ESTRUTURA E FUNCIONAMENTO DE UM EXECUTÁVEL

Por: Fernando Birck (Fergo)

INTRODUÇÃO

Neste guia vamos falar um pouco sobre a estrutura e o funcionamento de um arquivo executável para o Microsoft Windows, mais conhecido pela sua extensão: EXE.
Creio que todo o usuário de Windows já ouviu alguma vez esse nome, e sabe que é o arquivo principal de qualquer aplicativo, contendo o código do programa compilado.

Os executáveis (ou binários), também recebem a denominação de PE. Essa sigla vem de “Portable Executable”, que em português significaria “Executável Portável”. Essa denominação vem de um padrão estabelecido pela Microsoft nos primórdios do Windows, onde decidiram criar um formato de binário capaz de rodar em qualquer outra versão do Windows. Teoricamente eles conseguiram, pois o formato do arquivo permaneceu inalterado desde o Windows 95.

Os arquivos PE não se restringem apenas aos EXE. O mesmo formato é utilizado para as bibliotecas de linkagem (DLL), componentes ActiveX (OCX), entre diversos outros. Isso significa que o padrão de todos esses arquivos é semelhante, variando apenas algumas funções.

Vamos dar uma atenção maior aos arquivos EXE, pois além de serem os mais “famosos”, são os que levam o formato PE da forma mais abrangente possível. Para tal, criei um pequeno aplicativo, contendo uma janela e um botão, que será o programa de testes, onde faremos as análises. Apesar de simples, é o suficiente para termos um executável completo e puro (programado em Assembly).

FERRAMENTAS

Para o nosso estudo, vamos precisar de somente algumas ferramentas.

- **Executável de testes**
  - Esse é o programa que vamos utilizar para realizar nossos "experimentos". Pequeno, sem funcionalidade nenhuma. Porém, é puro e completo.
  - www.fergonez.net/download.php?file=tut_teste.zip

- **WinHex**
  - Um editor Hexadecimal. Fundamental para qualquer análise de binários compilados. É nele que vemos identificar as estruturas do nosso executável PE.
  - http://www.winhex.com/winhex/

- **LordPE**
  - Um ótimo aplicativo para desmembrar executáveis e DLLs. Através dele é possível explorar com facilidade o formato complexo dos arquivos PE.
  - http://scifi.pages.at/yoda9k/LordPE/info.htm

ESTRUTURA E FUNCIONAMENTO

Um executável no formato PE possui uma estrutura um tanto quanto complexa, mas ao mesmo tempo muito organizada e versátil.

O arquivo é organizado basicamente desta maneira:

- Cabeçalho DOS
- Cabeçalho Windows
- Tabela de Seções
- Seção 1
- Seção 2
- Seção N...

O cabeçalho DOS não tem utilidade prática dentro do sistema Windows, ele serve apenas para apresentar uma mensagem avisando o usuário que o aplicativo em questão não pode ser utilizado em modo texto.

Já o cabeçalho do Windows é de extrema importância. É nele que estão todas as informações básicas necessárias para que o aplicativo funcione, como o número de seções, tamanho de cada seção e início das mesmas, onde iniciar a execução do código, dentro dezenas de outras configurações. Veremos isso mais adiante.

O EXE é divido em seções, que variam de acordo com o compilador utilizado ou que são modificadas pelo usuário. Cada seção fica responsável por uma característica no PE. As informações referentes a cada uma das seções ficam armazenadas na "Tabela de Seções". Abaixo estão listadas as seções mais comuns (e oficiais) de um binário para Win32.

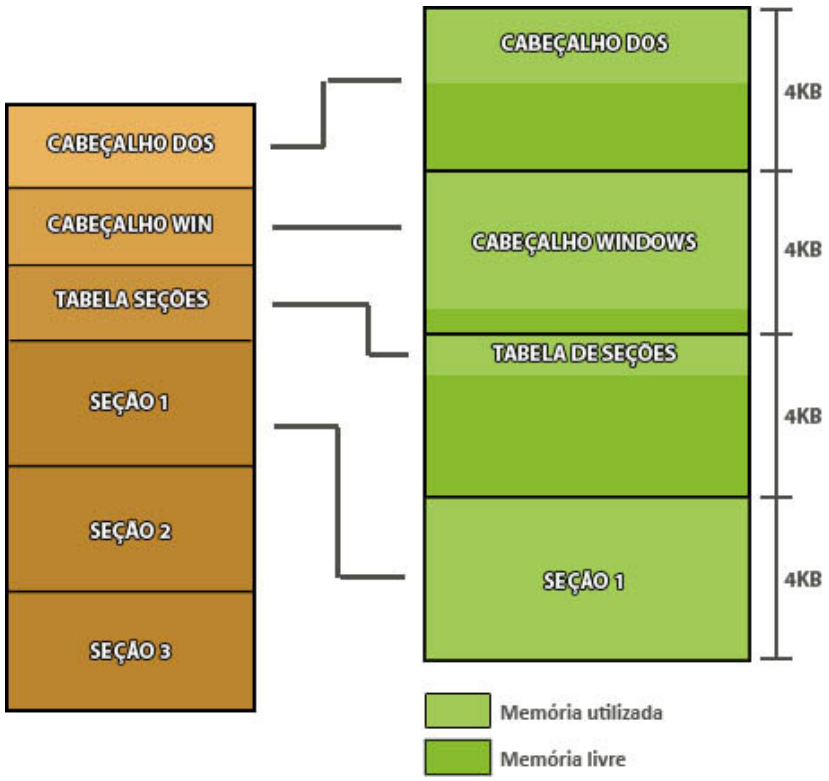
- Seção de código - Code Section (.text ou .code)
- Seção de recursos – Resource Section (.rsrc)
- Seção de dados – Data Section (.data)
- Seção de exportação – Export data section (.edata)
- Seção de importação - Import data section (.idata)
- Informações de debug - Debug information (.debug)

Veremos o que cada uma dessas seções comporta mais adiante, onde entraremos em detalhes mais técnicos.

Uma característica interessante sobre os arquivos PE é que eles são armazenados na memória mantendo a estrutura semelhante ao do arquivo no disco.
Quando o usuário requisita a execução de um aplicativo, o Windows Loader (parte do Kernel do Windows responsável por iniciar e organizar o binário na memória) analisa o cabeçalho do PE. Feito isso, ele possui as informações necessárias para poder copiar o executável do disco rígido para a RAM. No entanto, ele não é jogado para a RAM exatamente da mesma forma que ele se encontra no Windows. O Loader precisa fazer alguns ajustes.

Esses ajustes são necessários devido à forma com que o S.O. da Microsoft gerencia a memória, utilizando uma memória virtual paginada. Quando as seções são carregadas para a memória, o Windows alinha cada uma delas para caber em páginas de 4KB. É como se ele dividisse a RAM em diversos pedaços de 4KB e criasse um índice de cada trecho. Exemplo:

Supondo que você tem um trecho de dados com um tamanho de 5KB, e o Windows precisa alocar esses 5KB na memória. Inicialmente ele verifica no índice se existem páginas livres onde esses dados possam ser armazenados. Caso existam, ele vai colocar os primeiros 4KB em uma página, e os outros 1KB restantes na página seguinte. Nesta última, vão sobrar 3KB livres, que acabam sendo desperdiçados. A figura abaixo demonstra melhor a situação:



O conceito por trás da memória virtual é que ao invés de deixar o software controlar diretamente a memória, o programa chama o gerenciamento do Windows, que por sua vez vai consultar e analisar as leituras e gravações na RAM.

As vantagens por trás disso é a possibilidade de criar diversos espaços de endereçamento, que consiste em restringir o acesso a determinado trecho de memória somente ao aplicativo que originou a criação do mesmo, evitando que um aplicativo corrompa a memória utilizada por outra aplicação (como ocorria com os Win 9x).

Além do alinhamento na memória, ele também possui um alinhamento em disco. O alinhamento em disco segue a mesma teoria, mas as “páginas” não são divididas em 4KB, pois isso ocasionaria em um desperdício muito grande de espaço. No arquivo elas são dividas em trechos de 512 bytes (normalmente), o que explica o fato de qualquer executável padrão possui um tamanho múltiplo de 512.

Vamos então nos focar melhor em cada trecho do executável, começando pelo cabeçalho DOS.

ESPECIFICAÇÃO TÉCNICA

Cabeçalho DOS

O arquivo PE começa com um cabeçalho DOS que ocupa os primeiros 64 bytes do arquivo. A função deste cabeçalho é verificar se o executável é ou não um arquivo executável válido, assim como identificar se o programa pode ser rodado via MS-DOS ou necessita do Windows. Para o caso de aplicativos programados para o Windows, a única função do cabeçalho DOS é exibir esta mensagem (caso seja rodado a partir do MS-DOS):

"This program must be run under Microsoft Windows"

Este texto fica armazenado logo após o cabeçalho DOS, numa área chamada "*DOS Stub*". Essa área tem como função armazenar dados que possam ser utilizados na execução do arquivo. É no *DOS Stub* que fica armazenada a instrução para imprimir o texto destacado acima.

Abaixo vou adicionar a estrutura oficial desse cabeçalho, que é utilizada pelos programadores. Não há a necessidade de entender o significado de cada item, mas vou ressaltar os dois mais importantes.

```
IMAGE_DOS_HEADER STRUCT
  e_magic           WORD      ?
  e_cblp            WORD      ?
  e_cp              WORD      ?
  e_crlc            WORD      ?
  e_cparhdr         WORD      ?
  e_minalloc        WORD      ?
  e_maxalloc        WORD      ?
  e_ss              WORD      ?
  e_sp              WORD      ?
  e_csum            WORD      ?
  e_ip              WORD      ?
  e_cs              WORD      ?
  e_lfarlc          WORD      ?
  e_ovno            WORD      ?
  e_res             WORD    4 dup(?)
  e_oemid           WORD      ?
  e_oeminfo         WORD      ?
  e_res2            WORD   10 dup(?)
  e_lfanew          DWORD      ?
IMAGE_DOS_HEADER ENDS
```

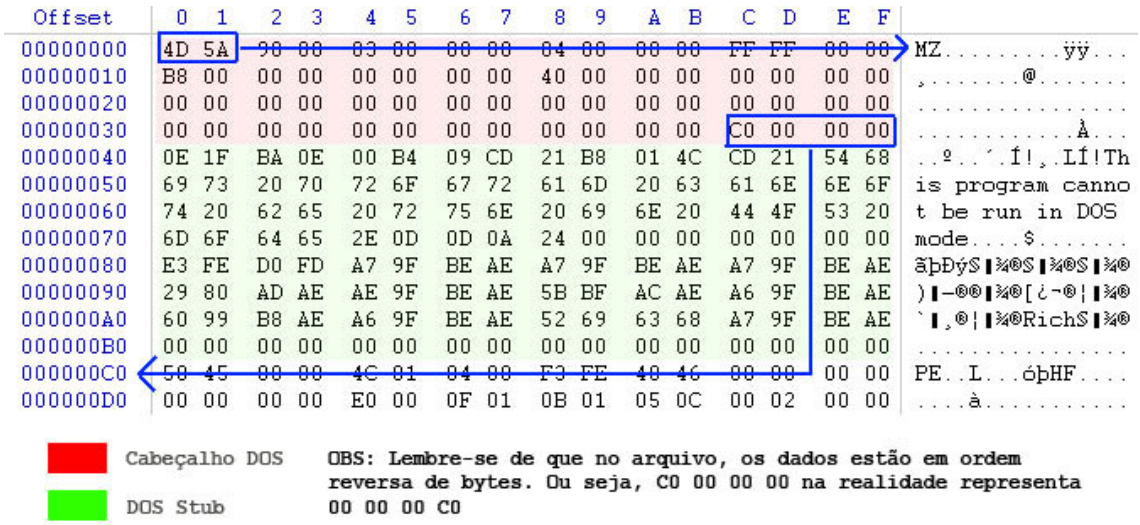
Como pode ver, temos diversos itens com tamanhos WORD e DWORD, que se forem somados, fecham os 64 bytes iniciais do cabeçalho. De todos esses nomes, vou destacar os 2 mais importantes.

- **e_magic** - É um valor de 2 bytes (WORD) que identifica um executável do DOS. Nesses 2 bytes ficam armazenados a sigla "MZ" (Mark Zbikowsky, um dos idealizadores do MS-DOS). Essa sigla é um dos dados que o Windows Loader

verifica na hora de rodar um aplicativo, e se ela não existir, ele deixa de reconhecer o arquivo como executável.

- **e_lfanew** - Armazena o offset (posição) no arquivo onde está localizado o cabeçalho WIN (falaremos dele logo em seguida).

Veja a imagem abaixo, que representa a estrutura do cabeçalho DOS, dentro do aplicativo de testes (utilize o WinHex para visualizar, caso queira)



Nesta imagem podemos notar claramente aqueles dois dados mencionados anteriormente. Os dois primeiros bytes (4D 5A) compõe o "*e_magic*", contendo a sigla MZ (valores ASCII para 4D e 5A). Já no final do cabeçalho DOS (offset 0000003Ch) nos temos os "*e_lfanew*", que indica o local no arquivo onde está localizado o cabeçalho PE (veja a seta indicativa).

Cabeçalho Windows

O cabeçalho Windows, ou cabeçalho PE, contém as informações fundamentais para o aplicativo. É nele que estão indicadas todas as características do arquivo.

Ele é composto por um conjunto de estruturas, que variam de tamanho conforme a complexidade do aplicativo e/ou o número de seções que nele estão armazenados.

A primeira dessas estruturas é o cabeçalho do NT

```
IMAGE_NT_HEADERS STRUCT
  Signature          DWORD                   ?
  FileHeader         IMAGE_FILE_HEADER       <>
  OptionalHeader     IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

Como podemos notar, ele é composto por 3 itens. O primeiro ("*Signature*") possui a mesma função do "*e_magic*". Ele apenas identifica o cabeçalho NT, e deve ser composto pela sigla PE, seguido de dois bytes nulos, fechando os 4 bytes da DWORD

Em seguida temos o "*FileHeader*", ocupando os próximos 20 bytes do cabeçalho NT, contendo informações sobre a estrutura física do arquivo executável. Veja a estrutura do "*FileHeader*" abaixo:

```
IMAGE_FILE_HEADER STRUCT
  Machine                WORD    ?
  NumberOfSections       WORD    ?
  TimeDateStamp          DWORD   ?
  PointerToSymbolTable   DWORD   ?
  NumberOfSymbols        DWORD   ?
  SizeOfOptionalHeader   WORD    ?
  Characteristics        WORD    ?
IMAGE_FILE_HEADER ENDS
```

Dessa estrutura, os dados mais importantes são:

- **NumberOfSections** – Indica o número de seções contida no aplicativo (reveja a introdução, caso necessário.

- **Characteristics** – Informa se o arquivo em questão se trata de um EXE, DLL ou OCX.

Voltando ao cabeçalho NT, temos por último uma outra estrutura, chamada de "*OptionalHeader*". Apesar do nome, ela deve SEMPRE existir. Essa estrutura possui um tamanho de 224 bytes, sendo que os últimos 128 são reservados para o diretório de dados, que veremos adiante.

É certamente a maior estrutura, contendo o maior número de valores.

```
IMAGE_OPTIONAL_HEADER32 STRUCT
  Magic                        WORD        ?
  MajorLinkerVersion           BYTE        ?
  MinorLinkerVersion           BYTE        ?
  SizeOfCode                   DWORD       ?
  SizeOfInitializedData        DWORD       ?
  SizeOfUninitializedData      DWORD       ?
  AddressOfEntryPoint          DWORD       ?
  BaseOfCode                   DWORD       ?
  BaseOfData                   DWORD       ?
  ImageBase                    DWORD       ?
  SectionAlignment             DWORD       ?
  FileAlignment                DWORD       ?
  MajorOperatingSystemVersion  WORD        ?
  MinorOperatingSystemVersion  WORD        ?
  MajorImageVersion            WORD        ?
  MinorImageVersion            WORD        ?
  MajorSubsystemVersion        WORD        ?
  MinorSubsystemVersion        WORD        ?
  Win32VersionValue            DWORD       ?
  SizeOfImage                  DWORD       ?
  SizeOfHeaders                DWORD       ?
  CheckSum                     DWORD       ?
  Subsystem                    WORD        ?
  DllCharacteristics           WORD        ?
  SizeOfStackReserve           DWORD       ?
```

```
  SizeOfStackCommit             DWORD       ?
  SizeOfHeapReserve             DWORD       ?
  SizeOfHeapCommit              DWORD       ?
  LoaderFlags                   DWORD       ?
  NumberOfRvaAndSizes           DWORD       ?
  DataDirectory                 IMAGE_DATA_DIRECTORY
IMAGE_OPTIONAL_HEADER32 ENDS
```

Bastante coisa não? Os nomes das variáveis na maioria dos casos explicam o seu propósito, mas como fiz anteriormente, colocarei aqui uma explicação mais profunda sobre algum desses valores.

- **AddressOfEntryPoint** – Indica o endereço relativo (RVA – *Relative Virtual Address*) da primeira instrução a ser executada pelo aplicativo, assim que carregado na memória. Para maiores informações sobre endereços relativos, veja ao apêndice no final do guia.

- **ImageBase** – E a posição no espaço relativo da memória (restrita ao aplicativo) que o Windows carregará o aplicativo. Na maioria dos casos, é utilizado o VA (*Virtual Address*, endereço relativo) 400000h.

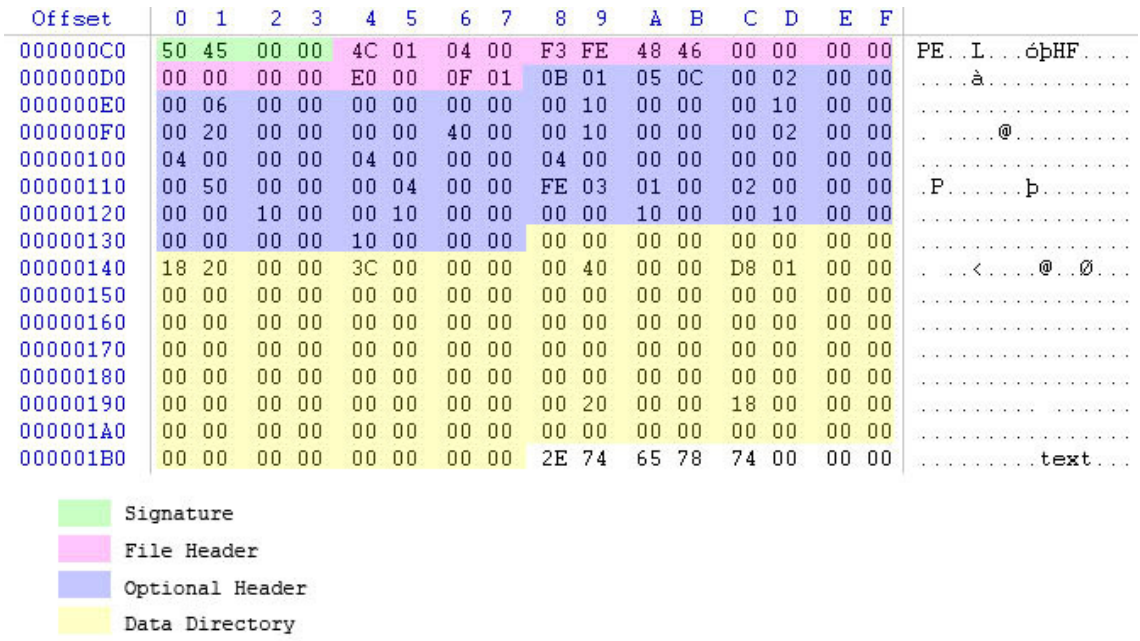
- **SectionAligment** – É o alinhamento de cada uma das seções do executável na memória. Nós falamos um pouco sobre isso na introdução deste guia, e lá foi mencionado que normalmente se utiliza um tamanho de 4096 bytes (4KB), logo, o valor do *SectionAligment* costuma ser 1000h (1000 em hexadecimal representa o valor 4096 no sistema decimal.

- **FileAligment** – Semelhante ao *SectionAligment*, mas representa o alinhamento das seções no arquivo em disco, e não na memória. Normalmente as seções ficam alinhadas em trechos de 512 bytes, o que nos daria o valor 200h em hexadecimal.

- **SizeOfImage** – Tamanho total do arquivo PE após carregado na memória, incluindo os espaços vazios deixados pelo *SectionAlignment*.

- **DataDirectory** – 16 estruturas do tipo *IMAGE_DATA_DIRECTORY*. Essas estruturas (mais precisamente, diretórios) contêm informações referentes às seções dentro do executável, como a tabela de Imports/Export, Code, Data, etc. Analisaremos ela detalhadamente mais a diante.

Veja a imagem abaixo, que ilustra o cabeçalho WIN (PE Header) dentro do editor Hexadecimal:



Uma outra forma de visualizar o cabeçalho do arquivo PE é utilizando algum programa específico para isso, como o caso do PEiD, LordPE ou até mesmo um debugger com a opção de desmembrar o cabeçalho (como é o caso do OllyDbg).

Para finalizar o cabeçalho Windows, precisamos falar sobre o *IMAGE_DATA_DIRECTORY*. Como mencionado logo acima, ele compõe os últimos 128 bytes do PE Header sendo uma estrutura importante, contendo o endereço (RVA) e tamanho dos diretórios do executável. Segue abaixo a estrutura do *IMAGE_DATA_DIRECTORY*:

```
IMAGE_DATA_DIRECTORY STRUCT
  VirtualAddress            DWORD       ?
  ISize                     DWORD       ?
IMAGE_DATA_DIRECTORY ENDS
```

Um tanto quanto simples. Podem ver que se trata apenas de dois valores DWORD (cada um com 4 bytes, totalizando 8 por estrutura). Essa estrutura é utilizada pelos 16 diretórios de dados, que são listados a seguir:

```
IMAGE_DIRECTORY_ENTRY_EXPORT                equ  0
IMAGE_DIRECTORY_ENTRY_IMPORT                equ  1
IMAGE_DIRECTORY_ENTRY_RESOURCE              equ  2
IMAGE_DIRECTORY_ENTRY_EXCEPTION             equ  3
IMAGE_DIRECTORY_ENTRY_SECURITY              equ  4
IMAGE_DIRECTORY_ENTRY_BASERELOC             equ  5
IMAGE_DIRECTORY_ENTRY_DEBUG                 equ  6
IMAGE_DIRECTORY_ENTRY_COPYRIGHT             equ  7
IMAGE_DIRECTORY_ENTRY_GLOBALPTR             equ  8
IMAGE_DIRECTORY_ENTRY_TLS                   equ  9
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG           equ 10
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT          equ 11
IMAGE_DIRECTORY_ENTRY_IAT                   equ 12
```

Para cada uma dessas entradas há uma estrutura do tipo *IMAGE_DATA_DIRECTORY*. Temos 16 entradas de diretórios, e cada uma delas possui oito bytes, totalizando os 128 bytes finais do cabeçalho WIN. Veja a marcação em amarelo na imagem da página anterior (pode verificar que há 128 bytes marcados em amarelo).

Podemos agora sair do cabeçalho WIN e partir para a tabela de seções

Tabela de Seções

A tabela de seções funciona de forma semelhante ao *IMAGE_DATA_DIRECTORY*. Nessa tabela estão contidas diversas informações referentes a cada uma das seções presente no executável (como o tamanho, endereço e características). A quantidade de itens na tabela de vai variar dependendo do número de seções contidos no aplicativo (pode ser obtida na entrada *NumberOfSections* do cabeçalho WIN).

```
IMAGE_SECTION_HEADER STRUCT
    Name1                   BYTE IMAGE_SIZEOF_SHORT_NAME dup(?)
    union Misc
        PhysicalAddress     DWORD  ?
        VirtualSize         DWORD  ?
    ends
    VirtualAddress          DWORD  ?
    SizeOfRawData           DWORD  ?
    PointerToRawData        DWORD  ?
    PointerToRelocations    DWORD  ?
    PointerToLinenumbers    DWORD  ?
    NumberOfRelocations     WORD   ?
    NumberOfLinenumbers     WORD   ?
    Characteristics         DWORD  ?
IMAGE_SECTION_HEADER ENDS

IMAGE_SIZEOF_SHORT_NAME     equ 8
```

- **Name1** – Nome da seção, apenas para diferenciar as seções. Não tem efeito real sobre o aplicativo. No máximo 8 caracteres.

- **VirtualAddress** – RVA do início da sessão. O valor aqui contido é somado com o *ImageBase* do cabeçalho WIN para ter o endereço real da seção.

- **SizeOfRawData** – Tamanho total da seção em disco, levando em consideração o alinhamento utilizado na compilação (512 bytes, para o nosso caso).

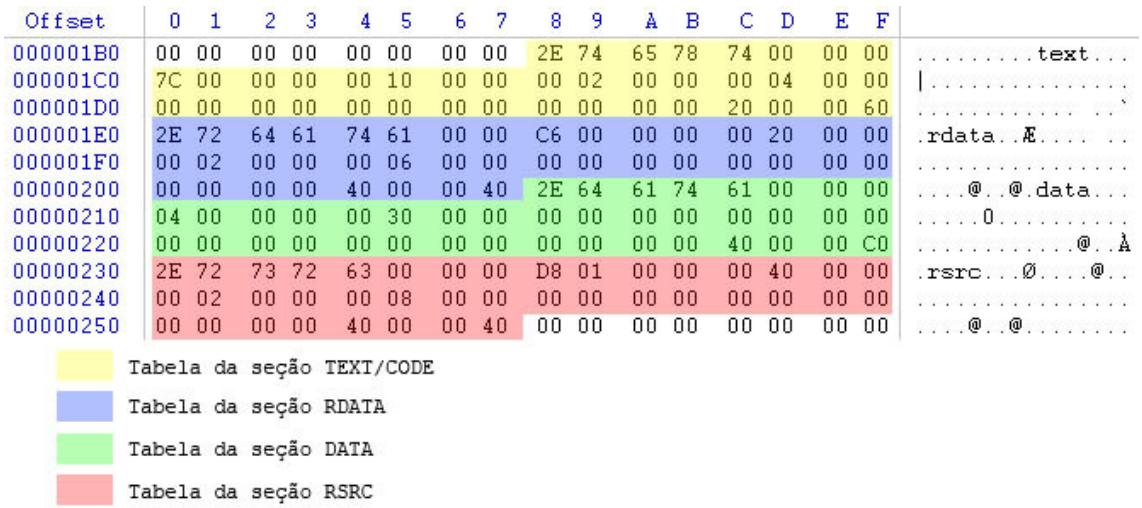
- **PointerToRawData** – Posição da seção dentro do arquivo (não na memória). Esse valor nos dá diretamente a posição da seção dentro do arquivo, podendo ser facilmente encontrada em um editor hexadecimal.

- **Characteristics** – São as características propriamente ditas da seção. Se ela é de somente leitura/escrita, possui dados não inicializados, etc.

Na introdução do tutorial vimos que existem diversas seções dentro de um arquivo PE, sendo algumas delas “oficiais”. O nosso aplicativo de teste é composto por apenas 4 seções:

- **CODE** (.text) - Contém as instruções e o código do programa.
- **RDATA** (.rdata) – Dados gerais (incluindo tabela de seções).
- **DATA** (.data) – Variáveis inicializadas.
- **RSRC** (.rsrc) – Resources (textos e disposição de itens na janela).

Podemos então analisar a tabela dessas 4 seções dentro do WinHex:



As seções

Como foi dito anteriormente, o arquivo PE pode conter infinitas seções, sendo que algumas delas são oficiais e estão presentes na maioria dos executáveis. Abaixo vamos descrever qual a função de cada uma

- **Seção de código** (CODE/TEXT)

  Dentro desta seção fica armazenado o código compilado do aplicativo, contendo todas as instruções em assembly para o funcionamento do programa. Qualquer alteração feita no código de um aplicativo vai resultar numa mudança dos dados presentes dentro deste trecho do arquivo.

- **Seção de dados** (DATA)

  Essa seção pode ser subdivida em 3 outras seções, sendo elas:

  - **BSS** – Contém todas as variáveis não inicializadas (sem um valor definido) do aplicativo.
  - **RDATA** – Dados de somente leitura. Podem ser strings, constantes ou até mesmo dados da *Import Table*.

- **DATA –** Todas as outras variáveis que não se encaixam em nenhuma das duas outras seções

- **Seção de recursos** (RSRC)

  Esse trecho é utilizado para armazenar qualquer outro tipo de dado dentro de um executável. Nela ficam armazenados os ícones, imagens, disposição dos itens na janela, menus, etc.

  Ela é um pouco diferente das outras seções, pois possui uma outra subdivisão interna, separando cada recurso. Um bom modo de ver essas subdivisões e os dados nela contidos é utilizar um "*Resource Editor*", facilmente encontrado na internet. Recomendo o *ResHack*, que é gratuito e poderoso.

- **Seção de exportação** (EDATA)

  Armazena o diretório de exportação, contendo informações sobre os nomes e endereços das funções contidas em uma DLL.

  Os arquivos DLL podem ser definidos por 2 tipos de funções: as internas e externas. As externas podem ser chamadas por qualquer outro módulo. Já as funções internas ficam restritas ao módulo "dono" da mesma.

  As DLLs nos dão a possibilidade de "modularizar" aplicativos, contendo funções genéricas que podem ser utilizadas por qualquer programa. Um bom exemplo disso é o próprio Kernel do Windows, que é subdividido em diversas DLLs que controlam o sistema (kernel.dll, user32.dll, gdi32.dll, entre outras).

- **Seção de importação** (IDATA)

  Esta seção funciona de forma semelhante a anterior. Ao invés de ser voltada para os arquivos DLL (como a de exportação), a seção de importação tem a finalidade de montar um bando de dados de todas as funções utilizadas por um executável, assim como o endereço e as características de cada rotina importada. Seria como dizer que a seção de exportação "fornece" funções para o uso e a de importação "busca" essas funções exportadas.

  Poderíamos explorar melhor a seção de importação, mas ela é um tanto quanto complexa, e ficaria um pouco fora do intuito deste guia.
  Caso queira maiores informações, verifique no site da Microsoft pelo formato e funcionamento da API do Windows.

- **Seção de debug** (DEBUG)

  Presente normalmente nas compilações de aplicativos em estágio de desenvolvimento, essa seção contém dados úteis para o programador, que podem o auxiliar no tratamento de erros.

APÊNDICE – ENDEREÇAMENTO VIRTUAL

Entendendo o endereçamento virtual

O Windows trabalha com uma forma de endereçamento virtual de memória. Isso quer dizer basicamente que os aplicativos não trabalham com endereços absolutos baseados no arquivo, mas sim na memória. Podemos citar três formas de endereçamento: Offset, VA e RVA.

- **Offset - RawOffset**

  Indica o posicionamento "bruto" de algo dentro de um arquivo.
  Por exemplo: *PointerToRawData* na tabela de seções trabalha com offsets, pois indica em qual byte (e não endereço de memória) no arquivo executável se encontra determinada seção.

- **VA – Virtual Address**

  Endereço virtual "absoluto" na memória. Coloquei entre aspas, pois ele é absoluto apenas quando trabalhamos com o espaço de endereçamento criado para o aplicativo, e não para a toda a memória presente no computador. O VA "começa" no valor determinado pelo *ImageBase*, no cabeçalho PE

- **RVA – Relative Virtual Address**

  É o endereço relativo a partir do início do endereçamento de memória destinado ao aplicativo em questão.

Tendo essas diferenciações, podemos formar pequenas equações que talvez esclareçam um pouco as coisas:

RVA = VA – ImageBase

VA = RVA + ImageBase

Vamos para um exemplo. Suponha que um aplicativo qualquer possua um *ImageBase* com valor 400000h. O VA relativo ao início do espaço de memória destinado ao aplicativo passa a ser 400000h. Suponho agora que o aplicativo inicie sua execução no RVA 1000h. Pela formula acima, podemos descobrir que o VA do início da execução do programa está no endereço 401000h (RVA + ImageBase = 400000h + 1000h).

Não é tão complicado quanto parece, só é preciso cautela para não confundir as nomenclaturas.

Conversão entre Offset e VA

Para fazer a conversão de um Offset (posição de determinado byte no arquivo executável) para um VA, é necessário conhecer alguns dados do aplicativo.

Primeiramente devemos saber em qual seção o nosso offset está localizado. Para isso basta comparar o offset que você possui com o *PointerToRawData* e o *SizeOfRawData* de cada uma das seções. Fica mais fácil de entender através de um exemplo (vou utilizar o nosso aplicativo de testes).

Digamos que eu queira descobrir o VA do offset 00000900h. Abaixo está uma tabela com o *RawOffset* e o *RawSize* de cada seção (retirado da tabela de seções)

Seção	RawOffset	RawSize	VirtualOffset
.TEXT	00000400h	00000200h	00001000h
.RDATA	00000600h	00000200h	00002000h
.DATA	00000000h	00000000h	00003000h
.RSRC	00000800h	00000200h	00004000h

Está bem claro que o nosso offset está contido dentro da seção de recursos, pois ela vai de 00000800h até 0000A00h (800h + 200h) e o nosso offset está bem no meio dela (00000900h).

Sabemos então que o nosso offset está 100h bytes a frente do início da seção de recursos. Como vimos que as seções são copiadas para a memória da mesma forma que elas estão no arquivo em disco, o VA que queremos descobrir também está 100h bytes a frente do *VirtualOffset* da seção de recursos. Então basta somar o 100h ao *VirtualOffset* da seção e incluir o *ImageBase*.

VA = RawOffset – RawOffset da seção + VirtualOffset da seção + ImageBase

No nosso exemplo, tendo a *ImageBase* como 00400000h, esses cálculos seriam:

VA = 00000900h – 00000800h + 00004000h + 00400000h
VA = 00404100h

Analiticamente, também é possível descobrir o *Offset* através de um VA:

RawOffset = VA – VirtualOffset da seção – ImageBase + RawOffset da seção

Fazendo o processo inverso para o nosso exemplo, tendo o VA 00404100h e querendo saber o *RawOffset*:

RawOffset = 00404100h – 00004000h – 00400000h + 00000800h
RawOfsset = 00000900h

CONCLUSÃO

É isso aí. Espero que esse tutorial tenha atingido o seu objetivo, que era de dar uma visão geral sobre o formato dos executáveis, assim como colocar informações úteis para programadores que pretendem se aventurar nesse mundo. Deixei de lado algumas informações, como a seção de relocação, pois é raro de aparecer.

Fiquei satisfeito com o resultado e devo dizer que ao mesmo tempo em que escrevia este artigo, aprendi algumas coisas novas sobre o formato, que passaram despercebidas quando eu comecei a me interessar pelo assunto.

A vantagem de entender e dominar esse tipo de arquivo é que você passa a ter a possibilidade de "customizar" o executável, seja para modificar ou proteger seu software, alterando um pouco a disposição e os endereços padrões estabelecidos. Para quem um dia pensa em fazer um compilador, editor de recursos ou simplesmente um visualizador de arquivos PE, creio que este tutorial possa ajudar.

Gostaria de deixar um agradecimento especial ao fórum Guia do Hardware, por ceder um espaço onde eu possa publicar estes artigos, assim como receber críticas e sugestões do mesmo.

Obrigado e até a próxima!

Artigo escrito por Fernando Birck - 2007
Website: www.fergonez.net

REFERÊNCIAS

- **The Portable Executable Format**
  - http://www.nikse.dk/petxt.html

- **Iczelion Win32 Assembly**
  - http://win32assembly.online.fr

- **Win32 Programming Reference**
  - ftp://ftp.cs.virginia.edu/pub/lcc-win32/win32hlp.exe

- **Windows EXE File Structure – Microsoft**
  - http://www.wotsit.org/download.asp?f=exe-win&sc=233430147