aRC-FL-Cracking 005 (01/08/2003)

Nopear o no nopear, he ahí el dilema

(por Furious Logic [aRC])

Advertencia

Antes de poner en práctica el presente documento y cualquier material asociado al mismo, sea éste naturaleza tangible o intangible usted debe estar totalmente de acuerdo con todos los términos y condiciones siguientes:

Del software

Cualquier software se proporciona tal como está, sin ninguna garantía expresa ni implícita de ningún tipo.

aRC no garantiza ni asume responsabilidad alguna en cuanto a la integridad o exactitud de cualquier información contenida en el software.

Ni los miembros ni los colaboradores ni los invitados aRC se hacen responsables por el uso que se le pueda dar al software.

Al examinar, copiar, ejecutar, instalar o utilizar el software, el lector está aceptando su total conformidad con todos los términos y condiciones enunciados.

Del documento

Al abrir este documento, el lector acepta incondicionalmente su total y exclusiva responsabilidad legal, de acuerdo a las leyes vigentes en su país, por el uso de las técnicas experimentales, educativas y/o de investigación aquí vertidas en materia de programación especializada de computadoras.

En caso de discrepar con alguno de los puntos descritos, deberá eliminar inmediatamente el presente documento y todo material asociado al mismo.

Agradecimientos

A Lotus Word Pro de Lotus Corporation (TM), incluido en la suite de oficina Lotus Millenium 9.

A FontLab 3.00F de FontLab Developers Group por permitirnos asignar permiso completo a las fuentes true type protegidas contra copia y a Acrobat Distiller 5.0 de Adobe Systems por su excelente resultado en la creación del documento electrónico en formato PDF.

En sus inicios, este documento se desarrolló en base a cálculos de los nanosegundos o cantidad de ciclos de reloj que tomaban las instrucciones para completar su trabajo. No obstante, durante esta etapa, aprendimos que una sorprendente variación se ha dado lugar en este aspecto. Muchos códigos mnemónicos ya en el año 1995, empleaban solamente 1 ó menos de 1 ciclo de reloj, contrariamente a los 5 a 20 ciclos que teníamos documentados hasta ese entonces. Así, hemos reorientado completamente la base técnica para este material, descartando por completo el tema de optimización de velocidad en términos de ciclos de reloj para utilizar, en su lugar, técnicas y combinaciones de instrucciones no muy comunes. Queda comprobado, una vez más, que la enseñanza es uno de los más valiosos métodos de aprendizaje. Agradecemos a los lectores por habernos dado la oportunidad de enriquecer nuestra base teórica.

Objetivos

Perfeccionar nuestra programación en lenguaje ensamblador en lo referente a sustitución de bytes en un código fuente.

Motivar al lector a crear sus propias combinaciones de instrucciones assembler sustitutorias.

Demostrar que siempre existen procedimientos más elegantes de programación, que los comúnmente utilizados o mecánicamente aprendidos.

Alcances

Programar una rutina, no es solo buscar cumplir con un objetivo específico de tiempo para lanzar al mercado el antivirus, y venderlo como "pan caliente" a cuanto incauto crea que es necesario comprar nuevas versiones cada 15 días, habiendo afamados productos cuyos costos de venta son asequibles y con actualizaciones gratuitas anuales.

En el ámbito del cracking, no es solo anular una llamada a un procedimiento o eliminar los clásicos esquemas de protección "incrackeables" estilo TrialWarp. Lo que se intenta, es escribir un código óptimo tanto en tamaño como en velocidad. Al mismo tiempo, es necesario conocer en detalle la función específica de cada instrucción assembler y los registros de los que se vale para realizar su tarea.

Uno de los inconvenientes de la programación en lenguajes como C/C++ o Delphi, es su estilo predefinido de código ejecutable y de allí que nos acerquemos más al lenguaje ensamblador para tener el control total de una rutina en el momento de desensamblar. Los códigos mnemónicos son óptimos en tamaño y recientemente también en velocidad, hablando en términos de nanosegundos, a pesar de ello, es posible realizar algunas mejoras.

Presentamos al lector, algunas pautas que le ayudarán a mejorar sus técnicas de codificación en ensamblador mayormente orientadas al cracking. Debemos aclarar, que estamos asumiendo que el lector tiene conocimiento suficiente del lenguaje ensamblador y entiende las técnicas básicas de cracking.

Deben acompañar a este documento los siguientes archivos para sus prácticas:

- ✓ ejemplo1.exe (42,496 Bytes)
- ✓ ejemplo2.exe (42,496 Bytes)
- ✓ ejemplo3.exe (42,496 Bytes)

Conceptos, como de costumbre

A pedido del público lector, algunos breves conceptos:

MIPS

(1) Son las siglas de *Millions of instruction per second* o "millones de instrucciones por segundo".
(2) Es la cantidad de millones de instrucciones por segundo que una computadora es capaz de realizar durante un segundo.

MFLOPS

(1) Son las siglas tomadas de *Million floating-point operations per second* o "millones de operaciones de punto flotante por segundo".
(2) Es la cantidad de millones de operaciones con números reales que el co-procesador matemático de su computadora puede ejecutar en el transcurso de un segundo.

Whestone

(1) Prueba de velocidad que mide la cantidad de MIPS o millones de instrucciones por segundo que el microprocesador o CPU de una computadora, es capaz de realizar durante el transcurso de un segundo.
(2) El resultado se da en unidades standard denominadas Whestones.

Dhrystone

(1) Prueba de velocidad que mide la cantidad de MFLOPS que el co-procesador matemático o FPU de una computadora es capaz de realizar durante un segundo.
(2) El resultado se da en unidades standard denominadas Dhrystone.

Y dice así:

Para los especialistas en electrónica, no es suficiente evaluar un procesador por el "numerito" que esta pintado en el chip que dice 1,000 ó 2,000 MHz. Ellos evalúan los microprocesadores por su desempeño en la ejecución de un óptimo programa predefinido que no consume demasiado tiempo, al que denominan Whestone Benchmark o Dhrystone Benchmark, "medición Whestone" y "medición Dhrystone" respectivamente.

Por supuesto, aprendiendo de todo y de todos los que nos rodean, vamos a imaginar que estamos haciendo un programa o un procedimiento para utilizarlo en nuestro Crack-Benchmark. Así que este programa debe consumir pocos recursos del procesador. A nuestro favor, están las variadas instrucciones de los procesadores CISC, así que vamos a jugar un poco con el ensamblador para ver qué sucede. Recordemos que la computadora sólo hace lo que le decimos. No hay lugar para expresiones como: "éso se borró solito" ni tampoco para: "pero yo no hice nada".

Necesitamos algunas herramientas:

1. Un depurador como SoftIce 4.05, TRW2000 1.23, OllyDbg 1.09, TWX2002 o Hackman 7.3.
2. Un desensamblador como W32Dasm 8.93 o IDA Pro 4.30.
3. Un editor hexadecimal como BIEW 5.3.2, HIEW 6.82 o QView 2.69. Este último no es muy versátil.
4. Un administrador de archivos como Total Commander 5.51 o WinNc 3000 Professional 3.0.
5. Su archivador oficial de investigación y sus hojas para los bosquejos.
6. Un usuario bien configurado, optimizado y que no tenga bugs.
7. Por supuesto, como todas las veces que usamos Winbugs, nuestro backup completo del disco duro creado con Ghost (deberíamos cobrar por la publicidad) o similar para cuando se cuelgue esta chatarra de pseudo-sistema operativo.

Aunque el término salto está sumamente extendido entre los codificadores de habla española, resulta más apropiado utilizar la palabra "bifurcación" como traducción del término inglés *jump*, que es el mnemónico al que se refiere. Autores como Miguel Ángel Rodríguez-Roselló, Willian[SIC] H. Murray III y Chris H. Papas, apoyan esta traducción, aunque Maynard Kong en "Lenguaje Ensamblador Macro Assembler" no lo hace.

A partir de ahora y en todas las lecciones, utilizaremos la traducción "bifurcación" cuando queramos referirnos al mnemónico *jump* (jmp, je, jne, jb, ja, jae, jbe, jnz, etc.)

Caso 1

Desensamblemos el archivo `ejemplo1.exe`. Ubicamos el simpático nag y la bifurcación que lo genera. Se desea que la bifurcación se realice siempre, porque cuando no se realiza aparece un molesto nag. Podríamos hacer un simple reemplazo de bytes en la dirección `00401F6` y adiós nag.

El código original es:

```
...
:004010F4 3BC1            cmp eax, ecx
:004010F6 7216            jb 0040110E
...
```

Y la solución más común sería:

```
...
:004010F4 3BC1            cmp eax, ecx
:004010F6 EB16            jmp 0040110E
...
```

Editamos el byte **40110F6**, probamos el programa crackeado y, efectivamente, la bifurcación ocurre sin importar el resultado de la comparación hecha en **004010F4**. Sin embargo, antes de celebrar esta "maravilla de código ensamblador" examinemos las líneas anteriores a la comparación.

```
...
:004010E6 33C0          xor eax, eax
:004010E8 E82B010000    call 00401218
:004010ED 33C9          xor ecx, ecx
:004010EF B9A0BB0D00    mov ecx, 000DBBA0
:004010F4 3BC1          cmp eax, ecx
:004010F6 7216          jb 0040110E
...
```

Ya tenemos la idea completa. El **call** ubicado en **4010E8** llama a **GetTickCount** para determinar cuántos milisegundos han transcurrido desde que el sistema se inició. El resultado, como en toda API, se almacenará en **eax**. Se inicializa **ecx** a **0** con el clásico **xor** a sí mismo, se le asigna **DBBA0** y finalmente se compara **eax<ecx**, o lo que es lo mismo **MilisegundosTranscurridos<DBBA0**. Como **DBBA0** está en milisegundos en el sistema hexadecimal, lo convertimos a minutos en el sistema decimal: **(DBBA0h/1000d*60) = 15d**. Es decir, el nag solo aparece si los minutos transcurridos de iniciado Winbugs son mayores o iguales a 15 minutos. Antes de los 15 minutos no aparecerá.

Con esto en mente podríamos ampliar la asignación a su máxima expresión:

```
...
:004010E6 33C0          xor eax, eax
:004010E8 E82B010000    call 00401218
:004010ED 33C9          xor ecx, ecx
:004010EF B9FFFFFFFF    mov ecx, FFFFFFFF
:004010F4 3BC1          cmp eax, ecx
:004010F6 7216          jb 0040110E
...
```

Este cambio presenta 2 inconvenientes. El primero de ellos es que **FFFFFFFF** equivale a 49 días con 16 horas aproximadamente. El "simpático" nag solo aparecerá si el programa es ejecutado después de transcurrido ese tiempo de haber cargado Winbugs. Como codificadores, debemos ser exactos y nunca permitir esta clase de probabilidades de error. Otro inconveniente, es que se editan 4 Bytes ó 32 bits con una duración de algunos nanosegundos. ¡Qué despilfarro de espacio! Busquemos otra salida de estilo más simple y exacto.

```
...
:004010E6 33C0          xor eax, eax
:004010E8 E82B010000    call 00401218
:004010ED 33C9          xor ecx, ecx
:004010EF E31D          jcxz 0040110E
:004010F1 BB0D003BC1    mov ebx, C13B000D
:004010F6 7216          jb 0040110E
:004010F8 6840200400    push 00042040
:004010FD 680E304000    push 0040300E
:00401102 6813304000    push 00403013
:00401107 6A00          push 00000000
:00401109 E84C010000    call 0040125A
:0040110E 6A00          push 00000000
...
```

Esta fue una salida de 2 bytes. Analicemos los cambios ocurridos en el código. En **4010ED** inicializamos **ecx** a **0**, pero luego reemplazamos la asignación que anteriormente se hacía con **ecx=DBBA0** para mantenerlo constante en **0**. Utilizamos la instrucción **jcxz** o **jecxz**, ambas con el mismo código de operación **E3xx**, para realizar una bifurcación si **ECX=0**, lo que en nuestro caso siempre es afirmativo. Lo más interesante es que en **4010EF** hemos sobreescrito el código anterior de 5 bytes **B9A0BB0D00** con los 2 bytes **E31D**, dando como resultado un excedente de 3 bytes que al ser agregados a los 2 bytes que antes eran el **cmp**, representado por **3BC1** de la instrucción siguiente, pasaron a crear **BB0D003BC1**, un curioso **mov** que no nos preocupa en absoluto porque nunca se ejecuta. Aún más, todo

el contenido comprendido entre 4010F1 y 40110D inclusive, está totalmente descartado porque nunca llega a ejecutarse. Una firma digital podría ser agregada allí sin causar la menor molestia.

Realizar este tipo de crack requiere del entendimiento completo del funcionamiento de las rutinas involucradas. Es evidente que si el programa retornase a una de las instrucciones resaltadas en color rojo, tendríamos que rellenar los 3 bytes excedentes con código inofensivo, para no alterar el correcto funcionamiento de los mnemónicos consecuentes. Veamos otra técnica más compacta.

```
...
:004010E6 33C0          xor eax, eax
:004010E8 E82B010000    call 00401218
:004010ED 33C0          xor eax, eax
:004010EF B9A0BB0D00    mov ecx, 000DBBA0
:004010F4 3BC1          cmp eax, ecx
:004010F6 7216          jb 0040110E
...
```

Ahora sí estamos contentos. El resultado `eax` devuelto por el `call` de `4010E8` es ignorado al mantener la inicialización previa de `eax` en `0`. No nos interesa el valor asignado a `ecx`, porque la comparación `0<ecx` de `4010F4` siempre devolverá afirmativo, especialmente tomando en cuenta que el valor de `ecx` es constante.

Caso 2

Desensamblemos el archivo `ejemplo2.exe`. Se desea que la bifurcación nunca se realice porque cuando se realiza, se presenta el molesto nag. El código fuente es:

```
...
:004010E1 E882010000    call 00401268
:004010E6 EB2E          jmp 00401116
...
```

El call que vemos no tiene relación con el mensaje. Es solo una actualización de la ventana ya dibujada con un `UpdateWindow`. La bifurcación siguiente nos conduce directamente al llamado de la función generadora del nag. Cuando se desea eliminar código, frecuentemente hacemos esto: "nopear" (spanglish que significa rellenar o reemplazar con códigos `nop`).

```
...
:004010E1 E882010000    call 00401268
:004010E6 90            nop
:004010E7 90            nop
...
```

Hemos leído más de una vez que el "cracking es un arte". Seamos más creativos, a ver qué otras posibilidades podemos encontrar en este arte.

```
; Versión 1
...
:004010E1 E882010000    call 00401268
:004010E6 6690          nop
...

; Versión 2:
...
:004010E1 E882010000    call 00401268
:004010E6 87C0          xchg eax, eax
...
```

En realidad, esperábamos algo más que simples "nop optimizados". Sigamos intentando y esta vez tomemos nuestro tiempo para pensar en algo mejor. ¿Es posible o no?

```
...
```

```
:004010E1 E882010000      call 00401268
:004010E6 40              inc eax
:004010E7 48              dec eax
...
```

Esto se ve mucho mejor. Únicamente teníamos que saber que **1-1=0**. Entonces, ¿nopear o no nopear? ¡No nopear! Ésa es la consigna. El lector se preguntará: ¿qué tienen de malo las instrucciones **nop** y **xchg** presentadas en la primera y segunda versión de los **nop** optimizados? Para responder a esta pregunta necesitamos examinar el funcionamiento de ambas instrucciones.

Nop, cuyo código de operación es **90** para 1 Byte, **6690** para 2 Bytes y **666790** para 3 Bytes, no realiza ninguna operación, solo ocupa espacio y afecta únicamente al registro **EIP**. **Nop** es un alias para el uso de **xchg eax,eax**. Es decir, **nop** y **xchg eax,eax** son lo mismo. Un **nop** es un candidato perfecto a asignar el tiempo de su hilo multitarea o *thread* a otro programa que sí aproveche ese valioso tiempo.

Xchg es una instrucción que intercambia el contenido del primer operando con el segundo sin afectar a ninguno de los indicadores o registros de estado. Sólo en su formato **xchg eax,eax** ú **87C0** debe ser evitada a todas luces.

Siempre que se desee reemplazar código en una rutina, es aconsejable emplear mnemónicos que efectúen alguna operación. La última técnica depende del valor devuelto por el **call** de **4010E1**. En todo caso, se entiende que el valor de **eax** queda inalterado después de **eax+1-1**. Veamos algunas técnicas sugeridas que debieran ampliarnos la visión acerca del lenguaje ensamblador.

```
; Reemplazo de 2 bytes para EAX<FFFFFFFF [ a+01-01=a ]:
      40          inc eax
      48          dec eax

; Reemplazo de 2 bytes para EAX>00000000 [ a-01+01=a ]:
      48          dec eax
      40          inc eax

; Reemplazo de 2 bytes [ a<->b ==> b<->a ]:
      91          xchg eax, ecx
      91          xchg eax, ecx

; Reemplazo de 2 bytes [ a+0=a ]:
      3400        or al, 0

; Reemplazo de 3 bytes para 00000000<=EAX<000000FF [ a+01-1=a ]:
      40          inc eax
      FEC8        dec al

; Reemplazo de 3 bytes para 00000000<EAX<=000000FF [ a-01+1=a ]:
      48          dec eax
      FEC0        inc al

; Reemplazo de 3 bytes [ abcd<-abcd ]:
      C1C032      rol eax, 32

; Reemplazo de 3 bytes [ abcd->abcd ]:
      C1C832      ror eax, 32

; Reemplazo de 4 bytes [ -(-a)=a ]:
      F6D0        not al
      F6D0        not al

; Reemplazo de 4 bytes para AL<FF [ a+1-1=a ]:
      FEC0        inc al
      FEC8        dec al

; Reemplazo de 4 bytes para AL>00 [ a-1+1=a ]:
      FEC8        dec al
      FEC0        inc al

; Reemplazo de 4 bytes [ abcd->dcba->abcd ]:
      0FC8        bswap eax
```

```
      0FC8          bswap eax

; Reemplazo de 4 bytes [ ab<-ab ]:
      66C1C016      rol ax, 16

; Reemplazo de 4 bytes [ ab->ab ]:
      66C1C816      ror ax, 16

; Reemplazo de 5 bytes [ (a xor 00)->STACK<-a ]:
      83C800        xor eax, 0
      50            push eax
      58            pop eax

; Reemplazo de 6 bytes [ ab->ba->ab ]:
      660FC8        bswap ax
      660FC8        bswap ax

; Reemplazo de 6 bytes [ -(-a)=a ]:
      66F7D0        not ax
      66F7D0        not ax

; Reemplazo de 6 bytes [ -(-a)=a ]:
      67F7D0        not eax
      67F7D0        not eax
```

La lista puede extenderse, pero no deseamos interferir en el "hobbie" del cracker. A partir de esta lista, todo dependerá de su creatividad y dedicación. Hemos presentado toda una gama de colores. En suma, es la labor del artista matizarlos y emplearlos según su inspiración

Caso 3

Desensamblemos el archivo `ejemplo3.exe`. Aquí nos encontramos con una situación muy peculiar, la utilización de la llamada a un procedimiento con el empleo del mnemónico `call` y el posterior análisis del resultado devuelto por aquélla.

```
...
:0040109E 8D45D8          lea eax, dword ptr [ebp-28]
:004010A1 50              push eax
:004010A2 E843020000      call 004012EA
...
```

Interesante. Ese `call` llama a la API `MessageBoxIndirect` y no a `MessageBox`, pero resulta lo mismo para nosotros. Retrocedemos para ubicar su origen y nos percatamos que todo está incluido en un procedimiento llamado desde:

```
...
:00401179 E890010000      call 0040130E
:0040117E E8CBFEFFFF      call 0040104E
:00401183 7215            jb 0040119A
:00401185 33C0            xor eax, eax
...
```

El primer call que vemos es un `UpdateWindow` que no nos interesa, pero el segundo `call` es el que debe desaparecer. Normalmente reemplazaríamos los `5` bytes del `call` con código inofensivo o mejor aún iríamos al primer byte del `call`, `40104E` y cambiaríamos el `push ebp`:

```
...
:0040104E 55              push ebp
:0040104F 8BEC            mov ebp, esp
:00401051 83C4D8          add esp, FFFFFFD8
:00401054 C745D828000000  mov [ebp-28], 00000028
...
```

Por un mnemónico **ret**, **C3**:

```
...
:0040104E C3              ret
:0040104F 8BEC            mov ebp, esp
:00401051 83C4D8          add esp, FFFFFFD8
:00401054 C745D828000000  mov [ebp-28], 00000028
...
```

Sin embargo, al hacer esos cambios con el editor hexadecimal, nos encontramos con una pequeña sorpresa muy común a los programas demos. No solo presentan un molesto nag, sino que algunos verifican inmediatamente que no ha sido eliminado. Esta verificación de integridad se ve claramente en la bifurcación **jb** de **401183**. En alguna parte del **call** anterior, el que contiene el **MessageBoxIndirect**, se realiza un **cmp** o un **test**. A buscarlo entonces empezando con la dirección que indica el **call**, **40104E**.

```
...
:0040104E 55              push ebp
:0040104F 8BEC            mov ebp, esp
:00401051 83C4D8          add esp, FFFFFFD8
:00401054 C745D828000000  mov [ebp-28], 00000028
:0040105B C745DC00000000  mov [ebp-24], 00000000
:00401062 FF3580304000    push dword ptr [00403080]
:00401068 8F45E0          pop [ebp-20]
:0040106B C745E413304000  mov [ebp-1C], 00403013
:00401072 C745E80E304000  mov [ebp-18], 0040300E
:00401079 C745EC40200400  mov [ebp-14], 00042040
:00401080 C745F000000000  mov [ebp-10], 00000000
:00401087 C745F400000000  mov [ebp-0C], 00000000
:0040108E C745F800000000  mov [ebp-08], 00000000
:00401095 C745FC09000000  mov [ebp-04], 00000009
:0040109C 33C0            xor eax, eax
:0040109E 8D45D8          lea eax, dword ptr [ebp-28]
:004010A1 50              push eax
:004010A2 E843020000      call 004012EA
:004010A7 F9              stc
:004010A8 C9              leave
:004010A9 C3              ret
...
```

Un momento, está sucediendo algo extraño. Este es el código completo y no contiene ningún **cmp** ni tampoco vemos ningún **test** aunque lo busquemos con lupa. El **call** anterior a este, el de **401179**, era un **UpdateWindow** así que estamos seguros que el lugar correcto es aquí. Debemos resolver el misterio porque estamos para eso.

Examinemos las líneas involucradas. Las líneas **40104E** hasta **4010A1** emplean mnemónicos **push**, **pop**, **add**, **mov**, **xor** y **lea**. De acuerdo a la documentación de cada una de ellas, ninguna se asemeja ni se puede utilizar como un **cmp** o un **test**. El **call** que le sigue tampoco, pero ese mnemónico **stc** en **4010a7** se ve sumamente sospechoso. **Stc**, establece el indicador de acarreo **CF** a **0**. -¿Y qué hay con eso?- Pues volvamos a analizar el **call** que llamó este procedimiento.

```
...
:00401179 E890010000      call 0040130E
:0040117E E8CBFEFFFF      call 0040104E
:00401183 7215            jb 0040119A
:00401185 33C0            xor eax, eax
...
```

Dirección **401183**. Veamos la documentación acerca del **jb** o consultemos la tabla que aparece en aRC-FL-Cracking 005 Tiempo de cracking. En cualquier caso, esto es lo que descubrimos. El código de operación o código hexadecimal de un **jb** es **72xx**. No obstante, tenemos otras instrucciones con el mismo código hexadecimal. Ellas son, **jnae** que es lo mismo pero enunciado como negación de su

complemento lógico, y **jc**. ¡Éso es lo que buscamos!. Los 3 desensambladores optaron por utilizar **jb** en su lista muerta, aunque pudieron haber elegido tanto **jnae** como **jc** de esta forma:

```
...
:00401179 E890010000      call 0040130E
:0040117E E8CBFEFFFF      call 0040104E
:00401183 7215            jc 0040119A
:00401185 33C0            xor eax, eax
...
```

Ahora todo está claro como el agua y ya vemos los primeros rayos de sol en el horizonte. El **call** en **40117E** muestra al nag, activa el indicador de acarreo, retorna y enseguida se verifica si no fue alterado. Muy bien, entendiendo el funcionamiento, podemos aplicar ingeniería reversa. Podríamos anular el **call** con un simple **ret** y reemplazar el **jb** por un **jmp**, pero eso no sería lo más apropiado para nuestro nivel. Modiquemos un poco esta técnica. El código original del **call** es:

```
...
:0040104E 55              push ebp
:0040104F 8BEC            mov ebp, esp
:00401051 83C4D8          add esp, FFFFFFD8
:00401054 C745D828000000  mov [ebp-28], 00000028
...
```

Y en su lugar colocaremos:

```
...
:0040104E F9              stc
:0040104F C3              ret
:00401050 EC              in al, dx
:00401051 83C4D8          add esp, FFFFFFD8
:00401054 C745D828000000  mov [ebp-28], 00000028
...
```

El byte sobrante, **EC**, se convierte en **in al,dx** y junto los bytes **401051** hasta **4010A9** son descartados. Nuestro código está listo.

Amigo lector, el cracking es un arte y como tal requiere creatividad, lógica, deducción y sentido común además de conocimientos de programación. No se mecanice, cambie sus técnicas, sea innovador sin preocuparse por el tiempo que le tome. Esto no es un concurso de quien crackea más rápido. Si usted llegó hasta este párrafo, entonces va por buen camino. No se conforme con lo que ha aprendido siga superándose.

En el siglo XVIII, gracias al movimiento de la Ilustración, surgieron los Enciclopedistas cuya ideología era más bien política antes que filosófica e incluso muy cercana a la anarquía en sus expresiones más recalcitrantes. Sin embargo, descartando por completo el tema político, siempre hay algo que aprender. Ello es, la tendencia a la no especialización, sino al aprendizaje de toda ciencia sin llegar a los extremos por supuesto. De allí el término "Enciclopedista". En lugar de limitarse tan solo a un campo o especialidad, ellos propugnaban el cultivo de toda materia susceptible de aprenderse metódicamente.

Ésta es la moraleja: "Todo lo susceptible a aprenderse, debe ser metódicamente aprendido".

Completando el crack

Ahora que usted ha practicado con el lenguaje ensamblador, su trabajo es corregir el impase que aún permanece pendiente con el programa Hedit32 1.2: Cuando se ingresa un nombre y código válidos, el programa dice que el código no es correcto.

Bibliografía recomendada

Seguido, una relación de algunos libros y documentos que el codificador debiera adquirir para desarrollarse en todas sus áreas y no solo en la especialidad del cracking. Una búsqueda exhaustiva en Internet aumentará exponencialmente esta bibliografía. Eso dependerá de cuánto les interese el tema.

Finalmente, solo resaltar que no existe nada como un buen libro. Visite la biblioteca a menudo.

Libros

- ✓ H. Murray III, Willian[SIC] y H. Pappas, Chris. *80386/80286 Programación en lenguaje ensamblador*. Madrid, Ediciones La Colina S.A., 1987.
- ✓ Cunningham, Stephen K. *Learn Microsoft Assembler in a Day*. (formato HTML)
- ✓ Kauler, Barry. *Windows Assembly Languaje & Systems Programming*. Kansas, R&D Books, 1997.
- ✓ Kong, Maynard. *Lenguaje ensamblador Macro Assembler*. Lima, Fondo Editorial de la Pontificia Universidad Católica del Perú, 1989.
- ✓ Petzold, Charles. *Programming Windows*. Washington , Microsoft Press, 1998.
- ✓ Pietrek, Matt. *Windows 95 System Programming Secrets*. California, IDG Books Worldwide, Inc., 1995.
- ✓ Rodríguez-Roselló, Miguel Ángel. *8088-8086/8087 Programación ensamblador en entorno MS DOS*. Madrid, Ediciones Anaya Multimedia S.A., 1988.

Documentos

- ✓ s/a. *Enfoques en Ingeniería de Software*. (Formato HTML)
- ✓ s/a. *Informatifobia*. (Formato HTML)
- ✓ s/a. *Introducción al ASM*. (Formato TXT)
- ✓ s/a. *Run Trace*, (Tutorial for OllyDbg 1.06). (Formato HTML)
- ✓ s/a. *The Art of Assembly Language Programming*. (Formato PDF)
- ✓ s/a. *Win32 Developer's Reference*. (Formato HLP)
- ✓ AESOFT, *ASM por AESOFT*. (Formato TXT)
- ✓ Fog, Agner. *How to optimize for the pentium family of Microprocessors*. (Formato HLP)
- ✓ García de Celis, Ciriaco. *El universo digital del IBM PC, AT y PS/2*. 4ta. Edición. (Formato HTML)
- ✓ Giménez, José A. *Pseudocódigo*. (Formato DOC)
- ✓ Hide, Randall. *The Art Of Assembly Language*. (Formato PDF para el HLA, High Level Assembler)
- ✓ Iczelion. *PE Tutorials*. (Formato HTML)
- ✓ _____. *Win32 Assembly*. (Formato HTML)
- ✓ Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual (Volume 2, Instruction Set Reference)*. (Formato PDF)
- ✓ Intel Corporation, *Intel Architecture Optimization*. (Formato PDF)
- ✓ Juffa, Norbert. *Everything you always wanted to know about Math Coprocessors*. (Formato TXT)
- ✓ Kath, Randy. *The Portable Executable File Format from Top to Bottom*. (Formato HTML)
- ✓ Mammon. *Mammon_'s Tales To His Grandson, Breathing Life into Dead Listings*. (Formato HTM)
- ✓ O'Leary, Michael J. *Portable Executable Format*. (Formato TXT)
- ✓ Pérez Pérez, Hugo. *Curso extenso de ensamblador*. (Formato HTML)
- ✓ TIS Committe. *Tool Interface Standard (TIS) Formats Specification for Windows 1.0*. (Formato PDF)

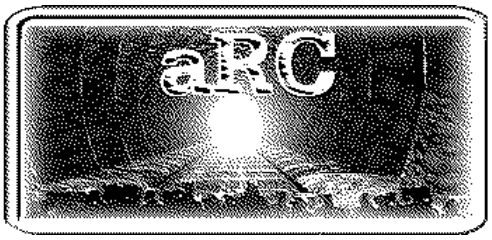
Derechos de autor

Las preguntas, consultas, sugerencias y correcciones son todas bienvenidas aunque las respuestas puedan tardar unos días en llegarles.

El autor puede ser contactado en:

IRC CyberArmy /server -m irc.cyberarmy.com: #arc3000, #arc
IRC DalNet: #arc3000
Email: furiouslogic@eml.cc

Furious Logic
Cofounder of



aDVANCED rESEARCH cOMMUNITY
http://arc3000.web1000.com
Lista: http://beam.to/arc3000
IRC CyberArmy: #arc3000, #arc
IRC DalNET: #arc3000

"Porque buscamos la libertad que sólo en el conocimiento podemos encontrar"