de 1,2 a 2 GHz.

▪ Ensamblador

Aquí explicaré una pequeña base de ensamblador en win32, no me voy a meter en estructuras de los ejecutables ni cosas del estilo. Antes de nada deben saber:

¿Qué es el ensamblador? El ensamblador (en inglés Assembly) es un lenguaje de programación que se inventó para facilitar la creación de programas, ya que en binario es bastante complicado. El ensamblador es un lenguaje de bajo nivel.

Mientras un lenguaje sea más parecido al humano es de un nivel más alto, por ejemplo:

Alto nivel: Pascal.
Medio nivel: C.
Bajo nivel: Ensamblador (ASM).
El nivel más bajo: Lenguaje Máquina (binario).

*Nota: El nivel medio no está bien definido por internet aún.

Como pueden ver el ensamblador (abreviatura: ASM), es el lenguaje de más bajo nivel que se puede aprender ya que nadie aprende binario, por lo complicado que es.
Hay que saber que en cada arquitectura de microprocesadores el ensamblador cambia, no es igual el ensamblador del Motorola 68000 que el de INTEL o AMD.

¿Qué ventajas nos ofrece el ensamblador? Antes de hablar de las ventajas hay que saber como se mide la velocidad de un ordenador:

La frecuencia de reloj: indica a la velocidad que realiza una CPU sus operaciones básicas (sumar, restar, mover valores de un registro a otro...) y se mide en hercios.
Un hercio representa una repetición de un evento por cada segundo.

Como podrán suponer un programa será más rápido mientras menos ciclos haga para hacer algo:
No es lo mismo para imprimir/mostrar en la pantalla la palabra: Hola, que se hagan 80 ciclos, que 100, obviamente el de 100 será un programa más lento (en el mismo ordenador).

Mientras un lenguaje sea de más nivel, más ciclos hará para conseguir algo que en ensamblador sería más rápido, ya que casi en ensamblador interacciona nuestro código con el microprocesador.

Pero una desventaja es que si cambiamos de arquitectura el mismo código que os servía en INTEL no vale (a no ser que sean compatibles). Los lenguajes de alto nivel suelen hacerse más portables, esto es, que un código en C lo

compilan en INTEL, lo ejecutan y funciona, hacen esto en otra arquitectura que exista un compilador de C y ese mismo código funcionará.

Entonces, ¿si ensamblador es más difícil y menos portable, aunque sea más rápido para qué es útil? Cualquier lenguaje de programación (menos los interpretados) son pasados a ensamblador, y si conocen ensamblador pueden modificar el comportamiento del mismo, sin tener el código fuente del programa, o simplemente ver lo que hace y comprender por qué (normalmente a esta clase de cosas se le llama ingeniería inversa).

*Nota:

C: Es un lenguaje de programación creado por Ken Thompson y Denis M.Ritchie en Bell 1969.
INTEL (INTegrated ELectronics): Empresa que fabrica microprocesadores.
AMD (Advanced Micro Devices): Empresa que fabrica microprocesadores.
*AMD Athlon es compatible con el microprocesador de INTEL.

¿Cómo se crea un programa ejecutable en windows? Primero se necesita un compilador, esto es un programa que transforme lo que han escrito en código objeto (un fichero con lenguaje máquina), una vez que tengan todos los códigos fuentes pasados a código objeto deben usar un enlazador, para convertirlos en un programa ejecutable.
En ensamblador se usa una herramienta llamada también ensamblador que pasa nuestro código a código objeto ejecutable directamente por el microprocesador que lo ha generado.

Ensambladores para win32: TASM, MASM32 …
Enlazadores: TLINK.

Clases de microprocesadores:

Los microprocesadores entonces según su número de instrucciones los podemos clasificar en dos clases básicas:

RISC (Reduced Instruction Set Computer): Microprocesadores con un conjunto de instrucciones reducidas: MIPS, PowerPC …

CISC (Complex Instruction Set Computer): Microprocesadores con un amplio conjunto de instrucciones: INTEL, AMD …

Un conjunto de instrucciones (en inglés ISA: Instruction Set Architecture): Son unos comandos que una CPU (Central Processing Unit) debe entender, la CPU suele estar en el microprocesador.

Sistemas de numeración:

Un sistema de numeración es un conjunto de reglas y símbolos que nos sirven para representar los números válidos en él. Nosotros vamos a ver en este apartado el decimal, binario, hexadecimal y octal.
Los sistemas mencionados anteriormente son sistemas de numeración posicionales, esto quiere decir, que el valor de un número varía según cual sea y en que posición esté por ejemplo:
1000 y 0001 no es lo mismo, aunque estén formados por las mismas cifras.
80 y 30 no es lo mismo, aunque estén formados por el mismo número de símbolos, etc.

Los sistemas de numeración posicional son los que usamos en la actualidad.

*Nota: La base es el número de cifras distintas que usa el sistema.

Sistema Decimal: (Solo voy a tratar de explicar los números enteros) Es el que usamos las personas la mayoría de veces, cuando vamos a la compra, para contar el dinero etc, en este sistema para representar los números se usa como base el número 10, esto quiere decir que tenemos las cifras: 0 1 2 3 4 5 6 7 8 9.

Así que para ir aumentando el valor de la cifra se sigue el orden del 0-9 y una vez recorrido los 10 lugares hacemos lo mismo pero partiendo del 1 y a la derecha seguimos incrementando en uno… y así sucesivamente:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
…….									
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109

Ojeando esta pequeña tabla vemos rápidamente la relación: la 1º columna si la miramos de arriba a abajo es como si contáramos del 0-9 pero con un 0 a la derecha, y dado que el 0 es el símbolo en el sistema decimal que menos valor tiene esa cifra será la más pequeña que empiece por el número indicado a la izquierda, y en la última columna si combinamos el número de la izquierda con el 9 (poniéndolo a la derecha) que es el símbolo que más vale en el sistema decimal tendremos la cifra más grande que se puede obtener con el número de la izquierda.

Así vemos que el sistema decimal (recuerden que solo hablo de los números enteros) el número de más peso (lo que es lo mismo: el más significativo o el que más vale) está a la izquierda del todo y el que menos vale (el de menor peso o menos significativo) a la derecha del todo.

Visto que los números decimales tienen de base 10, conociendo sus símbolos y sabiendo que es un sistema posicional, vamos a descomponer el número 9835:

Sabemos que 9835 es: 9000 + 800 + 30 + 5

O lo que es lo mismo:

9835 = 9 * 10^3 + 8 * 10^2 + 3 * 10^1 + 5 * 10^0 = 9835

Para que vean claro de donde salen esas potencias (ya sabíamos que la base era 10), vamos a ver que lugar ocupa cada cifra:

Posición	3	2	1	0
Número	9	8	3	5

Recordemos que el número de menor peso era el de más a la derecha.
Una forma de representarlo con álgebra:

Nm = NpT * 10^T-1 + NpT-1 * 10^T-2 + NpT-2 * 10^T-3 ... = Nm

Nm: La cifra.
N: Un número.
p: posición.
T: El total de la suma de posiciones.

Significado de las combinaciones:

NpT: El número que está en la posición más alta.
T-1: El total de posiciones - 1.

*Nota: a T le empezamos restando 1 por qué la posición 0 también se cuenta.

Último ejemplo: 702002

Sabemos que 702002 es: 700000 + 00000 + 2000 + 000 + 00 + 2

O lo que es lo mismo:

702002 = 7 * 10^5 + 0 * 10^4 + 2 * 10^3 + 0 * 10^2 + 0 * 10^1 + 2 * 10^0 = 702002

Aunque se podría simplificar así:

702002 = 7 * 10^5 + 2 * 10^3 + 2 * 10^0 = 702002

Y como sabemos que cualquier número elevado a la 0 da 1, el último número lo ponemos tal cual:

702002 = 7 * 10^5 + 2 * 10^3 + 2 = 702002

Tabla:

Posición	5	4	3	2	1	0
Número	7	0	2	0	0	2

Resumen:
Para descomponer una cifra en este sistema decimal se multiplica cada cifra por 10 ^ (elevado a) Su Posición y se indica la suma con la siguiente posición y así sucesivamente..

Sistema Hexadecimal: Este sistema es posicional además de ser un estándar en la informática, ya que la capacidad de procesamiento funciona siempre con múltiplos de 8 (8 bits, 16 bits, 32 bits, 64 bits …). Funciona como el decimal pero añadiendo 5 letras más del 0-F: 0 1 2 3 4 5 6 7 8 9 A B C D E F. Como vemos son 16 símbolos (De ahí su nombre Hexa). Solo vamos a trabajar con número hexadecimales enteros en este apartado.
Funciona exactamente como el decimal: recorremos en orden la lista de símbolos para ir aumentando el valor de cada cifra:

```
   0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
  10  11  12  13  14  15  16  17  18  19  1A  1B  1C  1D  1E  1F
  20  21  22  23  24  25  26  27  28  29  2A  2B  2C  2D  2E  2F
  30  31  32  33  34  35  36  37  38  39  3A  3B  3C  3D  3E  3F
.....
 200 201 202 203 204 205 206 207 208 209 20A 20B 20C 20D 20E 20F
```

Al igual que en el decimal, binario y octal en el hexadecimal, el números de más peso (el que más vale o el más significativo) es el que más a la izquierda esté y el menos significativo (el que menos vale o el de menor peso) a la derecha (en los enteros).

Los números hexadecimales tienen de base 16, pero como vamos a descomponerlo en hexadecimal y no en decimal pasamos el 16 decimal a hexadecimal (es 10). Vamos a descomponer el número 1D3F9:

Sabemos que 1D3F9 es: 10000 + D000 + 300 + F0 + 9

O lo que es lo mismo:

1D3F9 = 1 * 10^4 + D * 10^3 + 3 * 10^2 + F * 10^1 + 9 * 10^0 = 1D3F9

Para que vean claro de donde salen esas potencias (ya sabíamos que la base era 16, pero pasado a hexadecimal es 10), vamos a ver que lugar ocupa cada cifra:

Posición	4	3	2	1	0
Número	1	D	3	F	9

(El de más peso es el de más a la izquierda).

Una forma de representarlo con álgebra:

Nm = NpT * 10^T-1 + NpT-1 * 10^T-2 + NpT-2 * 10^T-3 ... = Nm

Nm: La cifra.
N: Un número.
p: posición.
T: El total de la suma de posiciones.

Significado de las combinaciones:

NpT: El número que está en la posición más alta.
T-1: El total de posiciones - 1.

*Nota: a T le empezamos restando 1 por qué la posición 0 también se cuenta.

Último ejemplo: F23BE9

Sabemos que F23BE9 es: F00000 + 20000 + 3000 + B00 + E0 + 9

O lo que es lo mismo:

F23BE9 = F * 10^5 + 2 * 10^4 + 3 * 10^3 + B * 10^2 + E * 10^1 + 9 * 10^0 = F23BE9

Aunque se podría simplificar así:

F23BE9 = F * 10^5 + 2 * 10^4 + 3 * 10^3 + B * 10^2 + E * 10 + 9 = F23BE9

Tabla:

Posición	5	4	3	2	1	0
Número	F	2	3	B	E	9

Resumen:
Para descomponer una cifra en este sistema hexadecimal se multiplica cada cifra por 10 ^ Su Posición y se indica la suma con la siguiente posición y así sucesivamente..
Recordar que al ser base 16 y operar en hexadecimal se pasa el 16 a hexadecimal que es 10.

Sistema Octal: Este sistema es posicional. Los símbolos de este sistema son del 0-7: 0 1 2 3 4 6 7. Como vemos son 8 símbolos (De ahí su nombre Oct). Solo vamos a trabajar con número Octales enteros en este apartado.
Funciona exactamente como el decimal: recorremos en orden la lista de símbolos para ir aumentando el valor de cada cifra:

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
.....							
440	441	442	443	444	445	446	447

Al igual que en el decimal, binario y hexadecimal en el octal, el números de más peso (el que más vale o más significativo) es el que más a la izquierda esté y el menos significativo (el de menos peso o el que menos vale) a la derecha (en los enteros).

Como vamos a descomponerlo en octal y no en decimal pasamos el 8 decimal a octal (es 10) y Vamos a descomponer el número 1D3F9:

Sabemos que es 36721: 30000 + 6000 + 700 + 20 + 1

O lo que es lo mismo:

36721 = 3 * 10^4 + 6 * 10^3 + 7 * 10^2 + 2 * 10^1 + 1 * 10^0 = 36721

Para que vean claro de donde salen esas potencias (ya sabíamos que la base era 8, pero pasado a Octal es 10), vamos a ver que lugar ocupa cada cifra:

Posición	4	3	2	1	0
Número	3	6	7	2	1

(El de más peso es el de más a la izquierda).

Una forma de representarlo con álgebra:

Nm = NpT * 10^T-1 + NpT-1 * 10^T-2 + NpT-2 * 10^T-3 ... = Nm

Nm: La cifra.
N: Un número.
p: posición.
T: El total de la suma de posiciones.

Significado de las combinaciones:

NpT: El número que está en la posición más alta.
T-1: El total de posiciones - 1.

*Nota: a T le empezamos restando 1 por qué la posición 0 también se cuenta.

Último ejemplo: 273461

Sabemos que 273461 es: 200000 + 70000 + 3000 + 400 + 60 + 1

O lo que es lo mismo:

273461 = 2 * 10^5 + 7 * 10^4 + 3 * 10^3 + 4 * 10^2 + 6 * 10^1 + 1 * 10^0 = 273461

Aunque se podría simplificar así:

273461 = 2 * 10^5 + 7 * 10^4 + 3 * 10^3 + 4 * 10^2 + 6 * 10 + 1 = 73461

Tabla:

Posición	5	4	3	2	1	0
Número	2	7	3	4	6	1

Resumen:
Para descomponer una cifra en este sistema octal se multiplica cada cifra por 10 ^ Su Posición y se indica la suma con la siguiente posición y así sucesivamente..
Recordar que al ser base 8 y operar en octal se pasa a el 8 a octal quedando 10.

Sistema binario: Hasta ahora los sistemas de numeración anteriores han sido muy parecidos al que usamos habitualmente (decimal). Pero este tiene sus peculiaridades, así que me voy a extender un poco en este sistema:

Este sistema usa base 2, por lo cual tenemos los números 0-1. Los ordenadores trabajan con el sistema binario por qué internamente trabajan con dos niveles de voltaje. Así que tenemos 1: Encendido, 2: Apagado.

Unidades: el valor más pequeño que hay en binario se le llama bit, es una cifra (un cero o un uno). Luego tenemos el siguiente al que se le llama nibble que son 4 bits, después el byte que son 8 Bits, Kilobyte 1024 bytes, Megabyte 1.048.576 bytes, Gigabyte 1.073.741.824 bytes, Terabyte 1.099.511.627.776, Petabyte 1.125.899.906.842.624 bytes, Exabyte 1.152.921.504.606.846.976 bytes, Zettabyte 1.180.591.620.717.411.303.424 bytes, Yottabyte 1.208.925.819.614.629.174.706.176 bytes y por último el Brontobyte que son: 1.237.940.039.285.380.274.899.124.224 bytes.

Los números siempre serán potencias de 2.

*Nota: Bit significa dígito binario (BInary digiT).

Para aclararnos:

1 bit: 1 bit.
1 nibble: 4 bits.
1 Byte: 8 bits.
1 Kilobyte (KB~K~K-byte): 2^10 bytes.
1 Megabyte (MB): 2^20 bytes.
1 Gigabyte (GB~GiB): 2^30 bytes.
1 Terabyte (TB): 2^40 bytes.
1 Petabyte (PB): 2^50 bytes.
1 Exabyte (EB): 2^60 bytes.
1 Zettabyte (ZB): 2^70 bytes.
1 Yottabyte (YB): 2^80 bytes.
1 Brontobyte (BB): 2^90 bytes.

*Nota: lo que hay entre paréntesis son las posible abreviaturas

1 bit: 1 bit.
1 kilobit (kbit): 2^10 bits.
1 megabit (Mbit): 2^20 bits.
1 gigabit (Gbit): 2^30 bits.
1 terabit (Tbit): 2^40 bits.

1 petabit (Pbit): 2^50 bits.
1 exabit (Ebit): 2^60 bits.
1 zettabit (Zbit): 2^70 bits.
1 yottabit (Ybit): 2^80 bits.
1 brontobit (Bbit): 2^90 bits.

Así que un bit es una cifra ya sea 0 o 1, ejemplo:

1 bit: 0 2 bits: 01
1 bit: 1 2 bits: 10

1 byte son 8 bits, ejemplo:

1 byte: 10101010 2 bytes: 0000100100111111
1 byte: 11111111 2 bytes: 1110100101010101

Este sistema funciona igual que los anteriores, el bit más significativo es el de la izquierda y el bit menos significativo el de la derecha, para hacer el número mayor se recorre en orden los símbolos 0-1 y se empieza a recorrer siguiendo la serie desde la izquierda y el de la derecha se modifica en el siguiente aumento y así sucesivamente… para verlo más claro:

0	1
10	11
100	101
110	111

Una vez entendido esto vamos a ver como se suma en binario, primero hay que saber:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10

Veamos unos ejemplo algo más difíciles:

```
  100010101010101010        111111111        101010101
+ 010101111110000011      + 001101010      + 111111111
  ___________________       _________        _________
  111000101000101101        1001101001       1101010100
```

Aquí se empieza a sumar como en el sistema decimal de derecha a izquierda, como vimos arriba 1 + 1 = 10, así que ponemos un 0 en el resultado y nos llevamos 1, esto es lo que se llama acarreo (en inglés carry), ese 1 que nos llevamos lo sumamos en la siguiente operación y en la última suma se pone todo el resultado, es lo mismo que se hace en el decimal.

Ejemplo para ver la explicación anterior:

```
  1101          *Proceso: 1 + 0 = 1, 0 + 1 = 1, 1 + 1 = 10
+ 0110                    (ponemos el 0 y sumamos el 1 al siguiente)
  ____                    1 + 0 = 1, pero le sumamos el 1 de acarreo anterior al resultado:
  10011                   1 + 1  = 10.
```

```
  0111          *Proceso: 1 + 0 = 1, 1 + 1 = 10, 1 + 0 + 1 = 10, 0 + 1 + 1 = 10
+ 1010                    Recordemos como funcionaba el acarreo, como en el decimal.
  ____
  10001
```

```
  1111          *Proceso: 1 + 1 = 10, 1 + 1 + 1 = 11, 1 + 1 + 1 = 11, 1 + 1 + 1 = 11
+ 1111                    Recordemos como funcionaba el acarreo, como en el decimal.
  ____
  11110
```

```
  1000101000    *Proceso: 0 + 1  = 1, 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 0 + 1 = 1, 1 + 0 = 1
+ 0101010101              0 + 1 = 1, 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1
  __________
  1101111101
```

(Para estos ejemplos he puesto el 0 a la izquierda del todo, pero no se suelen poner, en el decimal tampoco lo hacemos).

Vamos a ver como se resta en binario, primero hay que saber:

0 - 0 = 0
0 - 1 = 1 , y nos llevamos 1 (acarreo).
1 - 0 = 1
1 - 1 = 0

En binario se resta exactamente igual que en decimal.

Términos de la resta:

```
    M          <- Minuendo.
-   S          <- Sustraendo.
    _
    D          <- Diferencia.
```

El acarreo funciona como en el sistema decimal:

```
    22     Pasos: 2 – 7 =  5 y me llevo 1.
-   17            Se suma al sustraendo lo que nos llevábamos: 1 + 1 = 2.
    __            Se hace la resta normal tomando como minuendo el resultado de la suma anterior: 2 – 2 = 0.
    05
```

Entonces vamos a ver unos ejemplos:

1º)

```
    1110
-   0111
    ____
    0111
```

Pasos:

a)

```
    1110
-   0111      0 – 1 = 1 y nos llevamos 1.
    ____
       1
```

b)

```
    1110      Primero le sumamos a la cifra del sustraendo el 1 que nos llevábamos
-   0111       1 + 1 = 10, pero como en la suma sería 0, 1 – 0 = 1, y nos llevamos
    ____              el 1 de la suma.
      11
```

c)

```
    1110      Le sumamos a la cifra del sustraendo el 1, que nos llevábamos
-   0111      1 + 1 = 10, sería 0 y nos llevamos 1 en la siguiente operación.
    ____      1 – 0 = 1.
     111
```

d)

```
    1110      Le sumamos a la cifra del sustraendo el 1, que nos llevábamos
-   0111      0 + 1 = 1, y hacemos la resta con la cifra del minuendo – el resultado
    ____      de la suma anterior al sustraendo: 1 – 1 = 0.
    0111
```

2º)

```
    101
-   100
    ____
    001
```

Pasos:

a)

```
    101       1 – 0 = 1.
-   100
    ____
      1
```

b)

```
    101
-   100       0 – 0 = 0.
    ____
     01
```

c)

```
    101
```

- 100 1 – 1 = 0.

____

001

3º)

101111001101

- 100111100011

____

000111101010

Pasos:

a)

101111001101 1 – 1 = 0.

- 100111100011

____

0

b)

101111001101

- 100111100011 0 – 1 = 1, y me llevo 1.

____

10

c)

101111001101

- 100111100011 0 + 1 = 1, 1 – 1 = 0.

____

010

d)

101111001101

- 100111100011 1 – 0 = 1.

____

1010

e)

101111001101

- 100111100011 0 – 0 = 0.

____

01010

f)

101111001101

- 100111100011 0 – 1 = 1, me llevo 1.

____

101010

g)

101111001101

- 100111100011 1 + 1 = 0 y me llevo 1, 1 – 0 = 1.

____

1101010

h)

101111001101

- 100111100011 1 + 1 = 0 y me llevo 1, 1 – 0 = 1.

____

11101010

i)

101111001101

- 100111100011 1 + 1 = 0 y me llevo 1, 1 – 0 = 1.

____

111101010

j)

101111001101

- 100111100011 0 + 1 = 1, 1 – 1 = 0.

____

0111101010

k)

```
      101111001101
-     100111100011        0 – 0 = 0.
      ____________
       00111101010
```

l)

```
      101111001101
-     100111100011        1 – 1 = 0.
      ____________
      000111101010
```

*Nota: recuerden el resultado sería: 111101010.

Multiplicar en binario:

Se hace igual que en decimal, cualquier número multiplicado por 0 da 0, y solo dará 1 cuando este se multiplique por él mismo, vamos a ver unos ejemplos:

1)

```
      1101010111
           x 101
      __________
      1101010111
+    0000000000
     1101010111
   _____________
   1000010110011
```

2)

```
      1110
       x 0
      ____
      0000
```

3)

```
      1011
       x 1
      _____
       1011
```

4)

```
      11111101
           x 10
      ________
      00000000
+     11111101
      ________
      111111010
```

5)

```
       1111
        x 11
       ____
       1111
+      1111
       _____
      101101
```

Dividir en binario:

Voy a hacerlo como se aprende a dividir:

Términos de la división:

```
D     |   d
r     |__________
        c
```

D: dividendo.
d: divisor.
r: resto.
c: cociente/resultado.

La prueba se hace:
D = (d x c) + r

```
  575       |   25
- 50        |__________
  __         23  <- Resultado.  _
  075      x 25                  |
- 75       ___                   |
  _____    115                   |
Resto ->    0    +  46            |>  Prueba.
                    ____          |
                    575           |
                                  _|
```

*Nota: se multiplica la cifra del cociente por todo el divisor y se resta a la cifra/s del dividendo.

Vamos a ver unos ejemplos:

1º)

```
  101      |   101
- 101      |__________
  _____      1
   000  <- Resto
```

2º)

```
  1111011  |   101
- 101      |__________
  ___             11000  <- Resultado.
  0101             x 101                  _
- 101              ____                    |
  ____            11000                   |
  00000    +   00000                      |
 -  000       11000                       |
 _____     ________                       |> Prueba.
  000001     1111000                      |
- 000000   + 0000011                      |
  ______    _______                       |
   0000011   1111011  <- Dividendo.      _|
-  0000000
  ________
   0000011   <- Resto.
```

Pasos:

a)

```
  1111011  |   101
- 101      |__________
  ___             1
  0101
```

b)

```
  1111011  |   101
- 101      |__________
  ___             11
  0101
- 101
  ____
  00000
```

c)

```
  1111011  |   101
- 101      |__________
  ___             110
  0101
- 101
  ____
  00000
 -  000
 _____
  000001
```

d)

```
  1111011  |   101
- 101      |__________
  ___             1100
  0101
- 101
  ____
  00000
 -  000
 _____
  000001
- 000000
  ______
   0000011
```

e)

```
  1111011  |   101
- 101      |__________
  ___           11000  <- Resultado.
  0101
- 101
  ____
  00000
-   000
  _____
  000001
- 000000
  ______
   0000011
-  0000000
  ________
   0000011   <- Resto.
```

f)

```
  1111011 |   101
- 101     |__________
  ___           11000  <- Resultado.
  0101           x 101                  _
- 101           ____                    |
  ____          11000                   |
  00000    +   00000                    |
-   000       11000                     |
  _____      ________                   |> Prueba.
  000001      1111000                   |
- 000000   +  0000011                   |
  ______     _______                    |
   0000011    1111011  <- Dividendo.  _|
-  0000000
  ________
   0000011   <- Resto.
```

3º)

```
  10111    |   11
- 11       |__________
  ___           111  <- Resultado.     _
  0101          x 11                    |
- 11            ___                     |
  ____          111                     |
  00101    + 111                        |
-   11        ___                       |> Prueba.
  _____      10101                      |
Resto ->     00010   + 00010            |
                      _____              |
                      10111 <- Dividendo.  _|
```

Pasos:

a)

```
  10111    |   11
- 11       |__________
  ___           1
  0101
```

b)

```
  10111    |   11
- 11       |__________
  ___           11
  0101
-   11
  ____
  00101
```

c)

```
                 10111    |   11
               -  11      |_________
                  ___         111   <- Resultado.
                 0101
               -   11
                  ____
                 00101
                 -   11
                 _____
Resto ->         00010
```

d)

```
                 10111    |   11
               -  11      |_________
                  ___         111   <- Resultado.    _
                 0101         x 11                   |
               -   11          ___                   |
                  ____         111                   |
                 00101    + 111                      |
                 -   11       ___                    |> Prueba.
                 _____       10101                   |
Resto ->         00010    + 00010                    |
                             _____                   |
                            10111 <- Dividendo.     _|
```

(Recuerden que yo pongo los ceros a la izquierda del todo, pero no es necesario, lo hago por claridad para el usuario iniciado).

Cambios de base: Para cambiar de base hexadecimal, octal y binario a decimal se multiplicada cada cifra por su base elevada a su peso y se suma a la siguiente que se la ha hecho lo mismo, y así hasta terminar.
Para pasar de hexadecimal, octal y binario a decimal, se divide la cifra entre la base, y el resultado entre la base de nuevo, y cuando nos de el resultado menor a la base, cogemos el último resultado y los restos desde el último, y ese es el número, para aclararnos, vamos a ver unos ejemplos:

De binario a decimal:

11011 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 27

Simplificado:

11011 = 1 * 2^4 + 1 * 2^3 + 1 * 2 + 1 = 27

De hexadecimal a decimal:

Al tener base 16 la equivalencia es:

Decimal: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Hexadecimal: 0 1 2 3 4 5 6 7 8 9 A B C D E F

3F8A6 = 3 * 16^4 + 15 * 16^3 + 8 * 16^2 + 10 * 16^1 + 6 * 16^0 = 260.262

Simplificado:

3F8A6 = 3 * 16^4 + 15 * 16^3 + 8 * 16^2 + 10 * 16 + 6 = 260.262

De octal a decimal:

726351 = 7 * 8^5 + 2 * 8^4 + 6 * 8^3 + 3 * 8^2 + 5 * 8^1 + 1 * 8^0 = 240.873

Simplificado:

726351 = 7 * 8^5 + 2 * 8^4 + 6 * 8^3 + 3 * 8^2 + 5 * 8 + 1 = 240.873

De decimal a binario:

```
          56     |   2
          16     |_________
 6º ->     0     28
```

```
            28      |   2
            08      |___________
 5º ->       0      14


            14      |   2
 4º ->       0      |___________
                     7

            7       |   2
 3º  ->     1       |___________
                     3


            3       |   2
 2º  ->     1       |___________
                     1       <- 1º
```

56 = 111000

De decimal a octal:

```
            793       |   8
             73       |___________
 4º ->        1        99


             99       |   8
             19       |___________
 3º ->        3        12


             12       |   8
 2º ->        4        |___________
                       1     <- 1º
```

793 = 1431

De decimal a hexadecimal: (Los números que cogemos para hacer la cifra se pasa a hexadecimal, ya que estamos operando en decimal.)

```
              89352      |   16
              093        |___________
               135            5584
                072
 5º ->            08


              5584      |   16
              078       |___________
               144         349
 4º (0)->       00


              349      |   16
              029      |___________
 3º (D)->     13        21


               21      |   16
        2 ->   05      |___________
                       1       <- 1º
```

89352 = 15D08

*Nota: Hay otras formas de hacer las conversiones, he puesto la más famosas.

Con estos cambios de base serán suficientes, pueden encontrar más en algún buscador web.

Todas estas operaciones que hemos visto hasta ahora se pueden hacer con la calculadora de windows, poniéndola en modo científica, lo he enseñado por si a alguien le interesaba.

Notación: Los distintos sistemas de numeración tienen unas formas de representarse:

(N quiere decir el número).

Hexadecimal:	0xN	xN	\xN	Nh	N
Decimal:				Nd	N
Octal:				No	N
Binario:				Nb	N

*Nota: Esto son solo algunas formas de mostrar estos sistemas (hay más).

Los registros:

Están en el microprocesador, para saber que es un registro hay que saber como se divide la memoria en un ordenador. La memoria se divide en niveles: mientras más alto sea el nivel más rápido podremos acceder a lo que hay en el mismo.

¿Dónde se encuentran los registro? En el nivel más alto, así que un registro es una a la que el microprocesador puede acceder para leer, borrar, editar más rápidamente que en ningún otro sitio.

¿Para qué se usan los registros? Para almacenar datos que usarán los programas.

¿Qué finalidad tienen? Pues muy amplia: desde usarlo para guardar resultados de operaciones, guardar la siguiente instrucción a ejecutar…

Registros y usos habituales:

Datos: Todos los registros de datos son de 32 bits. Antes hay que saber algo de sus nombres, normalmente un registro de datos se reconoce por que acaba en una X, y como estamos en 32 bits una E al principio (de Extended) y en medio se pone la inicial de su función. A su vez los registros los podemos descomponer para acceder a una parte del registro en especial, para acceder a los últimos 16 bits de un registro se accede como al de 32 pero sin la E, para acceder a los primeros 8 bits (de los últimos 16 bits) se usa el nombre del registro con una H (de high) al final, y para acceder a los últimos 8 bits (de los últimos 16 bits) se pone una L (de low) al final del nombre. Pero a los primeros 16 bits del registro no hay forma de acceder independientemente, para aclarar un poco esto pasemos a verlos:

EAX: Es el Acumulador, se usa para operaciones o guardar cualquier cosa.
Descomposición del registro:
EAX 32 bits.
AX 16 bits.
AH|AL 8 bits cada uno.

Veamos un ejemplo para verlo más claro:

```
[       EAX      ]          32 bits.
       [    AX    ]          16 bits.
        [AH] [AL]
        8 bits 8 bits.
```

Imaginemos que tenemos en el registro:

```
EAX el valor: 11110010100010110011101001010111      32 bits EAX.
                          AX:     0011101001010111      16 bits últimos de EAX.
                             AH:  00111010                8 bits primeros de AX.
                              AL:           01010111      8 bits últimos de AX.
```

EBX: Es el Base, se usa para guardar datos que usará el programa.
Descomposición del registro:
EBX 32 bits.
BX 16 bits.
BH|BL 8 bits cada uno.

Veamos un ejemplo para verlo más claro:

```
[       EBX      ]          32 bits.
       [    BX    ]          16 bits.
        [BH] [BL]
        8 bits 8 bits.
```

Imaginemos que tenemos en el registro:

EBX el valor: 10101110101011111111111101010111 32 bits EBX.
BX: 1111111101010111 16 bits últimos de EBX.
BH: 11111111 8 bits primeros de BX.
BL: 01010111 8 bits últimos de BX.

ECX: Es el Contador, se usa para bucles, incrementos, decrementos automáticos…
Descomposición del registro:
ECX 32 bits.
CX 16 bits.
CH|CL 8 bits cada uno.

Veamos un ejemplo para verlo más claro:

[ECX] 32 bits.
[CX] 16 bits.
[CH] [CL]
8 bits 8 bits.

Imaginemos que tenemos en el registro:

ECX el valor: 00001011010101011011111011110100 32 bits ECX.
CX: 1011111011110100 16 bits últimos de ECX.
CH: 10111110 8 bits primeros de CX.
CL: 11110100 8 bits últimos de CX.

EDX: Es el de Datos, sirve para meter cualquier cosa.
Descomposición del registro:
EDX 32 bits.
DX 16 bits.
DH|DL 8 bits cada uno.

Veamos un ejemplo para verlo más claro:

[EDX] 32 bits.
[DX] 16 bits.
[DH] [DL]
8 bits 8 bits.

Imaginemos que tenemos en el registro:

EDX el valor: 11110101010111011011000001011101 32 bits EDX.
DX: 1011000001011101 16 bits últimos de EDX.
DH: 10110000 8 bits primeros de DX.
DL: 01011101 8 bits últimos de DX.

Segmento: Estos registros sirven para diferenciar las instrucciones de los datos, entre otras cosas:

CS: Es el de Código, en él se encuentra la dirección del segmento donde están las instrucciones del programa.
El registro CS es de 16 bits.

Veamos un ejemplo para verlo más claro:

[CS] 16 bits.

Un ejemplo de lo que podemos tener en CS: 1011111111010010

DS: Es el de Datos, en él se encuentra la dirección del segmento donde están las datos del programa.
El registro DS es de 16 bits.

Veamos un ejemplo para verlo más claro:

[DS] 16 bits.

Un ejemplo de lo que podemos tener en DS: 1000011100000001

ES: Es el registro de segmento Extra de DS, en él se encuentra la dirección del segmento donde están otros datos a los que no apunta DS.
El registro ES es de 16 bits.

Veamos un ejemplo para verlo más claro:

[ES] 16 bits.

Un ejemplo de lo que podemos tener en ES: 1110001011110010

FS y GS: Son registros de segmento extra.
Veamos un ejemplo para verlo más claro:

[FS] 16 bits.
[GS] 16 bits.

Un ejemplo de lo que podemos en FS: 1111111111001010
Un ejemplo de lo que podemos en GS: 1111111111111000

SS: Es el registro del segmento de la Pila (en ingles Stack), en él se encuentra la dirección del segmento donde está la pila (ya veremos más adelante que es esto).

El registro SS es de 16 bits.

Veamos un ejemplo para verlo más claro:

[SS] 16 bits.

Un ejemplo de lo que podemos en SS: 1000000100000001

Punteros de pila: Estos registros apuntan a distintas zonas de la pila.

EBP: Es el puntero Base, en el está la dirección de la base de la pila (donde empieza).
Descomposición del registro:
EBP 32 bits.
BP 16 bits.

Veamos un ejemplo para verlo más claro:

[EBP] 32 bits.
[BP] 16 bits.

Imaginemos que tenemos en el registro:

EBP el valor: 11110101010111011011000001111101 32 bits EBP.
BP: 1011000001111101 16 bits últimos de EBP.

ESP: Es el puntero que apunta al último dato metido en la pila.
Descomposición del registro:
ESP 32 bits.
SP 16 bits.

Veamos un ejemplo para verlo más claro:

[ESP] 32 bits.
[SP] 16 bits.

Imaginemos que tenemos en el registro:

ESP el valor: 11110101010111011011000001000101 32 bits ESP.
SP: 1011000001000101 16 bits últimos de ESP.

EIP: Es el puntero que apunta a la siguiente Instrucción que el microprocesador tiene que ejecutar.
Descomposición del registro:
EIP 32 bits.
IP 16 bits.

Veamos un ejemplo para verlo más claro:

[EIP] 32 bits.
[IP] 16 bits.

Imaginemos que tenemos en el registro:

EIP el valor: 11110101010111011011000001000101 32 bits EIP.
IP: 1011000001000101 16 bits últimos de EIP.

Índice: Estos registros se utilizan como índices para algunas instrucciones (lo veremos más detallado, más adelante):

EDI: Se usa como operando de Destino.
Descomposición del registro:
EDI 32 bits.
DI 16 bits.

Veamos un ejemplo para verlo más claro:

[EDI] 32 bits.
[DI] 16 bits.

Imaginemos que tenemos en el registro:

EDI el valor: 10010101010111011011000001000111 32 bits EDI.
DI: 1011000001000111 16 bits últimos de EDI.

ESI: Se usa como operando de origen.
Descomposición del registro:
ESI 32 bits.
SI 16 bits.

Veamos un ejemplo para verlo más claro:

[ESI] 32 bits.
[SI] 16 bits.

Imaginemos que tenemos en el registro:

ESI el valor: 11110101010111011011011101000101 32 bits ESI.
SI: 1011011101000101 16 bits últimos de ESI.

Eflags: Este es un registro un tanto especial ya que no nos importa todos los bits que estén dentro, nos importa como estén algunos: activados (1) o desactivados (0), algunos bits tienen su propio nombre y según su estado nos puede indicar desde si hay acarreo en alguna operación, si la operación da 0, etc.
Existen los flags que son los primeros 16 bits del Eflags y los Eflags con son todos (los 32):

[Eflags] 32 bits.
[flags] 16 bits.

Vamos a ver el nombre de cada bit de los Eflags y los flags y su utilidad:

*Nota: seguiremos la estructura: Nombre(Número del Bit.)

Flags de estado:

AF (4): Flag de carry auxiliar (AuxilliaryCarry Flag), este bit se pone a 1 si se produce acarreo en la artimética BCD, si no se produce acarreo se queda a 0.

CF (0): Flag de acarreo (Carry Flag), este bit se pone a 1 si en una operación se produce un acarreo, por ejemplo: cuando el resultado de una operación no cabe en el registro que estamos usando para acumular el resultado.

OF (11): Flag de overflow (Overflow Flag), este bit se pone a 1 cuando se produce un desbordamiento, imaginémonos que queremos meter algo en un registro y no cabe por completo, aquí ha ocurrido un desbordamiento y se activará el bit Overflow.

PF (2): Flag de paridad (Parity Flag), este bit se pone a 1 cuando el resultado de una operación tiene como resultado un número par de unos, por ejemplo: 1010110.

SF (7): Flag de signo (Sign Flag), este bit se pone a 1 cuando después de una operación el bit más significativo del resultado (el de más a la izquierda) es 1.

ZF (6): Flag de cero (Zero Flag), este bit se pone a 1 cuando comparamos dos valores y estos son iguales, también se pone a 1 cuando el resultado de una operación ha sido 0.

Flags de control:

DF (10): Flag de dirección de tratado de cadenas de caracteres (Direction Flag), este bit si se pone a 1 las operaciones con cadenas de caracteres se harán en sentido contrario.

IF (9): Flag de interrupción (Interrupt Flag), si este bit se pone a 1 las interrupciones estarán permitidas, las interrupciones se producen cuando llamamos a una función de la BIOS (Basic Input/Output System) o del S.O (Sistema Operativo).

TF (8): Flag de desvío (Trap Flag), si este bit se pone a 1 se podrá usar un depurador para ver el proceso de la ejecución de un programa.

He hecho una pequeña clasificación de los flags y dentro de cada una están por orden alfabético, veamos un pequeño esquema:

Bit nº:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Nombre:	-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

*Nota: donde he puesto – es que su uso no nos va a servir a nosotros y no voy a mencionarlos.

Vamos a ver un ejemplo, supongamos que tenemos en los flags: 000011101101001, veremos como quedaría:

Bit nº:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Valor:	0	0	0	0	1	1	1	0	1	1	0	1	0	0	0	1
Nombre:	-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

*Nota: faltan otros 16 bits a la izquierda, que serían la parte alta de los Eflags, y esta parte más la de los flags forman los Eflags por completo.En la arquitectura 80386 cambia el nombre del bit 12 y 13 y se añade al 14 el NT.

En los primeros 16 bits de los Eflags solo nos van a interesar 2 nuevos:

RF (16): Flag de uso para la depuración (Resume Flag), si este bit se pone 1 se ignorarán las excepciones de depuración.

VM (17): Flag del modo Virtual de 8086 (Virtual-8086 Mode), si este bit se pone a 1 cuando el 80386 está en modo protegido entra al modo virtual, manejando los segmentos como lo haría un 8086.

La memoria:

La memoria básicamente sirve para almacenar datos, hay varios tipos de memoria que conviene conocer:

RAM (Random Access Memory), memoria de acceso aleatorio: La memoria RAM es la memoria principal en un ordenador, en esta memoria se puede leer, escribir, modificar y borrar, es volátil, esto es, cuando no tiene energía eléctrica se borra, su acceso es más rápido que a un disco duro por ejemplo. Hay dos clases de memorias RAM:

> Estática: Mientras esté alimentada por la corriente eléctrica mantiene sus datos intactos.
> Dinámica: La información va desapareciendo con el tiempo, así que hay que refrescarla, esto es, rescribir lo que hay en cada sitio cada cierto tiempo.

ROM (Read Only Memory), memoria de solo lectura: La memoria ROM suele contener información del sistema, el programa de arranque del ordenador… En esta memoria la información que tiene no se puede borrar y siempre estará inalterable.

Ya sabemos los dos típicos tipos de memoria, pero claro la memoria también necesita ser transportada, por ejemplo cuando ejecutamos un programa se carga ese programa (o parte del mismo) en memoria RAM, esto se consigue gracias a los buses:

Bus de datos, se encarga de mover datos entre los distintos dispositivos del hardware de entrada y salida (E/S), por ejemplo: un teclado es un dispositivo de entrada y el monitor de salida.

Bus de direcciones: por este bus circulan las direcciones de memoria.

Bus de control: se encarga de enviar señales de control: carga, selección, lectura, escritura de memoria…

*Nota: A la unión de estos tres buses se la llama “Bus de Sistema”, no hablo del BUS PCI, USB … por qué solo quiero dar una idea básica y no extenderme mucho.

Para saber cuantos bits pueden ser enviados a la vez, se necesita saber el ancho del bus (también llamado ancho de canal).
Los buses están formados por: número de líneas y la frecuencia de trabajo del bus, se multiplican y esto es el ancho de banda, se mide en Mb/s.

El bus de datos actualmente (Pentium) tiene de ancho 64 bits, la frecuencia de reloj es directamente proporcional con la velocidad del procesador.

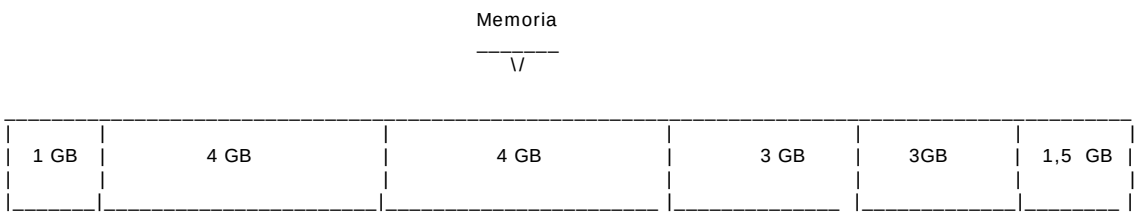
El bus de direcciones actualmente (Pentium) tiene de ancho 64 bits, esto quiere decir que la CPU podrá acceder a 2^32 posiciones de memoria (4294967296).

El bus de control actualmente (Pentium) tiene de ancho 16 bits.

*Nota: La memoria está dividida en páginas (en win32 segmentación paginada).

Como hemos visto los datos están siempre en algun tipo de memoria, ahora hay que entender como la CPU construye las direcciones de memoria.

La memoria está segmentada y paginada, en Pentium el límite de cada segmento es de 2^32 (4294967296 bits, 4 GB) como máximo, ya que 32 bits es lo máximo que podemos direccionar con un registro de 32 bits, vamos a verlo gráficamente:



Con el segmento + desplazamiento, accedemos a una posición de memoria, para conseguir esto se necesitan dos registros:

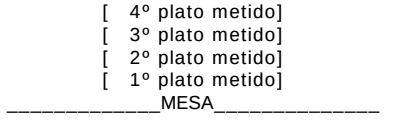
Registro_Segmento:Registro_Desplazamiento.

En el primero indicaremos el segmento (este registro es siempre de 16 bits), en el de desplazamiento lo que nos tenemos que desplazar para llegar a la posición que nos interesa (este registro es de 32 bits).

De una dirección de memoria a otra hay 1 byte de separación.

La pila:

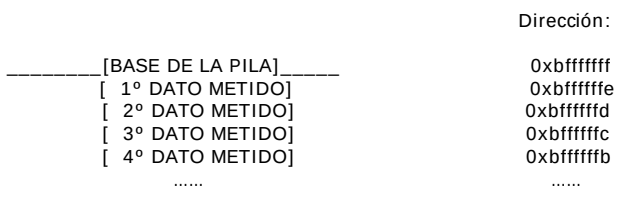
Es un bloque de memoria que usa el S.O y las distintas aplicaciones del sistema, por ejemplo cuando le pasamos argumentos a una función, se meten en la pila y se llama a la función (en bajo nivel).
La pila tiene una estructura LIFO (Last In First Out), última en entrar primera en salir, para entenderlo mejor:
Una estructura LIFO es como una pila de platos:



Como se puede ver para sacar el primer plato que metimos, antes hay que sacar los de encima, si no es muy probable que se nos caigan todos, la idea de LIFO es esta.

En el apartado instrucciones veremos que instrucciones se usan para meter o sacar algo de la pila.

La pila crece hacia abajo, esto es:



Como se puede apreciar en las direcciones, la base está en la dirección más alta y a medida que se meten elementos se va a una posición más baja.

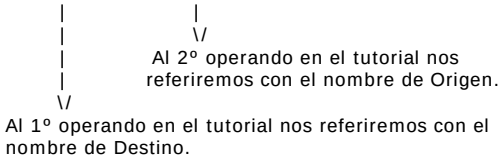
*Nota: la base de la pila no tiene por qué estar en la dirección 0xbfffffff, yo la he puesto para que os hagáis una idea.

El registro EBP siempre apunta a la dirección de la base de la pila.
El registro ESP siempre apunta al último dato introducido en la pila.

Instrucciones:

En este apartado vamos a ver algunas de las instrucciones más básicas, una instrucciones no es ni más que un comando que enviamos al microprocesador para que haga algo: sumar dos datos, restar dos datos….
Hay que saber que la sintaxis generalmente es: INSTRUCCIÓN DESTINO, ORIGEN.

Así que las instrucciones tienen la sintaxis básica : Instrucción 1º Operando, 2º Operando



Transferencia:

MOV: esta instrucción sirve para copiar origen en destino, Ejemplo:

```
MOV EAX, 3 ; Esta instrucción dará a EAX el valor 3.
```

Vamos a ver otro ejemplo:

```
MOV ECX, 2
MOV EAX, ECX
```

Estas dos instrucciones hacen: primero damos a ECX el valor 2, y luego lo copiamos a EAX, por lo cual EAX y ECX ahora valen lo mismo (2).

```
MOV BX, 23 ; Damos a BX el valor 23.

MOV EAX, 0x0040101D
MOV [EAX], 3
```

Cuidado con esto que puede llegar a engañar, lo que hacemos es: damos a EAX el valor 0040101D que es Una dirección de memoria, y al usar [EAX] copiamos el 3 a donde apunta EAX, y EAX sigue valiendo lo mismo.

```
MOV EBX, [EAX] ; Ponemos a EBX el valor que había en la dirección que apuntaba EAX.

MOV [00401111], EAX ; Ponemos el valor de EAX en la dirección de memoria 00401111.
```

No se permite usar MOV con dos direcciones de memoria:

```
MOV [EAX], [EBX] ; <- NO SE PUEDE.
MOV [0040101D], [00401111]  <- NO SE PUEDE.
```

Habría que usar un registro intermedio:

```
MOV EAX, [0040101D]
MOV [00401111], EAX ; Estas dos instrucciones: damos a EAX el valor de lo que haya en 0040101D
                      Seguidamente copiamos a la dirección 00401111 el valor de EAX.
```

Resumiendo se puede hacer:

```
MOV REGISTRO, REGISTRO
```

```
MOV REGISTRO, DATO
MOV MEMORIA, REGISTRO
MOV REGISTRO, MEMORIA
```

NUNCA: MOV MEMORIA, MEMORIA.

A veces es necesario indicar el tamaño de lo que estamos manipulando, esto se hace así:

```
MOV EAX, DWORD PTR [EBX] + 2 ; Damos a EAX un valor de 32 bits, que está en la dirección a la que
                               apunta EBX + 2, si apuntara a 0040001, cogeríamos lo que hay en 0040003.
```

Tenemos DWORD 32 bits, WORD 16 bits, BYTE 8 bits.

También valen cosas como:
MOV EAX, [EBX] – [ECX] + [EDX]

LEA: Esta instrucción es parecida a MOV, pero con LEA pasamos directamente el desplazamiento.

Vamos a ver un ejemplo:

```
LEA EBX, [EBP + 1F] ; En EBX obtendríamos la dirección que está señalada por EBP + 1F
                      en vez del dato que hay en la dirección.
```

*Nota: Por lo demás es igual que MOV.

XCHG: Esa instrucción intercambia los valores: entre dos registros, registro y memoria…
Igual que MOV no se permite un cambio entre MEMORIA y MEMORIA.

Vamos a ver un ejemplo:

```
MOV EAX, 3
MOV EBX, 19
XCHG EAX, EBX
```

Le damos a EAX el valor 3, luego a EBX el valor 19 y los intercambiamos, así que después de que el microprocesador ejecute la instrucción XCHG, EAX valdrá 19 y EBX 3.

*Nota: da igual que registro pongas primero ya que se intercambian los dos.

Resumiendo se puede hacer:

```
XCHG REGISTRO, REGISTRO
XCHG MEMORIA, REGISTRO
XCHG REGISTRO, MEMORIA
```

También se puede especificar de que tipo es el cambio:
XCHG AX, BYTE PTR BX

Último ejemplo:

```
XCHG [00300101],  EAX ; Intercambia lo que haya en la dirección especificada con EAX.
```

STC: Esta instrucción pone a 1 el CF (Flag de Carry).

CLC: Esta instrucción pone a 0 el CF (Fag de Carry).

STI: Esta instrucción pone a 1 el IF (Flag de interrupción).

CLI: Esta instrucción pone a 0 el IF (Flag de interrupción).

STD: Esta instrucción pone a 1 el DF (Flag de Dirección).

CLD: Esta instrucción pone a 0 el DF (Flag de Dirección).

CMP: compara destino con origen, es lo mismo que hacer un SUB, solo que el resultado no se almacena en ningún sitio, lo que ocurren es que se pueden modificar los flags.

Posibilidades:

```
CMP REGISTRO, REGISTRO
CMP DATO, DATO
CMP REGISTRO, DATO
CMP DATO, REGISTRO
CMP MEMORIA, REGISTRO
CMP REGISTRO, MEMORIA
CMP MEMORIA, DATO
CMP DATO, MEMORIA
```

PUSH: esta instrucción pone un valor en la pila.

Vamos a ver unos ejemplos:

```
MOV EAX, 3                          Al final de estas instrucciones la pila quedará así:
PUSH EAX        →                       BASE DE LA PILA
MOV EBX, 12                              \/
PUSH EBX        →                        3
MOV EDX, 5                               12
PUSH EDX        →                        5
```

POP: esta instrucción saca el último valor puesto en la pila, poniéndolo en el operando que especifiquemos.

Vamos a ver unos ejemplos:

```
MOV EAX, 12
MOV EBX, 15
PUSH EAX
PUSH EBX
```

La pila hasta aquí está:

```
BASE DE LA PILA
      \/
      12
      15
```

<- Cuando hagamos un POP, cogerá el primer valor que esté debajo del todo (el que hemos metido el último), sacándolo de la pila y poniéndolo donde le indiquemos.

```
POP EAX          ;EAX vale 15
POP EBX          ;EBX vale 12
```

La pila ahora está vacía.

PUSHF: esta instrucción pone en la pila los flags.

POPF: esta instrucción saca de la cima de la pila los bits que haya poniéndolos en el registro de los flags.

PUSHFD: esta instrucción pone en la pila los Eflags.

POPFD: esta instrucción saca de la cima de la pila los bits que haya poniéndolos en el registro de los Eflags.

PUSHA: esta instrucción pone en la pila los registros generales del 8086 en este orden: AX, CX, DX, BX, SP, BP, DI, SI.

POPA: esta instrucción saca de la pila en el orden inverso a PUSHA los registros generales del 8086.

PUSHAD: esta instrucción pone en la pila los registros generales de 32 bits en este orden: EAX, ECX, EDX, EBX, ESP, EBP, EDI, ESI.

POPAD: esta instrucción saca de la pila en el orden inverso a PUSHA los registros generales de 32 bits.

ROL: esta instrucción produce una rotación de bits hacia la izquierda con el operando destino que pongamos con la cantidad que pongamos en origen rotará un cierto número de veces.

Vamos a ver un pequeño esquema: Tenemos en la parte baja de AX (que es AL y puede almacenar 4 bits) por ejemplo:

```
Posición:   [3][2][1][0]
Valor:       1  0  0  1
```

Ahora hacemos un ROL AL, 1:

```
[2][1][0][3]        ; Tener en cuenta que esta son las posiciones de los bits del registro sin rotar.

 0  0  1  1         ; Esto es con el valor que quedaría el registro
```

Otro ejemplo más complicado:

```
MOV EAX, 1011111
ROL EAX, 3
```

;EAX al final valdrá:

;Proceso:

[6] [5] [4] [3] [2] [1] [0]
1 0 1 1 1 1 1

;Damos números de posiciones para moverlas a ellas y no a los números, ya que al ser ceros y unos podemos liarnos.

;Para no equivocarnos lo hacemos de una rotación en una:

1º Rotación: [5][4][3][2][1][0][6]
2º Rotación: [4][3][2][1][0][6][5]
3º Rotación: [3][2][1][0][6][5][4]

(Como cuando se jugaba a los relevos en el colegio, pues lo mismo.)

;Ahora sustituimos las posiciones de la 3º rotación:

Posiciones: [3][2][1][0][6][5][4]
Valor: 1 1 1 1 1 0 1

Así que EAX vale después del ROL: 1111101.

Posibilidades:

```
ROL REGISTRO, ROTACIONES
ROL MEMORIA, ROTACIONES
```

*Nota: podríamos indicarle un número distinto a binario para que hagan las rotaciones, por ejemplo :

```
MOV EAX, 765
ROL EAX, 4
```

Proceso: pasar el número que le hemos dado a EAX a binario, para hacer las rotaciones.

ROR: esta instrucción produce una rotación de bits hacia la derecha con el operando destino que pongamos y la cantidad que pongamos en origen rotará.

Vamos a ver un pequeño esquema: Tenemos en la parte baja de AX (que es AL y puede almacenar 4 bits) por Ejemplo:

Posición: [3][2][1][0]
Valor: 1 0 0 1

Ahora hacemos un ROR AL, 1:

[0][3][2][1] ; Tener en cuenta que esta son las posiciones de los bits del registro sin rotar.

1 1 0 0 ; Esto es con el valor que quedaría el registro

Otro ejemplo más complicado:

```
MOV EAX, 1011111
ROR EAX, 3
```

;EAX al final valdrá:

;Proceso:

[6] [5] [4] [3] [2] [1] [0]
1 0 1 1 1 1 1

;Damos números de posiciones para moverlas a ellas y no a los números, ya que al ser ceros y unos podemos liarnos.

;Para no equivocarnos lo hacemos de una rotación en una:

1º Rotación: [0][6][5][4][3][2][1]
2º Rotación: [1][0][6][5][4][3][2]
3º Rotación: [2][1][0][6][5][4][3]

(Como cuando se jugaba a los relevos en el colegio, pues lo mismo.)

;Ahora sustituimos las posiciones de la 3º rotación:

Posiciones: [2][1][0][6][5][4][3]
Valor: 1 1 1 1 0 1 1

Así que EAX vale después del ROR: 1111011.

Posibilidades:

```
ROR REGISTRO, ROTACIONES
ROR MEMORIA, ROTACIONES
```

*Nota: podríamos indicarle un número distinto a binario para que hagan las rotaciones, por ejemplo :

```
MOV EAX, 765
ROR EAX, 4
```

Proceso: pasar el número que le hemos dado a EAX a binario, para hacer las rotaciones.

RCL: esta instrucción produce una rotación hacia la izquierda un tanto especial, ya que se tiene en cuenta el CF (flag de carry), también tiene en cuenta dos operando el destino que es lo que queremos indicar para que haga el RCL y el número de rotaciones en origen.

Vamos a ver como funciona:

```
Posición:        [CF]    [3][2][1][0]
Valor:            1       1  0  0  0
```

Como vemos el flag de carry le hemos dado valor 1, podríamos darle cualquier otro, vamos a ver que pasaría en un:
MOV EAX, 1000
RCL EAX, 1:

```
___ [CF]  <- [3][2][1][0]
|    1       0  0  0  1
|_____________________/\
```

;Es bastante simple en vez de pasar a la parte de atrás pasa de izquierda a derecha los valores por el CF antes de llegar a la otra posición.

Vamos a ver otro ejemplo (teniendo el CF a 1):

```
MOV EAX, 100001
RCL EAX, 3
```

```
;Vamos por pasos:

Posición: [CF]   [5][4][3][2][1][0]
Valor:     1      1  0  0  0  0  1


;Primer paso:   [5]    [4][3][2][1][0][CF]
                 1      0  0  0  0  1    1

;Segundo paso:  [4]    [3][2][1][0][CF][5]
                 0      0  0  0  1    1   1



;Tercer paso:     [3]    [2][1][0][CF][5][4]
                  0      0  0  1    1   1  0
```

Así que el resultado de EAX es 001110, ya que el valor que quede alejado a la izquierda (donde estaba en un comienzo el CF no entra en el resultado), es decir solo entran las que estén juntas.

Posibilidades:

```
RCL REGISTRO, ROTACIONES
RCL MEMORIA, ROTACIONES
```

*Nota: podríamos indicarle un número distinto a binario para que hagan las rotaciones, por ejemplo :

```
MOV EAX, 765
RCL EAX, 4
```

Proceso: pasar el número que le hemos dado a EAX a binario, para hacer las rotaciones.

RCR: esta instrucción produce una rotación hacia la derecha un tanto especial, ya que se tiene en cuenta el CF (flag de carry), también tiene en cuenta dos operando el destino que es lo que queremos indicar para que haga el RCR y el número de rotaciones en origen.

Vamos a ver como funciona:

```
Posición:        [CF]    [3][2][1][0]
Valor:            1       1  0  0  0
```

Como vemos el flag de carry le hemos dado valor 1, podríamos darle cualquier otro, vamos a ver que pasaría en un:
MOV EAX, 1000
RCR EAX, 1:

```
|-> [CF]  -> [3][2][1][0]
|    0       1  0  0  0
|_____________________|
```

;Es bastante simple el valor que hay en el CF pasa a la primera posición, Y entra al CF el valor del bit que estaba en último lugar.

Vamos a ver otro ejemplo (teniendo el CF a 1):

```
MOV EAX, 100001
RCR EAX, 3

                ;Vamos por pasos:

Posición: [CF]  [5][4][3][2][1][0]
Valor:     1     1  0  0  0  0  1


                ;Primer paso:  [0]   [CF][5][4][3][2][1][0]
                                1     1  1  0  0  0  0  0

               ;Segundo paso:  [1]   [0][CF][5][4][3][2]
                                0     1  1  1  0  0  0

                ;Tercer paso:  [2]   [1][0][CF][5][4][3]
                                0     0  1  1  1  0  0
```

Así que el resultado de EAX es 011100, ya que el valor que quede alejado a la izquierda (donde estaba en un comienzo el CF no entra en el resultado), es decir solo entran las que estén juntas.

Posibilidades:

```
RCR REGISTRO, ROTACIONES
RCR MEMORIA, ROTACIONES
```

*Nota: podríamos indicarle un número distinto a binario para que hagan las rotaciones, por ejemplo :

```
MOV EAX, 765
RCR EAX, 4
```

Proceso: pasar el número que le hemos dado a EAX a binario, para hacer las rotaciones.

SAL: Es lo mismo que SHL (SHL está explicado más abajo).

SAR: esta instrucción produce un desplazamiento de bits, pero mantiene el signo, recibe un destino y un origen que puede ser 1 o si es más CL.

Vamos a ver un ejemplo:

```
Posición:    [4]  [3][2][1][0]   [BASURA]
Valor:        1    0  0  0  1
```

Vamos a ver que pasa en un:

```
MOV EAX, 10001
SAR EAX, 1

[4]  [n][3][2][1]   [0] ->   [BASURA]
 1    1  0  0  0
```

Como vemos el bit más significativo siempre se queda igual, el de más a la derecha se va a la basura y entra un bit nuevo por la izquierda con valor del bit más significativo en este caso 1.

Vamos a ver un ejemplo más complicado:

```
MOV EAX, 0111001
MOV CL, 3
SAR EAX, CL

[6]  [5][4][3][2][1][0]     [BASURA]
 0    1  1  1  0  0  1


                Vamos a ver ahora los pasos:

                1º paso:  [6]     [n][5][4][3][2][1]     [BASURA]
                           0       0  1  1  1  0  0       [0]

                2º paso:  [6]     [n2][n][5][4][3][2]    [BASURA]
                           0       0   0  1  1  1  0      [1][0]


                3º paso:  [6]     [n3][n2][n][5][4][3]   [BASURA]
                           0       0   0   0  1  1  1     [2][1][0]

                        Como podemos ver al final EAX vale 0000111.
```

*Nota: esto del CL puede que no sea siempre necesario y se puede pasar el valor directamente.

Posibilidades:

```
SAR MEMORIA, 1
SAR REGISTRO, 1
SAR MEMORIA, CL
```

SAL REGISTRO, CL

Operaciones aritméticas:

INC: Esta instrucción incrementa en una unidad el destino que le pasemos.

Vamos a ver un ejemplo:

```
MOV EAX, 2
INC EAX        ;Al hacer esto EAX valdrá 3.
```

Se puede aumentar en 1 lo que haya en una dirección de memoria apuntada por un registro:
INC, [BX+1] ; Le suma 1 a lo que haya en la siguiente dirección a la que apunta BX.

O directamente a una dirección:

INC WORD PTR [738492]

DEC: Esta instrucción decrementa en una unidad el destino que le pasemos.

Vamos a ver un ejemplo:

```
MOV EAX, 2
DEC EAX        ;Al hacer esto EAX valdrá 1.
```

Se puede decrementar en 1 lo que haya en una dirección de memoria apuntada por un registro:
DEC, [BX] ; Le resta 1 a lo que haya en la dirección que apunta BX.

O directamente a una dirección:

DEC WORD PTR [738492]

ADD: Suma los dos valores que le pasemos, guardando el resultado en destino.

Vamos a ver un ejemplo:

```
MOV EAX, 4
MOV EBX, 9
ADD EAX, EBX    ; Ahora EAX vale 13 y EBX se queda en 9.
```

Podemos hacerlo con registros que apunten a direcciones de memoria:
ADD [EBX], 12 ; Suma 12 al contenido de la dirección a la que apunte EBX.

O directamente a la dirección:
ADD [12345], WORD PTR EAX ; Suma a la dirección 12345 el valor que haya en EAX.

*Nota: Nunca de dirección de memoria a dirección de memoria.

SUB: Resta destino menos el origen, guardando el resultado en destino.

Vamos a ver un ejemplo:

```
MOV EAX, 13
MOV EBX, 10
SUB EAX, EBX    ; Ahora EAX vale 3, y EBX se queda en 10.
```

Podemos hacerlo con registros que apunten a direcciones de memoria:
SUB [EBX], 12 ; Resta el valor de la dirección que apunte [EBX] menos 12, y guarda el resultado en la dirección de memoria.

O directamente a la dirección:
SUB [12345], BYTE PTR EAX

*Nota: Nunca de dirección de memoria a dirección de memoria.

ADC: Suma los dos valores que le pasemos, guardando el resultado en destino, pero tiene en cuenta el acarreo.

Vamos a ver un ejemplo:

```
MOV EAX, 4
MOV EBX, 9
ADC EAX, EBX    ; Ahora EAX vale 13 y EBX se queda en 9.
```

Podemos hacerlo con registros que apunten a direcciones de memoria:
ADC [EBX], 12 ; Suma 12 al contenido de la dirección a la que apunte EBX.

O directamente a la dirección:
ADC [12345], WORD PTR EAX ; Suma a la dirección 12345 el valor que haya en EAX.

*Nota: Nunca de dirección de memoria a dirección de memoria.

SBB: Resta destino menos el origen, guardando el resultado en destino y teniendo en cuenta el acarreo.

Vamos a ver un ejemplo:

```
MOV EAX, 13
MOV EBX, 10
SBB EAX, EBX    ; Ahora EAX vale 3, y EBX se queda en 10.
```

Podemos hacerlo con registros que apunten a direcciones de memoria:

```
SBB [EBX], 12 ; Resta el valor de la dirección que apunte [EBX] menos 12, y guarda el resultado en la
                dirección de memoria.
```

O directamente a la dirección:

```
SBB [12345], BYTE PTR EAX
```

*Nota: Nunca de dirección de memoria a dirección de memoria, se usan muy poco esta instrucción.

MUL: Multiplica el valor que le pasemos con el valor que haya en EAX, poniendo el resultado en EDX (la parte más alta) y EAX (la parte menos significativa).

Vamos a ver un ejemplo:

```
MOV EBX, 12
MOV EAX, 8
MUL EBX             ;Guardamos el resultado de la operación 8 por 12 en los registros EDX y EAX.
```

Podemos hacerlo con registros que apunten a direcciones de memoria:

```
MOV EAX, 2
MUL [EBX]           ; Multiplica 2 por el resultado que haya en la dirección a la que apunta EBX.
```

O directamente a la dirección:

```
MOV EBX, 93
MUL WORD PTR [198373634]    ; Multiplica 93 por el valor que haya en la dirección 198373634.
```

DIV: Divide el valor de EAX entre el que le pasemos, poniendo el resultado en EAX o si no cabe, se guarda en los registros EDX y EAX.

Reglas:

1) Si el valor que le pasamos a la instrucción es de 8 bits el dividendo debe ser de 16 bits y estará en AX, dejando en AL el cociente/resultado y en AH el resto.

2) Si el valor que le pasamos a la instrucción es de 16 bits el dividendo debe ser de 32 bits y estará en DX (la parte más significativa del dividendo) y AX (la parte menos significativa del dividendo), dejando en AX el cociente/resultado y en DX el resto.

3) Si el valor que le pasamos a la instrucción es de 32 bits el dividendo debe ser de 64 bits y estará en EAX (la parte menos significativa del dividendo) y EDX (la parte más significativa del dividendo), dejando en EAX el cociente/resultado y en EDX el resto.

Vamos a ver un ejemplo:

```
MOV BL, 5
MOV AX, 10
DIV EBX             ;Guardamos el resultado de la operación 10:5 en AL, dejando el
                    Resto en AH.
```

Podemos hacerlo con registros que apunten a direcciones de memoria:

```
MOV EAX, 234
DIV [EBX]           ; Divide 234: el resultado que haya en la dirección a la que apunta EBX.
```

O directamente a la dirección:

```
MOV EBX, 93
DIV WORD PTR [198373634]    ; Divide 93: el valor que haya en la dirección 198373634.
```

IMUL: Multiplica el valor que le pasemos con el valor que haya en EAX, poniendo el resultado en EDX (la parte más alta) y EAX (la parte menos significativa). Es lo mismo que MUL solo que se tiene en cuenta que trabajamos con números con signo.

Vamos a ver un ejemplo:

```
MOV EBX, 12
MOV EAX, 8
IMUL EBX            ;Guardamos el resultado de la operación 8 por 12 en los registros EDX y EAX.
```

Podemos hacerlo con registros que apunten a direcciones de memoria:

```
MOV EAX, 2
IMUL [EBX]          ; Multiplica 2 por lo  que haya en la dirección a la que apunta EBX.
```

O directamente a la dirección:

```
MOV EBX, 93
```

```
IMUL WORD PTR [198373634]     ; Multiplica 93 por el valor que haya en la dirección 198373634.
```

IDIV: Divide el valor de EAX entre el que le pasemos, poniendo el resultado en EAX o si no cabe en los registros EAX y EDX, en AH quedará el resto y en EAX o EAX Y EDX el cociente, es lo mismo que DIV solo que se tiene en cuenta que trabajamos con números con signo.

Reglas:

1) Si el valor que le pasamos a la instrucción es de 8 bits el dividendo debe ser de 16 bits y estará en AX, dejando en AL el cociente/resultado y en AH el resto.

2) Si el valor que le pasamos a la instrucción es de 16 bits el dividendo debe ser de 32 bits y estará en DX (la parte más significativa del dividendo) y AX (la parte menos significativa del dividendo), dejando en AX el cociente/resultado y en DX el resto.

3) Si el valor que le pasamos a la instrucción es de 32 bits el dividendo debe ser de 64 bits y estará en EAX (la parte menos significativa del dividendo) y EDX (la parte más significativa del dividendo), dejando en EAX el cociente/resultado y en EDX el resto.

Vamos a ver un ejemplo:

```
MOV BL, 5
MOV AX, 10
DIV EBX             ;Guardamos el resultado de la operación 10:5 en AL, dejando el
                    Resto en AH.
```

Podemos hacerlo con registros que apunten a direcciones de memoria:

```
MOV EAX, 234
IDIV [EBX]          ; Divide 234: el resultado que haya en la dirección a la que apunta EBX.
```

O directamente a la dirección:

```
MOV EBX, 93
IDIV WORD PTR [198373634]     ; Divide 93: el valor que haya en la dirección 198373634.
```

*Nota: Al final todo queda en binario, es decir lo que se hace es la conversión de los números del sistema de numeración que usemos a binario y realiza la operación correspondiente.

Operaciones lógicas:

AND: Se hace un “Y” lógico con el operando destino y el origen, almacenando el resultado en el destino.
El AND es muy sencillo, vamos a ver como funciona: se comparan los bits de un operando con los correspondientes del otro y si los dos son 1 se pone 1 y si no 0, ejemplos:

```
        1010          1111111       10000         10101010       11101111000111
AND     1111    AND   1111111  AND  10000    AND  10101010  AND  10011001111010
        ____          _______       _____         ________       ______________
        1010          1111111       10000         10101010       10001001000010
```

Vamos a ver unos ejemplos de como sería en un código de Ensamblador:

```
MOV EBX, 123
MOV EAX, 345
AND EAX, EBX
```

Primeros es hacer la conversión a binario: 123 = 1111011, 345 = 101011001:

```
      1111011
AND 101011001          ;Cuando no hay nada arriba se supone que es 0.
    _________
    001011001
```

Hacemos la conversión de 1011001 a decimal: 89

EAX valdrá 89 que es el resultado del AND y EBX se quedará como estaba, con 345.

Posibilidades:

```
AND REGISTRO, REGISTRO
AND REGISTRO, MEMORIA
AND MEMORIA, REGISTRO
AND REGISTRO, DATO
AND MEMORIA, DATO
```

TEST: Esta operación es como el AND pero con una diferencia, no guarda el resultado en ningún sitio, solo afecta a los flags.

Posibilidades:

```
TEST REGISTRO, REGISTRO
TEST REGISTRO, MEMORIA
TEST MEMORIA, REGISTRO
TEST REGISTRO, DATO
TEST MEMORIA, DATO
```

OR: Se hace un “O” lógico con el operando destino y el origen, almacenando el resultado en el destino.
El OR es muy sencillo, vamos a ver como funciona: se comparan los bits de un operando con los correspondientes del otro y si los dos son 0 se pone 0 y si no 1, ejemplos:

```
        1010          1111111       10000        10101010        11101111000111
OR      1111    OR    1111111  OR   10000   OR   10101010   OR   10011001111010
        ____          _______       _____        ________        ______________
        1111          1111111       10000        10101010        11111111111111
```

Vamos a ver unos ejemplos de como sería en un código de Ensamblador:

```
MOV EBX, 123
MOV EAX, 345
OR EAX, EBX
```

Primeros es hacer la conversión a binario: 123 = 1111011, 345 = 101011001:

```
    1111011
OR  101011001        ;Cuando no hay nada arriba se supone que es 0.
    _________
    101111011
```

Hacemos la conversión de 101111011 a decimal: 379
EAX valdrá 379 que es el resultado del OR y EBX se quedará como estaba, con 345.

Posibilidades:

```
OR REGISTRO, REGISTRO
OR REGISTRO, MEMORIA
OR MEMORIA, REGISTRO
OR REGISTRO, DATO
OR MEMORIA, DATO
```

XOR: Se hace un “O” exclusivo lógico con el operando destino y el origen, almacenando el resultado en el destino.
El XOR es muy sencillo, vamos a ver como funciona: se comparan los bits de un operando con los correspondientes del otro y si los dos no son iguales se pone valor 1, si son iguales 0, ejemplos:

```
        1010          1111111       10000        10101010        11101111000111
XOR     1111    XOR   1111111  XOR  10000   XOR  10101010   XOR  10011001111010
        ____          _______       _____        ________        ______________
        0101          0000000       00000        00000000        01110110111101
```

Vamos a ver unos ejemplos de como sería en un código de Ensamblador:

```
MOV EBX, 123
MOV EAX, 345
XOR EAX, EBX
```

Primeros es hacer la conversión a binario: 123 = 1111011, 345 = 101011001:

```
    1111011
XOR 101011001        ;Cuando no hay nada arriba se supone que es 0.
    _________
    100100010
```

Hacemos la conversión de 100100010 a decimal: 290
EAX valdrá 290 que es el resultado del XOR y EBX se quedará como estaba, con 345.

Posibilidades:

```
XOR REGISTRO, REGISTRO
XOR REGISTRO, MEMORIA
XOR MEMORIA, REGISTRO
XOR REGISTRO, DATO
XOR MEMORIA, DATO
```

NOT: Se hace un “NO” lógico a diferencia de las demás operaciones lógica el NOT y NEG solo funcionan con un operando, guardando en este mismo el resultado; lo que hace es poner cuando haya un 1 un 0, y cuando haya un 0 un 1 (poner lo contrario), vamos a ver unos ejemplos:

NOT 1010101 = 0101010 NOT 111111 = 000000 NOT 1111 = 0000 NOT 1100011101 = 0011100010

Vamos a ver unos ejemplos de como sería en un código de Ensamblador:

```
MOV EAX, 345
NOT EAX
```

Primeros es hacer la conversión a binario: 345 = 101011001:

```
NOT 101011001
    _________
    010100110
```

Hacemos la conversión de 010100110 a decimal: 166
EAX valdrá 166.

Posibilidades:

```
NOT REGISTRO
```

NOT MEMORIA

NEG: Se hace un “NEG” lógico a diferencia de las demás operaciones lógica el NEG y NOT solo funcionan con un operando, guardando en este mismo el resultado; lo que hace es poner el número en negativo en complemento a 2.

SHL: esta instrucción produce un desplazamiento de bits hacia la izquierda, tiene en cuenta dos operandos, el destino: que es el valor que vamos a desplazar y en origen se pone el número de rotaciones, que puede ser 1, si son más se indicará el número de desplazamientos en el registro CL.

Vamos a ver como funciona:

```
Posición:        [BASURA]   [3][2][1][0]
Valor:                       1  0  0  0
```

Como podemos ver teniendo este registro con esas posiciones vamos a ver que pasa en un:

```
MOV EAX, 1000
SHL EAX, 1:

[BASURA] <- [3]  [2][1][0][N]      ;Como vemos el primer bit de la izquierda se va a la “basura” y en
             1    0  0  0  0       la izquierda entra uno nuevo con valor, así que después de esto
                                   el resultado de EAX es 0000.
```

Vamos a ver otro ejemplo:

```
MOV EAX, 110101
MOV CL, 3
SHL EAX, CL

                ;Vamos por pasos:

Posición: [BASURA]  [5][4][3][2][1][0]
Valor:               1  1  0  1  0  1


                ;Primer paso:  [BASURA]  [4][3][2][1][0][n]
                                [5]        1  0  1  0  1  0

               ;Segundo paso:  [BASURA]     [3][2][1][0][n][n2]
                                [5][4]       0  1  0  1  0  0

                ;Tercer paso:  [BASURA]       [2][1][0][n][n2][n3]
                                [5][4][3]      1  0  1  0  0  0
```

Como vemos los “n” son los bits nuevos que entran con valor 0 siempre, y los de la izquierda se Van a la “basura” esta basura en realidad no existe, simplemente desaparecen, lo he puesto para que quedase más claro, cada vez que se va uno por la izquierda entra otro por la derecha con valor 0. Después de esto EAX acaba valiendo 101000.

Posibilidades:

```
SHL REGISTRO, 1
SHL REGISTRO, CL
SHL MEMORIA, 1
SHL MEMORIA, CL
```

*Nota: cuando es más que 1 los bits que queremos desplazar, se pone el valor en CL, usando SHL destino, CL.

*Nota: esto del CL puede que no sea siempre necesario indicarlo para que se produzcan más desplazamientos y puede que se admita pasar el valor directamente.

SHR: esta instrucción produce un desplazamiento de bits hacia la derecha, tiene en cuenta dos operandos, el destino: que es el valor que vamos a desplazar y en origen se pone el número de rotaciones, que puede ser 1, si son más se indicará el número de desplazamientos en el registro CL.

Vamos a ver como funciona:

```
Posición:        [3][2][1][0]      [BASURA]
Valor:            1  0  0  0
```

Como podemos ver teniendo este registro con esas posiciones vamos a ver que pasa en un:

```
MOV EAX, 1000
SHR EAX, 1:

 [N][3][2][1]      [0] ->  [BASURA]  ;Como vemos sale el bit de más a la derecha
  0  1  0  0         0                para la basura y entra uno nuevo por la
                                      izquierda con valor 0.
```

Vamos a ver otro ejemplo:

```
MOV EAX, 110101
MOV CL, 3
SHR EAX, CL
                ;Vamos por pasos:

Posición:          [5][4][3][2][1][0]        [BASURA]
Valor:              1  1  0  1  0  1
```

```
;Primer paso:      [n][5][4][3][2][1]      [BASURA]
                    0  1  1  0  1  0       [0]

;Segundo paso:     [n2][n][5][4][3][2]       [BASURA]
                     0  0  1  1  0  1         [1][0]


;Tercer paso:     [n3][n2][n][5][4][3]       [BASURA]
                    0   0  0  1  1  0         [2][1][0]
```

Como vemos los “n” son los bits nuevos que entran con valor 0 siempre, y los de la derecha se Van a la “basura” esta basura en realidad no existe, simplemente desaparecen lo he puesto para que quedase más claro, cada vez que se va uno por la derecha entra otro por la izquierda con valor 0. Después de esto EAX acaba valiendo 000110.

Posibilidades:

```
SHR REGISTRO, 1
SHR REGISTRO, CL
SHR MEMORIA, 1
SHR MEMORIA, CL
```

*Nota: cuando es más que 1 los bits que queremos desplazar, se pone el valor en CL, usando: SHR destino, CL.

*Nota: esto del CL puede que no sea siempre necesario indicarlo para que se produzcan más desplazamientos y puede que se admita pasar el valor directamente.

Otras instrucciones sin clasificar:

NOP: Esta instrucción no hace absolutamente nada, solo se usa para perder tiempo o para rellenar. Es el equivalente a: XCHG EAX, EAX

INT: Esta instrucción sirve para llamar a una interrupción del S.O o de la BIOS, no voy a extenderme más, por que no entra en el objetivo de este tutorial.

Saltos:

CALL: esta instrucción, sirve para llamar a una subrutina, solo tiene un operando. Cuando usemos: CALL DIRECIÓN se empezará a ejecutar lo que hay a partir de la dirección indicada.

RET: esta instrucción sirve para volver de la llamada de una subrutina (de un CALL, que es lo mismo).

Vamos a ver un ejemplo del funcionamiento de CALL y RET:

```
Dirección        Entry_Point              Estado de los registros
                      \/

00400001         MOV EAX, 3               EAX = 3
00400005         MOV EBX, 45              EBX = 45
00400009         CALL 00400112
0040000E         ADD EAX, EBX             EAX =5  ,  EBX = 3
.........          .........
00400112         MOV EAX, 2               EAX = 2
00400116         MOV EBX, 3               EBX = 3
0040011A         RET
....               ........
```

El Entry Point es donde se empieza a ejecutar el código, como podemos ver es como cuando llamamos a una función en otro lenguaje, al hacer el CALL pasa a ejecutarse lo que hay a partir de la dirección 00400112, y al hacer el RET sigue ejecutándose a partir de la siguiente instrucción al CALL, que es: 0040000E.

Así que el código anterior sería equivalente a:

```
MOV EAX, 3
MOV EBX, 45
MOV EAX, 2
MOV EBX, 3
ADD EAX, EBX
```

En realidad el RET lo que hace es sacar el último valor metido en la pila (POP) y saltar a esa dirección, el CALL lo que hace es meter en la pila el valor de la siguiente instrucción, así que cuidado si alguien hace un PUSH antes de hacer un RET por qué puede que no vaya a la dirección que hay después del CALL que llamó a este procedimiento.

JMP: esta instrucción es un salto incondicional, esto quiere decir que salta a la dirección que le pasemos.

Vamos a ver un ejemplo:

JMP 00401234

Saltará a la dirección 00401234, es decir, se empezará a ejecutar lo que hay a partir de esa dirección.

Vamos a ver otro ejemplo:

Dirección	Entry_Point \/	Estado de los registros	Registros después del JMP
00400001	MOV EAX, 3	EAX = 3	
00400005	MOV EBX, 45	EBX = 45	
00400009	ADD EAX, EBX	EAX = 48 EBX = 45	EAX = 138, EBX = 45
0040000E	ADD EAX, EBX	EAX 93 EBX = 45	EAX = 183 EBX = 45
00400012	JMP 00400009		

Cuando se termine de ejecutar esto, EAX valdrá 183 Y EBX 45.

JZ/JE: este salto salta a la dirección indicada si el resultado de una operación es cero, es decir, que el ZF tiene que estar activado (1).

Vamos a ver un ejemplo:

Dirección	Entry_Point \/	Estado del ZF	Estado de los registros
00400001	MOV EAX, 12	0	EAX 12
00400005	MOV EBX, 12	0	EBX 12
00400009	CMP EAX, EBX	1	EAX = 12, EBX = 12
0040000E	JZ 00400023		
….	….		
00400023	MOV EAX, 3	1	EAX = 3
00400027	CMP EAX, EBX	0	EAX = 3, EBX = 12
….	…		

Como vemos en el ejemplo, al principio EAX y EBX valen lo mismo, y el CMP al compararlos da 0, ya que es equivalente a restar uno al otro sin almacenar el resultado, solo pone el ZF a 1 para indicarlo.
Así que entrará después del CMP a la dirección 00400023, si el ZF llega a estar a 0 no entraría.

Esto es muy importante por que en muchos programas lo que hacen es comparar el serial que metimos Con el original y hacen un JE a la zona que nos registra el programa, si no son iguales los seriales irá a la zona de serial incorrecto.

*Nota: JE y JZ son la misma instrucción.

JNZ/JNE: este salto salta a la dirección indicada si el resultado de una operación no es cero, es decir, que el ZF tiene que estar desactivado (0).

Vamos a ver un ejemplo:

Dirección	Entry_Point \/	Estado del ZF	Estado de los registros
00400001	MOV EAX, 23	0	EAX 23
00400005	MOV EBX, 98	0	EBX 98
00400009	CMP EAX, EBX	0	EAX = 23, EBX = 98
0040000E	JNZ 00400023		
….	….		
00400023	MOV EAX, 98	0	EAX = 98
00400027	CMP EAX, EBX	1	EAX = 98, EBX = 98
….	…		

Como vemos en el ejemplo, al principio EAX y EBX no valen lo mismo, y el CMP al compararlos da un resultado distinto a 0, por lo cual no se activa el ZF.
Así que entrará después del CMP a la dirección 00400023, si el ZF llega a estar a 1 no entraría.

*Nota: JNE y JNZ son la misma instrucción.

JA/JNBE: este salto salta si el CF está desactivado (0) y el ZF también está desactivado (0).
Estos dos flags se desactivan a la vez cuando después de una operación el operando destino es superior.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el ZF y el CF a 1:

Dirección		Estado del ZF	Estado del CF	Estado de los registros
	.....			
	\/			
00400001	MOV EAX, 99	1	1	EAX 99
00400005	MOV EBX, 5	1	1	EBX 5
00400009	CMP EAX, EBX	0	0	EAX = 99, EBX = 5
0040000E	JNB 00400023			
....	....			
00400023	MOV EAX, 98	0	0	EAX = 98
00400027	MOV EBX, 45	0	0	EAX = 98, EBX = 45
....	...			

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JA y JNBE son la misma instrucción.

JAE/JNB: este salto salta si el CF está desactivado (0).
Este flag se desactiva cuando después de una operación el destino es superior o igual.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el el CF a 1:

Dirección		Estado del CF	Estado de los registros
	.....		
	\/		
00400001	MOV EAX, 99	1	EAX 99
00400005	MOV EBX, 99	1	EBX 99
00400009	CMP EAX, EBX	0	EAX = 99, EBX = 99
0040000E	JAE 00400023		
....	....		
00400023	MOV EAX, 98	0	EAX = 98
00400027	MOV EBX, 45	0	EAX = 98, EBX = 45
....	...		

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JAE y JNB son la misma instrucción.

JB/JNAE: este salto salta si el CF está activado (1).
Este flag se activa cuando después de una operación el destino es inferior.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el el CF a 0:

Dirección		Estado del CF	Estado de los registros
	.....		
	\/		
00400001	MOV EAX, 5	0	EAX 5
00400005	MOV EBX, 87	0	EBX 87
00400009	CMP EAX, EBX	1	EAX = 5, EBX = 87
0040000E	JB 00400023		
....	....		
00400023	MOV EAX, 98	1	EAX = 98
00400027	MOV EBX, 45	1	EAX = 98, EBX = 45
....	...		

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JB y JNAE son la misma instrucción.

JBE/JNA: este salto salta si el CF está activado (1) o el ZF está activado (1).
Estos flag se activan cuando después de una operación el destino es inferior o igual.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el ZF y el CF a 0:

Dirección		Estado del ZF	Estado del CF	Estado de los registros
	.....			
	\/			
00400001	MOV EAX, 5	0	0	EAX 5
00400005	MOV EBX, 78	0	0	EBX 78
00400009	CMP EAX, EBX	1	1	EAX = 5, EBX = 78
0040000E	JBE 00400023			
....	....			
00400023	MOV EAX, 98	1	1	EAX = 98

00400027	MOV EBX, 45	1	1	EAX = 98, EBX = 45
....	...			

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JBE y JNA son la misma instrucción.

JG/JNLE: este salto esta basado en datos con signo y salta si el ZF está desactivado (0) y el SF tiene el mismo valor que el OF (sea 1 o 0). Estos flag se ponen así cuando después de una operación el destino es mayor.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el SF y el OF a 0 y el ZF a 1:

Dirección		Estado del ZF	Estado del SF	Estado del OF	Estado de los registros
	.....				
	\/	1	0	0	
00400001	MOV EAX, 87	1	0	0	EAX 87
00400005	MOV EBX, 2	1	0	0	EBX 2
00400009	CMP EAX, EBX	0	0		EAX = 87, EBX = 2
0040000E	JG 00400023				
....	....				
00400023	MOV EAX, 98	0	0	0	EAX = 98
00400027	MOV EBX, 45	0	0	0	EAX = 98, EBX = 45
....	...				

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JG y JNLE son la misma instrucción.

JGE/JNL: este salto esta basado en datos con signo y salta si SF tiene el mismo valor que el OF (sea 1 o 0). Estos flag se ponen así cuando después de una operación el destino es mayor o igual.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el SF y el OF a 1:

Dirección		Estado del SF	Estado del OF	Estado de los registros
	.....			
	\/	1	1	
00400001	MOV EAX, 87	1	1	EAX 87
00400005	MOV EBX, 87	1	1	EBX 87
00400009	CMP EAX, EBX	1	1	EAX = 87, EBX = 87
0040000E	JNL 00400023			
....	....			
00400023	MOV EAX, 98	1	1	EAX = 98
00400027	MOV EBX, 45	1	1	EAX = 98, EBX = 45
....	...			

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JGE y JNL son la misma instrucción.

JL/JNGE: este salto esta basado en datos con signo y salta si SF es 0 y el OF es 1. Estos flag se ponen así cuando después de una operación el destino es menor.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el SF 0 y el OF a 1:

Dirección		Estado del SF	Estado del OF	Estado de los registros
	.....			
	\/	0	1	
00400001	MOV EAX, 3	0	1	EAX 3
00400005	MOV EBX, 87	0	1	EBX 87
00400009	CMP EAX, EBX	0	1	EAX = 3, EBX = 87
0040000E	JL 00400023			
....	....			
00400023	MOV EAX, 98	0	1	EAX = 98
00400027	MOV EBX, 45	0	1	EAX = 98, EBX = 45
....	...			

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JL y JNGE son la misma instrucción.

JLE/JNG: este salto esta basado en datos con signo y salta si ZF es 1 o ZF es 1 y a la vez OF es 0. Estos flag se ponen así cuando después de una operación el destino es menor o igual.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el ZF a 1:

Dirección		Estado del ZF	Estado de los registros
	.....		
	\/	1	
00400001	MOV EAX, 3	1	EAX 3
00400005	MOV EBX, 3	1	EBX 3
00400009	CMP EAX, EBX	1	EAX = 3, EBX = 3
0040000E	JLE 00400023		
….	….		
00400023	MOV EAX, 98	1	EAX = 98
00400027	MOV EBX, 45	1	EAX = 98, EBX = 45
….	…		

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JL y JNGE son la misma instrucción.

JS: este salto esta basado en datos con signo y salta si SF es 1. Este flag se activa cuando después de una operación el resultado es un número negativo.

Vamos a ver un ejemplo:

Dirección		Estado del SF	Estado de los registros
	.....		
	\/	0	
00400001	MOV EAX, 20	0	EAX 20
00400005	MOV EBX, 90	0	EBX 90
00400009	SUB EAX, EBX	1	EAX = -70, EBX = 90
0040000E	JS 00400023		
….	….		
00400023	MOV EAX, 98	1	EAX = 98
00400027	MOV EBX, 45	1	EAX = 98, EBX = 45
….	…		

Como vemos entrará a la dirección 00400023 después del CMP.

JNS: este salto esta basado en datos con signo y salta si SF es 0. Este flag se desactiva cuando después de una operación el resultado no es un número negativo.

Vamos a ver un ejemplo:

Dirección		Estado del SF	Estado de los registros
	.....		
	\/	0	
00400001	MOV EAX, 90	0	EAX 90
00400005	MOV EBX, 20	0	EBX 20
00400009	CMP EAX, EBX	0	EAX = 90, EBX = 20
0040000E	JNS 00400023		
….	….		
00400023	MOV EAX, 98	0	EAX = 98
00400027	MOV EBX, 45	0	EAX = 98, EBX = 45
….	…		

Como vemos entrará a la dirección 00400023 después del CMP.

JC: este salto esta basado en datos con signo y salta si CF es 1. Este flag se activa cuando después de una operación el resultado se lleva acarreo.

Vamos a ver un ejemplo:

Dirección		Estado del CF
	.....	
	\/	
00400001	MOV EAX, 11111111111111111111111111111111	0
00400005	MOV EBX, 11111111111111111111111111111111	0
00400009	ADD EAX, EBX	1
0040000E	JNS 00400023	1
….	….	
00400023	MOV EAX, 98	1
00400027	MOV EBX, 45	1
….	…	

Como vemos entrará a la dirección 00400023 después del CMP.

JNC: este salto esta basado en datos con signo y salta si CF es 0. Este flag se desactiva cuando después de una operación el resultado no produce acarreo.

Vamos a ver un ejemplo:

Este será una parte de un programa en ejecución; anteriormente imaginemos que ha hecho unas operaciones que han puesto el CF a 1:

Dirección		Estado del CF	Estado de los registros
	.....		
	\/	1	
00400001	MOV EAX, 3	1	EAX 3
00400005	MOV EBX, 3	1	EBX 3
00400009	CMP EAX, EBX	0	EAX = 3, EBX = 3
0040000E	JNC 00400023		
....	....		
00400023	MOV EAX, 98	0	EAX = 98
00400027	MOV EBX, 45	0	EAX = 98, EBX = 45
....	...		

Como vemos entrará a la dirección 00400023 después del CMP.

JO: este salto esta basado en datos con signo y salta si OF es 1. Este flag se activa cuando después de una operación se produce un desbordamiento.

Vamos a ver un ejemplo:

Dirección		Estado del OF
	.....	
	\/	
00400001	MOV EAX, 11111111111111111111111111111111	0
00400005	MOV EBX, 11111111111111111111111111111111	0
00400009	ADD EAX, EBX	1
0040000E	JO 00400023	1
....	....	
00400023	MOV EAX, 98	1
00400027	MOV EBX, 45	1
....	...	

Como vemos entrará a la dirección 00400023 después del CMP.

JNO: este salto esta basado en datos con signo y salta si OF es 0. Este flag se desactiva cuando después de una operación no se produce un desbordamiento.

Vamos a ver un ejemplo:

Dirección		Estado del OF
	.....	
	\/	
00400001	MOV EAX, 11111	0
00400005	MOV EBX, 1111111111111111	0
00400009	ADD EAX, EBX	0
0040000E	JO 00400023	0
....	....	
00400023	MOV EAX, 98	0
00400027	MOV EBX, 45	0
....	...	

Como vemos entrará a la dirección 00400023 después del CMP.

JP/JPE: este salto esta basado en datos con signo y salta si PF es 1. Este flag se activa cuando después de una operación se produce paridad.

Vamos a ver un ejemplo:

Dirección		Estado del PF
	.....	
	\/	
00400001	MOV EAX, 101011	0
00400005	MOV EBX, 101010	0
00400009	ADD EAX, EBX	1
0040000E	JP 00400023	1
....	....	
00400023	MOV EAX, 98	1
00400027	MOV EBX, 45	1
....	...	

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JP y JPE son la misma instrucción.

JNP/JNPO: este salto esta basado en datos con signo y salta si PF es 0. Este flag se desactiva cuando después de una operación no se produce paridad.

Vamos a ver un ejemplo:

Dirección		Estado del PF
	.....	
	\/	
00400001	MOV EAX, 101010	0
00400005	MOV EBX, 101010	0
00400009	ADD EAX, EBX	0
0040000E	JNP 00400023	0
....	....	
00400023	MOV EAX, 98	0
00400027	MOV EBX, 45	0
....	...	

Como vemos entrará a la dirección 00400023 después del CMP.

*Nota: JNP y JNPO son la misma instrucción.

Otras:

LOOP: esta instrucción sirve para hacer bucles. Como contador se usará CX.

Vamos a ver un ejemplo:

Dirección	Instrucción
00400001	MOV CX, 6
00400005	MOV EBX, CX
00400009	MOV EAX, CX
0040000E	LOOP 00400005

Cada vez que se ejecute el LOOP se irá a la dirección que indique, restándole una unidad a CX, cuando CX sea 0 el LOOP ya no tendrá efecto y el programa seguirá su ejecución normal.

REP: esta instrucción repite solo operaciones de comparación, usa como contador CX.

REPZ/REPE: esta instrucción repite solo operaciones de comparación, usa como contador CX, pero necesita también que el ZF esté a 1, es decir, que lo que se compare sea igual.

REPNZ/REPNE: esta instrucción repite solo operaciones de comparación, usa como contador CX, pero necesita también que el ZF esté a 0, es decir, que lo que se compare sean distintas.

Tablas básicas:

Tabla de algunas instrucciones que considero importantes:

Vamos a usar la siguiente simbología:

1 - Se activa.
0 – Se desactiva.
? – No se sabe.
- No pasa nada.
* Se modifica según la operación.

Instrucción	Propósito	Efecto en los flags								
		AF	CF	DF	IF	OF	PF	SF	TF	ZF
ADC	Suma.	*	*	-	-	*	*	*	-	*

ADD	Suma.	*	*	-	-	*	*	*	-	*
AND	AND Lógico.	?	0	-	-	0	*	*	-	*
CALL	Llamada.	-	-	-	-	-	-	-	-	-
CMP	Comparación/resta.	*	*	-	-	*	*	*	-	*
DEC	Decrementa en 1 unidad.	*	-	-	-	*	*	*	-	*
DIV	Divide.	?	?	-	-	?	?	?	-	?
IDIV	Divide.	?	?	-	-	?	?	?	-	?
IMUL	Multiplica.	?	*	-	-	*	?	?	-	?
INC	Incrementa en 1 unidad.	*	-	-	-	*	*	*	-	*
IRET	Retorno.	*	*	*	*	*	*	*	*	*
JMP	Salto.	-	-	-	-	-	-	-	-	-
JE	Salto condicional, ZF=1	-	-	-	-	-	-	-	-	-
JZ	Salto condicional, ZF=1	-	-	-	-	-	-	-	-	-
JCX	Salto condicional, CX=0	-	-	-	-	-	-	-	-	-
JP	Salto condicional, PF=1	-	-	-	-	-	-	-	-	-
JPE	Salto condicional, PF=1	-	-	-	-	-	-	-	-	-
JA	Salto condicional, CF y ZF=0	-	-	-	-	-	-	-	-	-
JAE	Salto condicional, CF=0	-	-	-	-	-	-	-	-	-
JB	Salto condicional, CX=1	-	-	-	-	-	-	-	-	-
JBE	Salto condicional, CF o ZF=1	-	-	-	-	-	-	-	-	-
JNA	Salto condicional, CF y ZF=1	-	-	-	-	-	-	-	-	-
JNAE	Salto condicional, CF=1	-	-	-	-	-	-	-	-	-
JNB	Salto condicional, CF=0	-	-	-	-	-	-	-	-	-
JNBE	Salto condicional, CF=0	-	-	-	-	-	-	-	-	-
JC	Salto condicional, CF=1	-	-	-	-	-	-	-	-	-
JNC	Salto condicional, CF=0	-	-	-	-	-	-	-	-	-

JNE	Salto condicional, ZF=0	-	-	-	-	-	-	-	-	-
JNZ	Salto condicional, ZF=0	-	-	-	-	-	-	-	-	-
JNP	Salto condicional, PF=0	-	-	-	-	-	-	-	-	-
JPO	Salto condicional, PF=0	-	-	-	-	-	-	-	-	-
JG	Salto condicional, ZF y SF=OF	-	-	-	-	-	-	-	-	-
JGE	Salto condicional, SF=OF	-	-	-	-	-	-	-	-	-
JL	Salto condicional, SF < OF.	-	-	-	-	-	-	-	-	-
JLE	Salto condicional, ZF=1 o ZF<OF	-	-	-	-	-	-	-	-	-
JNG	Salto condicional, ZF=1 o ZF<OF	-	-	-	-	-	-	-	-	-
JNGE	Salto condicional, SF < OF	-	-	-	-	-	-	-	-	-
JNL	Salto condicional, SF=OF	-	-	-	-	-	-	-	-	-
JNLE	Salto condicional, ZF=0 y SF=OF	-	-	-	-	-	-	-	-	-
JO	Salto condicional, OF=1	-	-	-	-	-	-	-	-	-
JNO	Salto condicional, OF=0	-	-	-	-	-	-	-	-	-
JS	Salto condicional, SF=1	-	-	-	-	-	-	-	-	-
JNS	Salto condicional, SF=0	-	-	-	-	-	-	-	-	-
MUL	Multiplica	?	*	-	-	*	?	?	-	?
POP	Saca de la pila	-	-	-	-	-	-	-	-	-
POPF	Pone flags en la pila	*	*	*	*	*	*	*	*	*
PUSH	Saca de la pila	-	-	-	-	-	-	-	-	-
PUSHF	Saca flags de la pila	-	-	-	-	-	-	-	-	-
RCL	Rotación	-	*	-	-	*	-	-	-	-
RCR	Rotación	-	*	-	-	*	-	-	-	-
ROL	Rotación	-	*	-	-	*	-	-	-	-
ROR	Rotación	-	*	-	-	*	-	-	-	-
SAL/SHL	Desplazamiento	?	*	-	-	*	*	*	-	*

SAR	Desplazamiento	?	*	-	-	*	*	*	-	*
SHR	Desplazamiento	?	*	-	-	*	*	*	-	*
OR	OR Lógico	?	0	-	-	0	*	*	-	*
XOR	XOR Lógico	?	0	-	-	0	*	*	-	*

Tablas de registros básicas:

Tabla de 16 bits 8086

Datos	**AX, BX, CX, DX**															
Segmento	**CS, DS, ES, SS**															
Puntero de instrucciones	**IP**															
Punteros de pila	**BP SP**															
Índice	**DI, SI**															
Flags	Bit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	Flag	**CF**	**-**	**PF**	**-**	**AF**	**-**	**ZF**	**SF**	**TF**	**IF**	**DF**	**OF**	**-**	**-**	**-**	**-**

Tabla de 32 bits 80386

Datos	**EAX, EBX, ECX, EDX**																		
Segmento	**CS, DS, ES, SS**																		
Puntero de instrucciones	**EIP**																		
Punteros de pila	**EBP ESP**																		
Índice	**EDI, ESI**																		
Eflags	Bit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**16**	**15**	**17**
	Flag	**CF**	**-**	**PF**	**-**	**AF**	**-**	**ZF**	**SF**	**TF**	**IF**	**DF**	**OF**	**-**	**-**	**NT**	**RF**	**-**	**VM**

Complemento a uno:

Para hallar el complemento a uno de un número binario se cambian los 0 por 1 y viceversa:

Vamos a ver un ejemplo: 1010101001, en complemento a uno: 0101010110

Complemento a dos:

Para indicar el signo de un número en binario se usa el complemento a dos: viendo el bit más significativo si es un 0 el número es positivo, si el número es negativo será un 1.

Para hallar el complemento a dos de un número binario se pone primero a complemento uno y se le suma 1:

Vamos a ver un ejemplo: el número -119 en decimal, primero hallamos el 119 positivo en binario: 01110111, ahora aplicamos el complemento a uno: 1000100 y le sumamos 1: 10001001.

▪ Windows API

API significa Application Programming Interface (Interfaz de Programación de Aplicaciones), su objetivo es la comunicación entre el software de un Sistema Operativo, la API de Windows son un conjunto de funciones que nos proporciona este Sistema Operativo para ayudar al programador para ciertas cosas, por ejemplos las gráficas (GUI). Las API no forman parte del Sistema Operativo, son funciones que residen en distintas DLL (Dynamic Linking Library, Biblioteca de enlace dinámico) , con la API se puede crear una ventana, un icono, detectar la presencia de un depurador … La API ayuda a los programadores para que ciertas cosas no las tengan que programar ellos, simplemente usan la API adecuada. Las principales API en win32 son: kernel32.dll, gdi32.dll y user32.dll Hay dos categorías para las funciones de las API:

1) Las que acaben en "A" para ANSI (American National Standards Institute, Instituto Nacional Estadounidense de Estándares), por ejemplo la función CharUpperA del user32.dll

2) Las que acaben en "W" para UNICODE, por ejemplo la función lstrcmpW del kernel32.dll

Esto del ANSI es bastante sencillo, son solo caracteres terminados con un NULL y tiene de tamaño 1 byte.

Luego se creo UNICODE por que en otras lenguas hay más caracteres de los que se pueden formar con el ANSI, el UNICODE tiene un tamaño de 2 bytes, permitiendo 65536 caracteres distintos.

La biblioteca que se necesite (DLL) se carga en memoria cuando un programa la solicita, pero cualquier otro programa tiene acceso a las funciones que permite la misma.

*Nota: es conveniente bajarse la guía "Win32 Programmer´s Reference", que explica las funciones de la API de windows, nos va a ayudar mucho a la hora de investigar, a día de hoy la tenemos disponible online en: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_start_page.asp

▪ Ingeniería inversa

La ingeniería inversa es una técnica un tanto especial, ya que no se trata de crear algo, si no de saber como funciona algo que ya existe pero no sabemos parcialmente o globalmente como lo hace. En este tutorial vamos a ver algunos ejemplos prácticos de ingeniería inversa con software de win32, exactamente vamos a ver protección de software y algoritmos para desprotegerlo, es decir lo típico de cuando nos bajamos el serial de un programa que no hemos pagado, pues para estas y muchísimas más cosas sirve la ingeniería inversa, esto es totalmente educativo, usaremos algo llamado crackmes. En la ingeniería inversa de software el objetivo es entender el código en ensamblador, para modificar o entender como hace una determinada acción, normalmente como crea el serial el programa, o modificar el comportamiento del mismo, para que vaya a donde queramos, por ejemplo, a la zona del serial correcto.

*Nota: un crackme es un programa hecho para ser "crackeado", es decir, solo tiene el sistema de seguridad y nuestro trabajo es saltárnoslo, no es ilegal, ya que están hechos para eso, hacerlo con software comercial, es delito, así que no conviene hacerlo.

Técnicas:

Parchear: Esta técnica consiste en modificar el código ejecutable para conseguir algo, habitualmente es encontrar la zona donde compara el serial correcto con el incorrecto y va a una zona u otra.

Vamos a ver las dos zonas principales:

Chico bueno: es la zona de código donde se nos muestra el mensaje: serial correcto, y se nos registrar el programa.

Chico malo: es la zona de código donde se nos muestra el mensaje: serial incorrecto.

En la técnica de parcheo lo que vamos a hacer es modificar el salto ya sea condicional (un JNZ por ejemplo) o incondicional (JMP) que vaya a chico malo, para que vaya a la zona de chico bueno.

Vamos a ver un ejemplo:

Tenemos EAX con el serial que ha metido el usuario, por ejemplo: 2929393, y EBX con el serial correcto, por ejemplo: 99222:

……
00400000 CMP EAX, EBX
00400004 JZ 00400034

00400004 JNZ 00400134
…….

La zona de chico bueno será 00400034
La zona de chico malo será 004000134

Así que en este ejemplo lo que hacemos es:

1) Un CMP que pondrá el ZF a 1 si la resta de EAX y EBX da 0, es decir, que sean iguales, como en este caso no es así el ZF quedará a 0.

2) Un JZ 00400034, que lo que haremos es, entrar a 00400034 si el ZF está a 1, como este no es caso, no pasa nada.

3) Un JNZ 00400134, que lo que hacemos es, entrar a 00400134 si el ZF está a 0, como el ZF fue puesto a 0 por el CMP, entrará a esa dirección, que resulta ser el chico malo.

¿Cuál será el objetivo? Modificar el código fuente para que por ejemplo entre siempre a la zona de chico bueno, sea correcto o incorrecto el serial. El código tendría que quedar:

……
00400000 CMP EAX, EBX
00400004 JMP 00400034
00400004 JMP 00400034
…….

Como vemos siempre irá a la zona de chico bueno pero este código es algo absurdo, así que sería mejor algo así:

……
00400000 CMP EAX, EBX
00400004 JMP 00400034
00400004 NOP
…….

Así entrará de todas formas a 00400034 pero luego si volvemos a la siguiente instrucción será un NOP, es decir, una instrucción que no hace nada.

Aunque también podríamos hacer:

……
00400000 CMP EAX, EBX
00400004 JNZ 00400034
00400004 JZ 00400134
…….

Aunque esto tiene un problema, si el serial metido por el usuario es correcto entrará a la zona de chico malo, ya que hemos puesto los saltos contrarios a los que había, es decir, hacen lo inverso.

Podemos hacer todo lo que se nos ocurra, solo es saber ASM y hacer lo que estimemos adecuado. El problema principal es encontrar la zona donde se compara el serial correcto con el incorrecto.

¿Cuál es el problema de esta técnica? La gente que se dedica a la ingeniería inversa de software, no ve bien que se use esta técnica para que siempre vaya a la zona que queramos si hay otra vía, ya que hay que modificar el ejecutable y esto no conviene, es mejor entender el algoritmo que crea el serial, y poner un serial correcto y no hará falta alterar el funcionamiento del ejecutable, o incluso hacer un keygen, es decir, un programa que genere seriales para el programa.

Keygenning: esta técnica es simplemente entender el algoritmo de un programa por ejemplo, como crea un serial a partir de un nombre y crear un generador de claves con ese algoritmo.

Otros: hay muchísimos más métodos ya que hay muchas protecciones y no siempre se puede hacer esto.

Herramientas básica:

En este apartado vamos a resumir algunas de las herramientas básicas, que necesitaremos:

Depuradores: También llamados “Debuggers”, los depuradores son aplicaciones que nos permiten examinar Un programa, en este caso en ASM de win32, gracias a ellos podemos ver paso a paso como hace algo, y un montón de cosas más, es imprescindible.

Desensambladores: Cuando depuramos estamos viendo un programa en tiempo real de su ejecución, cuando lo desensamblamos no, solo vemos el código en ASM de win32 en este caso.

Descompresores/Desempacadores: Estos programas son muy útiles ya que a veces el software vienen comprimidos y esto nos dificulta la tarea, ya que hay que descomprimirlos, por que puede que no nos deje

depurar el software comprimido adecuadamente.

Editores hexadecimales: Gracias a este editor podremos modificar archivos en forma hexadecimal.

Identificadores de archivos : Gracias a esta clase de herramienta podemos tener cierta información sobre el programa, si está comprimido, con qué lo está …. El más conocido es el PEiD, lo tenéis disponible en http://www.peid.tk/ y http://www.fr33project.org, uno muy bueno también es el RDG.

Primeros objetivos:

En esta parte del tutorial vamos a usar las siguientes herramientas:

OllyDbg 1.10 (final versión), (depurador) disponible en http://www.ollydbg.de
o en la propia web http://www.fr33project.org

Hex Workshop 4.23, (Editor Hexadecimal) disponible en http://bpsoft.com o en http://www.fr33project.org

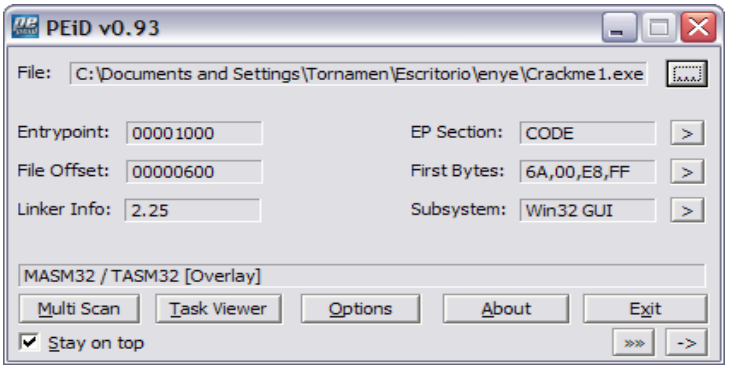
W32Dasm 8.93, (depurador y desensamblador) disponible en http://www.fr33project.org

PEiD 0.93, (identificador) disponible en http://www.peid.tk/ y http://www.fr33project.org

Y como crackme usaremos: el Crackme1.exe, este crackme pertenece a cruhedead, es muy conocido, lo he elegido por si no os sale con este tutorial, podéis encontrar otro que seguro que sí.

Manos a la obra:

1) Lo primero es conocer a nuestro enemigo y ver si está comprimido, así que abrimos el PEiD, hacemos clic en los puntos suspensivos que hay arriba a la derecha, seleccionamos el Crackme1.exe y damos a abrir:

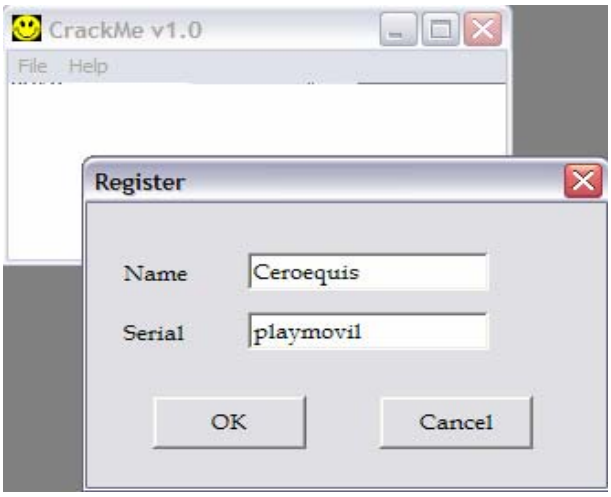


Vamos a ver que son algunas de los campos:

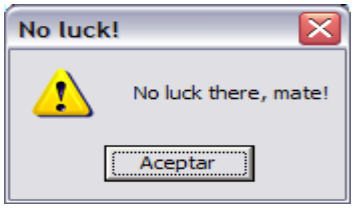
Entrypoint: muestra en donde empieza a ejecutarse el programa.
Subsystem: muestra el "sub system" en este caso es una aplicación GUI de Win32.

Una de las cosas más importantes de aquí es la parte donde pone MASM32/TASM32, esto quiere decir que fue ensamblado con uno de estos programas, si estuviera comprimido con UPX por ejemplo, lo pondría justo en ese lugar. Una vez visto que no está comprimido pasamos a analizar el programa.

2) Conocer el ejecutable, vamos a abrir el Crackme1.exe y vemos que tiene: dos pestañas, la de File no sirve para nada, en Help tenemos la zona para registrarnos, damos a Help – Register, y nos saldrá una ventana preguntando Name (nombre) y Serial (clave). La estrategia a seguir normalmente para crackearlo es meter un usuario y una contraseña cualquiera para ver el mensaje de error y luego buscarlo con un depurador o un desensamblador. Pues vamos allá:

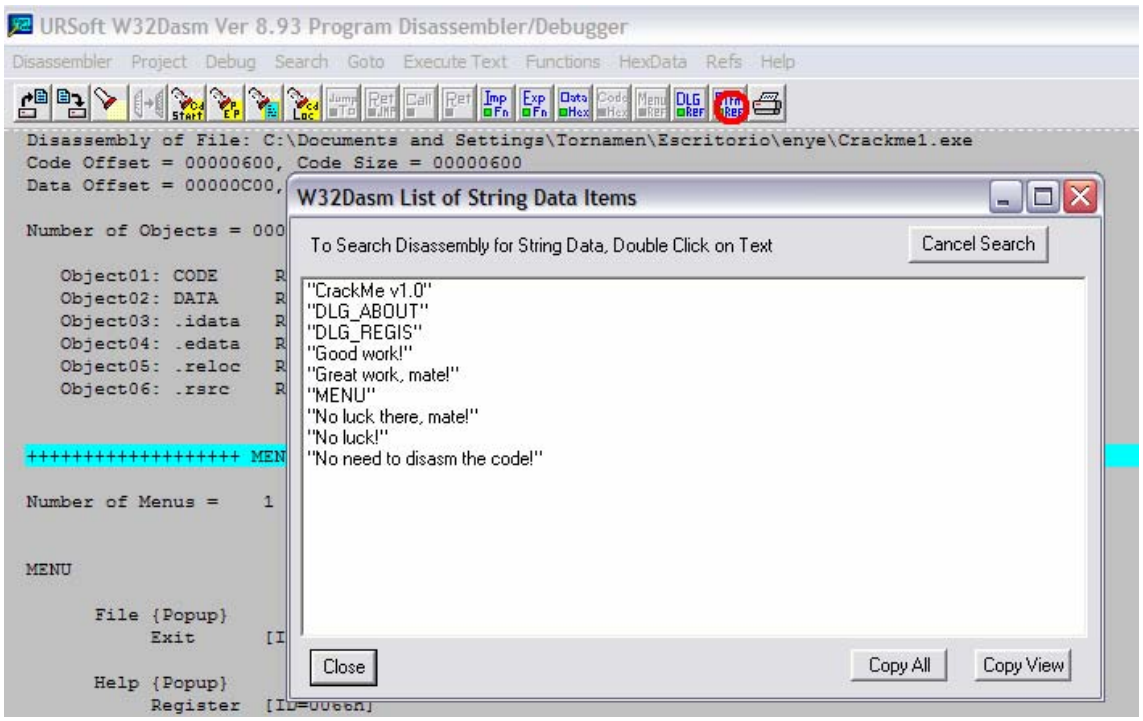


Yo en este caso he metido de Name Ceroequis y de password playmovil, pueden meter lo que quieran, ahora damos a OK:

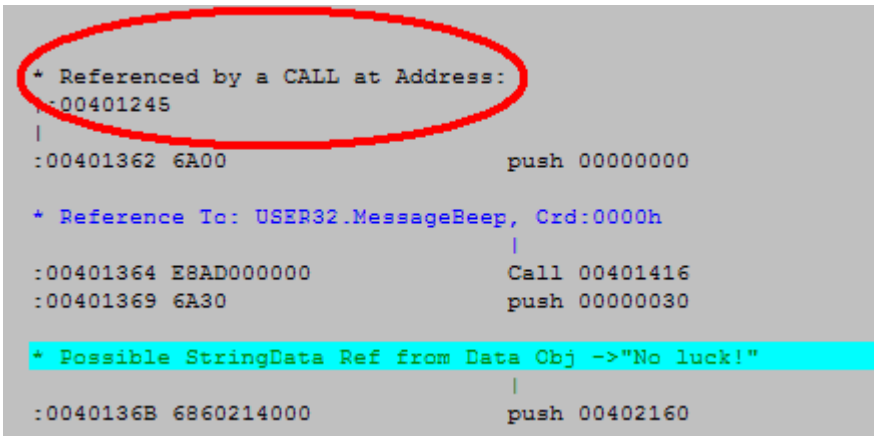


Ese es el mensaje que nos interesa: No luck there, mate!

3) Ya tenemos el mensaje de la zona de chico malo, ahora hay que pasar a la acción, cerramos el crackme y abrimos el W32Dasm. A continuación vamos a Disassembler – Open File to disassemble y seleccionamos el Crackme1.exe y damos a abrir, nos saldrá el código desensamblado, después hay que buscar el texto de chico malo, así que vamos a Stn Ref, que es el penúltimo botón, que está debajo de Refs, también podemos acceder desde Refs – String Data Referentes. (Esto lo hacemos para que se nos muestren las referencias a las strings, es decir a las cadenas de caracteres).

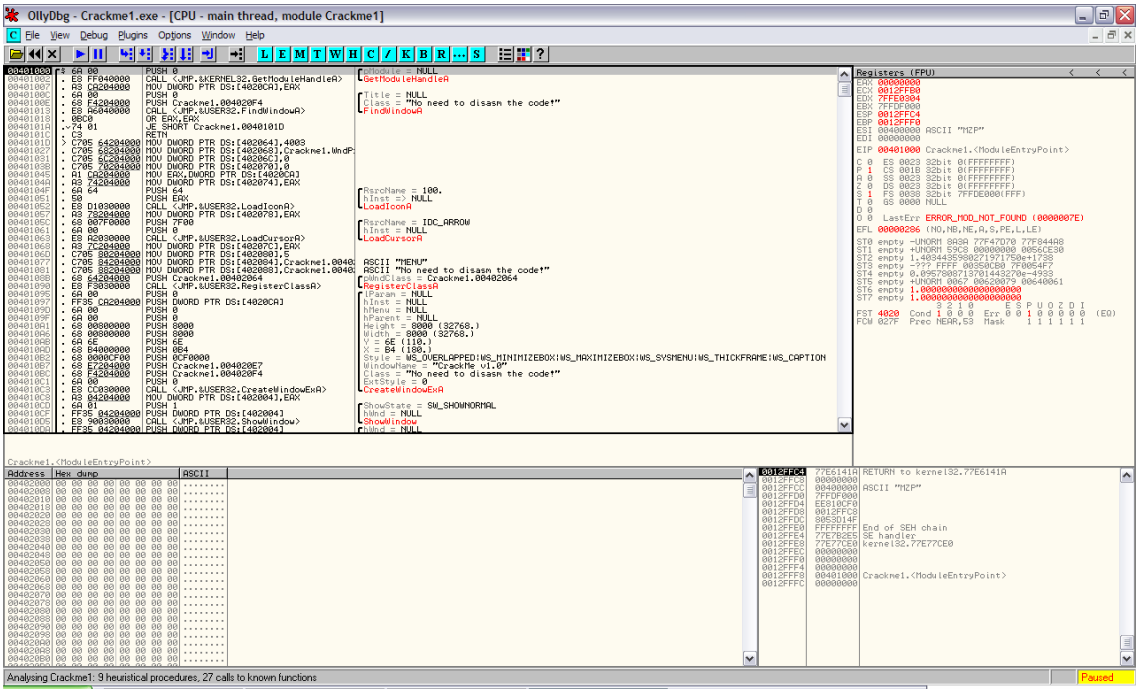


Lo que he rodeado en rojo es el botón al que hay que dar. Como vemos nos sale un listado con las cadenas de caracteres, mira que casualidad seguro que lo de Good Work! Es la zona de chico bueno, pero lo que andamos buscando es la chico malo, así que hacemos doble clic sobre: No luck There, mate! Y cerramos la ventana que ha listado las strings. Seguidamente aparecemos sobre la parte del código de chico malo, subimos un poco la ventana para ver desde donde ha sido llamada:

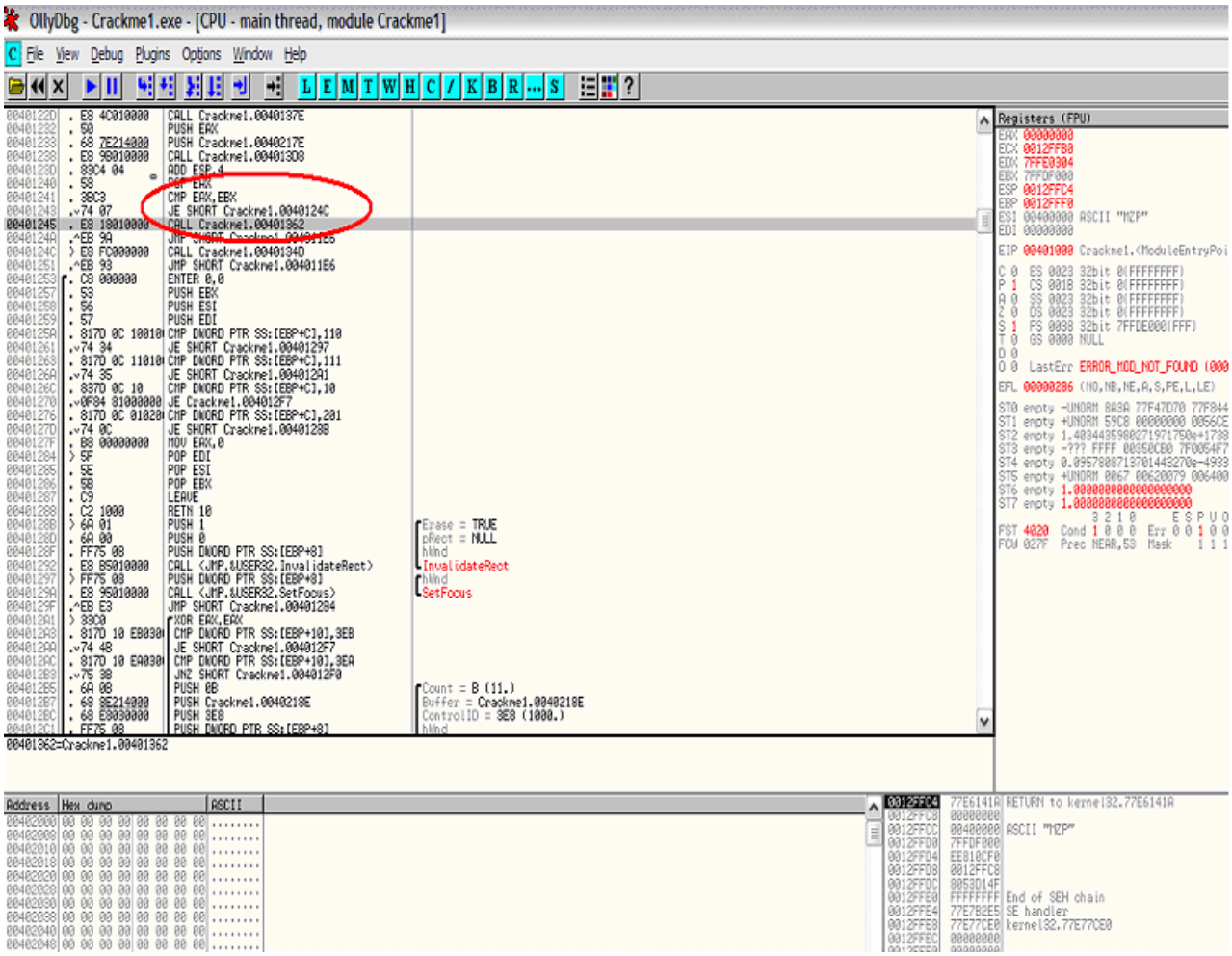


La zona rodeada es la que nos interesa, nos muestra desde que llamada (CALL) ha sido referenciada.

4) El siguiente paso es cerrar el W32Dasm, pero antes apuntar la referenced by a call at address: 00401245. Cerramos y abrimos el OllyDbg, vamos a file – open y seleccionamos el Crackme1.exe y damos a abrir. Atención: el OllyDbg tiene que estar así:

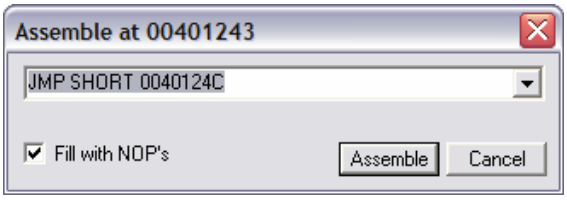


Las 4 ventanas tienen que estar tal cual las he puesto, para controlarlo todo, la más grande de la izquierda arriba, es el código, la de la derecha son los registros, la que está debajo de los registros es la pila y la de la izquierda de la pila es el código en hexadecimal.
Ahora lo que hacemos es: botón derecho sobre la ventana más grande en el centro de ella, Go to – Expression, (apretando Control + G sale directamente), nos saldrá un dialog, allí metemos la CALL que apuntamos antes: 00401245, para ir a la llamada que va a chico malo. Ya estamos sobre la llamada a chico malo, ahora subimos un poco y vemos:

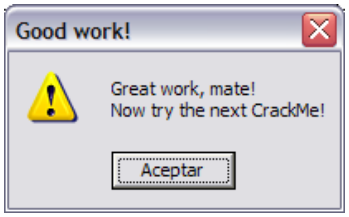


Como vemos hay un CMP EAX, EBX, luego un JE y debajo: el CALL de chico malo, por lo cual es bastante probable, que el CMP compare el serial correcto con el incorrecto y si está bien se active el ZF y salte con un JE a 0040124C, y si el serial es incorrecto entra con el CALL a chico malo que es al que hemos ido con el GO TO.

5) Es la hora de saber que es un Breakpoint, (abreviación: BP), cuando colocamos un breakpoint en una instrucción lo que pasará es que al ejecutar el programa y llegue a esa instrucción se parará y podremos seguir las siguientes instrucciones una a una para ver como crea el serial o lo que necesitemos saber. Pero antes vamos a ver como se parcharía este programa, es muy sencillo, ¿veis el JE? Vamos a poner un JMP para que salte siempre, para eso hacemos doble clic sobre: JE SHORT 0040124C, y lo sustituimos por: JMP SHORT 0040124C y damos a Assemble:



(Solo dar una vez a Assemble) y luego cerramos la ventana.
Ahora damos al botón de play o lo que es lo mismo F9 (o en la pestaña Debug – Run, que es lo mismo). Como vemos la ventana del Crackme1.exe se a abierto, (abajo a la izquierda) hacemos clic y metemos el usuario y la password que queramos, por ejemplo yo meteré: Name Ceroequis , Serial: playmovil, y damos a OK:



¡Lo hemos conseguido! como supuse era el salto condicional a la zona de chico malo, muy bien, ahora lo que tenemos que hacer es: dar a Aceptar y cerramos el programa.

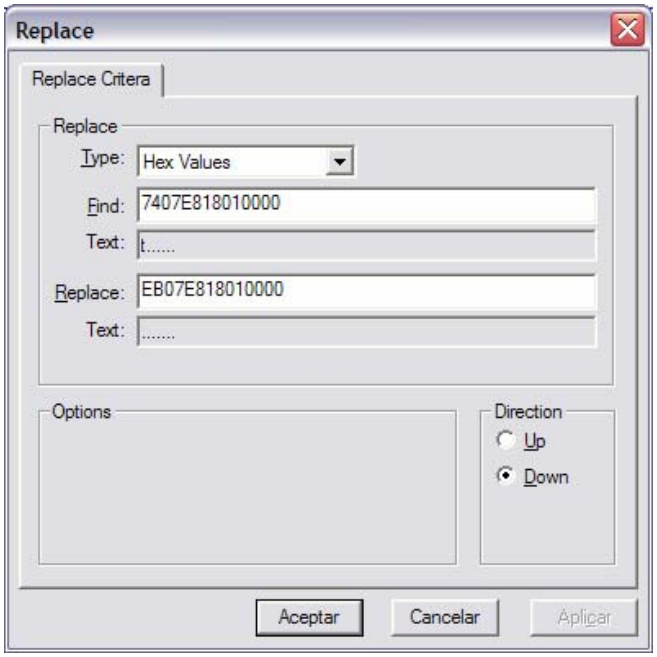
6) El OllyDbg nos ha dejado modificar el .exe y ejecutarlo, pero los cambios no se han guardado, para esto usaremos un editor hexadecimal, lo primero que hay que hacer es en el OllyDbg ir a Debug – Restart (o Control + F2). Hacemos esto para que el programa termine su ejecución y quede todo como estaba al principio, así que ahora hacemos otra vez lo del principio, botón derecho en el centro de la pantalla más grande GO TO – Expression: 00401245 y damos a OK, una vez allí subimos un poco, y donde está el JE miramos a la izquierda, en la columna hexadecimal, y vemos:

74 07 <- El 74 es el número hexadecimal del JZ y el JE. Como pueden existir varios saltos del estilo vamos a copiar también la parte de debajo:
E8 18010000

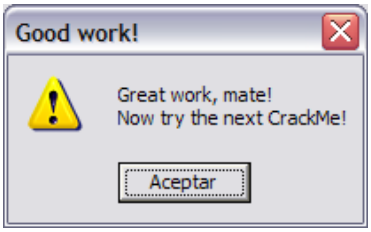
Así que tenemos: 7407E818010000
Vamos a tener que sustituir el 74 por el código hexadecimal del JMP que es: EB, lo podemos ver si hacemos doble clic sobre la instrucción y ponemos JMP donde pone JE y damos a Assemble, así que tenemos que sustituir:

7407E818010000
por:
EB07E818010000

7) Ahora abrimos el Hex Workshop y cerramos el OllyDbg, hacemos una copia del Crackme1.exe por si acaso y la llamamos: Copia-Crackme1.exe, en el Hex Workshop vamos a File – Open - Copia-Crackme1.exe y damos a abrir. Ahora vemos el código del programa en hexadecimal. Vamos a Edit – Replace o Control + H y ponemos esto:

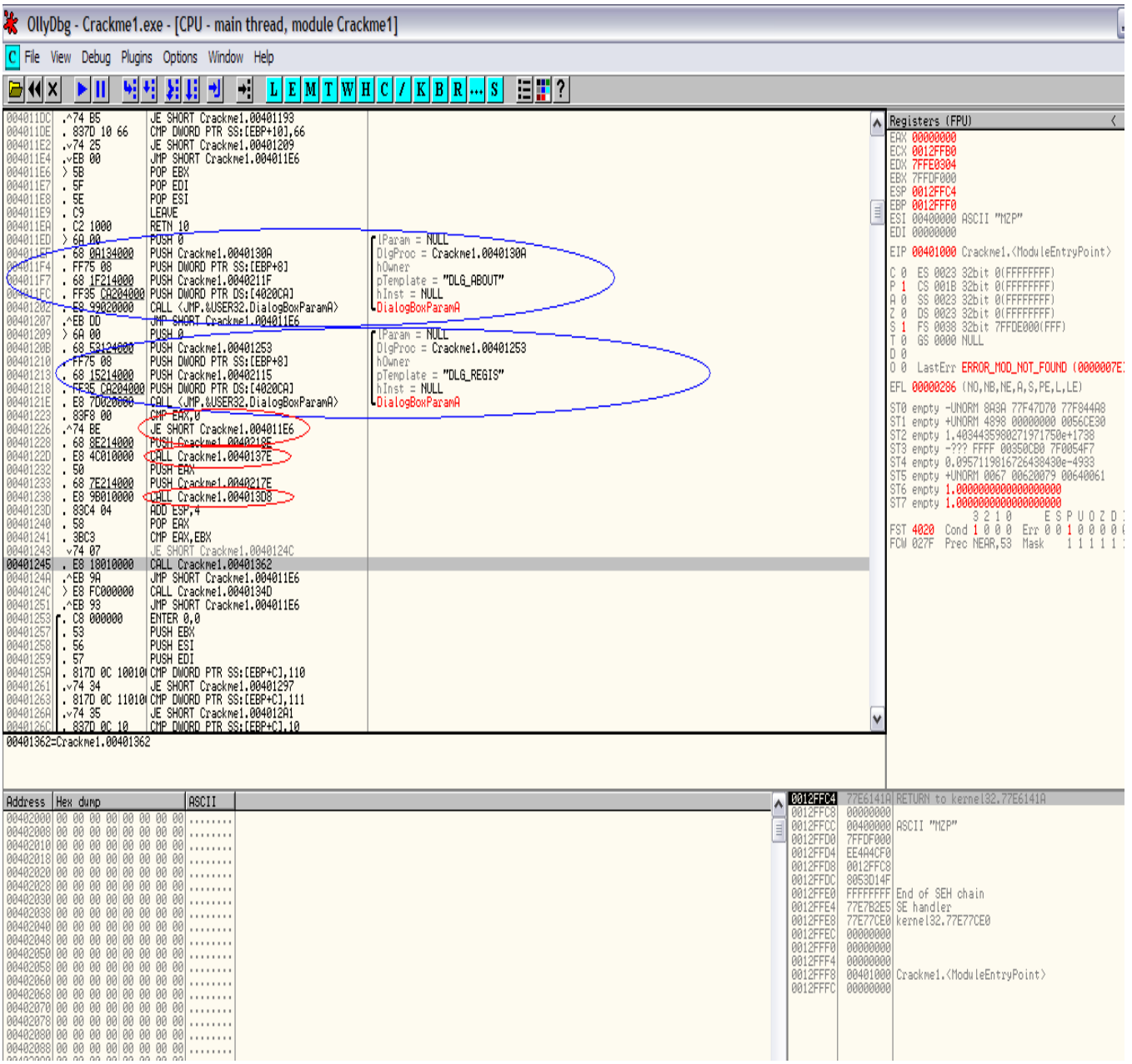


Y damos a aceptar. Nos sale un aviso y damos a Replace ALL, para que haya salido bien, solo se tiene que haber reemplazado un elemento, ahora vamos a File – Save y damos a Si. Cerramos el edito Hexadecimal y abrimos el Copia-Crackme1.exe, vamos a Help – Register y metemos un Name y un Serial cualquiera, yo meteré Name Ceroequis y Serial playmovil y damos a OK:



Ya hemos parcheado el programa original y cuando metamos un serial correcto o incorrecto nos lo dará por bueno (“en teoría”, ya que puede comprobar más cosas en otra parte del código para ir a chico malo, como veremos más adelante).

8) Ahora vamos a sacar un Name y un Serial correcto para no tener que modificar el programa. Primero cerramos todo lo que tenga que ver con el Crackme1.exe. Abrimos el OllyDbg, y abrimos con él el Crackme1.exe, que es el ejecutable sin modificar, ahora vamos donde siempre: botón derecho en el centro de la ventana más grande, GO TO – Expression: 00401245 y damos a OK. Ahora nuestro objetivo es ver como crea el serial, así que subimos un poco para ver el código que hay arriba:



Vamos a ver que son las cosas que hay por encima de la llamada a chico malo, empezamos por lo que he rodeado en azul:

```
004011ED   > 6A 00          PUSH 0                          ; /lParam = NULL
004011EF   . 68 0A134000    PUSH Crackme1.0040130A          ; |DlgProc = Crackme1.0040130A
004011F4   . FF75 08        PUSH DWORD PTR SS:[EBP+8]       ; |hOwner
004011F7   . 68 1F214000    PUSH Crackme1.0040211F          ; |pTemplate = "DLG_ABOUT"
004011FC   . FF35 CA204000  PUSH DWORD PTR DS:[4020CA]      ; |hInst = NULL
00401202   . E8 99020000    CALL <JMP.&USER32.DialogBoxParamA>      ; \DialogBoxParamA
```

Esto es nada más y nada menos que la creación del dialog de ABOUT, además de que se vé: “DLG_ABOUT”, esto no nos sirve de nada, pero un poco más debajo vemos:

```
00401209   > 6A 00          PUSH 0                          ; /lParam = NULL
0040120B   . 68 53124000    PUSH Crackme1.00401253          ; |DlgProc = Crackme1.00401253
00401210   . FF75 08        PUSH DWORD PTR SS:[EBP+8]       ; |hOwner
00401213   . 68 15214000    PUSH Crackme1.00402115          ; |pTemplate = "DLG_REGIS"
00401218   . FF35 CA204000  PUSH DWORD PTR DS:[4020CA]      ; |hInst = NULL
0040121E   . E8 7D020000    CALL <JMP.&USER32.DialogBoxParamA>      ; \DialogBoxParamA
```

Esto se ve a la legua que es el de register por el “DLG_REGIS”, muy bien pues debajo de esto están las acciones que pasan cuando damos a OK, cuando rodeé en rojo el:

```
00401226   .^74 BE          JE SHORT Crackme1.004011E6
```

Es por qué esto lo que hace es ir a la dirección de arriba del ABOUT, cuando EAX valga 0, esto no nos sirve, pero hay un CALL debajo, que es lo siguiente que he rodeado:

0040122D . E8 4C010000 CALL Crackme1.0040137E

Seguro que aquí ya hay algo interesante, vamos a poner un breakpoint, para eso nos ponemos encima pinchando con el clic izquierdo y damos a F2, se nos tiene que poner en rojo en la columna de direcciones, también podemos ponerlo con: botón derecho sobre la instrucción: Breakpoint – Toggle, esto lo haremos para ver que pasa de esa instrucción en adelante, así que ahora damos a F9 o al PLAY que es lo mismo y se nos abrirá el ejecutable (lo vemos abajo a la derecha) pinchamos y nos intentamos registrar con el Name y Serial que queramos, yo meteré Name Ceroequis y Serial: playmovil. Y cuando damos a OK, se nos minimiza el ejecutable, eso es que el breakpoint ha hecho efecto, ahora vamos al OllyDbg y vemos que estamos sobre la dirección:

0040122D . E8 4C010000 CALL Crackme1.0040137E

Pero si nos fijamos arriba:

00401228 . 68 8E214000 PUSH Crackme1.0040218E ; ASCII "Ceroequis"

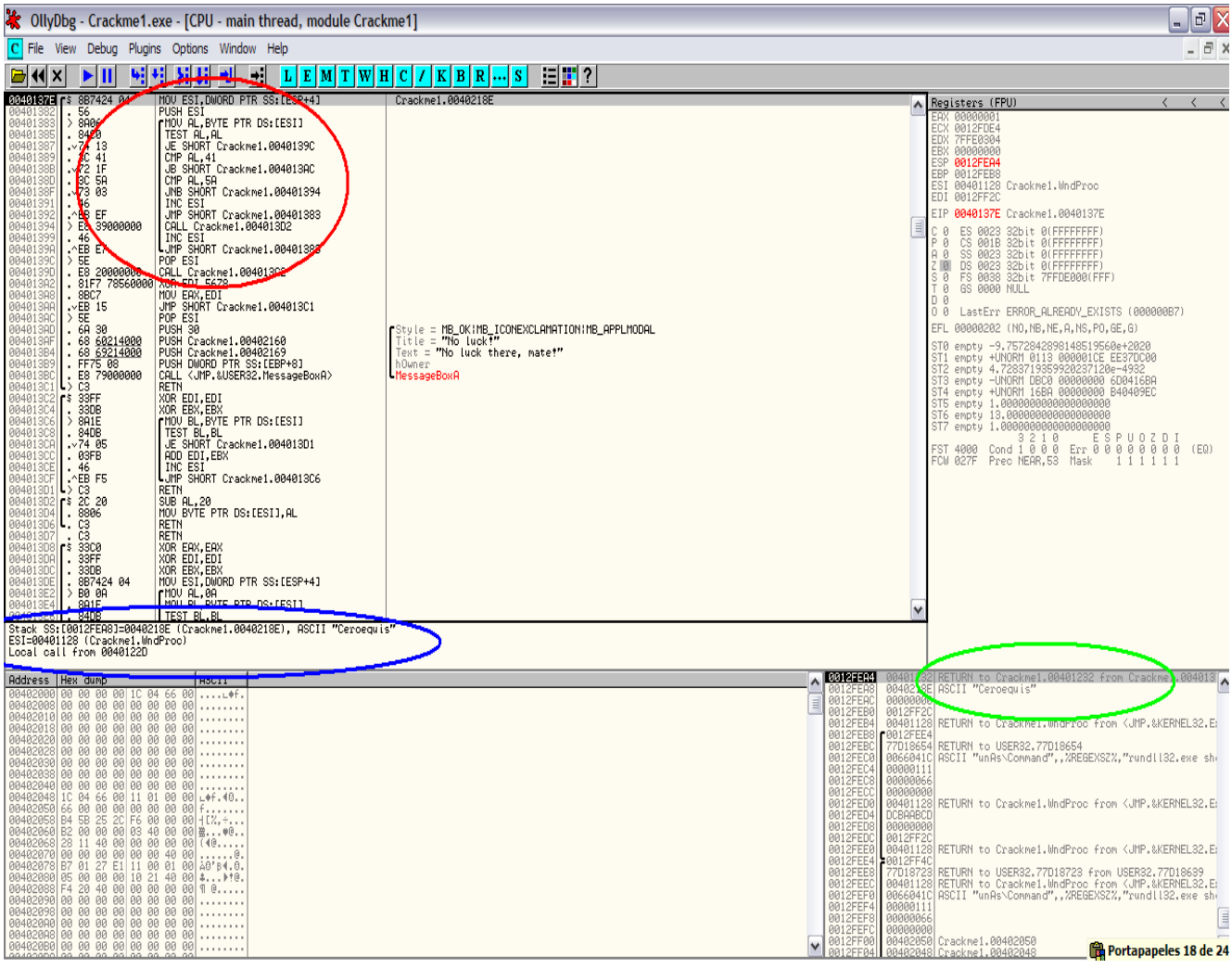
Pone en la pila el Name que hemos metido, ahora debemos saber:

¿Qué es "tracear"? Es seguir la ejecución del programa pero la controlaremos nosotros, hay dos formas de "tracear" básicas:

Step into (F7): Recorremos instrucción por instrucción entrando en todos los saltos.

Step over (F8): Recorremos instrucción por instrucción pero no entramos en los saltos, esto es útil cuando estamos depurando un programa y hace una llamada a una API, y lo que hace en esa llamada no nos interesa, pues gracias a step over no entramos y saltamos a la siguiente instrucción. Pero hay que tener claro que la llamada si se ejecuta, lo que pasa es que no la recorremos "por dentro".

9) Estamos sobre el breakpoint y tenemos que empezar a tracear, y como queremos entrar en la llamada (CALL) pues apretamos F7:



Como vemos aparecemos aquí, vamos a ver que es lo que hemos rodeado, lo que está en rojo:

```
0040137E  /$ 8B7424 04      MOV ESI,DWORD PTR SS:[ESP+4]          ;  Crackme1.0040218E
00401382  |. 56             PUSH ESI
                                                                  _
00401383  |> 8A06           /MOV AL,BYTE PTR DS:[ESI]             |
00401385  |. 84C0           |TEST AL,AL                           |
00401387  |. 74 13          |JE SHORT Crackme1.0040139C           |
00401389  |. 3C 41          |CMP AL,41                            |
0040138B  |. 72 1F          |JB SHORT Crackme1.004013AC           |
0040138D  |. 3C 5A          |CMP AL,5A                            |
0040138F  |. 73 03          |JNB SHORT Crackme1.00401394          | >  Esto es un Bucle.
00401391  |. 46             |INC ESI                              |
00401392  |.^EB EF          |JMP SHORT Crackme1.00401383          |
00401394  |> E8 39000000    |CALL Crackme1.004013D2               |
00401399  |. 46             |INC ESI                              |
0040139A  |.^EB E7          \JMP SHORT Crackme1.00401383          |
                                                                 _|
0040139C  |> 5E             POP ESI
0040139D  |. E8 20000000    CALL Crackme1.004013C2
```

Como vemos el OllyDbg nos marca el bucle con una linea negra a la izquierda del código, esto nos facilita las cosas para verlos. Esto lo primero que hace es:

> Mover a ESI un valor, y podemos saber lo que es, gracias a que en la parte de abajo, donde puse el círculo Azul sale:
>
> Stack SS:[0012FEA8]= (Crackme1.0040218E), ASCII “Ceroequis”
>
> Pues muy bien, ahora ESI apunta a donde está nuestro Name.
> Luego pone ESI en la pila (PUSH ESI).
> Y comienza el blucle.

Ahora vamos a ver lo que el bucle, pero antes damos una vez a F7 y nos ponemos sobre el PUSH, si nos fijamos a la derecha en la ventana de registro pone: ESI 0040218E ASCII “Ceroequis”. Damos a F7 otra vez y nos ponemos al principio del bucle, pero como vemos la pila se ha modificado al dar a F7 ya que se ejecutó la instrucción PUSH (la pila está donde puse el círculo verde). El bucle hace:

Mueve a AL un BYTE que está en DS:[ESI] ; Para saber que valor es damos a F7 una vez. Como vemos Abajo (donde antes nos ponía lo de Stack SS… (cículo azul)), nos pone: AL=43 (C), es decir, que ahora AL vale 43 y que en ASCII el número 43 vale C, y es la primera letra del Name, ya que ESI apuntaba a nuestro name y esto es que apunta a la primera letra de Ceroequis que es la C.

TEST AL, AL ; Hacemos un AND pero solo alteramos los flags, damos a F7 para ver que pasa, como vemos en los Flags que salen debajo de los registros, no se han modificado.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este no es el caso pues Nada, damos a F7.

CMP AL, 41 ; Compara AL con el número 41, es decir resta a AL – 41, pero solo modificando los flags, para ver que Pasa (aunque ya lo sabemos por que sabemos que AL vale 43), damos a F7, como suponíamos ningún Flag ha cambiado.

JB SHORT Crackme1.004013AC : Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el AF, CF y el SF se han puesto a 1 (se ponen en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este no es caso así que no Entrará, damos a F7 para ver que no entra.

INC ESI ; Incrementa en 1 a ESI, es decir ESI apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

Ya hemos pasado nuestra primera repetición por el bucle, como vemos no ha pasado absolutamente nada, aunque claro el lector debe saber que si a metido otro Name puede haber pasado, ya que como vemos hay varias operaciones con la primera letra y podría haber entrado en algún salto condicional.

Ahora vamos a ver la segunda repetición:

MOV AL,BYTE PTR DS:[ESI] ; Volvemos a estar en el principio, esto mueve a AL lo que haya en DS[ESI], Para ver lo que es miramos abajo (donde puse el círculo azul) y vemos: DS[0040218F]=65 (‘e’) <- Esto es lo que hay en DS[ESI], que es la segunda letra del Name que metimos.

AL=43 ('C') <- Esto es lo que vale AL antes de hacer el MOV, que es la primera letra del Name, como vimos en la primera repetición, damos a F7.

TEST AL, AL ; Ahora que AL vale lo que hay en la segunda letra del Name (65) o lo que es lo mismo la letra 'e' hace un TEST, para ver que pasa apretamos F7. Como vemos se modifican los flags CF poniéndose a 0 y PF poniéndose a 1.

JE SHORT Crackme1.0040139C ; Entrará en la dirección 0040139C si el ZF está a 1, este no es el caso así que F7.

CMP AL, 41 ; Resta AL – 41, damos a F7 para ver que pasa, aunque ya lo sabemos, no pasa nada.

JB SHORT Crackme1.004013AC ; Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el AF se ha puesto a 1 y PF se ha puesto a 0 (se ponen en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este es el caso así que Entrará, damos a F7 para ver que entra.

CALL Crackme1.004013D2 ; Hay una CALL a 004013D2, damos a F7 para entrar.

SUB AL,20 ; Resta AL – 20, guardando el resultado en AL, damos a F7 para ver el resultado, como vemos donde puse El círculo azul, sale: AL=45 ('E'), es decir ahora AL tiene el 45 que en ASCII es la letra E en mayúscula, así que lo que ha hecho es pasar la 'e' minúscula a mayúscula.

MOV BYTE PTR DS:[ESI], AL ; Mueve a DS[ESI] el resultado de AL, como podemos ver donde puse el círculo azul DS[0040218F] que es lo mismo que DS[ESI], es dónde está la letra 'e' y AL vale E, Damos a F7 para que se haga.

RETN ; Volvemos a la instrucción que había debajo del CALL que la llamó, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la siguiente letra del Name, damos a F7.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

En la segunda repetición del bucle como podemos ver lo que ha hecho es pasar la 2º letra del Name que metimos: Ceroequis en este caso la 'e' a mayúscula y se volverá a repetir, parece ser que este bucle lo que hace es dejar las mayúsculas como están y las minúsculas las pasa a mayúsculas, ahora donde está el Name hay: CEroequis.

Ahora vamos a ver la tercera repetición:

MOV AL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a AL, como vemos donde puse el círculo azul DS[00402190]=72 ('r'), así que cuando demos a F7 AL valdrá 72 que en ASCII Es la letra r, pues damos a F7.

TEST AL, AL ; Ahora que AL vale lo que hay en la tercera letra del Name (72) o lo que es lo mismo la letra 'r' hace un TEST, para ver que pasa apretamos F7. Como vemos se modifica el flag AF poniéndose a 0.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este no es el caso pues Nada, damos a F7.

CMP AL, 41 ; Compara AL con el número 41, es decir resta a AL – 41, pero solo modificando los flags, para ver que Pasa (aunque ya lo sabemos por que sabemos que AL vale 72), damos a F7, como suponíamos el flag PF Flag ha cambiado a 0.

JB SHORT Crackme1.004013AC : Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el PF y el AF se han puesto a 1 (se ponen en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este es el caso así que Entrará, damos a F7 para ver que entra.

CALL Crackme1.004013D2 ; Hay una CALL a 004013D2, damos a F7 para entrar.

SUB AL,20 ; Resta AL – 20, guardando el resultado en AL, damos a F7 para ver el resultado, como vemos donde puse El círculo azul, sale: AL=52 ('R'), es decir ahora AL tiene el 52 que en ASCII es la letra R en mayúscula, así que lo que ha hecho es pasar la 'r' minúscula a mayúscula.

MOV BYTE PTR DS:[ESI], AL ; Mueve a DS[ESI] el resultado de AL, como podemos ver donde puse el círculo azul DS[00402190] que es lo mismo que DS[ESI], es dónde está la letra 'r' y AL vale R, Damos a F7 para que se haga.

RETN ; Volvemos a la instrucción que había debajo del CALL que la llamó, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la siguiente letra del Name, damos a F7.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

En la tercera repetición del bucle como podemos ver lo que ha hecho es pasar la 3º letra del Name que metimos: Ceroequis en este caso la 'r' a mayúscula y se volverá a repetir, parece ser que este bucle lo que hace es dejar las mayúsculas como están y las minúsculas las pasa a mayúsculas, ahora donde está el Name hay: CERoequis.

Ahora vamos a ver la cuarta repetición:

MOV AL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a AL, como vemos donde puse el círculo azul DS[00402191]=6F ('o'), así que cuando demos a F7 AL valdrá 6F que en ASCII Es la letra o, pues damos a F7.

TEST AL, AL ; Ahora que AL vale lo que hay en la cuarta letra del Name (6F) o lo que es lo mismo la letra 'o' hace un TEST, para ver que pasa apretamos F7. Como vemos se modifica el flag PF poniéndose a 1.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este no es el caso pues Nada, damos a F7.

CMP AL, 41 ; Compara AL con el número 41, es decir resta a AL – 41, pero solo modificando los flags, para ver que Pasa (aunque ya lo sabemos por que sabemos que AL vale 6F), damos a F7, como suponíamos no ha cambiado nada.

JB SHORT Crackme1.004013AC : Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el PF ha puesto a 0 (se pone en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este es el caso así que Entrará, damos a F7 para ver que entra.

CALL Crackme1.004013D2 ; Hay una CALL a 004013D2, damos a F7 para entrar.

SUB AL,20 ; Resta AL – 20, guardando el resultado en AL, damos a F7 para ver el resultado, como vemos donde puse El círculo azul, sale: AL=4F ('O'), es decir ahora AL tiene el 4F que en ASCII es la letra O en mayúscula, así que lo que ha hecho es pasar la 'o' minúscula a mayúscula.

MOV BYTE PTR DS:[ESI], AL ; Mueve a DS[ESI] el resultado de AL, como podemos ver donde puse el círculo azul DS[00402191] que es lo mismo que DS[ESI], es dónde está la letra 'o' y AL vale O, Damos a F7 para que se haga.

RETN ; Volvemos a la instrucción que había debajo del CALL que la llamó, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la siguiente letra del Name, damos a F7.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

En la cuarta repetición del bucle como podemos ver lo que ha hecho es pasar la 4º letra del Name que metimos: Ceroequis en este caso la 'o' a mayúscula y se volverá a repetir, parece ser que este bucle lo que hace es dejar las mayúsculas como están y las minúsculas las pasa a mayúsculas, ahora donde está el Name hay: CEROequis.

Ahora vamos a ver la quinta repetición:

MOV AL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a AL, como vemos donde puse el círculo azul DS[00402192]=65 ('e'), así que cuando demos a F7 AL valdrá 65 que en ASCII Es la letra e, pues damos a F7.

TEST AL, AL ; Ahora que AL vale lo que hay en la quinta letra del Name (65) o lo que es lo mismo la letra 'e' hace un TEST, para ver que pasa apretamos F7. Como vemos se modifica el flag PF poniéndose a 1.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este no es el caso pues Nada, damos a F7.

CMP AL, 41 ; Compara AL con el número 41, es decir resta a AL – 41, pero solo modificando los flags, para ver que Pasa (aunque ya lo sabemos por que sabemos que AL vale 65), damos a F7, como suponíamos no ha cambiado nada.

JB SHORT Crackme1.004013AC : Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el PF ha puesto a 0 y el AF a 1 (se ponen en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este es el caso así que Entrará, damos a F7 para ver que entra.

CALL Crackme1.004013D2 ; Hay una CALL a 004013D2, damos a F7 para entrar.

SUB AL,20 ; Resta AL – 20, guardando el resultado en AL, damos a F7 para ver el resultado, como vemos donde puse El círculo azul, sale: AL=45 ('E'), es decir ahora AL tiene el 45 que en ASCII es la letra E en mayúscula, así que lo que ha hecho es pasar la 'e' minúscula a mayúscula.

MOV BYTE PTR DS:[ESI], AL ; Mueve a DS[ESI] el resultado de AL, como podemos ver donde puse el círculo azul DS[00402192] que es lo mismo que DS[ESI], es dónde está la letra 'e' y AL vale E, Damos a F7 para que se haga.

RETN ; Volvemos a la instrucción que había debajo del CALL que la llamó, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la siguiente letra del Name, damos a F7.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

En la quinta repetición del bucle como podemos ver lo que ha hecho es pasar la 5º letra del Name que metimos: Ceroequis en este caso la 'e' a mayúscula y se volverá a repetir, parece ser que este bucle lo que hace es dejar las mayúsculas como están y las minúsculas las pasa a mayúsculas, ahora donde está el Name hay: CEROEquis.

Ahora vamos a ver la sexta repetición:

MOV AL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a AL, como vemos donde puse el círculo azul DS[00402193]=71 ('q'), así que cuando demos a F7 AL valdrá 71 que en ASCII Es la letra q, pues damos a F7.

TEST AL, AL ; Ahora que AL vale lo que hay en la sexta letra del Name (71) o lo que es lo mismo la letra 'q' hace un TEST, para ver que pasa apretamos F7. Como vemos no se modifica ningún flag.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este no es el caso pues Nada, damos a F7.

CMP AL, 41 ; Compara AL con el número 41, es decir resta a AL – 41, pero solo modificando los flags, para ver que Pasa (aunque ya lo sabemos por que sabemos que AL vale 71), damos a F7, como suponíamos no ha cambiado nada.

JB SHORT Crackme1.004013AC : Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el el AF a 1 (se ponen en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este es el caso así que Entrará, damos a F7 para ver que entra.

CALL Crackme1.004013D2 ; Hay una CALL a 004013D2, damos a F7 para entrar.

SUB AL,20 ; Resta AL – 20, guardando el resultado en AL, damos a F7 para ver el resultado, como vemos donde puse El círculo azul, sale: AL=51 ('Q'), es decir ahora AL tiene el 51 que en ASCII es la letra Q en mayúscula, así que lo que ha hecho es pasar la 'q' minúscula a mayúscula.

MOV BYTE PTR DS:[ESI], AL ; Mueve a DS[ESI] el resultado de AL, como podemos ver donde puse el círculo azul DS[00402193] que es lo mismo que DS[ESI], es dónde está la letra 'q' y AL vale Q, Damos a F7 para que se haga.

RETN ; Volvemos a la instrucción que había debajo del CALL que la llamó, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la siguiente letra del Name, damos a F7.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

En la sexta repetición del bucle como podemos ver lo que ha hecho es pasar la 7º letra del Name que metimos: Ceroequis en este caso la 'q' a mayúscula y se volverá a repetir, parece ser que este bucle lo que hace es dejar las mayúsculas como están y las minúsculas las pasa a mayúsculas, ahora donde está el Name hay: CEROEQuis.

Ahora vamos a ver la octava repetición:

MOV AL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a AL, como vemos donde puse el círculo azul DS[00402194]=75 ('u'), así que cuando demos a F7 AL valdrá 75 que en ASCII Es la letra u, pues damos a F7.

TEST AL, AL ; Ahora que AL vale lo que hay en la octava letra del Name (75) o lo que es lo mismo la letra 'u' hace un TEST, para ver que pasa apretamos F7. Como vemos no se modifica ningún flag.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este no es el caso pues Nada, damos a F7.

CMP AL, 41 ; Compara AL con el número 41, es decir resta a AL – 41, pero solo modificando los flags, para ver que Pasa (aunque ya lo sabemos por que sabemos que AL vale 75), damos a F7, como suponíamos no ha cambiado nada.

JB SHORT Crackme1.004013AC : Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el AF y el PF se han puesto a 1 (se ponen en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este es el caso así que Entrará, damos a F7 para ver que entra.

CALL Crackme1.004013D2 ; Hay una CALL a 004013D2, damos a F7 para entrar.

SUB AL,20 ; Resta AL – 20, guardando el resultado en AL, damos a F7 para ver el resultado, como vemos donde puse El círculo azul, sale: AL=55 ('U'), es decir ahora AL tiene el 55 que en ASCII es la letra U en mayúscula, así que lo que ha hecho es pasar la 'u' minúscula a mayúscula.

MOV BYTE PTR DS:[ESI], AL ; Mueve a DS[ESI] el resultado de AL, como podemos ver donde puse el círculo azul DS[00402194] que es lo mismo que DS[ESI], es dónde está la letra 'u' y AL vale U, Damos a F7 para que se haga.

RETN ; Volvemos a la instrucción que había debajo del CALL que la llamó, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la siguiente letra del Name, damos a F7.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

En la octava repetición del bucle como podemos ver lo que ha hecho es pasar la 8º letra del Name que metimos: Ceroequis en este caso la 'u' a mayúscula y se volverá a repetir, parece ser que este bucle lo que hace es dejar las mayúsculas como están y las minúsculas las pasa a mayúsculas, ahora donde está el Name hay: CEROEQUis.

Ahora vamos a ver la novena repetición:

MOV AL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a AL, como vemos donde puse el círculo azul DS[00402195]=69 ('i'), así que cuando demos a F7 AL valdrá 69 que en ASCII Es la letra i, pues damos a F7.

TEST AL, AL ; Ahora que AL vale lo que hay en la novena letra del Name (69) o lo que es lo mismo la letra 'i' hace un TEST, para ver que pasa apretamos F7. Como vemos no se modifica ningún flag.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este no es el caso pues Nada, damos a F7.

CMP AL, 41 ; Compara AL con el número 41, es decir resta a AL – 41, pero solo modificando los flags, para ver que Pasa (aunque ya lo sabemos por que sabemos que AL vale 69), damos a F7, como suponíamos no ha cambiado nada.

JB SHORT Crackme1.004013AC : Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el AF se ha puesto a 1 (se pone en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este es el caso así que Entrará, damos a F7 para ver que entra.

CALL Crackme1.004013D2 ; Hay una CALL a 004013D2, damos a F7 para entrar.

SUB AL,20 ; Resta AL – 20, guardando el resultado en AL, damos a F7 para ver el resultado, como vemos donde puse El círculo azul, sale: AL=49 ('I'), es decir ahora AL tiene el 49 que en ASCII es la letra I en mayúscula, así que lo que ha hecho es pasar la 'i' minúscula a mayúscula.

MOV BYTE PTR DS:[ESI], AL ; Mueve a DS[ESI] el resultado de AL, como podemos ver donde puse el círculo azul DS[00402195] que es lo mismo que DS[ESI], es dónde está la letra 'i' y AL vale I, Damos a F7 para que se haga.

RETN ; Volvemos a la instrucción que había debajo del CALL que la llamó, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la siguiente letra del Name, damos a F7.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

En la novena repetición del bucle como podemos ver lo que ha hecho es pasar la 9º letra del Name que metimos: Ceroequis en este caso la 'i' a mayúscula y se volverá a repetir, parece ser que este bucle lo que hace es dejar las mayúsculas como están y las minúsculas las pasa a mayúsculas, ahora donde está el Name hay: CEROEQUIs.

Ahora vamos a ver la décima repetición:

MOV AL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a AL, como vemos donde puse el círculo azul DS[00402196]=73 ('s'), así que cuando demos a F7 AL valdrá 73 que en ASCII Es la letra s, pues damos a F7.

TEST AL, AL ; Ahora que AL vale lo que hay en la décima letra del Name (73) o lo que es lo mismo la letra 's' hace un TEST, para ver que pasa apretamos F7. Como vemos se modifica el flag PF poniéndose a 0.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este no es el caso pues Nada, damos a F7.

CMP AL, 41 ; Compara AL con el número 41, es decir resta a AL – 41, pero solo modificando los flags, para ver que Pasa (aunque ya lo sabemos por que sabemos que AL vale 73), damos a F7, como suponíamos no ha cambiado nada.

JB SHORT Crackme1.004013AC : Entra a la dirección 004013AC si CF está a 1, como este no es el caso no entrará, damos a F7.

CMP AL, 5A ; Compara AL con 5A, damos a F7 para ver que pasa, cuando lo hacemos vemos en la ventana de Registros que el AF se ha puesto a 1 (se pone en rojo además).

JNB SHORT Crackme1.00401394 ; Entrará en la dirección 00401394 si CF es 0 y ZF es 0, este es el caso así que Entrará, damos a F7 para ver que entra.

CALL Crackme1.004013D2 ; Hay una CALL a 004013D2, damos a F7 para entrar.

SUB AL,20 ; Resta AL – 20, guardando el resultado en AL, damos a F7 para ver el resultado, como vemos donde puse El círculo azul, sale: AL=53 ('S'), es decir ahora AL tiene el 63 que en ASCII es la letra S en mayúscula, así que lo que ha hecho es pasar la 's' minúscula a mayúscula.

MOV BYTE PTR DS:[ESI], AL ; Mueve a DS[ESI] el resultado de AL, como podemos ver donde puse el círculo azul DS[00402196] que es lo mismo que DS[ESI], es dónde está la letra 's' y AL vale S, Damos a F7 para que se haga.

RETN ; Volvemos a la instrucción que había debajo del CALL que la llamó, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la siguiente letra del Name, pero no hay más, damos a F7.

JMP SHORT Crackme1.00401383 ; Entrará siempre en la dirección 004001383, es decir al principio del bucle, damos a F7 para llegar.

En la décima repetición del bucle como podemos ver lo que ha hecho es pasar la 10º letra del Name que metimos: Ceroequis en este caso la 's' a mayúscula y se volverá a repetir, parece ser que este bucle lo que hace es dejar las mayúsculas como están y las minúsculas las pasa a mayúsculas, ahora donde está el Name hay: CEROEQUIS.

Final del bucle:

MOV AL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a AL, como vemos donde puse el círculo azul DS[00402197]=00 , así que cuando demos a F7 AL valdrá 0, pues damos a F7.

TEST AL, AL ; Como AL vale 0 este TEST pondrá PF y ZF a 1, así que damos a F7 para verlo.

JE SHORT Crackme1.0040139C ; Entra en la dirección 0040139C si el ZF está activado, como este es el caso pues entrará, damos a F7.

Ya estamos en la dirección 0040139C, es decir fuera del bucle:

POP ESI ; Saca el último valor metido en la pila poniéndolo en ESI, recordemos que era el Name Pero ahora está en mayúsculas todo, damos a F7.

CALL Crackme1.004013C2 ; Entra a la dirección 004013C2, damos a F7.

XOR EDI, EDI ; Pone EDI a 0, damos a F7.

XOR EBX, EBX ; Pone EBX a 0, damos a F7.

Ahora hemos entrado a otro bucle, veamos lo que hace:

```
004013C6  |> 8A1E          /MOV BL,BYTE PTR DS:[ESI]
004013C8  |. 84DB          |TEST BL,BL
004013CA  |. 74 05         |JE SHORT Crackme1.004013D1
004013CC  |. 03FB          |ADD EDI,EBX
004013CE  |. 46            |INC ESI
004013CF  |.^EB F5         \JMP SHORT Crackme1.004013C6
```

Ahora vamos a ver las repeticiones, 1º repetición:

MOV BL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul DS[0040218E]=43 (C) , así que cuando demos a F7 BL valdrá 43, que en ASCII es la letra C, recordemos que ESI apunta a la primera letra del Name , pues damos a F7

TEST BL, BL ; Ahora que BL vale lo que hay en la primera letra del Name (43) o lo que es lo mismo la letra 'C' hace un TEST, para ver que pasa apretamos F7. Como vemos se modifica el flag PF y ZF poniéndose a 0.

JE SHORT Crackme1.004013D1 ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues No entrará, damos a F7.

ADD EDI, EBX ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el Círculo azul EBX=00000043, EDI=00000000, damos a F7.

INC ESI ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6 ; Salta al inicio del bucle.

Como podemos ver lo que esto hace es sumar la 1º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso 0, guardándolo en EDI, así que al final de esta repetición EDI vale 43, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Ahora vamos a ver la 2º repetición:

MOV BL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul DS[0040218F]=45 (E) , así que cuando demos a F7 BL valdrá 45, que en ASCII es la letra E, recordemos que ESI apunta a la segunda letra del Name , pues damos a F7

TEST BL, BL ; Ahora que BL vale lo que hay en la segunda letra del Name (45) o lo que es lo mismo la letra 'E' hace un TEST, para ver que pasa apretamos F7. Como vemos no se modifica ningún flag.

JE SHORT Crackme1.004013D1 ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues No entrará, damos a F7.

ADD EDI, EBX ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el Círculo azul EBX=00000045, EDI=00000043, damos a F7.

INC ESI ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6 ; Salta al inicio del bucle.

Como podemos ver lo que esto hace es sumar la 2º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso 43, guardándolo en EDI, así que al final de esta repetición EDI vale 00000088,
Lo podemos ver en la ventana de registros, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Ahora vamos a ver la 3º repetición:

MOV BL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul DS[00402190]=52 (R) , así que cuando demos a F7 BL valdrá 52, que en ASCII es la letra R, recordemos que ESI apunta a la tercera letra del Name , pues damos a F7

TEST BL, BL ; Ahora que BL vale lo que hay en la tercera letra del Name (52) o lo que

es lo mismo la letra 'R' hace un TEST, para ver que pasa apretamos F7. Como vemos no se modifica ningún flag.

JE SHORT Crackme1.004013D1 ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues No entrará, damos a F7.

ADD EDI, EBX ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el Círculo azul EBX=00000052, EDI=00000088, damos a F7.

INC ESI ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6 ; Salta al inicio del bucle.

Como podemos ver lo que esto hace es sumar la 3º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso 88, guardándolo en EDI, así que al final de esta repetición EDI vale 000000DA, Lo podemos ver en la ventana de registros, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Ahora vamos a ver la 4º repetición:

MOV BL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul DS[00402191]=4F (O) , así que cuando demos a F7 BL valdrá 4F, que en ASCII es la letra O, recordemos que ESI apunta a la cuarta letra del Name , pues damos a F7

TEST BL, BL ; Ahora que BL vale lo que hay en la tercera letra del Name (4F) o lo que es lo mismo la letra 'O' hace un TEST, para ver que pasa apretamos F7. Como vemos no se modifica ningún flag.

JE SHORT Crackme1.004013D1 ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues No entrará, damos a F7.

ADD EDI, EBX ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el Círculo azul EBX=0000004F, EDI=000000DA, damos a F7.

INC ESI ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6 ; Salta al inicio del bucle.

Como podemos ver lo que esto hace es sumar la 4º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso DA, guardándolo en EDI, así que al final de esta repetición EDI vale 00000129, Lo podemos ver en la ventana de registros, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Ahora vamos a ver la 5º repetición:

MOV BL, BYTE PTR DS:[ESI] ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul DS[00402192]=45 (E) , así que cuando demos a F7 BL valdrá 45, que en ASCII es la letra E, recordemos que ESI apunta a la quinta letra del Name , pues damos a F7

TEST BL, BL ; Ahora que BL vale lo que hay en la quinta letra del Name (45) o lo que es lo mismo la letra 'E' hace un TEST, para ver que pasa apretamos F7. Como vemos no se modifica ningún flag.

JE SHORT Crackme1.004013D1 ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues No entrará, damos a F7.

ADD EDI, EBX ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el Círculo azul EBX=00000045, EDI=00000129, damos a F7.

INC ESI ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6 ; Salta al inicio del bucle.

Como podemos ver lo que esto hace es sumar la 5º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso 129, guardándolo en EDI, así que al final de esta repetición EDI vale 0000016E, Lo podemos ver en la ventana de registros, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Ahora vamos a ver la 6º repetición:

```
MOV BL, BYTE PTR DS:[ESI]        ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul
                                   DS[00402193]=51 (Q) , así que cuando demos a F7 BL valdrá 51, que en
                                   ASCII es la letra Q, recordemos que ESI apunta a la sexta letra del
                                   Name , pues damos a F7

TEST BL, BL                      ; Ahora que BL vale lo que hay en la sexta letra del Name (51) o lo que
                                   es lo mismo la letra 'Q' hace un TEST, para ver que pasa apretamos F7.
                                   Como vemos se pone el flag PF a 0.

JE SHORT Crackme1.004013D1       ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues
                                   No entrará, damos a F7.

ADD EDI, EBX                     ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el
                                   Círculo azul EBX=00000051, EDI=0000016E, damos a F7.

INC ESI                          ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6      ; Salta al inicio del bucle.
```

Como podemos ver lo que esto hace es sumar la 6º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso 16E, guardándolo en EDI, así que al final de esta repetición EDI vale 000001BF, Lo podemos ver en la ventana de registros, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Ahora vamos a ver la 7º repetición:

```
MOV BL, BYTE PTR DS:[ESI]        ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul
                                   DS[00402194]=55 (U) , así que cuando demos a F7 BL valdrá 55, que en
                                   ASCII es la letra U, recordemos que ESI apunta a la séptima letra del
                                   Name , pues damos a F7

TEST BL, BL                      ; Ahora que BL vale lo que hay en la séptima letra del Name (55) o lo que
                                   es lo mismo la letra 'U' hace un TEST, para ver que pasa apretamos F7.
                                   Como vemos se pone el flag PF a 1.

JE SHORT Crackme1.004013D1       ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues
                                   No entrará, damos a F7.

ADD EDI, EBX                     ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el
                                   Círculo azul EBX=00000055, EDI=000001BF, damos a F7.

INC ESI                          ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6      ; Salta al inicio del bucle.
```

Como podemos ver lo que esto hace es sumar la 7º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso 1BF, guardándolo en EDI, así que al final de esta repetición EDI vale 00000214, Lo podemos ver en la ventana de registros, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Ahora vamos a ver la 8º repetición:

```
MOV BL, BYTE PTR DS:[ESI]        ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul
                                   DS[00402195]=49 (I) , así que cuando demos a F7 BL valdrá 49, que en
                                   ASCII es la letra I, recordemos que ESI apunta a la octava letra del
                                   Name , pues damos a F7

TEST BL, BL                      ; Ahora que BL vale lo que hay en la octava letra del Name (49) o lo que
                                   es lo mismo la letra 'I' hace un TEST, para ver que pasa apretamos F7.
                                   Como vemos se pone el flag PF a 0.

JE SHORT Crackme1.004013D1       ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues
                                   No entrará, damos a F7.

ADD EDI, EBX                     ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el
                                   Círculo azul EBX=00000049, EDI=00000214, damos a F7.

INC ESI                          ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6      ; Salta al inicio del bucle.
```

Como podemos ver lo que esto hace es sumar la 8º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso 214, guardándolo en EDI, así que al final de esta repetición EDI vale 0000025D, Lo podemos ver en la ventana de registros, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Ahora vamos a ver la 9º repetición:

```
MOV BL, BYTE PTR DS:[ESI]          ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul
                                     DS[00402196]=53 (S) , así que cuando demos a F7 BL valdrá 53, que en
                                     ASCII es la letra S, recordemos que ESI apunta a la novena letra del
                                     Name , pues damos a F7

TEST BL, BL                        ; Ahora que BL vale lo que hay en la novena letra del Name (53) o lo que
                                     es lo mismo la letra 'S' hace un TEST, para ver que pasa apretamos F7.
                                     Como vemos no pasa nada.

JE SHORT Crackme1.004013D1     ;Entra en la dirección 004013D1 si el ZF está activado, como este no es el caso pues
                                 No entrará, damos a F7.

ADD EDI, EBX                   ; Suma el valor de EDI + EBX guardando el resultado en EDI, como vemos donde puse el
                                 Círculo azul EBX=00000053, EDI=0000025D, damos a F7.

INC ESI                        ; ESI se incrementa en uno y apuntará a la siguiente letra del Name.

JMP SHORT Crackme1.004013C6    ; Salta al inicio del bucle.
```

Como podemos ver lo que esto hace es sumar la 9º letra del name que está en BL (que pone EBX, pero es lo mismo en este caso) con lo que hay en EDI en este caso 25D, guardándolo en EDI, así que al final de esta repetición EDI vale 000002B0, Lo podemos ver en la ventana de registros, al parecer lo que hace es recorrer letra a letra el Name que ahora tenemos en mayúsculas y sumarlas guardándolas en EDI.

Final del bucle, la 10º repetición:

```
MOV BL, BYTE PTR DS:[ESI]          ; Mueve el valor de DS[ESI] a BL, como vemos donde puse el círculo azul
                                     DS[00402197]=00 , así que cuando demos a F7 BL valdrá 00
                                     pues damos a F7.

TEST BL, BL                        ; Ahora que BL vale 0, veamos que pasa al hacer el TEST, damos a F7 y
                                     Como vemos el ZF y el PF se han puesto a 1.

JE SHORT Crackme1.004013D1     ;Entra en la dirección 004013D1 si el ZF está activado, como este es el caso pues
                                 entra, damos a F7.
```

*Comentarios: Los números que vemos en las instrucciones y en los registros en principio son números hexadecimal como se ha podido observar.

Resumiendo lo que hemos visto hasta ahora es:

Un primer bucle, que pasa las letras de Name a Mayúsculas de la siguiente forma:

```
Bucle:
00401383  |> 8A06          /MOV AL,BYTE PTR DS:[ESI]
00401385  |. 84C0          |TEST AL,AL
00401387  |. 74 13         |JE SHORT Crackme1.0040139C
00401389  |. 3C 41         |CMP AL,41
0040138B  |. 72 1F         |JB SHORT Crackme1.004013AC
0040138D  |. 3C 5A         |CMP AL,5A
0040138F  |. 73 03         |JNB SHORT Crackme1.00401394
00401391  |. 46            |INC ESI
00401392  |.^EB EF         |JMP SHORT Crackme1.00401383
00401394  |> E8 39000000   |CALL Crackme1.004013D2  -______
00401399  |. 46            |INC ESI                        |
0040139A  |.^EB E7         \JMP SHORT Crackme1.00401383    |
                                                           |
El sitio al que lleva el CALL Crackme1.004013D2  :      <--- |
004013D2  /$ 2C 20          SUB AL,20
004013D4  |. 8806           MOV BYTE PTR DS:[ESI],AL
```

Lo primero que hay que saber para comprender esto es:

Rango de letras ASCII minusculas en decimal: es de 97 a 122.
Rango de letras ASCII mayusculas en decimal: es de 65 a 90.

Rango de letras ASCII minusculas en hexadecimal: es de 61 a 7A.
Rango de letras ASCII mayusculas en hexadecimal: es de 41 a 5A.

Pasar de una letra minuscula a una mayuscula se le suma en decimal: Letra + 32
Pasar de una letra mayuscula a una minuscula se le resta en decimal: Letra - 32

Pasar una letra minuscula a una mayuscula se le suma en hexadecimal: Letra + 20
Pasar una letra mayuscula a una minuscula se le resta en hexadecimal: Letra - 20

¿Y como sé eso? Muy fácil he hecho un pequeño programa en C, para sacarlo:

```
#include <stdio.h>
#include <stdlib.h>

int main( void ) {
   printf( "\nRango de letras ASCII minusculas en decimal: es de %d a %d. \n", 'a', 'z' );
   printf( "Rango de letras ASCII mayusculas en decimal: es de %d a %d. \n", 'A', 'Z' );

   printf( "\nRango de letras ASCII minusculas en hexadecimal: es de %X a %X. \n", 'a', 'z' );
   printf( "Rango de letras ASCII mayusculas en hexadecimal: es de %X a %X. \n\n", 'A', 'Z' );


   printf( "\nPasar de una letra minuscula a una mayuscula se le suma en decimal: Letra + %d", 'a'-'A' );
   printf( "\nPasar de una letra mayuscula a una minuscula se le resta en decimal: Letra - %d\n", 'a'-'A' );

   printf( "\nPasar una letra minuscula a una mayuscula se le suma en hexadecimal: Letra + %x", 'a'-'A' );
   printf( "\nPasar una letra mayuscula a una minuscula se le resta en hexadecimal: Letra - %x\n\n", 'a'-'A' );

   system("pause");
   return 0;
}
```

Como podemos ver el Crackme hace esto:

```
004013D2  /$ 2C 20         SUB AL,20
004013D4  |. 8806           MOV BYTE PTR DS:[ESI],AL
```

En AL está el valor ASCII de la letra minúscula, y se le resta 20 en hexadecimal, el rango de las minúsculas estaba en: 97-122, solo hay que restarle 20 en hexadecimal para obtener la letra mayúscula, una vez hace la resta la guarda en AL y mueve el valor a la posición de memoria donde estaba la letra en minúscula.

Así que el bucle solo hace conversiones de minúsculas a mayúsculas del Name. Por lo cual si el Crackme forma la serial a partir del Name va a ser el mismo serial para ceroequis, que para Ceroequis, que para CEROEQUIS.

Una vez que ha pasado las letras del Name a mayúsculas va a otro bucle que lo que hace es sumar los ASCII del Name ya puesto en mayúsculas, guardando la suma en principio en EDI, es decir:

```
004013C6  |> 8A1E           /MOV BL,BYTE PTR DS:[ESI]
004013C8  |. 84DB           |TEST BL,BL
004013CA  |. 74 05          |JE SHORT Crackme1.004013D1
004013CC  |. 03FB           |ADD EDI,EBX
004013CE  |. 46             |INC ESI
004013CF  |.^EB F5          \JMP SHORT Crackme1.004013C6
```

Vamos a ver un ejemplo de qué es lo que hace:

El name es: CEROE, para saber los ASCII:

```
#include <stdio.h>
#include <stdlib.h>

int main( void ) {
   printf( "\nASCII de CEROE: C=%d E=%d R=%d O=%d E=%d\n\n", 'C', 'E', 'R', 'O','E' );


   system("pause");
   return 0;
}
```

ASCII de CEROE: C=67 E=69 R=82 O=79 E=69, bien sabiendo esto y viendo el algoritmo lo que hace es:

Sumar 0 (EDI) + 67 (EBX) Guardando el resultado en EDI, EDI vale: 67
Sumar 67 (EDI) + 69 (EBX) Guardando el resultado en EDI, EDI vale: 136

Sumar 136 (EDI) + 82 (EBX) Guardando el resultado en EDI, EDI vale: 218
Sumar 218 (EDI) + 79 (EBX) Guardando el resultado en EDI, EDI vale: 297
Sumar 297 (EDI) + 69 (EBX) Guardando el resultado en EDI, EDI vale: 366

Así que EDI acaba valiendo 366, la suma de los ASCII del Name en mayúsculas.

¿Sencillo no?

Pero aún nos falta algo, si hemos visto bien el primer bucle, nos habremos dado cuenta que no ha entrado en:

```
0040138B  |. 72 1F        |JB SHORT Crackme1.004013AC
```

Primero apuntamos la dirección de donde estamos: 004013D1, sale a la izquierda en una columna.

Para ver que hay en el JB por si acaso, vamos a ponernos sobre él con clic izquierdo y damos al Intro (Enter), como vemos ahora estamos en:

```
004013AC  |> 5E           POP ESI                         ;  Crackme1.0040218E
```

y justo debajo:

```
004013AD  |. 6A 30        PUSH 30                         ; /Style = MB_OK|MB_ICONEXCLAMATION|MB_APPLMODAL
004013AF  |. 68 60214000    PUSH Crackme1.00402160        ; |Title = "No luck!"
004013B4  |. 68 69214000    PUSH Crackme1.00402169        ; |Text = "No luck there, mate!"
004013B9  |. FF75 08      PUSH DWORD PTR SS:[EBP+8]       ; |hOwner
004013BC  |. E8 79000000    CALL <JMP.&USER32.MessageBoxA>  ; \MessageBoxA
```

Así que si ese JB se hubiera ejecutado iríamos a la zona de chico malo, recordemos cuando se ejecuta JB: cuando el CF está activado y este flag se activa cuando hay un desbordamiento y justo antes del JB hay un:

```
00401389  |. 3C 41        |CMP AL,41
0040138B  |. 72 1F        |JB SHORT Crackme1.004013AC
```

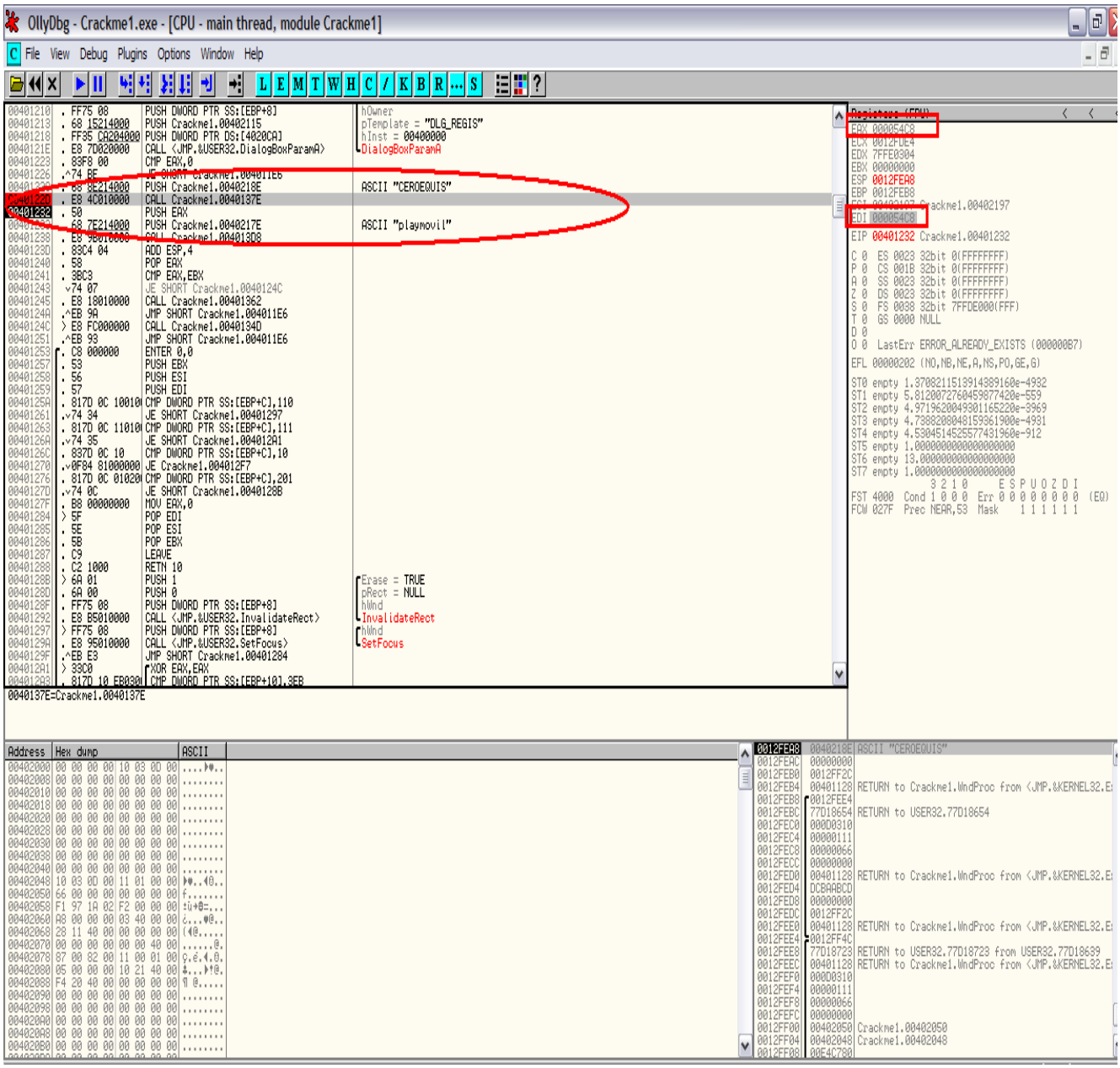
¿Y cuando hay un desbordamiento? Cuando en AL no hay una letra, lo comprobaremos al final.

Vamos a volver donde estábamos con: clic derecho en el centro de la pantalla grande: GO TO – Expression y metemos 004013D1 y damos a OK.

Ahora sigamos siguiendo el algoritmo, estamos en:

```
004013D1  \> C3           RETN                                ; Damos a F7.

004013A2  |. 81F7 78560000  XOR EDI, 5678                     ; Hace un XOR con EDI, que está la suma de los
                                                               ASCII del Name, con el número 5678, guardando
                                                              el resultado en EDI, damos a F7.

004013A8  |. 8BC7         MOV EAX,EDI                        ; Copia el valor de EDI a EAX.

004013AA  |. EB 15        JMP SHORT Crackme1.004013C1        ; Salta a la dirección 004013C1

004013C1  \> C3           RETN                                ; Damos a F7, y volvemos justo debajo del CALL
                                                              donde pusimos el breakpoint.
```

Ya estamos en la instrucción que hay debajo del 0040122D . E8 4C010000 CALL Crackme1.0040137E
Es decir, donde pusimos el breakpoint, podemos ver el breakpoint si subimos un poco más arriba:



Ahora EDI y EAX valen el XOR de la suma de los ASCII del name en mayúsculas, es decir: 000054C8, como podemos ver en la ventana de registros.

Ahora estamos sobre:

....
00401232 . 50 PUSH EAX
00401233 . 68 7E214000 PUSH Crackme1.0040217E ; ASCII "playmovil"
00401238 . E8 9B010000 CALL Crackme1.004013D8
....

Vamos a seguir traceando:

PUSH Crackme1.0040217E ; Pone en la pila EAX, que como sabemos es el XOR de la suma de los ASCII del name en mayúsculas, damos a F7 para que se haga.

PUSH Crackme1.0040217E ; Pone en la pila la dirección en la que está el Serial que metimos, damos a F7.

CALL Crackme1.004013D8 ; Entra a la dirección 004013D8, damos a F7 para entrar.

Ahora estamos:

```
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 Crackme1.004013F5                  |
004013EA  |. 80EB 30       |SUB BL,30                                   |
004013ED  |. 0FAFF8        |IMUL EDI,EAX                                |> Bucle.
004013F0  |. 03FB          |ADD EDI,EBX                                 |
004013F2  |. 46            |INC ESI                                     |
004013F3  |.^EB ED         \JMP SHORT Crackme1.004013E2                _|

004013F5  |> 81F7 34120000  XOR EDI,1234
004013FB  |. 8BDF          MOV EBX,EDI
004013FD  \. C3            RETN
```

Al ver que ha puesto lo del Serial en la pila, podemos suponer que aquí hace unas comprobaciones de la misma, vamos a ver lo que hace esta parte del código:

```
XOR EAX, EAX                        ;Pone EAX a 0, damos a F7.

XOR EDI, EDI                        ;Pone EDI a 0, damos a F7.

XOR EBX, EBX                        ;Pone EBX a 0, damos a F7.

MOV ESI, DWORD PTR SS:[ESP+4]       ;Como vemos donde puse el círculo azul: Stack SS:[0012FEA0]=0040217E,
                                    La parte que está entre corchetes es el valor de ESP+4, y lo que hay después:
                                    ASCII “playmovil”, es donde apunta ahora ESI, damos a F7.
```

Ahora empieza el bucle, 1º repetición:

```
MOV AL, 0A                          ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI]           ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo
                                      Azul: DS:[0040217E]=70 (‘p’), así que va a copiar el número 70 que en ASCII
                                    Equivale a la letra p, es decir, la primera letra del Serial, será copiada a BL, damos
                                    A F7.

TEST BL, BL                         ; Hace un TEST, damos a F7 para ver que pasa, como vemos el PF y el ZF se han
                                    puesto a 0.

JE SHORT Crackme1.004013F5          ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.

SUB BL, 30                          ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 70,
                                    damos a F7.

IMUL EDI, EAX                       ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A,
                                    EDI=00000000, damos a F7.

ADD EDI, EBX                        ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=00000040 y
                                    EDI=00000000, damos a F7.

INC ESI                             ; Incrementa ESI en 1, para que apunte a la segunda letra del Serial, damos a F7.

JMP SHORT Crackme1.004013E2         ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.
```

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX que estaban a 0, como podemos ver en los XOR de arriba del bucle, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=00000040 EBX=00000040

Lo podemos ver en la ventana de registros.

Vamos a ver la 2º repetición:

```
MOV AL, 0A                          ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI]           ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo
                                      Azul: DS:[0040217F]=6C (‘l’), así que va a copiar el número 6C que en ASCII
                                    Equivale a la letra l, es decir, la segunda letra del Serial, será copiada a BL, damos
                                    A F7.

TEST BL, BL                         ; Hace un TEST, damos a F7 para ver que pasa, como vemos el PF se ha
                                    puesto a 1.

JE SHORT Crackme1.004013F5          ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.
```

```
SUB BL, 30                    ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 6C,
                              damos a F7.

IMUL EDI, EAX                  ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A,
                               EDI=00000040, damos a F7.

ADD EDI, EBX                   ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=0000003C y
                               EDI=00000280, damos a F7.

INC ESI                       ; Incrementa ESI en 1, para que apunte a la tercera letra del Serial, damos a F7.

JMP SHORT Crackme1.004013E2   ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.
```

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=000002BC EBX=0000003C

Lo podemos ver en la ventana de registros.

Vamos a ver la 3º repetición:

```
MOV AL, 0A                    ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI]      ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo
                               Azul: DS:[00402180]=61 (‘1’), así que va a copiar el número 6A que en ASCII
                               Equivale a la letra a, es decir, la tercera letra del Serial, será copiada a BL, damos
                               A F7.

TEST BL, BL                   ; Hace un TEST, damos a F7 para ver que pasa, como vemos el AF se ha
                              puesto a 1.

JE SHORT Crackme1.004013F5    ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.

SUB BL, 30                    ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 61,
                              damos a F7.

IMUL EDI, EAX                  ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A,
                               EDI=000002BC, damos a F7.

ADD EDI, EBX                   ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=00000031 y
                               EDI=00001B58, damos a F7.

INC ESI                       ; Incrementa ESI en 1, para que apunte a la cuarta letra del Serial, damos a F7.

JMP SHORT Crackme1.004013E2   ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.
```

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=00001B89 EBX=00000031

Lo podemos ver en la ventana de registros.

Vamos a ver la 4º repetición:

```
MOV AL, 0A                    ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI]      ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo
                               Azul: DS:[00402181]=79 (‘y’), así que va a copiar el número 79 que en ASCII
                               Equivale a la letra y, es decir, la cuarta letra del Serial, será copiada a BL, damos
                               A F7.

TEST BL, BL                   ; Hace un TEST, damos a F7 para ver que pasa, como vemos el PF se ha
                              puesto a 0.

JE SHORT Crackme1.004013F5    ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.

SUB BL, 30                    ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 79,
                              damos a F7.

IMUL EDI, EAX                  ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A,
                               EDI=00001B89, damos a F7.

ADD EDI, EBX                   ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=00000049 y
                               EDI=0001135A, damos a F7.

INC ESI                       ; Incrementa ESI en 1, para que apunte a la quinta letra del Serial, damos a F7.

JMP SHORT Crackme1.004013E2   ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.
```

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=000113A3 EBX=00000049

Lo podemos ver en la ventana de registros.
Como podemos el único registro que se va conservando es EDI, los demás dentro del bucle van cogiendo distintos valores sin importar el anterior, puede que esto sea importante.

Vamos a ver la 5º repetición:

MOV AL, 0A ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI] ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo Azul: DS:[00402182]=6D ('m'), así que va a copiar el número 6D que en ASCII Equivale a la letra m, es decir, la quinta letra del Serial, será copiada a BL, damos A F7.

TEST BL, BL ; Hace un TEST, damos a F7 para ver que pasa, como vemos el PF se ha puesto a 0.

JE SHORT Crackme1.004013F5 ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.

SUB BL, 30 ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 6D, damos a F7.

IMUL EDI, EAX ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A, EDI=0000113A3, damos a F7.

ADD EDI, EBX ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=0000003D y EDI=000AC45E, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la sexta letra del Serial, damos a F7.

JMP SHORT Crackme1.004013E2 ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=000AC49B EBX=0000003D

Lo podemos ver en la ventana de registros.
Como podemos el único registro que se va conservando es EDI, los demás dentro del bucle van cogiendo distintos valores sin importar el anterior, puede que esto sea importante.

Vamos a ver la 6º repetición:

MOV AL, 0A ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI] ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo Azul: DS:[00402183]=6F ('o'), así que va a copiar el número 6F que en ASCII Equivale a la letra o, es decir, la sexta letra del Serial, será copiada a BL, damos A F7.

TEST BL, BL ; Hace un TEST, damos a F7 para ver que pasa, como vemos el PF se ha puesto a 1.

JE SHORT Crackme1.004013F5 ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.

SUB BL, 30 ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 6F, damos a F7.

IMUL EDI, EAX ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A, EDI=000AC49B, damos a F7.

ADD EDI, EBX ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=0000003F y EDI=006BAE0E, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la séptima letra del Serial, damos a F7.

JMP SHORT Crackme1.004013E2 ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=006BAE4D EBX=0000003F

Lo podemos ver en la ventana de registros.
Como podemos el único registro que se va conservando es EDI, los demás dentro del bucle van cogiendo distintos valores sin importar el anterior, puede que esto sea importante.

Vamos a ver la 7º repetición:

MOV AL, 0A ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI] ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo

Azul: DS:[00402184]=76 ('v'), así que va a copiar el número 76 que en ASCII Equivale a la letra v, es decir, la séptima letra del Serial, será copiada a BL, damos A F7.

TEST BL, BL ; Hace un TEST, damos a F7 para ver que pasa, como vemos el PF se ha puesto a 0.

JE SHORT Crackme1.004013F5 ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.

SUB BL, 30 ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 76, damos a F7.

IMUL EDI, EAX ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A, EDI=006BAE4D, damos a F7.

ADD EDI, EBX ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=00000046 y EDI=0434CF02, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la octava letra del Serial, damos a F7.

JMP SHORT Crackme1.004013E2 ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=0434CF48 EBX=00000046

Lo podemos ver en la ventana de registros.
Como podemos el único registro que se va conservando es EDI, los demás dentro del bucle van cogiendo distintos valores sin importar el anterior, puede que esto sea importante.

Vamos a ver la 8º repetición:

MOV AL, 0A ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI] ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo Azul: DS:[00402185]=69 ('i'), así que va a copiar el número 69 que en ASCII Equivale a la letra i, es decir, la octava letra del Serial, será copiada a BL, damos A F7.

TEST BL, BL ; Hace un TEST, damos a F7 para ver que pasa, como vemos el PF se ha puesto a 1.

JE SHORT Crackme1.004013F5 ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.

SUB BL, 30 ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 69, damos a F7.

IMUL EDI, EAX ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A, EDI=0434CF48, damos a F7.

ADD EDI, EBX ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=00000039 y EDI=2A1018D0, damos a F7.

INC ESI ; Incrementa ESI en 1, para que apunte a la novena letra del Serial, damos a F7.

JMP SHORT Crackme1.004013E2 ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=2A101909 EBX=00000039

Lo podemos ver en la ventana de registros.
Como podemos el único registro que se va conservando es EDI, los demás dentro del bucle van cogiendo distintos valores sin importar el anterior, puede que esto sea importante.

Vamos a ver la 9º repetición:

MOV AL, 0A ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI] ; Copia el valor de BYTE de DS:[ESI] a BL, como podemos ver donde puse el círculo Azul: DS:[00402186]=6C ('l'), así que va a copiar el número 6C que en ASCII Equivale a la letra l, es decir, la octava letra del Serial, será copiada a BL, damos A F7.

TEST BL, BL ; Hace un TEST, damos a F7 para ver que pasa, como vemos el PF se ha puesto a 1.

JE SHORT Crackme1.004013F5 ; Entra a la dirección 004013F5 si el ZF está a 1, este no es caso, damos a F7.

SUB BL, 30 ; Resta BL – 30, guardando el resultado en BL, sabemos que en BL está el valor 6C, damos a F7.

```
IMUL EDI, EAX                    ; Multiplica EDI x EAX, como vemos donde puse el círculo azul: EAX=0000000A,
                                 EDI=2A101909, damos a F7.

ADD EDI, EBX                     ; Suma EDI + EBX, como vemos donde puse el círculo azul EBX=0000003C y
                                  EDI=A4A0FA5A, damos a F7.

INC ESI                          ; Incrementa ESI en 1, para que apunte a la décima letra del Serial, pero como no
                                 existe apuntará a 0, damos a F7.

JMP SHORT Crackme1.004013E2  ; Entra a la dirección 004013E2, para comenzar una nueva repetición, damos a F7.
```

Cosas a destacar: resta BL – 30, guardando el resultado en BL, multiplica EDI y EAX, suma EDI + EBX, guardando el resultado en EDI, al final de esta repetición los registros quedan:

EAX=0000000A EDI=A4A0FA96 EBX=0000003C

Lo podemos ver en la ventana de registros.
Como podemos el único registro que se va conservando es EDI, los demás dentro del bucle van cogiendo distintos valores sin importar el anterior, puede que esto sea importante.

Última repetición:

```
MOV AL, 0A                       ; Copia el valor 0A a AL, damos a F7.

MOV BL, BYTE PTR DS:[ESI]        ; Copia el valor 0 a BL, damos a F7.

TEST BL, BL                      ; Hace un TEST poniendo el ZF a 1, damos a F7.

JE SHORT Crackme1.004013F5   ; Entra a la dirección 004013F5 si el ZF está a 1, este es caso, damos a F7 para
                                 entrar.
```

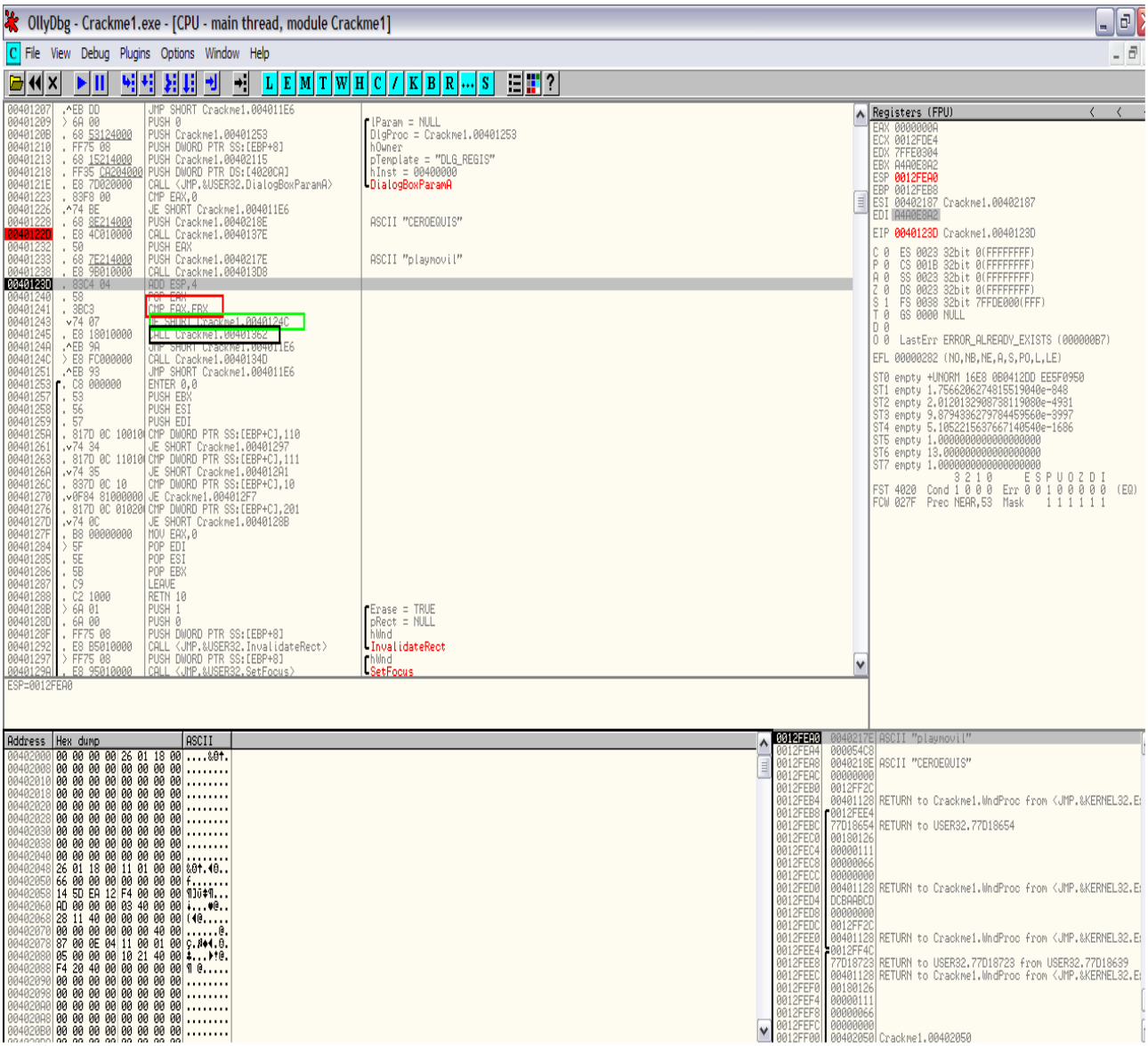
Como hemos visto hasta ahora lo que hace es hacer una clase de operaciones, y dejando los resultados de algunas en EDI, así que vamos a ver que hace lo que hay en ese registro, al entrar en el JE, vemos:

```
004013F5  |> 81F7 34120000  XOR EDI,1234      ; <- Estamos aquí
004013FB  |. 8BDF           MOV EBX,EDI
004013FD  \. C3             RETN                      ; Salimos del CALL que nos había traido.
```

Así que:

```
XOR EDI, 1234     ; Hace un XOR con el valor de EDI y 1234, damos a F7.
MOV EBX, EDI      ; Copia el valor de EDI a EBX, damos a F7.
RETN              ; Volvemos del CALL, damos a F7.
```

Como podemos ver en la ventana de registros EDI vale: A4A0E8A2 y EBX lo mismo, ya que lo acabábamos de ver como copiaba lo de EDI a EBX.
Ahora estamos:



El CMP EAX, EBX que he recuadrado en rojo es la comparación del Serial, el JE SHORT Crackme1.0040124C que he recuadrado en verde es el salto a chico bueno (la zona de serial correcto), el CALL Crackme1.00401362 que he rodeado en negro es el salto a chico malo (la zona de serial incorrecto).

Estamos sobre:

ADD ESP ,4 ; Suma 4 a ESP, damos a F7.

POP EAX ; Saca el valor al que apunte ESP y lo pone en EAX, damos a F7.

Ahora vemos en la ventana de registros: EAX=000054C8, recordemos que 000054C8 es : el XOR de la suma de los ASCII en mayúsculas, que vimos como se puso en la pila anteriormente, exactamente fue:

```
00401228   . 68 8E214000    PUSH Crackme1.0040218E              ;  ASCII "CEROEQUIS"
00401232   . 50             PUSH EAX
0040122D   . E8 4C010000    CALL Crackme1.0040137E              ; <- Aquí tenemos el breakpoint.

00401232   . 50             PUSH EAX                            ; <- Aquí pusimos en la pila el XOR de la suma de los ASCII.

00401233   . 68 7E214000    PUSH Crackme1.0040217E              ;  ASCII "playmovil"
00401238   . E8 9B010000    CALL Crackme1.004013D8
0040123D   . 83C4 04        ADD ESP,4
00401240   . 58             POP EAX
00401241   . 3BC3           CMP EAX,EBX
```

Ahora estamos sobre: CMP EAX, EBX, como vemos en la ventana de registros:

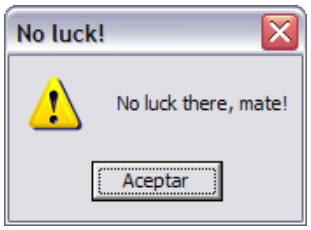
EAX=000054C8 ← XOR de la suma de los ASCII.

EBX=A4A0E8A2 ← Operaciones que hizo con el serial.

Por lo cual al hacer un CMP, no se activará el ZF, por que no son iguales, para que el serial fuera correcto debería ser igual EBX a EAX, para confirmar esto:

```
CMP EAX, EBX        ; Damos a F7.

JE SHORT Crackme1.0040124C        ; 0040124C es la zona de serial correcto, pero como el ZF no está activo no
                                  entrará, damos a F7.

CALL Crackme1.00401362            ; 00401362 es la zona de serial incorrecto, va a entrar.
```

Ahora no tener que trazar hasta que se termine la ejecución damos a F9 o al PLAY, y vamos a donde está el crackme en la barra de inicio:



Lo que suponíamos, pero ya sabemos como funciona el algoritmo, ahora solo hace falta que creemos un serial correcto.

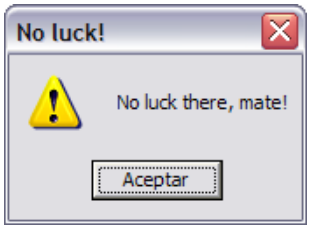
Ahora antes de buscar el serial correcto, ¿os acordáis del salto condicional que no se ejecutaba en el primer bucle?

```
0040138B  |. 72 1F         |JB SHORT Crackme1.004013AC
```

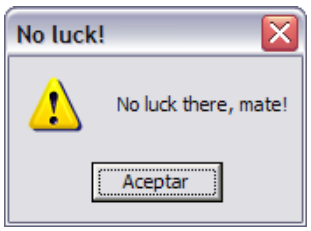
Vamos a ver como se comprueba que solo entrará si metes en el Name algún número: primero vamos a la pestaña: Debug – Restart o lo que es lo mismo CONTROL + F2, hacemos clic derecho en la ventana más grande en el centro: GO TO – Expresión 0040138B y damos a OK. Ahora ponemos un breakpoint dando a F2.
Ya tenemos el breakpoint ahora damos a F9, vamos al crackme y metemos Name: 2392 y Serial: cero y damos a OK, como vemos se para en el primer breakpoint, damos a F9 para que se ejecute hasta que llegue al próximo o al PLAY que es lo mismo. Ya estamos como vemos en la venta de registros: CF y SF a 1 y como salen en rojo quiere decir que se acaban de poner por la operación de la instrucción anterior, ahora damos a F7 y estamos:

```
004013AC  |> 5E            POP ESI                          ;  Crackme1.0040218E
004013AD  |. 6A 30         PUSH 30                           ; /Style = MB_OK|MB_ICONEXCLAMATION|MB_APPLMODAL
004013AF  |. 68 60214000    PUSH Crackme1.00402160            ; |Title = "No luck!"
004013B4  |. 68 69214000    PUSH Crackme1.00402169            ; |Text = "No luck there, mate!"
004013B9  |. FF75 08        PUSH DWORD PTR SS:[EBP+8]         ; |hOwner
004013BC  |. E8 79000000    CALL <JMP.&USER32.MessageBoxA>          ; \MessageBoxA
```

Como vemos es una zona de chico malo, así que ya sabemos que los números en el Name no valen, damos a F9 y vamos al crackme en la barra de inicio:



Este es uno de los motivos de por qué es mejor no parchear el programa tan rápidamente, ya que si no hemos visto el algoritmo esta clase de cosas podrían fastidiarnos el parcheo, si probamos el crackme al que parcheamos veremos que si metemos números en el name dará el error.
Vamos a comprobarlo: abrimos el Copia-Crackme1.exe y metemos de name: 19122 y de serial: 39393:



Como vemos esencial prestar atención al algoritmo y fijarse en los saltos que no entra, por si hay cosas de estas. Ahora nos queda obtener un Serial válido para un Name y luego crear un keygen.

9) Vamos a crear un Serial correcto para nuestro Name: lo primero cerrar el OllyDbg y cualquier cosa relacionada con el Crackme. Ahora vamos a usar lo que sabemos para crear un Name y un Serial correcto, recordemos que antes del CMP EAX, EBX para verificar si el serial es válido:

EAX=000054C8 ← XOR de la suma de los ASCII.

EBX=A4A0E8A2 ← Operaciones que hizo con el serial.

El Name introducido fue: Ceroequis
El serial introducido fue: playmovil

Sabemos que:

a) El Name siempre acaba en mayúsculas, por lo cual el serial para Ceroequis es igual al de CEROEQUIS.

b) Se suman los ASCII del Name en mayúsculas.

c) Se hace un XOR Name_Sumas_ASCII_Mayúsculas, 5678.

d) El serial introducido después de unas operaciones, tiene que ser igual al resultado del XOR Name_Sumas_ASCII_Mayúsculas, 5678.

También sabemos que forma el serial así:

a) Se coge el primer carácter del serial.

b) Se le resta 30, pero esto se hace por algo, si os fijáis antes de hacer el SUB BL, 30 en EBX que es donde se mueve el serial, no vale el valor del serial, si no otro valor, por que lo ha cogido en hexadecimal, y resta 30 para que en EBX este el número en decimal.

c) Multiplica EDI x EAX, recordemos que en EAX teníamos siempre el valor 0A, guarda el resultado en EDI, la primera repetición EDI vale 0.

d) Suma EDI + EBX guardando el resultado en EDI.

e) Cuando acaba el bucle hace un XOR EDI, 1234

f) Después de estas operaciones el resultado que tenemos en EDI tiene que ser igual al XOR Name_Sumas_ASCII_Mayúsculas, 5678.

Así que si metiéramos el Serial: 76493.

El serial que quedaría antes de la comparación sería:

1º repetición:
EDI(0) x EAX (0A) = EDI(0)
EDI(0) + EBX(7) = EDI(7)

2º repetición:
EDI(7) x EAX (0A) = EDI(46)
EDI(46) + EBX(6) = EBX(4C)

3º repetición:
EDI(4C) x EAX (0A) = EDI(2F8)
EDI(2F8) + EBX(4) = EBX(2FC)

4º repetición:
EDI(2FC) x EAX (0A) = EDI(1DD8)
EDI(1DD8) + EBX(9) = EBX(1DE1)

5º repetición:
EDI(1DE1) x EAX (0A) = EDI(12ACA)
EDI(12ACA) + EBX(3) = EBX(12ACD)

Hay que tener en cuenta que en cada repetición EBX va teniendo el carácter del serial correspondiente a la repetición, en la primera 7, en la segunda 6…

Al final del bucle se hace el:

XOR EDI(12ACD), 5678 = EDI(17CB5)

Finalmente se copia el valor de EDI a EBX, se pone el valor del xor de la suma de los ascii del name en mayúsculas en EAX y de hace un CMP EAX, EBX.
Todas estas operaciones se pueden hacer con la calculadora de windows: Ver – y marcamos: Científica, veremos que podemos operar en hexadecimal, hacer XOR…

Pues ya tenemos lo más difícil y lo más importante, entender el algoritmo, ahora mismo ya sabemos como funciona la protección pero hay que pensar en como podemos aprovecharnos de lo que sabemos:

Name Ceroequis:
EAX=000054C8 ← XOR de la suma de los ASCII.

Serial playmovil:
EBX=A4A0E8A2 ← Operaciones que hizo con el serial.

De alguna forma al final estos dos resultados tienen que ser iguales, para esto hay una regla que dice:

```
Serial =
        MOV EAX, X      ; X será la Suma de los ASCII del nick en mayúsculas del Name
                            Ceroequis.
        XOR EAX, 5678
        XOR EAX, 1234
```

Lo inverso de XOR es XOR sobre el XOR anterior, ahora lo vamos a comprobar:

La suma de los ASCII del nick en mayúsculas del Name Ceroequis era: 2B0 en hexadecimal.

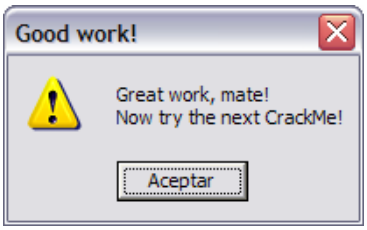
XOR EAX(2B0), 5678 = EAX(54C8)
XOR EAX(54C8), 1234 = EAX(46FC)

Ahora recordemos que cogía los números en decimal así que hacemos la conversión:

46FC hexadecimal = 18172 decimal.

Todas estas operaciones también se pueden hacer con la calculadora de windows.
Pues ya está según esto para el Name Ceroequis el Serial es: 18172.

Vamos a probarlo, abrimos el Crackme1.exe y metemos Name: Ceroequis y Serial: 18172:



Y si probamos con todas las letras en mayúsculas funcionará igual.

10) Keygenning/Haciendo un generador de claves: en este último punto vamos a crear un Keygen, es decir, un generador de claves, para que no tengamos que hacer todas estas operaciones a mano, yo usaré el lenguaje C, bien vamos a recapitular, nuestra misión es crear un programa que:

a) Tiene que crear un serial a partir de un Name introducido.
b) Comprobar que lo que hemos introducido solo son letras.
c) Un conversor de minúsculas a mayúsculas, dejando la mayúsculas como están.
d) Sumar todas las letras ASCII del Name.
e) Hacer un XOR con el resultado anterior y el número 5678.
f) Hacer un XOR con el resultado anterior y el número 1234.

Yo voy a usar el lenguaje de programación ANSI C, en pocos pasos:

a) Incluiremos las bibliotecas e ANSI C, para las funciones que vamos a necesitar y los números estáticos para los dos XOR:

```
#include <stdio.h>
#include <stdlib.h>
#define XOR1 0x5678
#define XOR2 0x1234
```

b) Crearemos una función que sumará los ASCII del Name:

```
int converascii ( char *nombre ) {
        int ascii = 0, i;

        for ( i = 0; nombre[i] != '\0' ;i++ ) {
                ascii += nombre[i];
        }

        return ascii;
}
```

c) Crearemos una función que aplicará los XOR una vez tengamos la suma de los ASCII del Name:

```
int creaserial ( int ascii ) {
        int serial;

        serial = ascii ^ XOR1;
        serial ^= XOR2;

        return serial;
}
```

d) Función para un mensaje para la intro y otro para la salida y los errores:

```
void intro ( void ) {
    printf( "\n\n" );
    printf( "           _____________________________________________\n" );
    printf( "          |   Crackme by \\0x90\\ para el CRACKME v1.0     |\n" );
    printf( "          |          De Cruhedead                         |\n" );
    printf( "          |      http://www.enye-sec.org                  |\n" );
    printf( "          |_______________________________________________|\n\n\n" );
    printf( " Nota: Hay que meter valores de la A-Z a-z sin espacios y un maximo 10 letras.\n\n" );
}

void salida ( void ) {
    printf( "\n\n   Solo se admiten letras de la A-Z a-z\n" );
    printf( "   Como maximo 10 letras\n\n" );
    system( "pause" );
    exit(1);
}

void final ( void ) {
    printf( "         _____________________________________________________\n" );
    printf( "        |  Keygen 1.0 by \\0x90\\ http://www.enye-sec.org  |\n" );
    printf( "        |_____________________________________________________|\n\n\n" );
}
```

e) Función para comprobar si las letras del Name son solo letras:

```
void compruebletras ( char *nombre ) {
   int i;

   for ( i = 0; nombre[i] != '\0'; i++ ) {
      if ( ! ( nombre[i] >= 'a' && nombre[i] <= 'z' || nombre[i] >= 'A' && nombre[i] <= 'Z' ) ) {
         salida();
      }
   }
}
```

f) Función para pasar letras minúsculas a mayúsculas, dejando las mayúsculas como están:

```
void conversion ( char *nombre ) {
    int i;

    for ( i = 0; nombre[i] != '\0'; i++ )
      if ( nombre[i] >= 'a' && nombre[i] <= 'z' ) {
         nombre[i] -= 32;
      }
}
```

e) Cuerpo del programa, pondremos como máximo 10 letras:

```
int main(void) {
   char nombre[10+1];
   int ascii, serial, tamayo;

   intro();

   printf( "\n\n   Meta el usuario: " );
   scanf( "%10s", nombre );
```

```
            compruebletras( nombre );
            conversion( nombre );

            ascii = converascii( nombre );

            serial = creaserial( ascii );
            printf( "   El serial es: %d\n\n\n\n", serial );

            final();
            system( "pause" );

            return 0;
        }
```

f) El código al final quedaría:

```
/*
Keygen en C por consola
by \0x90\ para el crackme de cruhedead
http://www.enye-sec.org
*/

#include <stdio.h>
#include <stdlib.h>
#define XOR1 0x5678
#define XOR2 0x1234


// Suma los datos de la string del nombre
int converascii ( char *nombre ) {
   int ascii = 0, i;

   for ( i = 0; nombre[i] != '\0' ;i++ ) {
      ascii += nombre[i];
   }

   return ascii;
}


// Crea el serial a partir del ascii
int creaserial ( int ascii ) {
   int serial;

   serial = ascii ^ XOR1;
   serial ^= XOR2;

   return serial;
}


// Muestra un mensaje al iniciar el programa
void intro ( void ) {
    printf( "\n\n" );
    printf( "            _____________________________________________\n" );
    printf( "           |    Crackme by \\0x90\\ para el CRACKME v1.0 |\n" );
    printf( "           |           De Cruhedead               |\n" );
    printf( "           |       http://www.enye-sec.org          |\n" );
    printf( "           |________________________________________|\n\n\n" );
    printf( " Nota: Hay que meter valores de la A-Z a-z sin espacios y un maximo 10 letras.\n\n" );
}


// Da un mensaje de salida cuando hemos metido algún caracter mal
void salida ( void ) {
    printf( "\n\n   Solo se admiten letras de la A-Z a-z\n" );
    printf( "   Como maximo 10 letras\n\n" );
    system( "pause" );
    exit(1);
}

// Muestra un pequeño texto al final de la ejecución
void final ( void ) {
    printf( "          _____________________________________________\n" );
    printf( "         |  Keygen 1.0 by \\0x90\\ http://www.enye-sec.org  |\n" );
    printf( "         |_____________________________________________|\n\n\n" );
}

// Compruebe si en el array nombre solo hay letras
void compruebletras ( char *nombre ) {
   int i;
```

```
   for ( i = 0; nombre[i] != '\0'; i++ ) {
      if ( ! ( nombre[i] >= 'a' && nombre[i] <= 'z' || nombre[i] >= 'A' && nombre[i] <= 'Z' ) ) {
         salida();
      }
   }
}


// Convierte minúsculas a mayúsculas, dejando las mayúsculas como están
void conversion ( char *nombre ) {
    int i;

    for ( i = 0; nombre[i] != '\0'; i++ )
      if ( nombre[i] >= 'a' && nombre[i] <= 'z' ) {
         nombre[i] -= 32;
      }
}

// Cuerpo del programa
int main(void) {
   char nombre[10+1];
   int ascii, serial, tamayo;

   intro();

   printf( "\n\n   Meta el usuario: " );
   scanf( "%10s", nombre );
   compruebletras( nombre );
   conversion( nombre );

   ascii = converascii( nombre );

   serial = creaserial( ascii );
   printf( "   El serial es: %d\n\n\n\n", serial );

   final();
   system( "pause" );

   return 0;
}
```

A continuación compilamos el código fuente y ejecutamos el keygen:

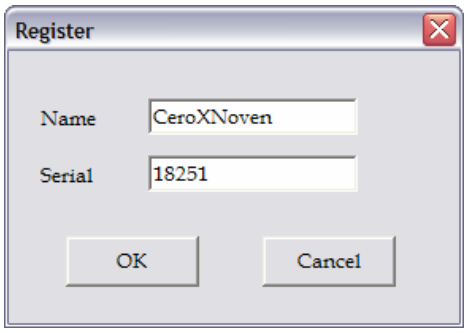
```
 ____________________________________________
|   Crackme by \0x90\ para el CRACKME v1.0   |
|           De Cruhedead                     |
|       http://www.enye-sec.org              |
|____________________________________________|
```

Nota: Hay que meter valores de la A-Z a-z sin espacios y un maximo 10 letras.

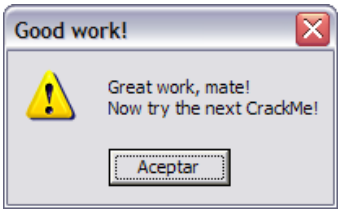
```
 Meta el usuario: CeroXNoven
 El serial es: 18251
```

```
 ____________________________________________________
|   Keygen 1.0 by \0x90\ http://www.enye-sec.org     |
|____________________________________________________|
```

Lo probamos en el Crackme1.exe:



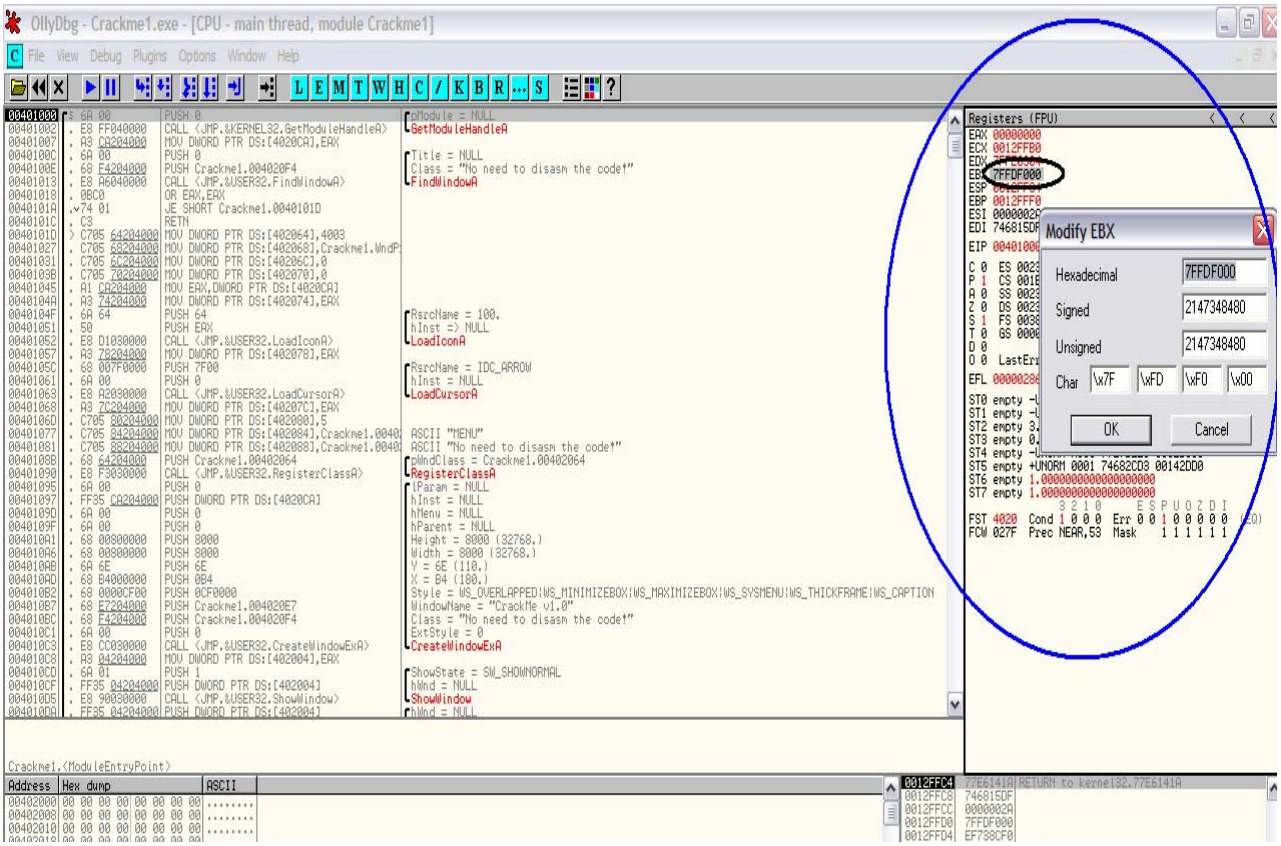
Damos a OK:



Por si no saben compilar tienen un keygen.exe con el tutorial y también el código fuente del mismo en el keygen.c

Ahora vamos a ver algunas cosas de OllyDbg:

1) Cuando estamos depurando alguna cosa en OllyDbg podemos cambiar el valor de un registro antes de que se ejecute una instrucción, hay que hacer doble clic sobre el valor que tiene el registro en la ventana de los registros o ver su valor en otras bases:



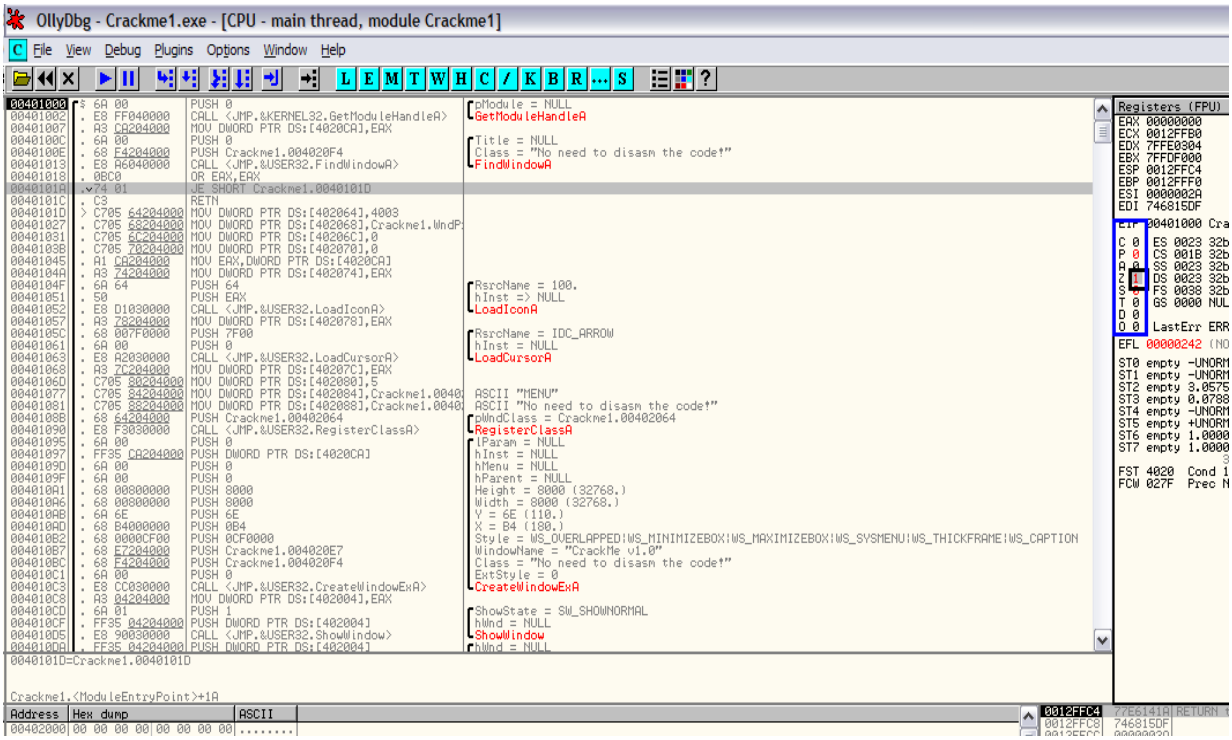
Se hace doble clic sobre lo que he rodeado en negro, lo que he rodeado en azul es la ventana de registros, podemos modificar el valor y damos a OK, una utilidad práctica de esto es por ejemplo:

Imaginémonos que estamos traceando/trazando un programa y estamos sobre:

CMP EAX, EBX ;EAX vale 00000030 y EBX 00000200
JZ ...

Para que entre en el JZ podemos modificar el registro EBX dándole el mismo valor que el de EAX y damos a F7, y se activará el ZF.

También podemos modificar el valor de los flags:



Se hace doble clic sobre el cuadrado negro y se cambiará si le damos de nuevo se volverá a cambiar, una utilidad práctica es:

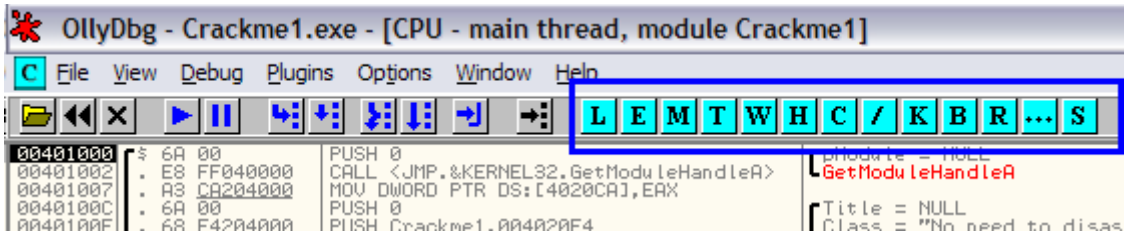
Estamos sobre un:

JZ ...

Y tenemos el ZF a 0, hacemos doble clic sobre su valor y se pondrá a 1 y al dar a F7 entrará.

Podemos atacar un proceso en ejecución: vamos a FILE – Attach y seleccionamos el proceso.

Vamos a ver para que son los siguientes botones:

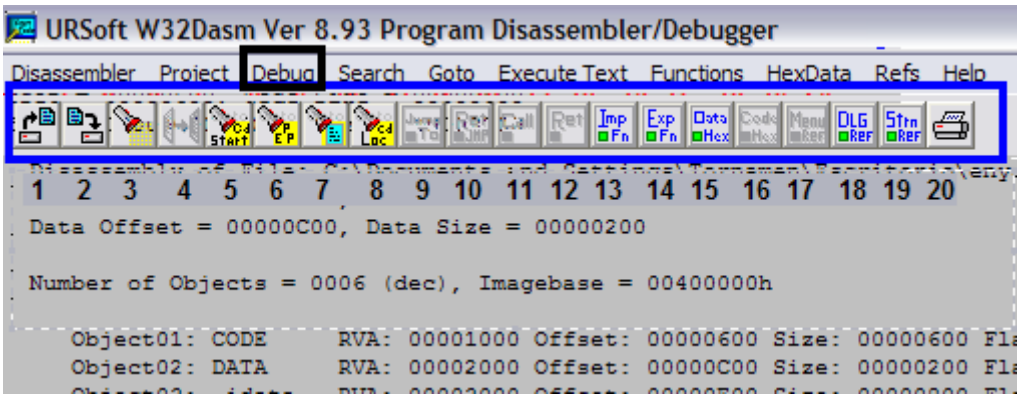


L: Muestra los logs
E: Muestra los módulos de Windows.
M: Muestra la memoria de Windows.
T: Muestra los threads.
W: Muestra la ventana de Windows.
H: Muestra los Handles.
C: Muetra la CPU.
/: Muestra los parches.
K: Muestra las llamadas de la pila.
B: Muestra los Breakpoints.
R: Muestra las referencias.
+++: Muestra el run trace.
S: Muestra el src.

En este crackme no era necesario usar el W32Dasm podemos acceder a las referencias de cadena que veíamos en el W32Dasm: dando al botón derecho sobre la ventana más grande en el centro: Search For – All Referenced Text Strings, allí nos saldrán, hacemos doble clic sobre la referencia que nos interese y estaremos allí. Damos botón derecho sobre la instrucción en que estemos: damos a: View Call Tree, para ver las CALL que llaman a la referencia donde estábamos.

El mejor documento sobre OllyDbg en español que he leído y recomiendo es:
OllyDbg © 2k3 [thEpOpE] - MrKhaki [WkT!] .:: www.thepope.tk

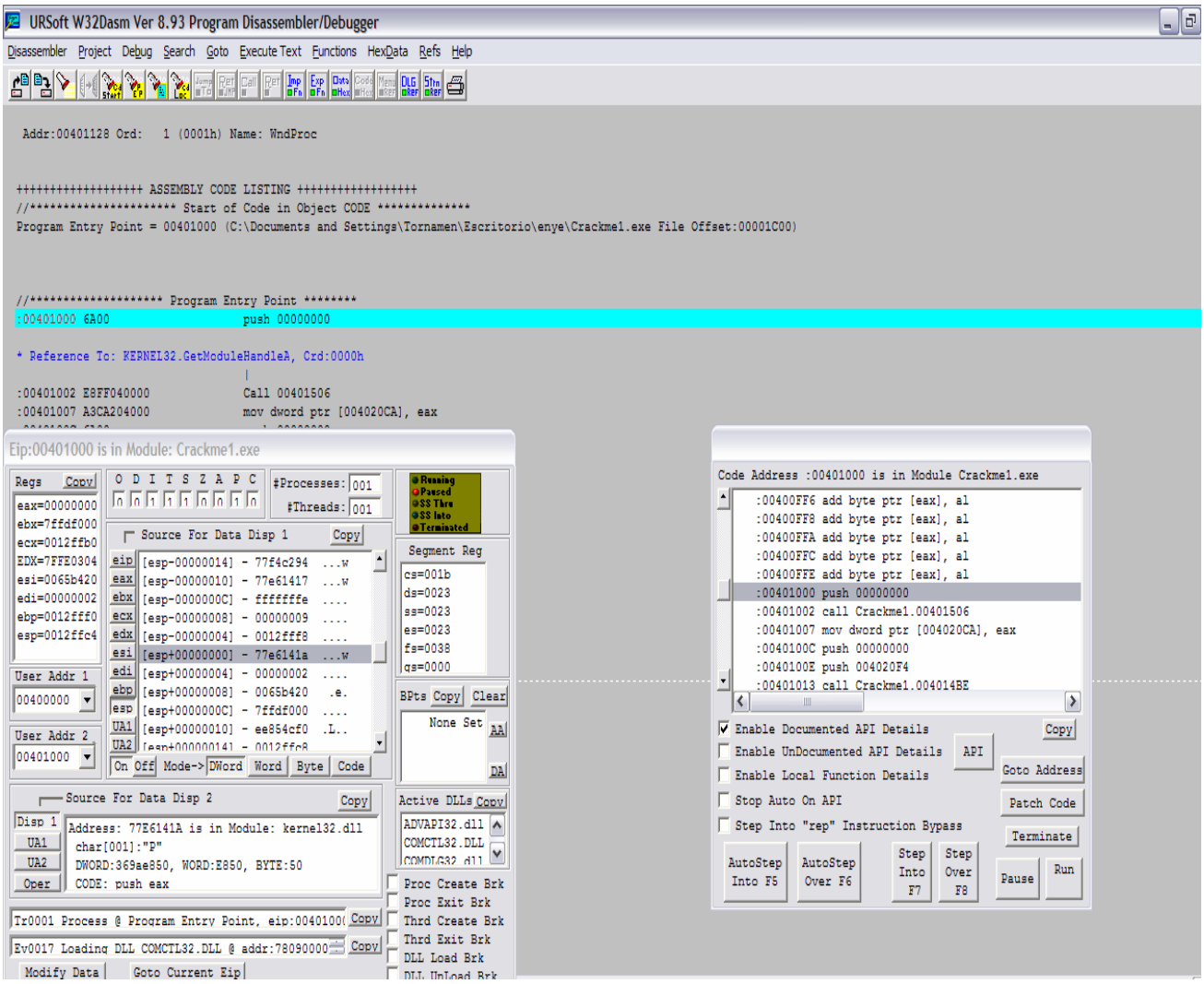
Ahora vamos a ver un poco el W32Dasm:



Vamos a ver las funciones de los botones recuadrados en azul de izquierda a derecha:

1º: Sirve para abrir un archivo para desensamblarlo.
2º: Guarda el Texto listado en un archivo ASCII.
3º: Sirve para buscar un texto o una palabra.
4º: Copia las líneas seleccionado al clipboard.
5º: Vamos a la parte del código listado.
6º: Vamos al entry point.
7º: Vamos a una determinada página.
8º: Vamos a una dirección que especifiquemos.
9º: Ejecutamos un salto.
10º: Volvemos de un salto anteior.
11º: Ejecutamos una CALL.
12º: Volvemos de una CALL anterior.
13º: Vemos los módulos importados.
14º: Vemos las funciones exportadas.
15º: Vemos los datos en hexadecimal.
16º: Vemos el código en hexadecimal.
17º: Vemos las Menu References.
18º: Vemos las Dialog References.
19º: Vemos las String References.
20º: Imprimimos una página o todo.

Ahora vamos a ver para que sirve el recuadro negro que he puesto el debug: abrimos el crackme con él, y vamos a debug – Load Proccess. Podemos meterle argumentos si el programa los acepta, este no es el caso, así que damos a OK en la ventana que nos sale:



Como podemos ver tiene varias de las opciones que nos permite el OllyDbg, podemos tracear y demás.

También podemos atacar un proceso en ejecución : Debug – Attach to an Active Proccess.

Podéis saber más de este programa en la web oficial.

Vamos a ver otro ejemplo práctico para terminar, usaremos para este solo OllyDbg y como crackme el: minicrackme.exe

1) Lo abrimos para ver como están las cosas:

Mini CraCKmE por \0x90\, http://www.enye-sec.org

Introduzca el password:

Metemos lo que sea, por ejemplo yo meteré: RLZ
Y se nos mostrará en pantalla:

Resultado: Mal, pruebe de nuevo.

Presione una tecla para continuar . . .

2) Abrimos el minicrackme.exe con el OllyDbg, botón derecho sobre la venta más grande en el centro: Search For – All Referenced Text Strings, como ya sabemos que chico malo es: Resultado: Mal, pruebe de nuevo, hacemos doble clic sobre él, subimos un poco para ver lo que hay arriba:

Primero vemos un bucle:

```
004013EC  |. C745 D4 000000>MOV DWORD PTR SS:[EBP-2C],0
004013F3  |> 837D D4 0E     /CMP DWORD PTR SS:[EBP-2C],0E
004013F7  |. 7E 02          |JLE SHORT minicrac.004013FB
```

```
004013F9  |. EB 27         |JMP SHORT minicrac.00401422
004013FB  |> 8D45 F8       |LEA EAX,DWORD PTR SS:[EBP-8]
004013FE  |. 0345 D4       |ADD EAX,DWORD PTR SS:[EBP-2C]
00401401  |. 8D48 F0       |LEA ECX,DWORD PTR DS:[EAX-10]
00401404  |. 8D45 F8       |LEA EAX,DWORD PTR SS:[EBP-8]
00401407  |. 0345 D4       |ADD EAX,DWORD PTR SS:[EBP-2C]
0040140A  |. 8D50 E0       |LEA EDX,DWORD PTR DS:[EAX-20]
0040140D  |. 0FB601        |MOVZX EAX,BYTE PTR DS:[ECX]
00401410  |. 3A02          |CMP AL,BYTE PTR DS:[EDX]
00401412  |. 74 07         |JE SHORT minicrac.0040141B
00401414  |. C745 D0 000000>|MOV DWORD PTR SS:[EBP-30],0
0040141B  |> 8D45 D4       |LEA EAX,DWORD PTR SS:[EBP-2C]
0040141E  |. FF00          |INC DWORD PTR DS:[EAX]
00401420  |.^EB D1         \JMP SHORT minicrac.004013F3
```

Luego la posible comparación del serial:

```
00401422  |> 837D D0 01    CMP DWORD PTR SS:[EBP-30],1          ; |
```

Más abajo vemos:

```
00401426  |. 75 0C         JNZ SHORT minicrac.00401434          ; |  <- Sí ZF es 0 entra en 00401434, es chico malo y parece que entra para comprobar si el serial es inválido.

00401428  |. C70424 1013400>MOV DWORD PTR SS:[ESP],minicrac.00401310 ; |ASCII "  _
 Resultado: Password Correcta.                                                  _| Chico bueno.

"
0040142F  |. E8 6C050000   CALL <JMP.&msvcrt.printf>             ; \printf
```

Otra posible comprobación de serial:

```
00401434  |> 837D D0 00    CMP DWORD PTR SS:[EBP-30],0          ; |
```

Más abajo vemos:

```
00401438  |. 75 0C         JNZ SHORT minicrac.00401446          ; |  <- Sí ZF es 0 se finaliza el programa.
0040143A  |. C70424 5013400>MOV DWORD PTR SS:[ESP],minicrac.00401350 ; |ASCII "  _
 Resultado: Mal, pruebe de nuevo.                                               _| Chico malo

"
00401441  |. E8 5A050000   CALL <JMP.&msvcrt.printf>             ; \printf
00401446  |> C70424 7613400>MOV DWORD PTR SS:[ESP],minicrac.00401376 ; |ASCII "pause"
0040144D  |. E8 2E050000   CALL <JMP.&msvcrt.system>             ; \system
00401452  |. B8 00000000   MOV EAX,0
00401457  |. C9            LEAVE
00401458  \. C3            RETN
```

3) Ahora que nos suponemos el funcionamiento pone un breakpoint (F2) sobre:

```
004013F3  |> 837D D4 0E    /CMP DWORD PTR SS:[EBP-2C],0E
```

Este es el inicio del bucle, ahora damos a play o a F9, vamos al programa en la barra de inicio: y metemos cualquier serial, por ejemplo: CeR, damos al intro y vamos al OllyDbg:

1º Repetición del bucle:

CMP DWORD PTR SS:[EBP-2C], 0E ; Movemos a SS[EBP-2C] el valor 0E, damos a F7.

JLE SHORT minicrac.004013FB ; Salta si ZF es 1, como no lo está no entrará, F7.

LEA EAX, DWORD PTR SS:[EBP-8] ; Hacemos un LEA a EAX, damos a F7.

ADD EAX, DWORD PTR SS:[EBP-2C] ; Suma lo que ha en EAX, que acabábamos de meter + SS[EBP-2C].

LEA ECX, DWORD PTR DS:[EAX-10] ; Hacemos un LEA a ECX, pero atención: abajo nos sale: …. (ASCII “enyecrackme”), damos a F7, parece que esto es el serial, válido.

Antes de nada para no seguir perdiendo el tiempo por si acaso abrimos el minicrackme.exe: y metemos: enyecrackme:

Mini CraCKmE por \0x90\, http://www.enye-sec.org

Introduzca el password: enyecrackme

Resultado: Password Correcta.

Que fácil ha sido, no siempre es así, pero a veces la intuición también llamada “Zen”, nos puede ahorrar trabajo, tenéis el código fuente de este crackme en el archivo: minicrackme.c

▪ Despedida

Pues esto ha sido todo, espero que se haya entendido lo mejor posible y me gustaría que los fallos o comentarios los enviaran a 0x90@enye-sec.org, tenéis un eNYe CraCKmE en la versión 1.0, que mete cosas un poco más avanzadas y lo he creado para que la gente después de leer este documento e informarse por internet lo pueda sacar, está en la web: http://www.enye-sec.org

Dedicado a:

El team eNYe SeC por su puesto.

Bruce-lee, hgates, DS, S-P-A-R-K, DeCoDe, shotgan, ^strahd^, crasse, yarte, sync, ^talli^, kania, Mrridk, [Shearer], khanete, membrive, pacorro, nethox, kibo, [thEpOpE] – MrKhaki, ozone y por último a \^snake^\.

Recomendaciones:

Dudas técnicas y específicas de ingeniería inversa: #Crackers y #latinreversers IRC-HISPANO.
Dudas de ensamblador en: #ASM, #Programacion, #Ensamblador, #win32 IRC-HISPANO.
Dudas de C: #C IRC-HISPANO.

Material web, buscar cualquier cosa de crackslatinos o de WkT, incluso algo de karpoff.

Nota al lector: Esto solo es un poco de este gran mundo, el usuario que quiera aprender de verdad debe leer muchísimo más.

▪ Bibliografía

Esta es la parte más importante de la bibliografía:

http://www.wikipedia.org
Curso ASM de AESOFT.
Curso ASM 29A.
http://www.fr33project.org
http://www.enye-sec.org
http://www.google.es
Página oficial de INTEL.

F I N

By \0x90\, http://www.enye-sec.org
0x90@enye-sec.org