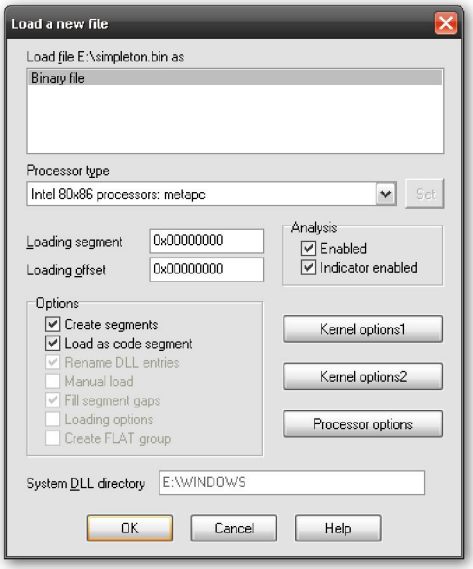
17.0.— Binary file y módulos de carga

En alguna ocasión cuando carguemos un archivo se nos puede mostrar el diálogo de carga de archivos con un pequeño problema desde la perspectiva del usuario. La lista de reconocimiento de tipos de archivos contiene una sola entrada, archivo binario (Binary file), eso nos indica que ningún módulo de carga reconoce el formato de archivo que se quiere cargar. Por suerte al menos sabemos con el lenguaje máquina que estamos tratando con lo cual podemos hace una selección correcta del tipo de procesador, porque eso es casi lo único que podremos hacer en dichos casos.



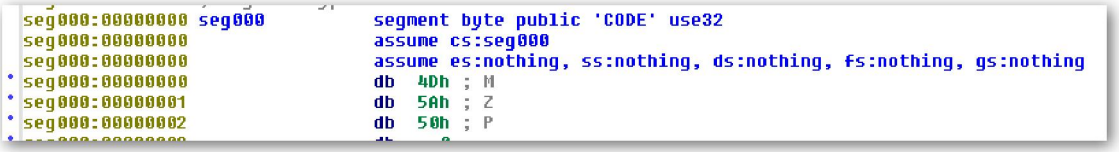
En este escrito estudiaremos las capacidades de IDA para ayudarnos a comprender archivos de tipo desconocido, a partir del análisis manual de formatos de archivo binario y utilizando éste como desarrollo de nuestros propios módulos de carga de IDA.

17.1.—Análisis de archivos desconocidos.

Existen un número infinito de formatos de archivo para almacenar código ejecutable. IDA conlleva módulos de carga para reconocer algunos de los formatos de archivo más comunes, pero no hay forma de colocar el creciente número de formatos existentes. Las imágenes de binarios pueden contener archivos ejecutables formateados para utilizar con sistemas operativos específicos, imágenes ROM extraídas de sistemas, imágenes de firmware extraídas de actualizaciones o simplemente bloques en crudo de lenguaje máquina talvez extraídos de capturas de paquetes de red. El formato de estas imágenes puede estar dictado por el sistema operativo (archivos ejecutables), por el procesador y arquitectura del sistema (imágenes ROM) o ninguno de los anteriores (esquema de datos de una shellcode adjuntada a una aplicación).

Asumiendo que disponemos de un módulo se procesador para desensamblar el código de los binarios desconocidos, nuestro trabajo será componer correctamente la imagen del archivo dentro de una base de datos IDA, antes de pasar la información a IDA de qué partes del binario representan código y qué partes representan datos. Para la mayor parte de tipos de procesador, el resultado de cargar de un archivo utilizando el formato binario es un simple listado de los contenidos del archivo apelmazado dentro de un solo segmento con dirección cero, como podemos ver a continuación.

Líneas iniciales de un archivo PE cargado en modo binario.



En algunos casos, dependiendo de lo sofisticado que sea el módulo de procesador seleccionado, puede realizarse algún tipo de desensamblado. Este puede ser el caso de cuando se selecciona un procesador con un microcontrolador adjuntado el cual puede hacer suposiciones respecto al esquema de memoria de imágenes ROM. Para los interesados en dichas aplicaciones, Andy Whittaker ha realizado una excelente disertación sobre ingeniería inversa de una imagen binaria con el microcontrolador C166 de Siemens.

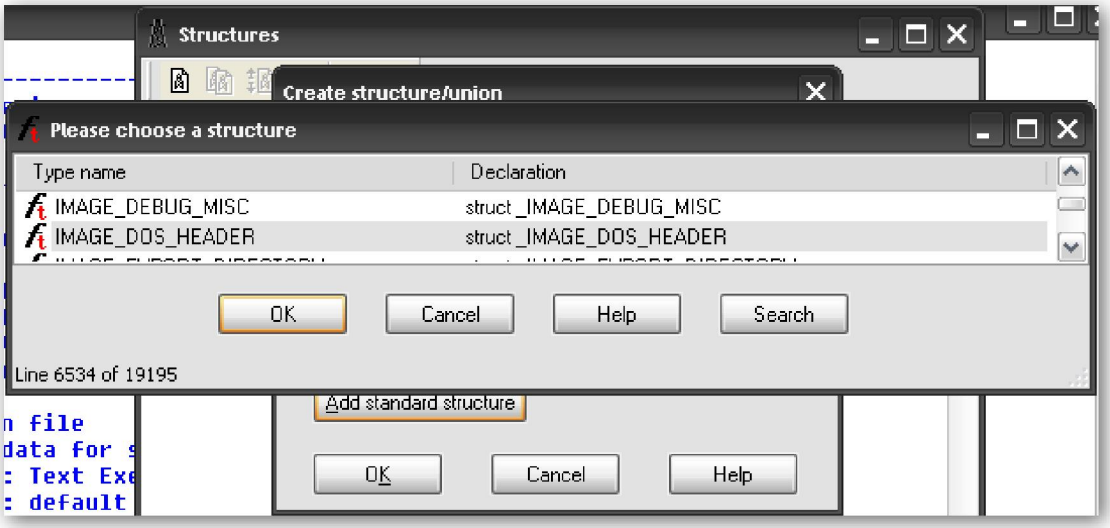
Cuando nos encaremos con archivos binarios, necesitaremos armarnos con distintos recursos relacionados con el archivo que tengamos entre manos. Dichos recursos pueden ser referencias de la CPU, referencias del sistema operativo, documentación del diseño del sistema y cualquier información del esquema de memoria obtenida a través de la depuración o análisis vía hardware.

En la siguiente sección, como ejemplo asumiremos que IDA no reconoce el formato de archivo PE de Windows. Al ser un formato muy conocido estamos muy familiarizados con él. Además podemos encontrar mucha información detallando la estructura de archivo PE, con lo cual la disección de este formato es una tarea relativamente fácil.

17.2.—Carga manual de un archivo PE Windows

Cuando podamos hallar documentación sobre el formato utilizado por un archivo en particular, tendremos muchas facilidades para intentar componer el archivo en una base de datos IDA. La figura anterior nos mostraba las primeras líneas de un archivo PE cargado en IDA como un archivo binario (binary file). Sin la ayuda de IDA y tomando las especificaciones de PE nos dicen que un archivo PE válido empezará con una estructura header **MS-DOS**. Una estructura de encabezado MS-DOS se inicia con una firma de dos bytes **4Dh 5Ah (MZ),** la cual podemos ver en las primeras dos líneas de la figura anterior.

Llegados a este punto, necesitamos comprender el esquema de un encabezado MS-DOS En las especificaciones de PE nos indicará que el valor de los cuatro bytes localizados en el offset **0x3C** del archivo, nos indica el offset del siguiente encabezado el cual necesitamos para hallar el encabezado PE. Ahora tenemos dos estrategias para ver los campos del header MS-DOS; una definir manualmente el tamaño apropiado del dato para cada campo del header MS-DOS o dos utilizar la característica de IDA **structure-creation** para definir y aplicar una estructura IMAGE_DOS_HEADER de acuerdo con las especificaciones del archivo PE. Utilizando la última opción tendríamos una vista como




```
HEADER:00000000 __ImageBase     dw 5A4Dh                ; e_magic
HEADER:00000000                 dw 50h                  ; e_cblp
HEADER:00000000                 dw 2                    ; e_cp
HEADER:00000000                 dw 0                    ; e_crlc
HEADER:00000000                 dw 4                    ; e_cparhdr
HEADER:00000000                 dw 0Fh                  ; e_minalloc
HEADER:00000000                 dw 0FFFFh               ; e_maxalloc
HEADER:00000000                 dw 0                    ; e_ss
HEADER:00000000                 dw 0B8h                 ; e_sp
HEADER:00000000                 dw 0                    ; e_csum
HEADER:00000000                 dw 0                    ; e_ip
HEADER:00000000                 dw 0                    ; e_cs
HEADER:00000000                 dw 40h                  ; e_lfarlc
HEADER:00000000                 dw 1Ah                  ; e_ovno
HEADER:00000000                 dw 4 dup(0)             ; e_res
HEADER:00000000                 dw 0                    ; e_oemid
HEADER:00000000                 dw 0                    ; e_oeminfo
HEADER:00000000                 dw 0Ah dup(0)           ; e_res2
HEADER:00000000                 dd 100h                 ; e_lfanew
```

El campo **e_lfanew** tiene un valor de **100h**, lo cual nos indica que el encabezado PE deberá encontrarse en el offset 100 (256 byte) de la base de datos. Examinemos los byte del offset 100, nos tendrá que mostrar el número mágico (**magic number**) de un

```
HEADER:00000100                 db  50h ; P
HEADER:00000101                 db  45h ; E
HEADER:00000102                 db    0
HEADER:00000103                 db    0
```

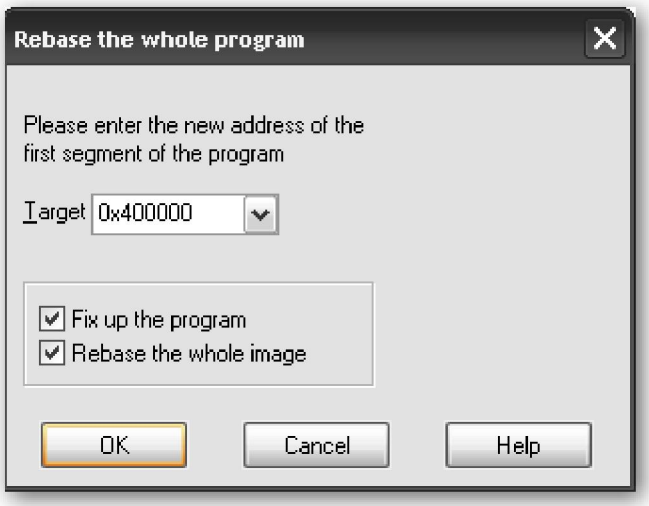
encabezado PE, **50h 45h (PE)** y permitirnos construir y aplicar en el offset 100 (basándonos en las especificaciones de PE) una estructura **IMAGE_NT_HEADERS** dentro de la base de datos. Una parte del resultado es el siguiente:

```
HEADER:00000100                 dd 4550h                ; Signature
HEADER:00000100                 dw 14Ch                 ; FileHeader.Machine
HEADER:00000100                 dw 6                    ; FileHeader.NumberOfSections
HEADER:00000100                 dd 0AD92429h            ; FileHeader.TimeDateStamp
HEADER:00000100                 dd 0                    ; FileHeader.PointerToSymbolTable
HEADER:00000100                 dd 0                    ; FileHeader.NumberOfSymbols
HEADER:00000100                 dw 0E0h                 ; FileHeader.SizeOfOptionalHeader
HEADER:00000100                 dw 818Eh                ; FileHeader.Characteristics
HEADER:00000100                 dw 10Bh                 ; OptionalHeader.Magic
HEADER:00000100                 db 2                    ; OptionalHeader.MajorLinkerVersion
HEADER:00000100                 db 19h                  ; OptionalHeader.MinorLinkerVersion
HEADER:00000100                 dd 600h                 ; OptionalHeader.SizeOfCode
HEADER:00000100                 dd 2200h                ; OptionalHeader.SizeOfInitializedData
HEADER:00000100                 dd 0                    ; OptionalHeader.SizeOfUninitializedData
HEADER:00000100                 dd 1000h                ; OptionalHeader.AddressOfEntryPoint
HEADER:00000100                 dd 1000h                ; OptionalHeader.BaseOfCode
HEADER:00000100                 dd 2000h                ; OptionalHeader.BaseOfData
HEADER:00000100                 dd 400000h              ; OptionalHeader.ImageBase
HEADER:00000100                 dd 1000h                ; OptionalHeader.SectionAlignment
HEADER:00000100                 dd 200h                 ; OptionalHeader.FileAlignment
HEADER:00000100                 dw 1                    ; OptionalHeader.MajorOperatingSystemVersion
HEADER:00000100                 dw 0                    ; OptionalHeader.MinorOperatingSystemVersion
```

El listado anterior y su explicación tienen mucha similitud con la exploración de estructuras MS-DOS y PE header realizadas en el escrito 7 (CRACKME.EXE). En este caso, sin embargo, el archivo ha sido cargado en IDA sin utilizar el cargador PE, y al contrario que en el escrito 7 que sólo era por curiosidad, en este estudio las estructuras de encabezado son esenciales para comprender el resto de la base de datos.

Una vez tengamos partes interesantes de información respecto al archivo, debemos de afinar el esquema de la base de datos. Primero el campo **Machine** del header PE, indica el tipo de CPU con el cual el archivo ha sido construido. En este ejemplo el valor es **14Ch** lo cual indica que el archivo utiliza el procesador **x86**. Si el tipo de máquina es algo parecido a 1C0h (ARM), en realidad deberíamos cerrar la base de datos e inicializar otra vez el análisis, intentando escoger un tipo correcto de procesador en el diálogo inicial. Una vez cargada la base de datos no es posible cambiar el tipo de procesador utilizado en la base de datos.

El campo **ImageBase** indica la dirección virtual base de la imagen del archivo cargado. Utilizando esta información podemos iniciar alguna información de direcciones virtuales en la base de datos. Utilizando la opción de menú **Edit>Segments>Rebase program…** , podremos especificar una nueva dirección base para el primer segmento del programa como se muestra en la figura



En el ejemplo actual sólo existe un segmento, ya que cuando un archivo es cargado en modo binario IDA crea un solo segmento para tener todo el archivo en él. Las dos

opciones de los checkbox del diálogo determinan cómo manejará la redistribución de las entradas cuando los segmento sean movidos y si IDA deberá mover cada segmento presente en la base de datos. En un archivo cargado en modo binario, IDA no tiene conciencia de la nueva situación de la información. De forma similar, con un solo segmento en el programa, por defecto la imagen entera será restablecida.

El campo **AddressOfEntryPoint** especifica la dirección virtual relativa (**RVA**) del punto de entrada (entry point) del programa. Una RVA es un offset relativo a la dirección virtual base del programa, mientras que el punto de entrada del programa representa la dirección de la primera instrucción que será ejecutada dentro del programa. En este caso un punto de entrada **RVA 1000h** indica que el programa empezará su ejecución en la dirección virtual **401000h (400000h + 1000h)**.

Esta es una información importante, es nuestra primera indicación de dónde deberemos empezar a buscar el código dentro de nuestra base de datos. Sin embargo, antes de poderlo hacer, necesitaremos componer correctamente el resto de la base de datos con sus direcciones virtuales apropiadas.

El formato PE utiliza secciones, rangos de memoria, para describir la composición del contenido del archivo. Analizando los encabezados de sección de cada una de ellas en el archivo, podremos completar el esquema básico de memoria virtual de la base de datos. El campo **NumberOfSections** nos indica el número de secciones que contiene el archivo PE; en nuestro caso son seis. Si nos referimos otra vez al esquema archivo PE

nos enseña que el conjunto de las estructuras de sección, están inmediatamente después que la estructura **IMAGE_NT_HEADERS** y que los elementos individuales del conjunto son las estructuras **IMAGE_SECTION_HEADER**, las cuales pueden ser definidas en la ventana **Structures** de IDA y aplicarlas (seis veces en nuestro caso) a los siguientes byte de la estructura **IMAGE_NT_HEADERS**.

Antes cuando hablamos de la creación de segmento, surgieron dos campos adicionales apuntando a **FileAlignment** y a **SectionAlignment**. Estos campos indican como son alineados los datos para cada sección (“La alineación describe la dirección o offset inicial de un bloque de datos. La dirección u offset debe ser un múltiplo constante del valor de alineación. Por ejemplo, cuando los datos son alineados con límite a 200h (512) byte, debemos empezar en una dirección u offset el cual sea divisible por 200h”) dentro del archivo y cómo esos mismos datos serán alineados cuando se compongan en memoria. En nuestro ejemplo, cada sección está alineada a un offset de **200h** byte

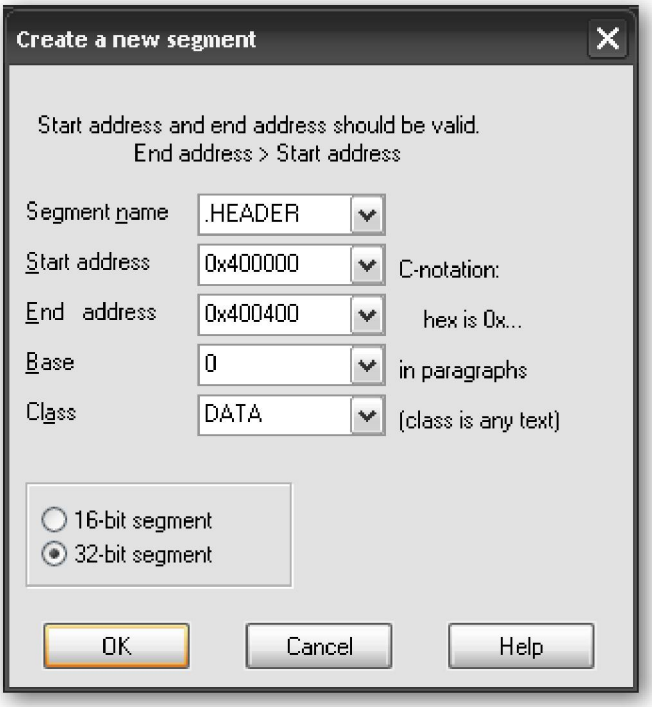
dentro del archivo; sin embargo, cuando se carguen en memoria, estas mismas secciones serán alineadas en direcciones que serán múltiplos de **1000h**. El valor pequeño **FileAlignment** proporciona los medios de guardar espacio cuando una imagen ejecutable es almacenada en un archivo, mientras que el valor mayor **SectionAlignment** normalmente corresponde al tamaño de la página de memoria virtual del sistema operativo. El comprender cómo son alineadas las secciones nos podrá ayudar a evitar errores cuando creemos secciones manualmente dentro de una base de datos.

Después de estructurar cada uno de los encabezados de sección, tendremos finalmente bastante información para empezar a crear los segmentos adicionales dentro de la base de datos. Aplicar la plantilla **IMAGE_SECTION_HEADER** en los byte inmediatos a la estructura **IMAGE_NT_HEADERS** nos produce el primer encabezado de sección y nos resultará los siguientes datos mostrados en nuestro ejemplo:

```
HEADER:004001F8                 db 43h, 4Fh, 44h, 45h, 4 dup(0); Name
HEADER:004001F8                 db 0, 10h, 2 dup(0)
HEADER:004001F8                 dd 1000h                ; VirtualAddress
HEADER:004001F8                 dd 600h                 ; SizeOfRawData
HEADER:004001F8                 dd 600h                 ; PointerToRawData
HEADER:004001F8                 dd 0                    ; PointerToRelocations
HEADER:004001F8                 dd 0                    ; PointerToLinenumbers
HEADER:004001F8                 dw 0                    ; NumberOfRelocations
HEADER:004001F8                 dw 0                    ; NumberOfLinenumbers
HEADER:004001F8                 dd 60000020h            ; Characteristics
```

El campo **Name** nos informa de la sección descrita en este caso (CODE). El resto de campos son potencialmente útiles para el formato de la base de datos, pero nos centraremos en tres que describen el esquema de la sección. El campo valor **(600h) PointerToRawData** indica el offset de archivo en el cual se puede encontrar la sección. Observemos que este valor es un múltiplo del valor de alineación de archivo, **200h**. Las secciones dentro de un archivo PE son colocadas en orden creciente tanto en offset como en dirección virtual. Desde el momento en que esta sección empieza en el offset de archivo **600h**, podemos concluir que los primeros 600h byte del archivo contienen datos de encabezado. Por lo tanto, aunque no nos dice nada, constituyen una sección la cual podemos agrupar como una sección dentro de la base de datos (.HEADER).

La Acción **Edit> Segments> Create Segment** es utilizada para crear manualmente segmentos en una base de datos. La siguiente figura muestra el diálogo de creación:



Al crear un segmento, podemos especificar el nombre que deseemos. Aquí hemos escogido **.HEADER** y describiremos adecuadamente el contenido de la sección. Podemos introducir manualmente las direcciones de inicio (inclusive) y final (exclusiva) de la sección o serán rellenadas automáticamente si tenemos resaltado el

rango de direcciones que compone la sección antes de abrir el diálogo. El valor base de la sección está descrito en el archivo del SDK **segment.hpp**. En pocas palabras, para binarios x86, IDA calcula la dirección virtual de un byte cambiando cuatro posiciones a la izquierda la base del segmento y añadiendo el offset al byte (**virtual = (base << 4) + offset**). Se deberá utilizar un valor base cero cuando no se utiliza la segmentación. La **Class** del segmento se puede utilizar para describir el contenido del segmento. Son reconocidos varios nombres de class como **CODE, DATA y BSS**. Las clases predefinidas de segmento también son descritas en **segment.hpp**.

Un efecto secundario de crear un segmento nuevo es que cualquier dato que haya sido definido dentro de los límites del segmento (tales como los encabezados previamente formateados) quedarán indefinidos. Una vez hayamos reaplicado todas las estructuras de encabezados explicadas previamente, retornemos al encabezado de la sección **.code** y observemos que el campo **VirtualAddress (1000h)** es una **RVA** que especifica la dirección de memoria en la cual el contenido de la sección tendrá que ser cargada y el campo **SizeOfRawData (600h)** cuantos byte de datos están presentes en el archivo. En otras palabras, este encabezado de sección nos dice que la sección **.code** se crea componiendo los 600h byte desde el offset **600h al BFFh** en la dirección virtual **401000h – 4015FFh**.

Debido a que nuestro archivo ha sido cargado en modo binario, todos los bytes de la sección **.code** están presentes en la base de datos; nosotros simplemente deberemos cambiarlos a sus localizaciones adecuadas. El final de la sección **.header** creada, tendría una vista similar a la siguiente

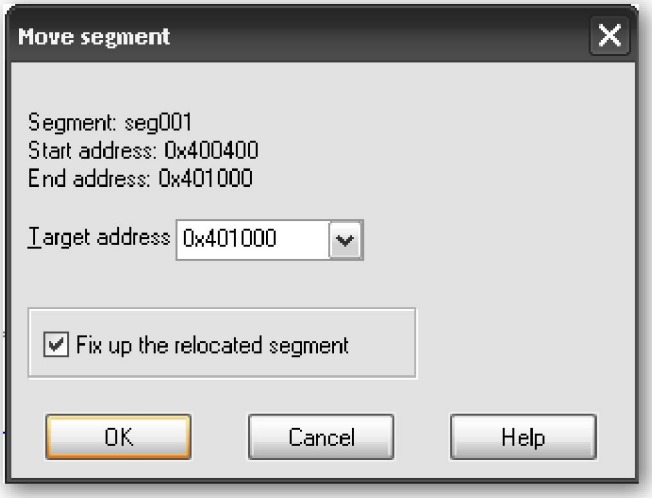
```
.HEADER:004003FF                db    0
.HEADER:004003FF _HEADER        ends
.HEADER:004003FF
seg001:004003F0 ; ===========================================================================
seg001:004003F0
seg001:004003F0 ; Segment type: Pure data
seg001:004003F0 seg001          segment page public 'DATA' use32
seg001:004003F0                 assume cs:seg001
```

Cuando se creó la sección **.header**, IDA partió el segmento original **seg000** en la nueva sección especificada .header y un nuevo segmento **seg001** el cual ocupa el resto de byte del **seg000**. Observemos que **seg001** empieza en la dirección **4003F0h** en vez de la dirección **400400h**, como era de esperar. Eso es debido a que IDA ha habilitado la base paragraph del seg001 a 1; entonces utilizando la fórmula presentada previamente, la dirección virtual se calcularía de la siguiente forma

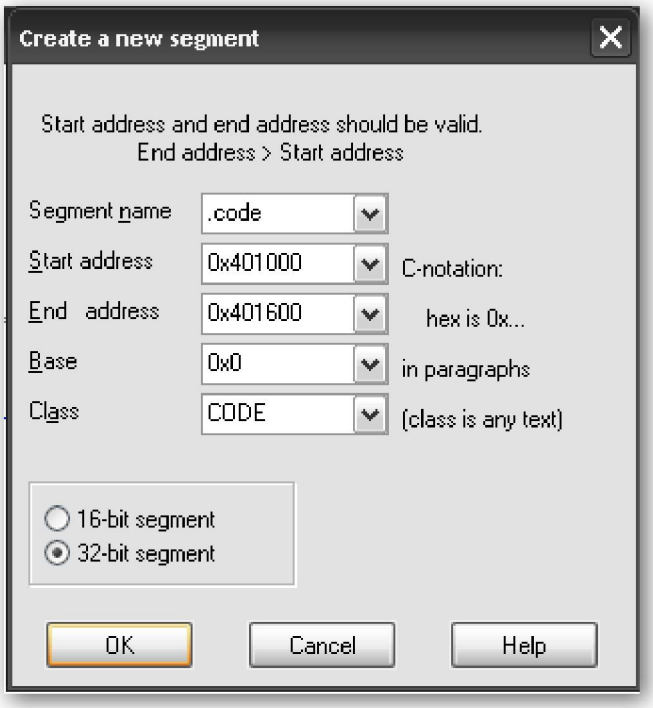
Virtual = (base << 4) + Offset
= (1 << 4) + 4003F0h
= 10h + 4003F0h
= 400400h

A pesar de cómo son mostradas las direcciones, el contenido de la sección **.code** reside en la base de datos como los primeros **600h** byte de **seg001**. Simplemente necesitamos mover la sección **.code** a su ubicación y tamaño correcto.

El primer paso para crear la sección **.code** implica mover **seg001** a la dirección virtual **401000h**. Utilizando la acción **Edit> Segments> Move Current Segment**, especificaremos una nueva dirección de inicio para **seg001** como vemos seguidamente



Podemos ver que el diálogo nos muestra como dirección de inicio del segmento a **400400h** en discrepancia con la del desensamblado, que muestra **4003F0**. Esto confirma nuestros cálculos previos. El próximo paso es colocar la sección .code en los primeros 600h byte del nuevo seg001 que ha sido movido, esto lo realizaremos con la acción **Edit> Segments> Create Segment** con los parámetros de la siguiente figura, los cuales provienen de los valores del encabezado de sección, utilizados para crear la sección.



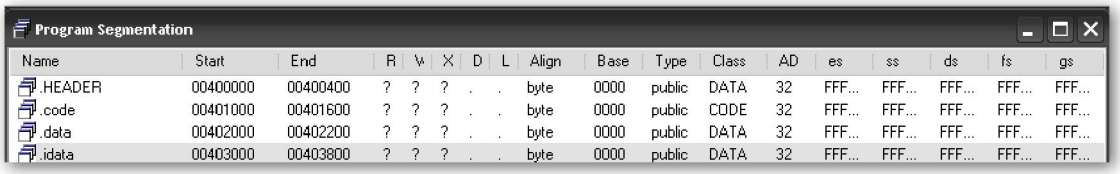
Tengamos presente que la dirección final es exclusiva. La creación de la sección ha partido a **seg001** en una nueva sección, **.code** más el resto de byte del archivo original, en una nueva sección llamada **seg002** inmediatamente después de la sección **.code**

```
.code:00401574 _code            ends
.code:00401574
seg002:004015F0 ; ===========================================
seg002:004015F0
seg002:004015F0 ; Segment type: Pure data
seg002:004015F0 seg002          segment page public 'DATA' use32
seg002:004015F0                 assume cs:seg002
```

Si retornamos a la sección headers, veremos ahora la segunda sección, que aparecerá como sigue una vez haya sido estructurada como una IMAGE_SECTION_HEADER:

```
.HEADER:00400220                db 44h, 41h, 54h, 41h, 4 dup(0); Name
.HEADER:00400220                dd 1000h                ; Misc.PhysicalAddress
.HEADER:00400220                dd 2000h                ; VirtualAddress
.HEADER:00400220                dd 200h                 ; SizeOfRawData
.HEADER:00400220                dd 0C00h                ; PointerToRawData
.HEADER:00400220                dd 0                    ; PointerToRelocations
.HEADER:00400220                dd 0                    ; PointerToLinenumbers
.HEADER:00400220                dw 0                    ; NumberOfRelocations
.HEADER:00400220                dw 0                    ; NumberOfLinenumbers
.HEADER:00400220                dd 0C0000040h           ; Characteristics
```

Utilizando los mismos campos con los que examinamos la sección .code, observamos que esta sección es llamada **.data**, que ocupa **200h** y que el archivo empieza en el offset **0C00h**, y se compondrá en **RVA 2000h (dirección virtual 402000h)**. Es importante observar a estas alturas que desde que movimos el segmento .code, no es fácil componer el campo **PointerToRawData** a un offset dentro de la base de datos. Por el contrario dependemos del hecho de que el contenido de **.data** sigue inmediatamente al contenido de la sección **.code**. En otras palabras, la sección **.data** actual reside en los primeros **200h** byte del **seg002**. La creación de la sección **.data** se hará de forma similar a la creación de la sección **.code**. El primer paso será mover **seg002 a 402000h**, y el segundo paso será crear la sección **.data** en el rango de direcciones **402000h-402200h**. Este ciclo es repetido para cada sección restante de la base de datos hasta que se haya realizado en cada sección. Cuando se ha completado, veremos en la ventana **Segments** el listado de las seis secciones que tiene el archivo:



Los signos de interrogación representan los valores desconocidos de permisos de cada sección. En los archivos PE, estos valores son especificados por los bit en el campo **Characteristics** de cada encabezado de sección. No hay forma de especificar manualmente los permisos de las secciones creadas, se tiene que realizar a través de scripts IDC o plug-in. La siguiente declaración IDC habilita el permiso de ejecución en la sección **.code** del listado anterior:

SetSegmentAttr (0x401000, SEGATTR_PERM, 1);

Desafortunadamente, IDC no define constantes simbólicas para cada uno de los permisos admisibles. Los usuarios de UNIX pueden encontrar fácil de recordar los permisos ya que los bit de permiso de sección corresponden a los bit de permiso de archivo de UNIX; así pues lectura es **4**, escritura es **2** y ejecución es **1**. Podemos combinar estos valores utilizando un **OR** para habilitar más de un permiso en una simple operación.

El último paso que realizaremos en el proceso de carga manual será tomar finalmente el módulo de procesador x86 para que nos realice cierta tarea. Una vez el binario ha sido compuesto en varias secciones, podemos retornar al punto de entrada que nos proporcionaba el encabezado (RVA 1000h o dirección virtual 401000h) y pedir a IDA que nos convierta los bytes de dicha ubicación a código. Cuando un archivo es cargado en modo binario, IDA no ejecuta ningún análisis automático del contenido del archivo.

Entre otras cosas no hace ningún intento para identificar el compilador utilizado para crear los binarios, tampoco para determinar las librerías ni las funciones importadas por el binario, ni cargar automáticamente ningún tipo de librerías o información de firma en la base de datos. Con toda probabilidad tendremos que trabajar mucho para producir un desensamblado como el que nos hace IDA de forma automática. De hecho aún no hemos hablado de otros aspectos de los encabezados PE y de cómo podríamos incorporar toda esa información adicional en nuestro proceso de carga manual.

Para acabar nuestra explicación sobre la carga manual, consideremos que necesitaríamos repetir cada uno de los pasos anteriores cada vez que quisiéramos abrir un binario con el mismo formato, o sea un desconocido por IDA. Con el tiempo, podríamos automatizar las acciones programando scripts IDC que ejecutarían algún análisis del header y la creación de segmentos. Esta es exactamente la motivación y el propósito de los módulos de carga de IDA, los cuales veremos en la siguiente sección.

Performance Bigundill@