Materia: Sistemas Operativos – Fac. de Informática – U.N.L.P.
Unidad: Archivos Ejecutables
Versión: Junio 2012
Autores: Pérez, Juan Pablo - Molinari, Lía
Palabras Claves: Ejecutable, Exe, Carga Dinámica, PE, Portable, Windows

PE Format – Portable Ejecutable

Introducción

El formato PE, Portable Executable, es un formato de archivo utilizado en las diferentes versiones (se los suele llamar sabores "flavors") de los Sistemas Operativos Microsoft Windows para almacenar programas ejecutables o librerías de funciones (DLLs).

Se puede pensar en el formato PE como una estructura de datos que encapsula la información necesaria para que el cargador (Loader) del Sistema Operativo pueda manejar los intercambios en el código ejecutable. Esto incluye referencias a librerías dinámicas para el "Linkeo" en ejecución, la API de tablas de elementos importados y exportados, recursos de datos y globales o locales a los hilos de ejecución. Archivos del sistema que se encuentran en este formato son los .EXE, .DLL, .OBJ, .SYS entre otros.

Microsoft introdujo el formato PE como parte de la especificación de Win32. Sin embargo, los archivos PE son una versión derivada del formato COFF (Common Object File Format) (Ref.1) original de VAX/VMS, situación que tiene sentido dado que gran parte del equipo de desarrollo de Windows NT provenía de DEC (Digital Equipment Corporation).

El termino "Portable Ejecutable" es utilizado ya que se buscaba obtener un formato compatible con todos los "sabores" de Windows para todas las CPUs. Este propósito se ha logrado, ya que el formato es utilizado en los descendientes de Windows NT, Windows 95, Windows CE, Xbox, etc.

El formato se ha mantenido limitado para limitar la brecha entre los sistemas basados en DOS y los NT. Por ejemplo, todos los programas con el formato PE incluyen un programa con el formato DOS que es el mensaje que por defecto se muestra al ejecutar un programa PE en un sistema DOS: "This program cannot run in DOS mode".

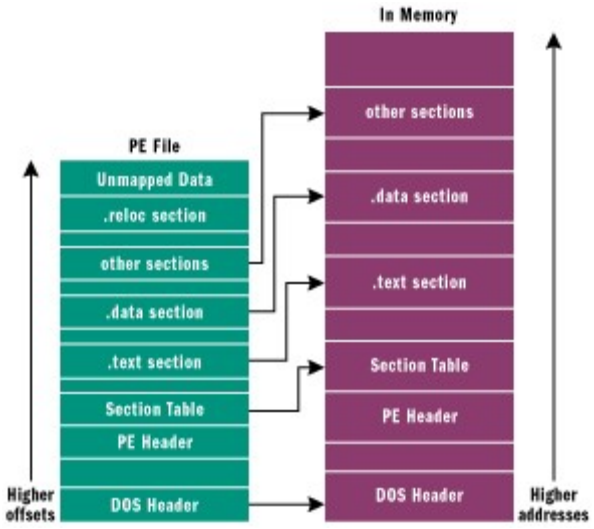
La aparición de las versiones de 64 bits de Windows llevo a realizar algunas modificaciones al formato, llamado ahora PE32+. No se agregaron nuevo campos, solo un campo fue eliminado. El resto de los cambios simplemente fueron la ampliación de algunos capos de 32 a 64 bits.

La diferencia entre los archivos .EXE y .DLL es solo semántica. Ambos archivos utilizan el mismo formato PE. La única diferencia entre ambos formatos es un único BIT que indica si el archivo debe ser tratado como .EXE o como .DLL. Incluso la extensión DLL es artificial, podemos

tener archivos DLL con otras extensiones, como por ejemplo .OCX (controles) o .CPL (applets del panel de control) son DLLs.

Una característica interesante de los archivos PE es que las estructuras de datos que se almacenan en el disco son las mismas a las utilizadas en la memoria. Cargar un programa ejecutable en memoria es simplemente "mapear" ciertos rangos de datos del archivo PE en el espacio de direcciones de usuario. Por ejemplo, la estructura de datos IMAGE_NT_HEADER (que veremos mas adelante) es idéntica en disco y en memoria.

Es importante remarcar que los archivos PE no son "mapeados" en la memoria como un único archivo, sino que el cargador examina el archivo PE y decide que porciones del archivo cargar. El mapeo es consistente en que los desplazamiento mayores en el archivo corresponden con las direcciones de memoria mayores. El desplazamiento de un elemento del archivo puede diferir de desplazamiento en la memoria. Toda la información necesaria para hacer esta traducción está disponible. En la siguiente imagen podemos ver como es el "mapeo":



Cuando un archivo PE es cargado en la memoria, vía el "loader" de Windows, la versión en la memoria es conocida como Modulo. La dirección de comienzo donde el archivo es cargado es llamada HMODULE. Dado un HMODULE podemos saber que estructura encontrar en está dirección.

El formato PE se encuentra definido en el archivo WINNT.H. Dentro de este archivo de cabeceras "headers" podemos encontrar todas las estructuras de datos, definiciones necesarias para trabajar con los archivos PE o sus estructuras equivalentes en la memoria.

El Formato

Un archivo con formato PE consiste en un conjunto de encabezados y secciones que le indican al cargador dinámico como presentar el archivo en la memoria.

Un archivo ejecutable consiste en diferentes regiones, cada una con diferente forma de protección, lo que requiere que cada comienzo de una de estas regiones se corresponda con el comienzo de una página (Windows implementa la administración de memoria mediante paginación por demanda). Por ejemplo, la región llamada ".text" , que almacena el código del programa en sí, es colocada como una región "ejecutable/solo lectura"; mientras que la sección ".data", que almacena variables globales, es marcada como "no ejecutable/lectura y escritura" . Este "ajuste" que se realiza entre las diferentes regiones y páginas de memoria, es realizada al momento de la carga y no en la imagen ejecutable en disco, tarea que es realizada por el cargador dinámico.

Las Secciones

Una sección en un archivo PE representa código o datos. Mientras que una sección de código representa código, existen múltiples tipos de datos. Además de los datos del programa (como variables globales), existen otros tipos de datos en las secciones, como las API de las tablas de importación y exportación de funciones, recursos y "relocations" (re-ubicaciones). Cada sección tiene su propio conjunto de atributos en memoria, por ejemplo, indicando si es código (execute), si es solo lectura (read only), lectura escritura (read/write), si los datos de la sección se comparten entre todos los procesos usando el ejecutable, etc.

Cada sección tiene un nombre diferente. Generalmente el nombre intenta representar el contenido de la sección. Por ejemplo, una sección llamada ".rdata" indica que es una sección de datos de solo lectura. La utilización de nombres en las secciones solo tiene como objetivo de organización para los seres humanos, y no son tenidos en cuenta por el Sistema Operativo.

Comúnmente los compiladores utilizan un conjunto de nombres estándares para las secciones que generan. Es posible, al programar, definir nuestras propias secciones. Por ejemplo, en Visual C++ uno puede indicarle al compilador una nueva sección de código o datos, por ejemplo:

```
#pragma data_seg( "MY_DATA" )
```

Cada sección tiene dos valores de alineamiento: uno en el archivo en el disco y otro en la memoria. El header del archivo PE (que veremos mas adelante) define estos valores, que pueden ser diferentes. Cada sección comienza en un desplazamiento que es un múltiplo del valor de alineación. Por ejemplo, un valor típico de alineamiento en el archivo suele ser 0x200, lo que indica que cada sección comienza en el archivo en un desplazamiento que es múltiplo de 0x200.

Veamos un ejemplo de dos secciones de una DLL del Sistema Operativo:

```
Section Table
```

```
  01 .text     VirtSize: 00074658  VirtAddr:  00001000
    raw data offs:   00000400  raw data size: 00074800
...
  02 .data     VirtSize: 000028CA  VirtAddr:  00076000
    raw data offs:   00074C00  raw data size: 00002400
```

En el ejemplo vemos 2 secciones, la sección ".text" y la ".data". El comienzo de la sección ".text" comienza en el desplazamiento 0x400 en el archivo PE, mientras que debe ser cargada con un desplazamiento 0x1000 bytes desde donde la DLL es cargada en memoria (En la memoria virtual). Para el caso de la sección ".data" los valores respectivos son de 0x74C00 y 0x76000.

Una vez cargado en la memoria el archivo PE, el comienzo de cada sección debe corresponderse con el comienzo del de una página. Esto es, el primer byte de cada sección corresponde con una página. Esto se debe a que los atributos de cada sección son diferentes, y los atributos son asignados a nivel de página. Por ejemplo: la sección de datos puede tener los atributos de lectura/escritura, por lo que las páginas de dicha sección también los deben tener. La sección de código tendrá el atributo de solo lectura, por lo que las páginas de dicha sección solo podrán ser leídas. Esto indica en una página no debería contener partes de estás 2 secciones, ya que los atributos entre ellas son disjuntos.

Direcciones Relativas Virtuales (RVAs)

En un archivo ejecutable, hay muchos lugares donde una dirección de memoria necesita ser utilizada. Por ejemplo, la dirección de una variable global que necesita ser referenciada. Los archivos PE pueden ser cargados en cualquier posición del espacio de direcciones del proceso. Si bien estos pueden tener una dirección "preferida" de carga, no se puede confiar en que el archivo sea cargado en la misma. Es por esto que necesitamos contar con una forma de especificar direcciones que son independientes de donde el archivo PE es cargado.

Para solucionar estas situaciones, y no tener que "hard-codear" direcciones en los archivos PEs, son utilizadas las direcciones relativas virtuales (RVAs) son utilizadas. Una RVA es simplemente un desplazamiento en la memoria, relativa a donde el archivo PE es cargado. Por ejemplo, consideremos un archivo .EXE cargado en la dirección 0x400000, y que su sección de código comienza en la dirección 0x401000. La RVA de la sección de código será:

```
target address) 0x401000 - (load address)0x400000  = (RVA)0x1000.
```

Para obtener una dirección actual, simplemente hay que realizar el proceso inverso (sumar a una RVA la dirección de carga del archivo). La dirección actual es llamada "Dirección Virtual" en el lenguaje PE.

El "Data Directory"

Existen muchas estructuras de datos dentro de los archivos ejecutables que deben ser localizadas rápidamente. Algunas de ellas son las que contienen información de importación y exportación de funciones, recursos o reubicaciones. Todas estas estructuras conocidas se encuentran de una manera consistente y puede ser ubicada en un lugar conocido como el Directorio de Datos (Data Directory).

El Data Directory es un arreglo de 16 estructuras. Cada entrada en el arreglo tiene un significado predefinido que indica a los que se refiere su contenido. Las definiciones (#defines) "IMAGE_DIRECTORY_ENTRY_XXX" son los índices (0 a 15) en el Data Directory. Algunos valores son:

Valor	Descripción
IMAGE_DIRECTORY_ENTRY_EXPORT	Apunta a la información que se exporta (una estructura IMAGE_EXPORT_DIRECTORY).
IMAGE_DIRECTORY_ENTRY_IMPORT	Apunta a la información que se importa (un arreglo de la estructura IMAGE_IMPORT_DESCRIPTOR).
IMAGE_DIRECTORY_ENTRY_RESOURCE	Apunta a la información de los recursos (una estructura IMAGE_RESOURCE_DIRECTORY

Otros ejemplos son de entradas pueden ser: IMAGE_DIRECTORY_ENTRY_EXCEPTION, IMAGE_DIRECTORY_ENTRY_SECURITY, IMAGE_DIRECTORY_ENTRY_DEBUG, etc.

Funciones Importadas

Cuando utilizamos código o datos de otro archivo (por ejemplo una DLL), estamos importándolo. Cuando un archivo PE es cargado a memoria por el cargador del Sistema Operativo, uno de los trabajos de este es localizar todas las funciones que son importadas, y los datos, para hacerlos accesibles (poder utilizar las direcciones).

Por ejemplo, la mayoría de las aplicaciones que se desarrolla para Windows tienen relacionada una DLL llamada KERNEL32.DLL, que funciona como puerta de acceso en modo usuario a las funciones del Sistema Operativo. A su vez KERNEL32.DLL utilizada a NTDLL.DLL que es la responsable de realizar las llamadas al sistema (System Calls). Del mismo modo, si importamos funciones de la librería GDI32.DLL (responsable de las funciones graficas) veremos que está importa de USER32.DLL, NTDLL.DLL y KERNEL32.DLL, las cuales el cargador asegura que estén cargadas y todas las importaciones resueltas.

Dentro de un archivo PE existe un arreglo de estructuras de datos, una por DLL que se importa. Cada una de estas estructuras contiene el nombre de la DLL y un puntero a un arreglo de

punteros a funciones (conocido como IAT, Import Address Table). Cada API importada tiene su propia posición en la IAT, donde la dirección de la función importada es colocada por el cargador. Una vez que el módulo es cargado, la IAT contiene la dirección que es utilizada cuando se llama a una función que fue importada.

Veamos un ejemplo de cómo es un llamado a una función importada. Existen 2 casos a considerar: La manera eficiente y la ineficiente. En el mejor de los casos, un llamado a una función importada luce así:

```
CALL DWORD PTR [0x00405030]
```

lo que representa un llamado a un puntero de función, que indica que la instrucción CALL enviada el control a las dirección 0x405030 que se encuentra en la IAT.

La forma menos eficiente de un llamado se vería así:

```
CALL 0x0040100C
•••
0x0040100C:
JMP        DWORD PTR [0x00405030]
```

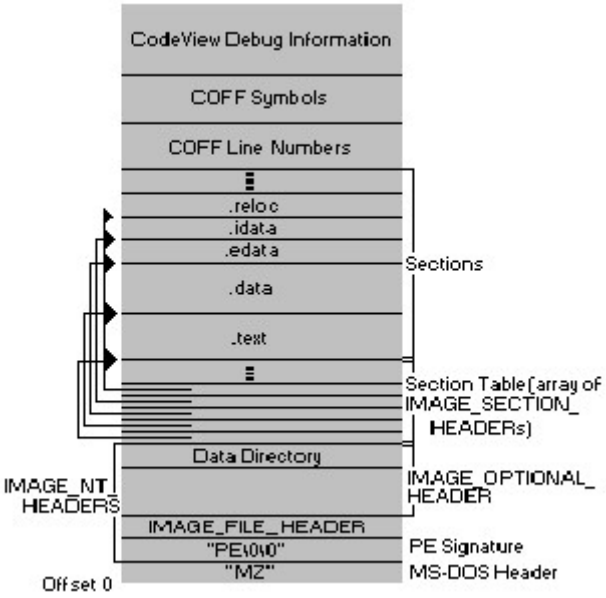
En está situación, la instrucción CALL transfiere el control a un pequeño "stub" (fragmento de código). El "stub" es un salto (JUMP) a la dirección cuyo valor es la dirección 0x405030. Recordemos que dicha dirección es la entrada en la IAT.

La eficiencia entre estas dos formas se da en la cantidad de código adicional (5 bytes) y en la cantidad de saltos.

La estructura del Archivo PE

Ahora que hemos analizado algunas generalidades de los archivos PE, pasemos a examinar el formato en que estos archivos son almacenados en el disco, comenzando por el comienzo del archivo y describiendo algunas de las estructuras de datos que se utilizan (Para una versión completa de la especificación del formato dirigirse a Ref. 4). Todas las estructuras que vamos a analizar se encuentran definidas en WINNT.H.

La siguiente imagen describe a grandes rasgos el formato de los archivos PE:



En la mayoría de los casos, las estructuras entre las versiones de 32 y 64 bits se corresponden. Por ejemplo IMAGE_NT_HEADERS32 y IMAGE_NT_HEADERS64. Estas estructuras son casi idénticas, excepto por algunos campos que se han ampliado en las versiones de 64 bits.

```
typedef struct _IMAGE_NT_HEADERS {
      DWORD Signature;
      IMAGE_FILE_HEADER FileHeader;
      IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS32,*PIMAGE_NT_HEADERS32;
typedef struct _IMAGE_NT_HEADERS64 {
      DWORD Signature;
      IMAGE_FILE_HEADER FileHeader;
      IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS64,*PIMAGE_NT_HEADERS64;
#ifdef _WIN64
typedef IMAGE_NT_HEADERS64    IMAGE_NT_HEADERS;
typedef PIMAGE_NT_HEADERS64   PIMAGE_NT_HEADERS;
#else
typedef IMAGE_NT_HEADERS32    IMAGE_NT_HEADERS;
typedef PIMAGE_NT_HEADERS32   PIMAGE_NT_HEADERS;
#endif
```

El Header MS-DOS

Todo archivo PE comienza con un pequeño programa MS-DOS ejecutable. Esto quedo por una cuestión de compatibilidad cuando surgieron las primeras versiones de Windows, para los usuarios que recién comenzaban a utilizarlo, de manera de indicarle al mismo que versión de

Windows necesitaba para ejecutar la aplicación en caso de que la misma se quiera ejecutar desde DOS.

Los primeros bytes de un archivo PE comienzan con el header tradicional de MS-DOS, llamado IMAGE_DOS_HEADER, una estructura con un conjunto de campos. Los únicos 2 valores importantes en está estructura son “e_magic” y “e_lfanew”. El campo e_lfanew contiene el desplazamiento en el archivo el header PE (IMAGE_NT_HEADERS). El campo e_magic contiene con el valor 0x5A4D. Este valor está definido como IMAGE_DOS_SIGNATURE y su valor en ASCII representa “MZ”, que son las iniciales de Mark Zbikowski, uno de los desarrolladores originales de MS-DOS.

El header IMAGE_NT_HEADERS

La estructura IMAGE_NET_HEADERS es la principal ubicación que especifica los detalles del contenido del archivo PE. Existe una versión de la estructura para 32 bits y otra para la de 64, como vimos al comienzo de la sección.

En un archivo PE valido, el campo Signature (Firma) contiene el valor 0x00004550, que en ASCII representa “PE00”, una constante definida en IMAGE_NT_SIGNATURE. El segundo campo, la estructura IMAGE_FILE_HEADER que contiene información básica acerca del archivo, siendo uno de los más importantes el tamaño de la sección del header opcional del archivo (que analizaremos en un momento). Los campos de la estructura IMAGE_FILE_HEADER son:

Tamaño	Campo	Descripción

WORD	Machine	Tipo de CPU para la que se compilo el archivo. Por ejemplo:
		`IMAGE_FILE_MACHINE_I386 0x014c // Intel 386`
		`IMAGE_FILE_MACHINE_IA64 0x0200 // Intel 64`

WORD	NumberOfSections	Indica la cantidad de secciones en la tabla de secciones. La tabla de secciones se encuentra luego de IMAGE_NT_HEADERS
DWORD	TimeDateStamp	Tiempo en el que el archivo fue creado.
DWORD	PointerToSymbolTable	Desplazamiento a la tabla de símbolos COFF.
DWORD	NumberOfSymbols	Cantidad de símbolos en la tabla de símbolos COFF.
WORD	SizeOfOptionalHeader	Tamaño del header de datos opcionales que se encuentra luego de la estructura IMAGE_FILE_HEADER
WORD	Characteristics	Un conjunto de flags (bits) que indican atributos en el archivo.

Algunos valores comunes que se pueden utilizar para los flags del campo Characteristics son:

Flag	Valor	Descripción
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Indica si la imagen es ejecutable o no
IMAGE_FILE_LARGE_ADDRESS_ AWARE	0x0020	Si la aplicación puede manejar más de 2GB

		como espació de direcciones de usuario.
	0x0040	Reservado
IMAGE_FILE_32BIT_MACHINE	0x0100	Si las instrucciones son para 32 bits o no.

Seguido a la estructura IMAGE_FILE_HEADER se encuentra la estructura IMAGE_OPTIONAL_HEADER. Si bien su nombre indica que este header es opcional, el mismo es requerido generalmente para la mayoría de las aplicaciones, no así para las DLLs.

Los campos de está estructura pueden ser analizado como en tres grandes grupos:

Desplazamiento (PE32/PE32+)	Tamaño (PE32/PE32+)	Grupo	Descripción
0	28/24	Campos estándares	Campos que están definidos paras todas las implementaciones de COFF, incluso UNIX
28/24	68/88	Campos específicos de Windows	Campos adicionales para soportar características de Windows.
96/112	Variable	Directorio de datos	Pares de Direcciones/Tamaños de tablas especiales que se encuentran en el archivo (por ejemplo las tablas de funciones importadas/exportadas)

El primer grupo de información común se describe con los siguientes campos:

Tamaño	Campo	Descripción
WORD	Magic	Una firma indicando el tipo de header. Los valores mas usados son: IMAGE_NT_OPTIONAL_HDR32_MAGIC = 0x10b (PE32) IMAGE_NT_OPTIONAL_HDR64_MAGIC = 0x20b (PE32+)
BYTE	MajorLinkerVersion	Versión mayor del compilador utilizado
BYTE	MinorLinkerVersion	Versión menor del compilador utilizado
DWORD	SizeOfCode	La suma del tamaño de todas las secciones de código
DWORD	SizeOfInitializedData	La suma del tamaño de todas las secciones de datos inicializados
DWORD	SizeOfUninitializedData	La suma del tamaño de todas las secciones de datos no inicializados
DWORD	AddressOfEntryPoint	La RVA del primer byte de código en el archivo. Para las DLL este valor puede ser 0.
DWORD	BaseOfCode	La RVA del primer byte de código cuando es cargado a memoria.
DWORD	BaseOfData	La RVA del primer byte de los datos cuando estos son cargados en memoria. Este campo no esta presente en la

		versión de 64 bits (PE32+).

El segundo grupo, de información especifica para Windows, contiene los siguientes campos:

Tamaño	Campo	Descripción
DWORD	ImageBase	La dirección preferida de carga del archivo en la memoria. Si el cargador logra colocarlo en está dirección, no es necesario realizar una re-ubicación (veremos mas adelante está situación). Los valores por defecto son: para EXEs 0x400000 y para DLLs 0x10000000.
DWORD	SectionAlignment	El alineamiento de las secciones cuando son cargadas en memoria. El valor por defecto es el tamaño de la página de la CPU para la que se compilo.
DWORD	FileAlignment	El alineamiento de las secciones dentro del archivo PE. Para x86 este valor suele ser 0x200 o 0x1000.
WORD	MajorOperatingSystemVersion	El número de versión mayor del Sistema Operativo Requerido.
WORD	MinorOperatingSystemVersion	El número de versión menor del Sistema Operativo Requerido.
WORD	MajorImageVersion	El número de versión mayor del archivo.
WORD	MinorImageVersion	El número de versión menor del archivo.
WORD	MajorSubsystemVersion	El número de versión mayor del sub-sistema requerido para el ejecutable.
WORD	MinorSubsystemVersion	El número de versión menor del sub-sistema requerido para el ejecutable.
DWORD	Win32VersionValue	Típicamente 0. No es utilizado
DWORD	SizeOfImage	Tamaño de la imagen cuando es cargada en memoria. Debe ser un múltiplo de SectionAlignment.
DWORD	SizeOfHeaders	La suma de los tamaños del header MS-DOS, el header PE y la tabla se secciones. Debe ser múltiplo de FileAlignment.
DWORD	CheckSum	Suma de comprobación (checksum) de la imagen

WORD	Subsystem	Un valor que representa el sub-sistema necesario. Algunos valor posibles son:
		`IMAGE_SUBSYSTEM_UNKNOWN = 0`
		`IMAGE_SUBSYSTEM_NATIVE = 1`
		`IMAGE_SUBSYSTEM_WINDOWS_GUI = 2`
		`IMAGE_SUBSYSTEM_WINDOWS_CUI = 3`
		`IMAGE_SUBSYSTEM_WINDOWS_CE_GUI = 9`
		`IMAGE_SUBSYSTEM_XBOX = 14`

WORD	DllCharacteristics	Flags utilizados para indicar características de la DLL. Corresponde a las constantes definidas como IMAGE_DLLCHARACTERISTICS_xxx
DWORD	SizeOfStackReserve	El tamaño máximo reservado de la pila (stack) del thread inicial.
DWORD	SizeOfStackCommit	El tamaño de la pila (stack) del thread inicial que se encuentra "comitido" (es posible utilizarlo)

DWORD	SizeOfHeapReserve	El tamaño inicial reservado para la heap del proceso. Por defecto es de 1Mb.
DWORD	SizeOfHeapCommit	El tamaño inicial "comitido" para la heap del proceso.
DWORD	LoaderFlags	No utilizado
DWORD	NumberOfRvaAndSizes	La cantidad de entradas que existen en el directorio de datos que continúa a este grupo de campos del IMAGE_OPTIONAL_HEADER. Actualmente contiene el valor 16.

El tercer, y último, grupo de los campos del IMAGE_OPTIONAL_HEADER es un arreglo de 16 entradas de estructuras IMAGE_DATA_DIRECTORY:

```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES 16

IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
```

Este arreglo es como una libreta de direcciones a ubicaciones importantes del archivo. Cada entrada contiene la siguiente estructura:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;     // RVA de los datos
    DWORD   Size;               // Tamaño de lo sdatos
} IMAGE_DATA_DIRECTORY;
```

La Tabla de Secciones

Luego de la estructura IMAGE_NT_HEADER se encuentra la Tabla de Secciones. Está tabla es un arreglo de estructuras IMAGE_SECTION_HEADER, la cual provee información sobre las secciones: ubicación, tamaño y características. La cantidad de entradas en está tabla se encuentra indicada en el campo "NumberOfSections" en el IMAGE_NT_HEADER. Los campos de la estructura son:

Tamaño	Campo	Descripción
BYTE	Name[8]	Nombre de la sección.
DWORD	VirtualSize	Indica el tamaño de sección.
DWORD	VirtualAddress	Indica la RVA donde la sección comienza en la memoria.
DWORD	SizeOfRawData	El tamaño (en bytes) de datos para la sección.
DWORD	PointerToRawData	El desplazamiento en el archivo donde los datos para la sección se encuentran.
DWORD	PointerToRelocations	El desplazamiento para las reubicaciones para la sección. Es 0 en los ejecutables.
DWORD	PointerToLinenumbers	El desplazamiento en el archivo para lo snumeros de linea
WORD	NumberOfRelocations	La cantidad de reubicaciones a las que apunta el campo PointerToRelocations. Es 0 en los ejecutables.
WORD	NumberOfLinenumbers	La cantidad de entradas a las que apunta el campo NumberOfRelocations.

DWORD	Characteristics	Conjunto de flags (unidos por un OR) que indican atributos para la sección.

Algunos de los posibles atributos para el campo Characteristics se pueden usar son:

Flag	Valor	Descripción
IMAGE_SCN_CNT_CODE	0x00000020	Indica que la sección contiene código ejecutable.
IMAGE_SCN_MEM_READ	0x40000000	La sección puede ser leída.
IMAGE_SCN_MEM_WRITE	0x80000000	La sección puede ser escrita.

La sección "Exports"

Cuando un archivo exporta código o datos, lo que se realiza es permitir que un conjunto de variables o funciones se puedan utilizar desde otros archivos. Para simplificar, utilizaremos el termino "símbolo" para referirnos a las funciones o variables que son exportadas. Como mínimo, para exportar algo, la dirección del símbolo exportado necesita ser obtenida de una manera diferente. Cada símbolo exportado posee un número ordinal asociado que será utilizado para buscarlo. Además, hay asociado un nombre (en formato ASCII) al símbolo, el que tradicionalmente, se corresponde con el nombre de la función o variable del código fuente (aunque puede ser diferente).

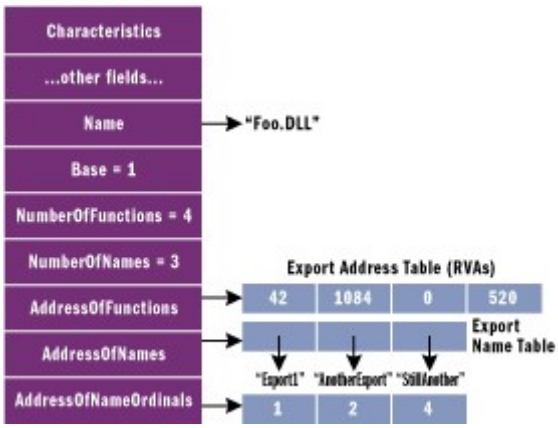
Normalmente, cuando un archivo importa un símbolo lo hace a través de su nombre en lugar de su número. Sin embargo, cuando la importación se hace utilizando su nombre, el sistema utiliza el nombre para localizar el número del símbolo y con este recuperar la dirección del mismo.

Para representar los símbolos exportados se utiliza la estructura IMAGE_EXPORT_DIRECTORY. Los campos de está estructura son:

Tamaño	Campo	Descripción
DWORD	Characteristics	Flags para los datos exportados. Actualmente no es utilizado
DWORD	TimeDateStamp	Momento en el que fue creado.
WORD	MajorVersion	Número de versión mayor. No es utilizado.
WORD	MinorVersion	Número de versión menor. No es utilizado.
DWORD	Name	Dirección RVA que apunta al nombre (ASCII)
DWORD	Base	El valor inicial utilizado para los números asignados a los símbolos. Generalmente es 1.
DWORD	NumberOfFunctions	Cantidad de entradas en la tabla de símbolos exportados (EAT)
DWORD	NumberOfNames	Cantidad de entradas en la tabla de nombre exportados (Export Names Table – ENT).
DWORD	AddressOfFunctions	La dirección RVA de la EAT. La EAT es un arreglo de RVAs, donde cada RVA en el arreglo corresponde a un símbolo

		exportado.
DWORD	AddressOfNames	La dirección RVA de la ENT. La ENT es un arreglo de RVAs a los nombres.
DWORD	AddressOfNameOrdinals	La dirección RVA de la tabla de números ordinales de los símbolos. Es un arreglo de WORDs.

Está estructura apunta a 3 arreglos y a la tabla de nombres. El único arreglo requerido es el EAT, que es el arreglo de los punteros que contienen las direcciones de los símbolos exportados. En la siguiente figura, vemos la estructura descripta:



Veamos un ejemplo de los símbolos exportados en el archivo KERNEL32.DLL:

```
exports table:
  Name:            KERNEL32.dll
  Characteristics: 00000000
  TimeDateStamp:   3B7DDFD8 -> Fri Aug 17 23:24:08 2001
  Version:         0.00
  Ordinal base:    00000001
  # of functions:  000003A0
  # of Names:      000003A0

  Entry Pt  Ordn  Name
  00012ADA     1  ActivateActCtx
  000082C2     2  AddAtomA
...
```

La sección "Imports"

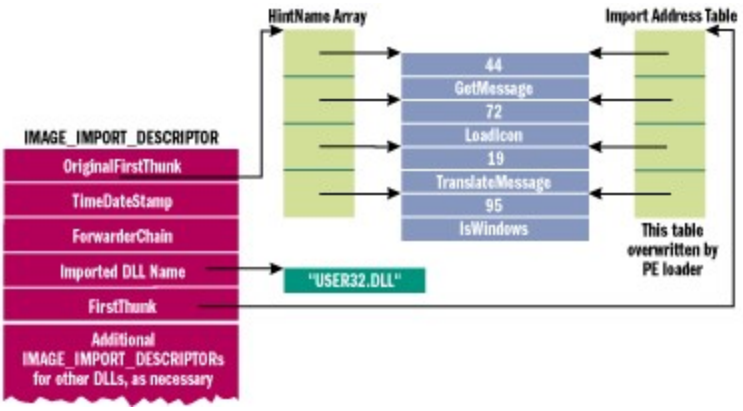
Lo opuesto a lo analizado sobre los símbolos exportados, son los importados, o sea, las funciones o variables utilizadas en un archivo PE que pertenecen a otro. La estructura que

define los símbolos importados en un archivo es IMAGE_IMPORT_DESCRIPTOR. La entrada en el Directorio de Datos (analizado antes) es un arreglo de está estructura. Los campos de IMAGE_IMPORT_DESCRIPTOR son:

Tamaño	Campo	Descripción
DWORD	OriginalFirstThunk	Contiene la dirección RVA a la tabla de nombres importados (INT – Import Name Table). Esta tabla es un arreglo de estructuras IMAGE_THUNK_DATA. Cuando este valor es 0 se indica que es el final de la tabla de IMAGE_IMPORT_DESCRIPTORs.
DWORD	TimeDateStamp	Este valor es cero si el ejecutable no se encuentra ligado a la DLL.
DWORD	ForwarderChain	Es el índice a la primera función redireccionada. Este documento no alcanza está funcionalidad
DWORD	Name	La dirección RVA al nombre del archivo importado.
DWORD	FirstThunk	La dirección RVA de la tabla de direcciones importadas (IAT - Import Address Table). Esta tabla es un arreglo de estructuras IMAGE_THUNK_DATA.

Existe una estructura IMAGE_IMPORT_DESCRIPTOR para cada archivo importado. El final de la tabla de IMAGE_IMPORT_DESCRIPTOR es indicada con una entrada donde todos los campos contiene el valor 0.

Cada IMAGE_IMPORT_DESCRIPTOR referencia a dos arreglos idénticos, IAT y INT, de estructuras IMAGE_THUNK_DATA. La siguiente imagen muestra un ejemplo de un archivo que importa algunas funciones del archivo USER32.DLL:



La estructura IMAGE_THUNK_DATA es una unión de punteros. Cada elemento, en los arreglos, de esté tipo corresponde a una función importada. El final de ambos arreglos (IAT y INT) es indicado con un elemento con valor 0. La unión IMAGE_THUNK_DATA es un DWORD con las siguientes interpretaciones:

```
DWORD Function;        // Dirección de Memoria de la función importada
DWORD Ordinal;         // Número ordinal del símbolo exportado
DWORD AddressOfData;   // Dirección RVA a una estructura
                       // IMAGE_IMPORT_BY_NAME con el nombre
                       // del símbolo exportado
DWORD ForwarderString;// Dirección RVA a una función
                       //importada redireccionada
```

La estructura IMAGE_THUNK_DATA en la IAT tiene 2 propósitos. En el archivo PE, contiene o bien el número ordinal del símbolo importado o bien una dirección RVA a una estructura IMAGE_IMPORT_BY_NAME que contiene 2 campos: un WORD y una cadena de caracteres con el nombre del símbolo exportado.

Cuando el cargador trae a memoria el archivo PE sobrescribe cada entrada en la IAT con la dirección actual del símbolo importado. En Ref. 6 se explica detalles de cómo trabaja el cargador de Windows.

El otro arreglo, el INT, es esencialmente idéntico al IAT. La diferencia esta en que el INT no es sobrescrito por el cargador cuando el archivo es traído a memoria. ¿Por qué llevar 2 arreglos en paralelo con la misma información de símbolos importados de una DLL? La respuesta está en un concepto llamado “binding” (ligarse). Cuando se produce la ligadura, la tabla IAT en el archivo es sobrescrita, por lo que se utiliza la tabla INT para mantener la información original. Más información sobre el proceso de binding se puede encontrar en Ref. 7

La sección de Recursos (Resources)

En está sección es posible almacenar información en recursos como iconos, bitmaps o ventanas de diálogos.

Los recursos se encuentran en una sección llamada “.rsrc”. La entrada IMAGE_DIRECTORY_ENTRY_RESOURCE en el Data Directory contiene la dirección RVA y el tamaño para está sección. Lo recursos, dentro de la sección son organizamos de una forma similar a un sistema de archivos (File System), usando directorios y nodos hojas.

El puntero desde el Data Directory apunta a una estructura IMAGE_RESOURCE_DIRECTORY. Está estructura contiene campos que no son utilizados como “Characteristic”, “TimeDateStamp”, “MajorVersion”, “MinorVersion”. Los únicos campos interesantes de la misma son “NumberOfNamedEntries” y “NumberOfIdEntries”. Seguido a IMAGE_RESOURCE_DIRECTORY hay un arreglo de estructuras IMAGE_RESOURCE_DIRECTORY_ENTRY (entradas en el directorio). En esté arreglo existen tantas entradas como la suma de los campos “NumberOfNamedEntries” y “NumberOfIdEntries” mencionados.

Una estructura IMAGE_RESOURCE_DIRECTORY_ENTRY (que contiene dos campos de tipo DWORD) apunta a otro IMAGE_RESOURCE_DIRECTORY o a un dato de un recurso particular:

- Cuando IMAGE_RESOURCE_DIRECTORY_ENTRY apunta a otro IMAGE_RESOURCE_DIRECTORY, el BIT más significativo del segundo campo es 1, los restantes 31 bits son el desplazamiento al IMAGE_RESOURCE_DIRECTORY. El desplazamiento indicado es relativo al comienzo de la sección de recursos (no es una dirección RVA)

- Cuando IMAGE_RESOURCE_DIRECTORY_ENTRY apunta a un recurso, BIT más significativo del segundo campo es 0 y los restantes 31 bits son el desplazamiento al recurso (icono, imagen, etc.). El desplazamiento indicado es relativo al comienzo de la sección de recursos (no es una dirección RVA)

Las entradas en un directorio pueden tener un nombre o un valor numérico. El primer campo de IMAGE_RESOURCE_DIRECTORY_ENTRY indica esto:

- Cuando en el primer campo de IMAGE_RESOURCE_DIRECTORY_ENTRY, el BIT más significativo es 1, los restantes 31 bits son el desplazamiento al nombre del recurso.

- Cuando en el primer campo de IMAGE_RESOURCE_DIRECTORY_ENTRY, el BIT más significativo es 0, los 16 bits menos significativos contienen el número que identifica.

Veamos un ejemplo de los recursos de ADVAPI32.DLL:

```
Resources (RVA: 6B000)
ResDir (0) Entries:03 (Named:01, ID:02) TimeDate:00000000
    ____________________________________
    ResDir (MOFDATA) Entries:01 (Named:01, ID:00) TimeDate:00000000
        ResDir (MOFRESOURCENAME) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000128
            DataRVA: 6B6F0  DataSize: 190F5  CodePage: 0
    ____________________________________
    ResDir (STRING) Entries:01 (Named:00, ID:01) TimeDate:00000000
        ResDir (C36) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000138
            DataRVA: 6B1B0  DataSize: 0053C  CodePage: 0
    ____________________________________
    ResDir (RCDATA) Entries:01 (Named:00, ID:01) TimeDate:00000000
        ResDir (66) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000148
            DataRVA: 85908  DataSize: 0005C  CodePage: 0
```

Cada línea que comienza con “ResDir” corresponde a una estructura IMAGE_RESOURCE_DIRECTORY. Seguido a “ResDir” se encuentra el nombre del recurso, entre paréntesis (en el ejemplo “0”, “MOFDATA”, “STRING”, etc). Seguido al nombre se encuentra la cantidad de entradas en el directorio.

Reubicaciones Base "Base Relocations"

En muchas partes del archivo, podemos encontrar direcciónes de memoria. Cuando un ejecutable es enlazado, se le coloca una dirección de carga "preferida" (recordar que vimos que está información se encuentra en la estructura IMAGE_FILE_HEADER). Esta dirección solo es valida si el ejecutable es cargado en la misma.

Si el cargador necesita colocar el archivo PE en otra dirección de memoria, todas las direcciones que se utilizaron serán incorrectas (no las RVAs). Esto implicara trabajo extra al cargador.

La información de reubicaciones le dira al cargador cada posición en el archivo que necesita ser modificada en caso de que el mismo no se cargue en su dirección "preferida". El cargador no necesita conocer detalles de sobre como se utiliza esa dirección, solo necesita saber los lugares que necesitan ser modificados en forma consistente.

Veamos un ejemplo para tener la situación mas clara. Supongamos que tenemos la siguiente instrucción que carga el valor de una variable (que está en la dirección 0x0040D434) en el registro ECX:

```
00401020: 8B 0D 34 D4 40 00  mov ecx,dword ptr [0x0040D434]
```

La instrucción se encuentra en la dirección 0x00401020 y tiene 6 bytes de longitud. Los primeros 2 bytes (0x8B 0x0D) indican el código de instrucción. Los restantes 4 bytes almacenan una dirección (0x0040D434). Supongamos ademas, que está instrucción corresponde a un programa cuya dirección "preferida" de carga es of 0x00400000 y que la variable global se encuentra en la dirección RVA 0xD434.

Si el ejecutable se carga en la dirección 0x00400000, la instrucción puede ser ejecutada exactamente como está. Pero supongamos que se carga en la dirección 0x00500000. Si esto sucede, los ultimos 4 bytes de la instrucción necesitan ser cambiados a 0x0050D434.

El cargador compara que dirección de carga preferida con la actual dirección de carga y calcula un valor delta. Para el ejemplo, el valor delta seria 0x00100000. Este valor delta puede ser agregado al valor de la dirección de memoria de la instrucción para obtener la nueva dirección de la variable. En el ejemplo anterior, deberia existir información para la reubicación de la dirección 0x00401022, que es la ubicación de la dirección de memoria de la instrucción. (Notar que guarda la dirección de la dirección donde en encuentra la dirección de memoria a cambiar y no la de la instrucción).

En pocas palabras, las reubicaciones son una lista de ubicaciones (direcciónes de memoria) en el ejecutable donde el valor "delta" necesita ser sumado al valor que existe actualmente. La información sobre reubicaciones se encuentra en una sección llamada ".reloc", pero la forma

correcta de encontrarla es por medio de la entrada IMAGE_DIRECTORY_ENTRY_BASERELOC en el Data Directory.

La información de reubicaciones es un arreglo de estructuras IMAGE_BASE_RELOCATION. El campo "VirtualAddress" de la está estructura contiene la dirección RVA del rango de memoria donde la reubicación comienza (base). El campo "SizeOfBlock" indica cuantos bytes componen el rango de reubicaciones para está base, incluyendo el tamaño de IMAGE_BASE_RELOCATION.

Inmediatamente a IMAGE_BASE_RELOCATION hay un número variable de valores WORD. La cantidad de estos valores puede ser deducidad del campo "SizeOfBlock". Cada valor WORD consiste en 2 partes:

- Los 4 bits mas significativos indican el tipo de reubicación. Los diferentes tipos de reubicación se pueden encontrar definidos como constantes en WINNT.H, y sus nombres son de la forma IMAGE_REL_BASED_xxx
- Los 12 bits menos significativos son el desplazamiento relativos al valor indicado en el campo "VirtualAddress", donde la reubicación debe ser aplicada.

El header .NET

Los ejecutables producidos por Microsoft .NET son tambien archivos PE. Sin embargo, la mayoria del código y los datos es mínima. El proposito de un archivo .NET es llevar a memoria información especifica como metadata y el lenguaje intermedio (IL) que utiliza el framework.

Los archivos ejecutables .NET se encuentran ligados a MSCOREE.DLL. Está DLL es el punto de entrada un proceso .NET. Cuando el archivo .NET es cargado en memoria, el punto de entrada es un pequeño fragmento de código, el cual realiza un salto a una función exportada por MSCOREE.DLL (_CorExeMain or _CorDllMain).

A partir de alli, MSCOREE comienza la ejecución utilizando la metadata y el IL el archivo ejecutable.

El punto de entrada para la información .NET está definido en la estructura IMAGE_COR20_HEADER. Está estructura esta apuntada por la entrada IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR en el Data Directory. Para una descripción detallada de la estructura IMAGE_COR20_HEADER consultar Ref. 7.

Referencias

1. http://en.wikipedia.org/wiki/COFF (Junio 2010)
2. http://en.wikipedia.org/wiki/Portable_Executable (Junio 2010)
3. http://msdn.microsoft.com/en-us/library/ms809762.aspx (Junio 2010)
4. http://www.microsoft.com/whdc/system/platform/firmware/PECOFFdwn.mspx (Junio 2010)
5. http://msdn.microsoft.com/en-us/magazine/cc301805.aspx (Junio 2010)
6. http://msdn.microsoft.com/en-us/magazine/cc301727.aspx (Junio 2010)
7. http://msdn.microsoft.com/en-us/magazine/cc301808.aspx (Junio 2010)
8. http://www.wotsit.org/ (Junio 2010) – Sitio web con la definición de las estructuras que definen diferentes formatos de archivos, entre ellos PE, ELF, COFF, etc.
9. http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html (Junio 2010) - Herramienta para analizar archivos PE.
10. http://biew.sourceforge.net/ (Junio 2010) – Herramientas para analizar archivos PE, ELF y otros formatos de ejecutables