

# O formato de arquivo executável portátil de cima para baixo

Randy Kath  
Microsoft Developer Network Technology Group

[Baixe o exemplo EXEVIEW. \(Depois, use PKUNZIP.EXE -d para recriar a estrutura de diretório.\)](#)

[Baixe a amostra PEFILE. \(Depois, use PKUNZIP.EXE -d para recriar a estrutura de diretório.\)](#)

## Abstrato

O sistema operacional Windows NT™ versão 3.1 introduz um novo formato de arquivo executável chamado o formato de arquivo Portable Executable (PE). A especificação Portable Executable File Format, embora bastante vaga, foi disponibilizada ao público e está incluída no CD da Microsoft Developer Network (especificações e estratégia, especificações, especificações de formato de arquivo do Windows NT).

No entanto, essa especificação sozinha não fornece informações suficientes para tornar mais fácil, ou até mesmo razoável, que os desenvolvedores entendam o formato de arquivo PE. Este artigo destina-se a resolver esse problema. Nele, você encontrará uma explicação completa de todo o formato de arquivo PE, juntamente com descrições de todas as estruturas necessárias e exemplos de código-fonte que demonstram como usar essas informações.

Todos os exemplos de código-fonte que aparecem neste artigo são obtidos de uma biblioteca de vínculo dinâmico (DLL) chamada PEFILE.DLL. Eu escrevi essa DLL simplesmente com o objetivo de obter as informações importantes contidas em um arquivo PE. A DLL e seu código-fonte também estão incluídos neste CD como parte do aplicativo de amostra PEFile; sinta-se à vontade para usar a DLL em seus próprios aplicativos. Além disso, sinta-se à vontade para pegar o código-fonte e construir sobre ele para qualquer propósito específico que você possa ter. No final deste artigo, você encontrará uma breve lista das funções exportadas do PEFILE.DLL e uma explicação de como usá-las. Acho que você encontrará essas funções para facilitar o entendimento do formato do arquivo PE.

## Introdução

A recente adição do sistema operacional Microsoft® Windows NT™ à família de sistemas operacionais Windows™ trouxe muitas alterações ao ambiente de desenvolvimento e mais algumas mudanças nos próprios aplicativos. Uma das mudanças mais significativas é a introdução do formato de arquivo Portable Executable (PE). O novo formato de arquivo PE se baseia principalmente na especificação COFF (Common Object File Format) que é comum aos sistemas operacionais UNIX®. No entanto, para permanecer compatível com versões anteriores dos sistemas operacionais MS-DOS® e Windows, o formato de arquivo PE também mantém o antigo cabeçalho MZ familiar do MS-DOS.

Neste artigo, o formato de arquivo PE é explicado usando uma abordagem de cima para baixo. Este artigo discute cada um dos componentes do arquivo como eles ocorrem quando você percorre o conteúdo do arquivo, começando na parte superior e trabalhando seu caminho através do arquivo.

Grande parte da definição de componentes de arquivo individuais vem do arquivo WINNT.H, um arquivo incluído no SDK (Software Development Kit) do Microsoft Win32™ para Windows NT. Nele você encontrará definições de tipo de estrutura para cada um dos cabeçalhos de arquivo e diretórios de dados usados para representar vários componentes no arquivo. Em outros locais no arquivo, o WINNT.H não tem definição suficiente da estrutura do arquivo. Nestes locais, escolhi definir minhas próprias estruturas que podem ser usadas para acessar os dados do arquivo. Você encontrará essas estruturas definidas no PEFILE.H, um arquivo usado para criar o PEFILE.DLL. Todo o conjunto de arquivos de desenvolvimento PEFILE.H está incluído no aplicativo PEFile.

Além do código de exemplo PEFILE.DLL, um aplicativo de exemplo separado baseado em Win32 chamado EXEVIEW.EXE acompanha este artigo. Este exemplo foi criado para duas finalidades: Primeiro, eu precisava de uma maneira de testar as funções PEFILE.DLL, que em alguns casos exigiam várias visualizações de arquivos simultaneamente - daí o suporte a múltiplas visualizações. Em segundo lugar, muito do trabalho de descobrir o formato de arquivo PE envolve a capacidade de ver os dados de forma interativa. Por exemplo, para entender como a tabela de nomes de endereços de importação está estruturada, tive que visualizar o cabeçalho da seção .idata, o diretório de dados da imagem de importação, o cabeçalho opcional e o corpo da seção .idata real, todos simultaneamente. EXEVIEW.EXE é o exemplo perfeito para visualizar essas informações.

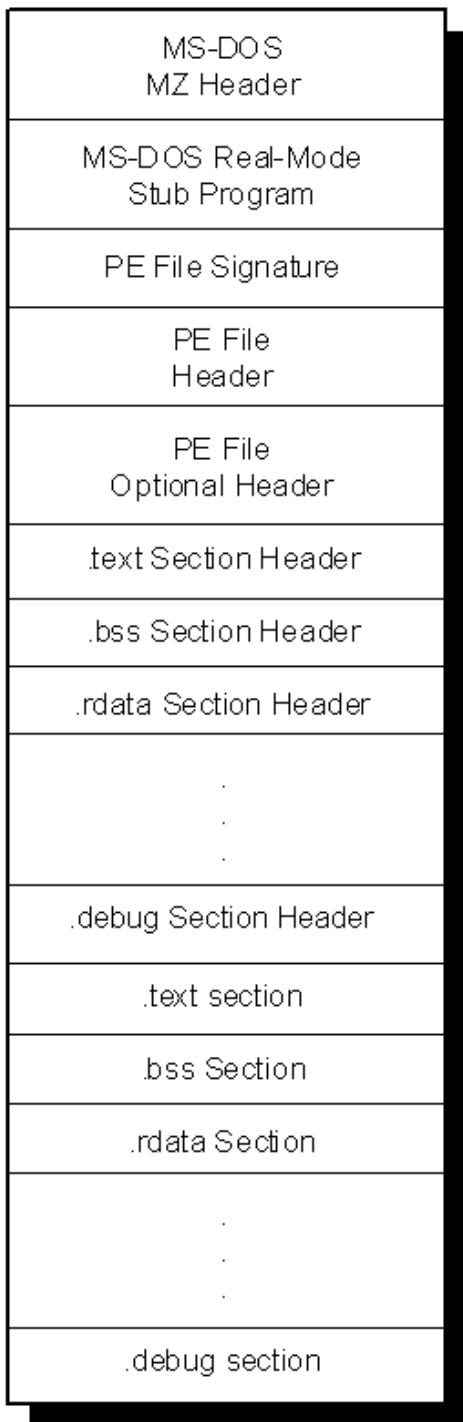
Sem mais delongas, vamos começar.

## Estrutura de arquivos PE

O formato de arquivo PE é organizado como um fluxo linear de dados. Ele começa com um cabeçalho do MS-DOS, um stub de programa em modo real e uma assinatura de arquivo PE. Imediatamente a seguir é um cabeçalho de arquivo PE e cabeçalho opcional. Além disso, todos os cabeçalhos de seção aparecem, seguidos por todos os corpos da seção. Fechando o arquivo estão algumas outras regiões de informações diversas, incluindo informações de realocação,

informações de tabela de símbolos, informações de números de linha e dados da tabela de cadeias. Tudo isso é mais facilmente absorvido olhando-o graficamente, como mostrado na Figura 1.

## PE File Format



**Figura 1. Estrutura de uma imagem de arquivo executável portátil**

Começando com a estrutura de cabeçalho de arquivo do MS-DOS, cada um dos componentes no formato de arquivo PE é discutido abaixo na ordem em que ocorre no arquivo. Grande parte dessa discussão é baseada em código de exemplo que demonstra como obter as informações no arquivo. Todo o código de amostra é obtido do arquivo PEFILE.C, o módulo de origem para PEFILE.DLL. Cada um desses exemplos aproveita um dos recursos mais interessantes do Windows NT, arquivos mapeados na memória. Os arquivos mapeados na memória permitem o uso de desreferência de ponteiro simples para acessar os dados contidos no arquivo. Cada um dos exemplos usa arquivos mapeados na memória para acessar dados em arquivos PE.

**Observação** Consulte a seção no final deste artigo para obter uma discussão sobre como usar o PEFILE.DLL.

## **Cabeçalho do modo MS-DOS / Real**

Como mencionado acima, o primeiro componente no formato de arquivo PE é o cabeçalho do MS-DOS. O cabeçalho do MS-DOS não é novo para o formato de arquivo PE. É o mesmo cabeçalho do MS-DOS que existe desde a versão 2 do sistema operacional MS-DOS. A principal razão para manter a mesma estrutura intacta no início do formato de arquivo

PE é que, quando você tenta carregar um arquivo criado no Windows versão 3.1 ou anterior, ou MS DOS versão 2.0 ou posterior, o sistema operacional pode ler o arquivo e entender que não é compatível. Em outras palavras, quando você tenta executar um executável do Windows NT no MS-DOS versão 6.0, você recebe esta mensagem: "Este programa não pode ser executado no modo DOS." Se o cabeçalho do MS-DOS não foi incluído como a primeira parte do formato de arquivo PE,

O cabeçalho do MS-DOS ocupa os primeiros 64 bytes do arquivo PE. Uma estrutura representando seu conteúdo é descrita abaixo:

## WINNT.H

```
typedef struct _IMAGE_DOS_HEADER { // cabeçalho do DOS .EXE
    USHORT e_magic; // Número mágico
    USHORT e_cblp; // Bytes na última página do arquivo
    USHORT e_cp; // Páginas no arquivo
    USHORT e_crlc; // Realocações
    USHORT e_cparhdr; // Tamanho do cabeçalho nos parágrafos
    USHORT e_minalloc; // Parâmetros mínimos extras necessários
    USHORT e_maxalloc; // Máximo de parágrafos extras necessários
    USHORT e_ss; // Valor inicial (relativo) de SS
    USHORT e_sp; // Valor inicial de SP
    USHORT e_csum; // Checksum
    USHORT e_ip; // Valor inicial do IP
    USHORT e_cs; // Valor inicial do CS (relativo)
    USHORT e_lfarlc; // Endereço do arquivo da tabela de realocação
    USHORT e_ovno; // Número de sobreposição
    USHORT e_res [4]; // Palavras reservadas
    USHORT e_oemid; // identificador OEM (para e_oeminfo)
    USHORT e_oeminfo; // informações do OEM; e_oemid specific
    USHORT e_res2 [10]; // Palavras reservadas
    LONG e_lfanew; // Endereço do arquivo do novo cabeçalho exe
} IMAGE_DOS_HEADER, * PIMAGE_DOS_HEADER;
```

O primeiro campo, **e\_magic**, é o chamado número mágico. Este campo é usado para identificar um tipo de arquivo compatível com o MS-DOS. Todos os arquivos executáveis compatíveis com o MS-DOS definem esse valor para 0x54AD, que representa os caracteres ASCII **MZ**. Por vezes, os cabeçalhos MS-DOS são referidos como cabeçalhos MZ. Muitos outros campos são importantes para sistemas operacionais MS-DOS, mas para o Windows NT, há realmente um campo mais importante nessa estrutura. O campo final, **e\_lfanew**, é um deslocamento de 4 bytes no arquivo onde o cabeçalho do arquivo PE está localizado. É necessário usar esse deslocamento para localizar o cabeçalho PE no arquivo. Para arquivos PE no Windows NT, o cabeçalho do arquivo PE ocorre logo após o cabeçalho do MS-DOS com apenas o programa stub de modo real entre eles.

## Programa de esboço em modo real

O programa stub de modo real é um programa real executado pelo MS-DOS quando o executável é carregado. Para um arquivo de imagem executável do MS-DOS real, o aplicativo começa a executar aqui. Para sistemas operacionais sucessivos, incluindo Windows, OS / 2® e Windows NT, um programa stub do MS-DOS é colocado aqui e é executado em vez do aplicativo real. Os programas normalmente não produzem mais do que uma linha de texto, como: "Este programa requer o Microsoft Windows v3.1 ou superior". Claro, quem cria o aplicativo é capaz de colocar qualquer stub que eles gostam aqui, o que significa que você pode ver muitas vezes coisas como: "Você não pode executar um aplicativo do Windows NT no OS / 2, simplesmente não é possível."

Ao criar um aplicativo para o Windows versão 3.1, o vinculador vincula um programa stub padrão chamado WINSTUB.EXE ao executável. Você pode substituir o comportamento de vinculador padrão substituindo seu próprio programa baseado em MS-DOS válido no lugar de WINSTUB e indicando isso para o vinculador com a instrução de definição de módulo **STUB**. Os aplicativos desenvolvidos para o Windows NT podem fazer o mesmo usando a opção - **STUB: linker** ao vincular o arquivo executável.

## Cabeçalho e Assinatura do Arquivo PE

O cabeçalho do arquivo PE está localizado indexando o campo **e\_lfanew** do cabeçalho do MS-DOS. O campo **e\_lfanew** simplesmente fornece o deslocamento no arquivo, portanto, adicione o endereço base mapeado na memória do arquivo para determinar o endereço real mapeado na memória. Por exemplo, a macro a seguir está incluída no arquivo de origem PEFILE.H:

## PEFILE.H

```
#define NTSIGNATURE (a) ((LPVOID) ((BYTE *) a + \
                           ((PIMAGE_DOS_HEADER) a) -> e_lfanew))
```

Ao manipular as informações do arquivo PE, descobri que havia vários locais no arquivo que eu precisava consultar com frequência. Como esses locais são apenas deslocamentos no arquivo, é mais fácil implementar esses locais como macros, pois eles fornecem um desempenho muito melhor do que as funções.

Observe que, em vez de recuperar o deslocamento do cabeçalho do arquivo PE, essa macro recupera o local da assinatura do arquivo PE. A partir dos executáveis do Windows e do OS / 2, os arquivos .EXE receberam assinaturas de arquivos para especificar o sistema operacional de destino pretendido. Para o formato de arquivo PE no Windows NT, essa assinatura ocorre imediatamente antes da estrutura do cabeçalho do arquivo PE. Nas versões do Windows e OS / 2, a assinatura é a primeira palavra do cabeçalho do arquivo. Além disso, para o formato de arquivo PE, o Windows NT usa um DWORD para a assinatura.

A macro apresentada acima retorna o deslocamento de onde a assinatura do arquivo aparece, independentemente de qual tipo de arquivo executável é. Portanto, dependendo se é uma assinatura de arquivo do Windows NT ou não, o cabeçalho do arquivo existe após a assinatura DWORD ou na assinatura WORD. Para resolver essa confusão, escrevi a função **ImageFileType** (a seguir), que retorna o tipo de arquivo de imagem:

### PEFILE.C

```
DWORD WINAPI ImageFileType (
    LPVOID lpFile)
{
    /* A assinatura do arquivo DOS vem em primeiro lugar. */
    if (* (USHORT *) lpFile == IMAGE_DOS_SIGNATURE)
    {
        /* Determinar a localização do cabeçalho do arquivo PE de
         * Cabeçalho DOS. */
        if (LOWORD (* (DWORD *) NTSIGNATURE (lpFile)) ==
            IMAGE_OS2_SIGNATURE ||
            LOWORD (* (DWORD *) NTSIGNATURE (lpFile)) ==
            IMAGE_OS2_SIGNATURE_LE)
            retornar (DWORD) LOWORD (* (DWORD *) NTSIGNATURE (lpFile));

        else if (* (DWORD *) NTSIGNATURE (lpFile) ==
            IMAGE_NT_SIGNATURE)
            retornar IMAGE_NT_SIGNATURE;

        outro
            retornar IMAGE_DOS_SIGNATURE;
    }

    outro
        /* Tipo de ficheiro desconhecido */
        return 0;
}
```

O código listado acima mostra rapidamente a utilidade da macro **NTSIGNATURE**. A macro facilita comparar os diferentes tipos de arquivos e retornar o apropriado para um determinado tipo de arquivo. Os quatro tipos de arquivo diferentes definidos no WINNT.H são:

### WINNT.H

```
#define IMAGE_DOS_SIGNATURE 0x5A4D // MZ
#define IMAGE_OS2_SIGNATURE 0x454E // NE
#define IMAGE_OS2_SIGNATURE_LE 0x454C // LE
#define IMAGE_NT_SIGNATURE 0x00004550 // PE00
```

A princípio, parece curioso que os tipos de arquivos executáveis do Windows não apareçam nesta lista. Mas, depois de uma pequena investigação, fica claro o motivo: não há diferença entre os executáveis do Windows e os executáveis do OS / 2, além da especificação de versão do sistema operacional. Ambos os sistemas operacionais compartilham a mesma estrutura de arquivos executáveis.

Voltando nossa atenção para o formato de arquivo PE do Windows NT, descobrimos que, uma vez que temos o local da assinatura do arquivo, o arquivo PE segue quatro bytes depois. A próxima macro identifica o cabeçalho do arquivo PE:

### PEFILE.C

```
#define PEFHROFFSET (a) ((LPVOID) ((BYTE *) a + \
    ((PIMAGE_DOS_HEADER) a) -> e_lfanew + SIZE_OF_NT_SIGNATURE))
```

A única diferença entre esta e a macro anterior é que esta adiciona na constante SIZE\_OF\_NT\_SIGNATURE. É triste dizer que essa constante não está definida no WINNT.H, mas é uma que eu defini no PEFILE.H como o tamanho de uma DWORD.

Agora que sabemos a localização do cabeçalho do arquivo PE, podemos examinar os dados no cabeçalho simplesmente atribuindo esse local a uma estrutura, como no exemplo a seguir:

```
PIMAGE_FILE_HEADER pfh;
pfh = (PIMAGE_FILE_HEADER) PEFHROFFSET (lpFile);
```

Neste exemplo, *lpFile* representa um ponteiro para a base do arquivo executável mapeado pela memória, e aí está a conveniência dos arquivos mapeados na memória. Nenhum arquivo I / O precisa ser executado; simplesmente desreferencia o ponteiro *pfh* para acessar informações no arquivo. A estrutura do cabeçalho do arquivo PE é definida como:

## WINNT.H

```
typedef struct _IMAGE_FILE_HEADER {
    Máquina USHORT;
    USHORT NumberOfSections;
    ULONG TimeStamp;
    ULONG PointerToSymbolTable;
    ULONG NumberOfSymbols;
    USHORT SizeOfOptionalHeader;
    Características do USHORT;
} IMAGE_FILE_HEADER, * PIMAGE_FILE_HEADER;

#define IMAGE_SIZEOF_FILE_HEADER 20
```

Observe que o tamanho da estrutura do cabeçalho do arquivo é convenientemente definido no arquivo de inclusão. Isto torna mais fácil para obter o tamanho da estrutura, mas eu achei mais fácil usar o **sizeof** função na própria estrutura, porque ele não exige me a lembrar o nome do **IMAGE\_SIZEOF\_FILE\_HEADER** constante, além do **IMAGE\_FILE\_HEADER** próprio nome estrutura. Por outro lado, lembrar o nome de todas as estruturas mostrou-se bastante desafiador, especialmente porque nenhuma dessas estruturas é documentada em qualquer lugar, exceto no arquivo de inclusão WINNT.H.

As informações no arquivo PE são basicamente informações de alto nível usadas pelo sistema ou aplicativos para determinar como tratar o arquivo. O primeiro campo é usado para indicar para que tipo de máquina o executável foi construído, como o DEC® Alpha, o MIPS R4000, o Intel® x86 ou algum outro processador. O sistema usa essas informações para determinar rapidamente como tratar o arquivo antes de prosseguir para o restante dos dados do arquivo.

As **características** campo identifica características específicas sobre o arquivo. Por exemplo, considere como arquivos de depuração separados são gerenciados para um executável. É possível remover informações de depuração de um arquivo PE e armazená-las em um arquivo de depuração (.DBG) para uso pelos depuradores. Para fazer isso, um depurador precisa saber se deseja localizar as informações de depuração em um arquivo separado ou não e se as informações foram removidas do arquivo ou não. Um depurador pode descobrir fazendo um drill down no arquivo executável procurando por informações de depuração. Para salvar o depurador de ter que pesquisar o arquivo, uma característica de arquivo que indica que o arquivo foi retirado (**IMAGE\_DEBUG\_STRIPPED**) foi inventado.

WINNT.H define vários outros sinalizadores que indicam informações de cabeçalho de arquivo muito da maneira que o exemplo descrito acima faz. Deixarei como um exercício para o leitor procurar as bandeiras para ver se alguma delas é interessante ou não. Eles estão localizados no WINNT.H imediatamente após a estrutura **IMAGE\_FILE\_HEADER** descrita acima.

Uma outra entrada útil na estrutura do cabeçalho do arquivo PE é o campo **NumberOfSections**. Acontece que você precisa saber quantas seções - mais especificamente, quantos cabeçalhos de seção e corpos de seção - estão no arquivo para extrair as informações facilmente. Cada cabeçalho de seção e corpo de seção é exibido seqüencialmente no arquivo, portanto, o número de seções é necessário para determinar onde os cabeçalhos e corpos de seção terminam. A função a seguir extrai o número de seções do cabeçalho do arquivo PE:

## PEFILE.C

```
int WINAPI NumOfSections (
    LPVOID lpFile)
{
    /* Número de seções é indicado no cabeçalho do arquivo. */
    return (int) ((PIMAGE_FILE_HEADER)
        PEFHDOFFSET (lpFile)) -> NumberOfSections;
}
```

Como você pode ver, o **PEFHDOFFSET** e as outras macros são muito úteis para se ter por perto.

## Cabeçalho Opcional de PE

Os 224 bytes seguintes no arquivo executável formam o cabeçalho opcional do PE. Embora seu nome seja "cabeçalho opcional", tenha certeza de que essa não é uma entrada opcional nos arquivos executáveis do PE. Um ponteiro para o cabeçalho opcional é obtido com a macro **OPTHDOFFSET** :

## PEFILE.H

```
#define OPTHDOFFSET (a) ((LPVOID) ((BYTE *) a + \
    ((PIMAGE_DOS_HEADER) a) -> e_lfanew + SIZE_OF_NT_SIGNATURE + \
    sizeof (IMAGE_FILE_HEADER)))
```

O cabeçalho opcional contém a maioria das informações significativas sobre a imagem executável, como tamanho inicial da pilha, local do ponto de entrada do programa, endereço base preferencial, versão do sistema operacional, informações de alinhamento da seção e assim por diante. A estrutura **IMAGE\_OPTIONAL\_HEADER** representa o cabeçalho opcional da seguinte maneira:

## WINNT.H

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // campos padrão.
    //
    USHORT Magic;
    UCHAR MajorLinkerVersion;
    UCHAR MinorLinkerVersion;
    ULONG SizeOfCode;
    ULONG SizeOfInitializedData;
    ULONG SizeOfUninitializedData;
    ULONG AddressOfEntryPoint;
    ULONG BaseOfCode;
    ULONG BaseOfData;
    //
    // campos adicionais do NT.
    //
    ULONG ImageBase;
    ALONG Section Alignment;
    ULONG FileAlignment;
    USHORT MajorOperatingSystemVersion;
    USHORT MinorOperatingSystemVersion;
    USHORT MajorImageVersion;
    USHORT MinorImageVersion;
    USHORT MajorSubsystemVersion;
    USHORT MinorSubsystemVersion;
    ULONG Reservado1;
    ULONG SizeOfImage;
    ULONG SizeOfHeaders;
    ULONG CheckSum;
    Subsistema USHORT;
    DllCharacteristics do USHORT;
    ULONG SizeOfStackReserve;
    ULONG SizeOfStackCommit;
    ULONG SizeOfHeapReserve;
    ULONG SizeOfHeapCommit;
    ULONG LoaderFlags;
    ULONG NumberOfRvaAndSizes;
    IMAGE_DIRECTORY DataDirectory [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, * PIMAGE_OPTIONAL_HEADER;
```

Como você pode ver, a lista de campos nessa estrutura é bastante longa. Em vez de aborrecê-lo com descrições de todos esses campos, eu simplesmente discutirei os úteis - isto é, úteis no contexto de explorar o formato de arquivo PE.

## Campos padrão

Primeiro, observe que a estrutura é dividida em "Campos padrão" e "Campos adicionais do NT". Os campos padrão são aqueles comuns ao COFF (Common Object File Format), que a maioria dos arquivos executáveis do UNIX usa. Embora os campos padrão mantenham os nomes definidos no COFF, o Windows NT, na verdade, usa alguns deles para propósitos diferentes que seriam melhor descritos com outros nomes.

- *Magic* . Não consegui rastrear para que esse campo é usado. Para o aplicativo de exemplo EXEVIEW.EXE, o valor é 0x010B ou 267.
- *MajorLinkerVersion* , *MinorLinkerVersion* . Indica a versão do vinculador que vinculou essa imagem. O préliminar Windows NT Software Development Kit (SDK), fornecido com compilar 438 do Windows NT, inclui versão de vinculador 2.39 (2.27 hex).
- *SizeOfCode* . Tamanho do código executável.
- *SizeOfInitializedData* . Tamanho dos dados inicializados.
- *SizeOfUninitializedData* . Tamanho dos dados não inicializados.
- *AddressOfEntryPoint* . Dos campos padrão, o campo *AddressOfEntryPoint* é o mais interessante para o formato de arquivo PE. Este campo indica a localização do ponto de entrada para o aplicativo e, talvez mais importante para os hackers do sistema, a localização do final da tabela de endereços de importação (IAT). A seguinte função demonstra como recuperar o ponto de entrada de uma imagem executável do Windows NT do cabeçalho opcional.

## PEFILE.C

```

    LPVOID WINAPI GetModuleEntryPoint (
        LPVOID lpFile)
    {
        PIMAGE_OPTIONAL_HEADER poh;

        poh = (PIMAGE_OPTIONAL_HEADER) OPTHDROFFSET (lpFile);

        if (poh != NULL)
            return (LPVOID) poh->AddressOfEntryPoint;
        outro
        return NULL;
    }
}

```

- *BaseOfCode* . Deslocamento relativo do código (seção ".text") na imagem carregada.
- *BaseOfData* . Deslocamento relativo de dados não inicializados (seção ".bss") na imagem carregada.

## Campos Adicionais do Windows NT

Os campos adicionais adicionados ao formato de arquivo PE do Windows NT fornecem suporte ao carregador para grande parte do comportamento do processo específico do Windows NT. A seguir, um resumo desses campos.

- *ImageBase* . Endereço base preferencial no espaço de endereço de um processo para mapear a imagem executável para. O vinculador que vem com o SDK do Microsoft Win32 para os padrões do Windows NT para 0x00400000, mas você pode substituir o padrão com o **-base: vinculador** switch.
- *SectionAlignment* . Cada seção é carregada no espaço de endereço de um processo sequencialmente, começando pelo *ImageBase* . O *SectionAlignment* determina a quantidade mínima de espaço que uma seção pode ocupar quando carregada - ou seja, as seções são alinhadas nos limites de *SectionAlignment* .

O alinhamento da seção não pode ser menor que o tamanho da página (atualmente 4096 bytes na plataforma x 86) e deve ser um múltiplo do tamanho da página, conforme determinado pelo comportamento do gerenciador de memória virtual do Windows NT. 4096 bytes é o x vinculador padrão 86, mas isso pode ser definido utilizando o **-ALIGN: vinculador** switch.

- *FileAlignment* . Granularidade mínima de partes da informação dentro do arquivo de imagem antes do carregamento. Por exemplo, o vinculador zera o corpo de uma seção (dados brutos para uma seção) até o limite *FileAlignment* mais próximo no arquivo. Versão 2.39 do vinculador mencionado anteriormente alinha arquivos de imagem em uma granularidade de 0x200 bytes. Este valor é restrito para ser uma potência de 2 entre 512 e 65.535.
- \* *MajorOperatingSystemVersion* . Indica a versão principal do sistema operacional Windows NT, atualmente definida como 1 para o Windows NT versão 1.0.
- *MinorOperatingSystemVersion* . Indica a versão secundária do sistema operacional Windows NT, atualmente definida como 0 para o Windows NT versão 1.0
- *MajorImageVersion* . Usado para indicar o número da versão principal do aplicativo; no Microsoft Excel versão 4.0, seria 4.
- *MinorImageVersion* . Usado para indicar o número da versão secundária do aplicativo; no Microsoft Excel versão 4.0, seria 0.
- *MajorSubsystemVersion* . Indica o número da versão principal do subsistema Windows NT Win32, atualmente definido como 3 para o Windows NT versão 3.10.
- *MinorSubsystemVersion* . Indica o número de versão secundária do subsistema Windows NT Win32, atualmente definido como 10 para o Windows NT versão 3.10.
- *Reservado1* . Finalidade desconhecida, atualmente não usada pelo sistema e definida como zero pelo vinculador.
- \* *SizeOfImage* . Indica a quantidade de espaço de endereço a ser reservado no espaço de endereço da imagem executável carregada. Este número é influenciado grandemente pelo *SectionAlignment* . Por exemplo, considere um sistema com um tamanho de página fixo de 4096 bytes. Se você tiver um executável com 11 seções, cada uma com menos de 4096 bytes, alinhadas em um limite de 65.536 bytes, o campo *SizeOfImage* será definido como  $11 * 65.536 = 720.896$  (176 páginas). O mesmo arquivo vinculado ao alinhamento de 4096 bytes resultaria em  $11 * 4096 = 45.056$  (11 páginas) para o *SizeOfImage* campo. Este é um exemplo simples em que cada seção requer menos de uma página de memória. Na realidade, o vinculador determina a exata *SizeOfImage* , calculando cada seção individualmente. Ele primeiro determina quantos bytes a seção requer, então arredonda para o limite de página mais próximo e, por fim, arredonda a contagem de páginas para o limite de *SectionAlignment* mais próximo . O total é então a soma do requisito individual de cada seção.
- *SizeOfHeaders* . Esse campo indica quanto espaço no arquivo é usado para representar todos os cabeçalhos de arquivo, incluindo o cabeçalho do MS-DOS, o cabeçalho do arquivo PE, o cabeçalho opcional PE e os cabeçalhos da seção PE. Os corpos da seção começam neste local no arquivo.
- *CheckSum* . Um valor de soma de verificação é usado para validar o arquivo executável no momento do carregamento. O valor é definido e verificado pelo vinculador. O algoritmo usado para criar esses valores de soma de verificação é uma informação proprietária e não será publicado.
- *Subsistema* . Campo usado para identificar o subsistema de destino para esse executável. Cada um dos possíveis valores de subsistema são listados no arquivo WINNT.H imediatamente após a estrutura **IMAGE\_OPTIONAL\_HEADER** .
- *DllCharacteristics* . Sinalizadores usados para indicar se uma imagem DLL inclui pontos de entrada para inicialização e finalização do processo e do encadeamento.

- *SizeOfStackReserve* , *SizeOfStackCommit* , *SizeOfHeapReserve* , *SizeOfHeapCommit* . Esses campos controlam a quantidade de espaço de endereço a ser reservada e confirmada para a pilha e o heap padrão. Tanto a pilha quanto o heap têm valores padrão de 1 página confirmada e 16 páginas reservadas. Estes valores são definidos com o ligante liga **-STACKSIZE:** e **-HEAPSIZE:** .
- *LoaderFlags* . Informa ao carregador se ele deve quebrar na carga, depurar na carga ou o padrão, que é permitir que as coisas funcionem normalmente.
- *NumberOfRvaAndSizes* . Este campo identifica o comprimento da matriz *DataDirectory* a seguir. É importante observar que esse campo é usado para identificar o tamanho da matriz, não o número de entradas válidas na matriz.
- *DataDirectory* . O diretório de dados indica onde encontrar outros componentes importantes de informações executáveis no arquivo. Na verdade, não é nada mais que uma matriz de estruturas **IMAGE\_DATA\_DIRECTORY** localizadas no final da estrutura de cabeçalho opcional. O atual formato de arquivo PE define 16 possíveis diretórios de dados, 11 dos quais estão sendo usados agora.

## Diretórios de dados

Conforme definido no WINNT.H, os diretórios de dados são:

### WINNT.H

```
// Entradas do diretório

// Diretório de exportação
#define IMAGE_DIRECTORY_ENTRY_EXPORT 0
// Diretório de Importação
#define IMAGE_DIRECTORY_ENTRY_IMPORT 1
// Diretório de Recursos
#define IMAGE_DIRECTORY_ENTRY_RESOURCE 2
// Diretório de exceções
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION 3
// Diretório de Segurança
#define IMAGE_DIRECTORY_ENTRY_SECURITY 4
// Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_BASERELOC 5
// Debug Directory
#define IMAGE_DIRECTORY_ENTRY_DEBUG 6
// Descrição String
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT 7
// Valor da Máquina (MIPS GP)
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR 8
// Diretório TLS
#define IMAGE_DIRECTORY_ENTRY_TLS 9
// Carregar Diretório de Configuração
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG 10
```

Cada diretório de dados é basicamente uma estrutura definida como **IMAGE\_DATA\_DIRECTORY** . E, embora as entradas do diretório de dados sejam as mesmas, cada tipo de diretório específico é totalmente exclusivo. A definição de cada diretório de dados definido é descrita em "Seções Predefinidas", mais adiante neste artigo.

### WINNT.H

```
typedef struct _IMAGE_DATA_DIRECTORY {
    ULONG VirtualAddress;
    Tamanho ULONG;
} IMAGE_DATA_DIRECTORY, * PIMAGE_DATA_DIRECTORY;
```

Cada entrada do diretório de dados especifica o tamanho e o endereço virtual relativo do diretório. Para localizar um diretório específico, você determina o endereço relativo da matriz do diretório de dados no cabeçalho opcional. Em seguida, use o endereço virtual para determinar em qual seção o diretório está. Depois de determinar qual seção contém o diretório, o cabeçalho da seção para essa seção é usado para localizar o local exato do deslocamento do arquivo do diretório de dados.

Portanto, para obter um diretório de dados, primeiro você precisa saber sobre as seções, que são descritas a seguir. Um exemplo de como localizar diretórios de dados segue imediatamente essa discussão.

## Seções de arquivos PE

A especificação do arquivo PE consiste nos cabeçalhos definidos até o momento e um objeto genérico chamado *seção* . As seções contêm o conteúdo do arquivo, incluindo código, dados, recursos e outras informações executáveis. Cada seção tem um cabeçalho e um corpo (os dados brutos). Os cabeçalhos de seção são descritos abaixo, mas os corpos de seção não possuem uma estrutura de arquivo rígida. Eles podem ser organizados de qualquer maneira que um vinculador deseja organizar-los, desde que o cabeçalho esteja cheio de informações suficientes para decifrar os dados.

### Cabeçalhos de Seção

Os cabeçalhos de seção estão localizados sequencialmente logo após o cabeçalho opcional no formato de arquivo PE. Cada cabeçalho de seção é de 40 bytes sem preenchimento entre eles. Os cabeçalhos de seção são definidos na seguinte estrutura:

## WINNT.H

```
#define IMAGE_SIZEOF_SHORT_NAME 8

typedef struct _IMAGE_SECTION_HEADER {
    Nome UCHAR [IMAGE_SIZEOF_SHORT_NAME];
    União {
        ULONG PhysicalAddress;
        ULONG VirtualSize;
    } Misc;
    ULONG VirtualAddress;
    ULONG SizeOfRawData;
    ULONG PointerToRawData;
    ULONG PointerToRelocations;
    ULONG PointerToLinenumbers;
    USHORT NumberOfRelocations;
    USHORT NumberOfLinenumbers;
    ULONG Características;
} IMAGE_SECTION_HEADER, * PIMAGE_SECTION_HEADER;
```

Como você obtém informações de cabeçalho de seção para uma seção específica? Como os cabeçalhos de seção são organizados seqüencialmente em nenhuma ordem específica, os cabeçalhos de seção devem estar localizados por nome. A função a seguir mostra como recuperar um cabeçalho de seção de um arquivo de imagem PE, de acordo com o nome da seção:

## PEFILE.C

```
BOOL WINAPI GetSectionHdrByName (
    LPVOID lpFile,
    IMAGE_SECTION_HEADER * sh,
    char * szSection)
{
    PIMAGE_SECTION_HEADER psh;
    int nSections = NumOfSections (lpFile);
    int i;

    if ((psh = (PIMAGE_SECTION_HEADER) SECHDROFFSET (lpFile)) != NULO)
    {
        /* encontrar a seção pelo nome */
        para (i = 0; i <nSections; i++)
        {
            if (!strcmp (psh-> nome, szSection))
            {
                /* copiar dados para o cabeçalho */
                CopyMemory ((LPVOID) sh,
                            (LPVOID) psh,
                            sizeof (IMAGE_SECTION_HEADER));
                retorno verdadeiro;
            }
            outro
            psh++;
        }
    }
    retorna falso;
}
```

A função simplesmente localiza o primeiro cabeçalho da seção através da macro **SECHDROFFSET**. Em seguida, a função percorre cada seção, comparando o nome de cada seção com o nome da seção que está procurando, até encontrar o nome certo. Quando a seção é encontrada, a função copia os dados do arquivo mapeado na memória para a estrutura passada para a função. Os campos da estrutura **IMAGE\_SECTION\_HEADER** podem ser acessados diretamente da estrutura.

## Campos de cabeçalho de seção

- *Nome* . Cada cabeçalho de seção tem um campo de *nome* com até oito caracteres, para o qual o primeiro caractere deve ser um período.
- *PhysicalAddress* ou *VirtualSize* . O segundo campo é um campo de união que não é usado atualmente.

- **VirtualAddress** . Este campo identifica o endereço virtual no espaço de endereço do processo para o qual carregar a seção. O endereço real é criado tomando o valor desse campo e adicionando-o ao endereço virtual *ImageBase* na estrutura de cabeçalho opcional. Tenha em mente, porém, que se este arquivo de imagem representa uma DLL, não há garantia de que a DLL será carregada para a localização do *ImageBase* solicitada. Portanto, assim que o arquivo é carregado em um processo, o valor real do *ImageBase* deve ser verificado programaticamente usando **GetModuleHandle** .
- **SizeOfRawData** . Este campo indica o tamanho *relativo* ao *FileAlignment* do corpo da seção. O tamanho real do corpo da seção será menor ou igual a um múltiplo de *FileAlignment* no arquivo. Depois que a imagem é carregada no espaço de endereço de um processo, o tamanho do corpo da seção torna-se menor ou igual a um múltiplo de *SectionAlignment* .
- **PointerToRawData** . Esse é um deslocamento para o local do corpo da seção no arquivo.
- **PointerToRelocations** , **PointerToLinenumbers** , **NumberOfRelocations** , **NumberOfLinenumbers** . Nenhum desses campos é usado no formato de arquivo PE.
- **Características** . Define as características da seção. Esses valores são encontrados no WINNT.H e na especificação Portable Executable Format localizada neste CD.

| Valor      | Definição                              |
|------------|--|
| 0x00000020 | Seção de código                        |
| 0x00000040 | Seção de dados inicializados           |
| 0x00000080 | Seção de dados não inicializados       |
| 0x04000000 | Seção não pode ser armazenada em cache |
| 0x08000000 | Seção não é paginável                  |
| 0x10000000 | Seção é compartilhada                  |
| 0x20000000 | Seção Executável                       |
| 0x40000000 | Seção legível                          |
| 0x80000000 | Seção gravável                         |

## Localizando Diretórios de Dados

Os diretórios de dados existem dentro do corpo de sua seção de dados correspondente. Normalmente, os diretórios de dados são a primeira estrutura dentro do corpo da seção, mas não necessariamente. Por esse motivo, você precisa recuperar informações do cabeçalho da seção e do cabeçalho opcional para localizar um diretório de dados específico.

Para tornar esse processo mais fácil, a seguinte função foi escrita para localizar o diretório de dados para qualquer um dos diretórios definidos no WINNT.H:

### PEFILE.C

```
LPVOID WINAPI ImageDirectoryOffset (
    LPVOID lpFile,
    DWORD dwIMAGE_DIRECTORY)
{
    PIMAGE_OPTIONAL_HEADER poh;
    PIMAGE_SECTION_HEADER psh;
    int nSections = NumOfSections (lpFile);
    int i = 0;
    LPVOID VAIImageDir;

    / * Deve ser 0 a (NumberOfRvaAndSizes-1). * /
    if (dwIMAGE_DIRECTORY >= poh-> NumberOfRvaAndSizes)
        return NULL;

    / * Recupere deslocamentos para cabeçalhos opcionais e de seção. * /
    poh = (PIMAGE_OPTIONAL_HEADER) OPTHDEROFFSET (lpFile);
    psh = (PIMAGE_SECTION_HEADER) SECHDROFFSET (lpFile);

    / * Localize o endereço virtual relativo do diretório de imagens. * /
    VAIImageDir = (LPVOID) poh-> DataDirectory
        [dwIMAGE_DIRECTORY] .VirtualAddress;

    / * Localize a seção que contém o diretório da imagem. * /
    while (i ++ <nSections)
    {
        if (psh-> VirtualAddress <= (DWORD) VAIImageDir &&
            psh-> VirtualAddress +
            psh-> SizeOfRawData > (DWORD) VAIImageDir)
```

```

    pausa;
    psh++;
}

if (i > nseções)
    return NULL;

/* Retorna o deslocamento do diretório de importação de imagens. */
return (LPVOID) (((int) lpFile +
    (int) VAIImageDir. psh-> VirtualAddress) +
    (int) psh-> PointerToRawData);
}

```

A função começa validando o número de entrada do diretório de dados solicitado. Em seguida, ele recupera ponteiros para o cabeçalho opcional e o cabeçalho da primeira seção. A partir do cabeçalho opcional, a função determina o endereço virtual do diretório de dados e usa esse valor para determinar em qual corpo de seção o diretório de dados está localizado. Depois que o corpo de seção apropriado foi identificado, o local específico do diretório de dados é encontrado traduzindo o endereço virtual relativo do diretório de dados para um endereço específico no arquivo.

## Seções Predefinidas

Um aplicativo para o Windows NT geralmente possui nove seções predefinidas chamadas .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata e .debug. Alguns aplicativos não precisam de todas essas seções, enquanto outros podem definir ainda mais seções para atender às suas necessidades específicas. Esse comportamento é semelhante ao código e segmentos de dados no MS-DOS e no Windows versão 3.1. Na verdade, a maneira como um aplicativo define uma seção exclusiva é usar as diretivas de compilador padrão para nomear segmentos de código e dados ou usar a opção de compilador de segmento de nome **-NT** - exatamente da mesma maneira na qual os aplicativos definem segmentos de código e dados exclusivos em Windows versão 3.1.

A seguir, é apresentada uma discussão sobre algumas das seções mais interessantes comuns aos arquivos típicos do Windows NT PE.

### Seção de código executável, .text

Uma diferença entre o Windows versão 3.1 eo Windows NT é que o comportamento padrão combina todos os segmentos de código (como eles são mencionados no Windows versão 3.1) em uma única seção chamada ".text" no Windows NT. Como o Windows NT usa um sistema de gerenciamento de memória virtual baseado em página, não há vantagem em separar o código em segmentos de código distintos. Consequentemente, ter uma seção de código grande é mais fácil de gerenciar tanto para o sistema operacional quanto para o desenvolvedor do aplicativo.

A seção .text também contém o ponto de entrada mencionado anteriormente. O IAT também vive na seção .text imediatamente antes do ponto de entrada do módulo. (A presença do IAT na seção .text faz sentido porque a tabela é realmente uma série de instruções de salto, para as quais o local específico para saltar é o endereço fixo.) Quando imagens executáveis do Windows NT são carregadas no espaço de endereço de um processo , o IAT é corrigido com a localização do endereço físico de cada função importada. Para encontrar o IAT na seção .text, o carregador simplesmente localiza o ponto de entrada do módulo e se baseia no fato de que o IAT ocorre imediatamente antes do ponto de entrada. E como cada entrada é do mesmo tamanho, é fácil voltar na mesa para encontrar o começo.

### Seções de dados, .bss, .rdata, .data

A seção .bss representa dados não inicializados para o aplicativo, incluindo todas as variáveis declaradas como estáticas em uma função ou módulo de origem.

A seção .rdata representa dados somente leitura, como seqüências de caracteres literais, constantes e informações do diretório de depuração.

Todas as outras variáveis (exceto variáveis automáticas, que aparecem na pilha) são armazenadas na seção .data. Basicamente, estas são variáveis globais de aplicativo ou módulo.

### Seção de Recursos, .rsrc

A seção .rsrc contém informações sobre recursos para um módulo. Ele começa com uma estrutura de diretórios de recursos, como a maioria das outras seções, mas os dados dessa seção são estruturados em uma árvore de recursos. O **IMAGE\_RESOURCE\_DIRECTORY** , mostrado abaixo, forma a raiz e os nós da árvore.

### WINNT.H

```

typedef struct _IMAGE_RESOURCE_DIRECTORY {
    ULONG Características;
    ULONG TimeStamp;
    USHORT MajorVersion;
    USHORT MinorVersion;
    USHORT NumberOfNamedEntries;

```

```
USHORT NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, * PIMAGE_RESOURCE_DIRECTORY;
```

Olhando para a estrutura de diretórios, você não encontrará nenhum ponteiro para os próximos nós. Em vez disso, existem dois campos, *NumberOfNamedEntries* e *NumberOfIdEntries*, usados para indicar quantas entradas estão anexadas ao diretório. Por anexo , quero dizer que as entradas do diretório seguem imediatamente após o diretório nos dados da seção. As entradas nomeadas aparecem primeiro em ordem alfabética crescente, seguidas pelas entradas de ID em ordem numérica crescente.

Uma entrada de diretório consiste em dois campos, conforme descrito na seguinte estrutura

#### **IMAGE\_RESOURCE\_DIRECTORY\_ENTRY :**

#### **WINNT.H**

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    Nome ULONG;
    ULONG OffsetToData;
} IMAGE_RESOURCE_DIRECTORY_ENTRY, * PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Os dois campos são usados para coisas diferentes, dependendo do nível da árvore. O campo *Nome* é usado para identificar um tipo de recurso, um nome de recurso ou um ID de idioma do recurso. O campo *OffsetToData* é sempre usado para apontar para um irmão na árvore, um nó de diretório ou um nó de folha.

Nós de folha são o nó mais baixo na árvore de recursos. Eles definem o tamanho e a localização dos dados reais do recurso. Cada nó folha é representado usando a seguinte estrutura **IMAGE\_RESOURCE\_DATA\_ENTRY** :

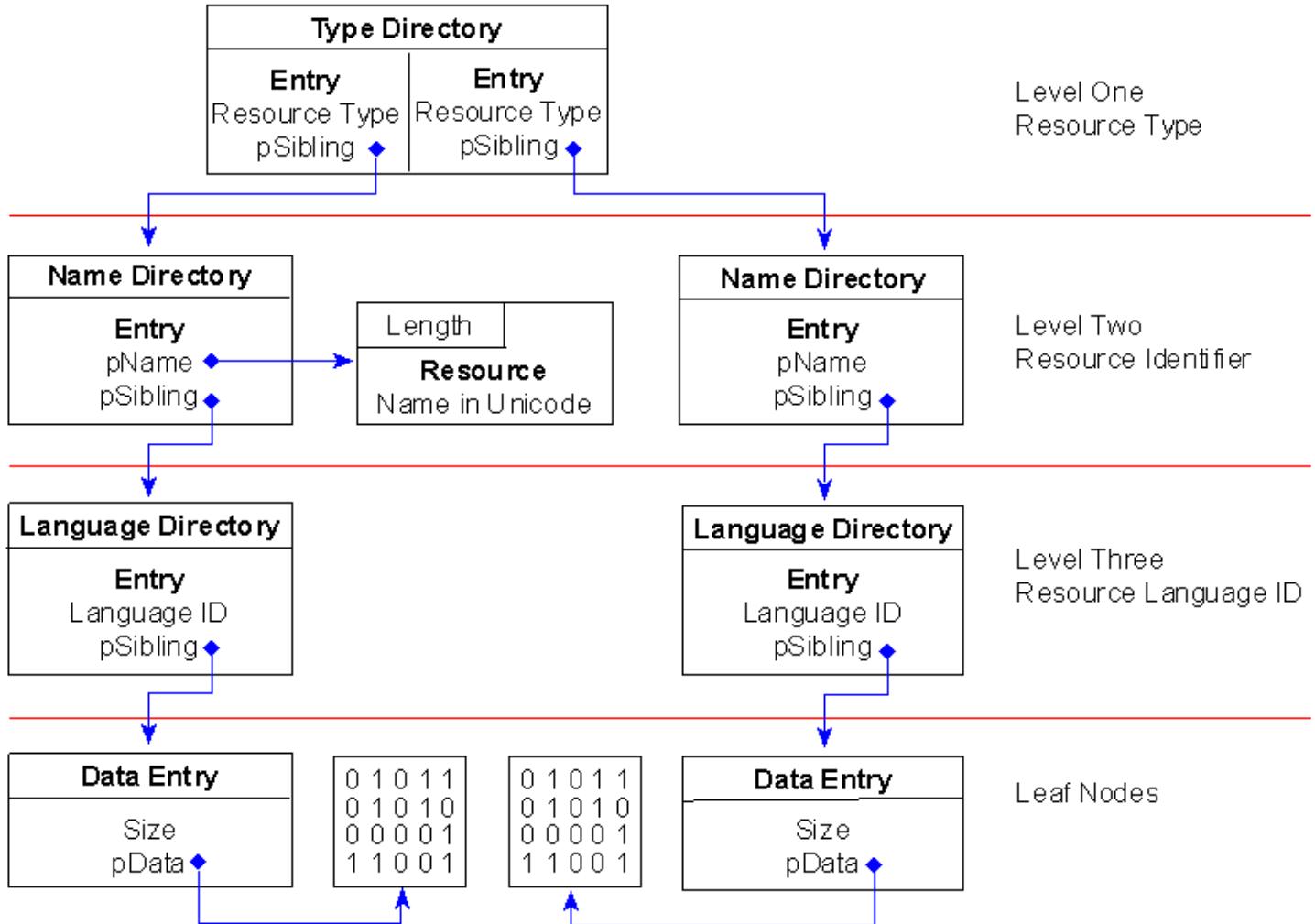
#### **WINNT.H**

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
    ULONG OffsetToData;
    Tamanho ULONG;
    ULONG CodePage;
    ULONG Reservado;
} IMAGE_RESOURCE_DATA_ENTRY, * PIMAGE_RESOURCE_DATA_ENTRY;
```

Os dois campos *OffsetToData* e *Size* indicam a localização e o tamanho dos dados reais do recurso. Como essa informação é usada principalmente por funções depois que o aplicativo foi carregado, faz mais sentido tornar o campo *OffsetToData* um endereço virtual relativo. Este é precisamente o caso. Curiosamente, todos os outros deslocamentos, como ponteiros de entradas de diretório para outros diretórios, são deslocamentos em relação ao local do nó raiz.

Para tornar tudo isso um pouco mais claro, considere a Figura 2.

## Resource Tree



**Figura 2. Uma estrutura de árvore de recurso simples**

A Figura 2 descreve uma árvore de recursos muito simples contendo apenas dois objetos de recursos, um menu e uma tabela de strings. Além disso, o menu e a tabela de strings têm apenas um item cada. No entanto, você pode ver como a árvore de recursos se torna complicada - mesmo com poucos recursos.

Na raiz da árvore, o primeiro diretório tem uma entrada para cada tipo de recurso que o arquivo contém, não importando quantos de cada tipo existam. Na Figura 2, existem duas entradas identificadas pela raiz, uma para o menu e outra para a tabela de strings. Se houvesse um ou mais recursos de diálogo incluídos no arquivo, o nó raiz teria mais uma entrada e, consequentemente, outra ramificação para os recursos da caixa de diálogo.

Os tipos básicos de recursos são identificados no arquivo WINUSER.H e estão listados abaixo:

### WINUSER.H

```

/* *
 * Tipos de recursos predefinidos
 */
#define RT_CURSOR MAKEINTRESOURCE (1)
#define RT_BITMAP MAKEINTRESOURCE (2)
#define RT_ICON MAKEINTRESOURCE (3)
#define RT_MENU MAKEINTRESOURCE (4)
#define RT_DIALOG MAKEINTRESOURCE (5)
#define RT_STRING MAKEINTRESOURCE (6)
#define RT_FONTDIR MAKEINTRESOURCE (7)
#define RT_FONT MAKEINTRESOURCE (8)
#define RT_ACCELERATOR MAKEINTRESOURCE (9)
#define RT_RCDATA MAKEINTRESOURCE (10)
#define RT_MESSAGETABLE MAKEINTRESOURCE (11)

```

No nível superior da árvore, os valores de `MAKEINTRESOURCE` listados acima são colocados no campo *Nome* de cada entrada de tipo, identificando os diferentes recursos por tipo.

Cada uma das entradas no diretório raiz aponta para um nó irmão no segundo nível da árvore. Esses nós também são diretórios, cada um com suas próprias entradas. Nesse nível, os diretórios são usados para identificar o nome de cada

recurso dentro de um determinado tipo. Se você tivesse vários menus definidos em seu aplicativo, haveria uma entrada para cada um aqui no segundo nível da árvore.

Como você provavelmente já sabe, os recursos podem ser identificados por nome ou por inteiro. Eles são diferenciados nesse nível da árvore por meio do campo *Nome* na estrutura de diretórios. Se o bit mais significativo do campo *Nome* for definido, os outros 31 bits serão usados como um deslocamento para uma estrutura **IMAGE\_RESOURCE\_DIR\_STRING\_U**.

## WINNT.H

```
typedef struct _IMAGE_RESOURCE_DIR_STRING_U {
    Comprimento USHORT;
    WCHAR NameString [1];
} IMAGE_RESOURCE_DIR_STRING_U, * PIMAGE_RESOURCE_DIR_STRING_U;
```

Essa estrutura é simplesmente um campo *Comprimento* de 2 bytes seguido por caracteres UNICODE de *Comprimento*.

Por outro lado, se o bit mais significativo do campo *Nome* for claro, os 31 bits inferiores serão usados para representar o ID inteiro do recurso. A Figura 2 mostra o recurso de menu como um recurso nomeado e a tabela de string como um recurso de ID.

Se houvesse dois recursos de menu, um identificado por nome e um por recurso, ambos teriam entradas imediatamente após o diretório de recursos do menu. A entrada de recurso nomeada apareceria primeiro, seguida pelo recurso identificado por inteiro. Os campos de diretório *NumberOfNamedEntries* e *NumberOfIdEntries* conteriam o valor 1, indicando a presença de uma entrada.

Abaixo do nível dois, a árvore de recursos não expande mais. O nível um se ramifica em diretórios representando cada tipo de recurso, e o nível dois ramifica em diretórios representando cada recurso por identificador. O nível três mapeia uma correspondência um-para-um entre os recursos individualmente identificados e seus respectivos IDs de idioma. Para indicar o ID do idioma de um recurso, o *nome* campo da estrutura de entrada de diretório é usado para indicar o idioma primário e o ID de sub-idioma para o recurso. O Win32 SDK para Windows NT lista os recursos de valor padrão. Para o valor 0x0409, 0x09 representa o idioma principal como LANG\_ENGLISH e 0x04 é definido como SUBLANG\_ENGLISH\_CAN para o sub-idioma. Todo o conjunto de IDs de idioma é definido no arquivo WINNT.H, incluído como parte do Win32 SDK para Windows NT.

Como o nó ID de idioma é o último nó de diretório na árvore, o campo *OffsetToData* na estrutura de entrada é um deslocamento para um nó folha - a estrutura **IMAGE\_RESOURCE\_DATA\_ENTRY** mencionada anteriormente.

Referindo-se novamente à Figura 2, você pode ver um nó de entrada de dados para cada entrada de diretório de idioma. Este nó indica simplesmente o tamanho dos dados do recurso e o endereço virtual relativo em que os dados do recurso estão localizados.

Uma vantagem de ter tanta estrutura na seção de dados de recursos, .rsrc, é que você pode obter uma grande quantidade de informações da seção sem acessar os próprios recursos. Por exemplo, você pode descobrir quantos deles existem de cada tipo de recurso, quais recursos - se houver algum - usam um determinado ID de idioma, se um determinado recurso existe ou não e o tamanho de tipos individuais de recursos. Para demonstrar como usar essas informações, a função a seguir mostra como determinar os diferentes tipos de recursos que um arquivo inclui:

## PEFILE.C

```
int WINAPI GetListOfResourceTypes (
    LPVOID lpFile,
    HANDLE hHeap,
    char ** pszResTypes)
{
    PIMAGE_RESOURCE_DIRECTORY prdRoot;
    PIMAGE_RESOURCE_DIRECTORY_ENTRY prde;
    char * pMem;
    int nCnt, i;

    / * Obtém o diretório raiz da árvore de recursos. * /
    if ((prdRoot = PIMAGE_RESOURCE_DIRECTORY) ImageDirectoryOffset
        (lpFile, IMAGE_DIRECTORY_ENTRY_RESOURCE) == NULL)
        return 0;

    / * Aloca espaço suficiente da pilha para cobrir todos os tipos. * /
    nCnt = prdRoot-> NumberOfIdEntries * (MAXRESOURCENAME + 1);
    * pszResTypes = (char *) HeapAlloc (hHeap,
                                         HEAP_ZERO_MEMORY,
                                         nCnt);
    if ((pMem = * pszResTypes) == NULL)
        return 0;

    / * Definir ponteiro para entrada do primeiro tipo de recurso. * /
    prde = (PIMAGE_RESOURCE_DIRECTORY_ENTRY) pMem;
```

```

prde = (PIMAGE_RESOURCE_DIRECTORY_ENTRY) ((DWORD) prdRoot +
    sizeof (IMAGE_RESOURCE_DIRECTORY));

/* Faz um loop por todos os tipos de entrada de diretório de recursos. */
para (i = 0; i <prdRoot-> NumberOfIdEntries; i++)
{
    if (LoadString (hDll, nome-> nome, pMem, MAXRESOURCENAME))
        pMem += strlen (pMem) + 1;

    prde++;
}

return nCnt;
}

```

Esta função retorna uma lista de nomes de tipos de recursos na string identificada por *pszResTypes*. Observe que, no coração dessa função, **LoadString** é chamado usando o campo *Nome* de cada entrada de diretório do tipo de recurso como o ID da sequência. Se você procurar no PEFILE.RC, verá que defini uma série de cadeias de tipos de recursos cujos IDs são definidos da mesma forma que os especificadores de tipos nas entradas de diretório. Há também uma função em PEFILE.DLL que retorna o número total de objetos de recursos na seção .rsrc. Seria bastante fácil expandir essas funções ou escrever novas funções que extraíam outras informações desta seção.

## Exportar seção de dados, .edata

A seção .edata contém dados de exportação para um aplicativo ou DLL. Quando presente, esta seção contém um diretório de exportação para obter as informações de exportação.

### WINNT.H

```

typedef struct _IMAGE_EXPORT_DIRECTORY {
    ULONG Características;
    ULONG TimeStamp;
    USHORT MajorVersion;
    USHORT MinorVersion;
    Nome ULONG;
    ULONG Base;
    ULONG NumberOfFunctions;
    ULONG NumberOfNames;
    PULONG * AddressOfFunctions;
    PULONG * AddressOfNames;
    PUSHORT * AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, * PIMAGE_EXPORT_DIRECTORY;

```

O campo *Nome* no diretório de exportação identifica o nome do módulo executável. Os campos *NumberOfFunctions* e *NumberOfNames* indicam quantas funções e nomes de função estão sendo exportados do módulo.

O campo *AddressOfFunctions* é um deslocamento para uma lista de pontos de entrada de função exportados. O campo *AddressOfNames* é o endereço de um deslocamento para o início de uma lista separada por nulos de nomes de função exportados. *AddressOfNameOrdinals* é um deslocamento para uma lista de valores ordinais (cada 2 bytes de comprimento) para as mesmas funções exportadas.

Os três campos *AddressOf ...* são endereços virtuais relativos no espaço de endereço de um processo depois que o módulo é carregado. Depois que o módulo é carregado, o endereço virtual relativo deve ser adicionado ao endereço base do módulo para obter a localização exata no espaço de endereço do processo. Antes que o arquivo seja carregado, no entanto, o endereço pode ser determinado subtraindo-se o endereço virtual do cabeçalho da seção (*VirtualAddress*) do endereço do campo especificado, adicionando o deslocamento do corpo da seção (*PointerToRawData*) ao resultado e usando esse valor como um deslocamento o arquivo de imagem. O exemplo a seguir ilustra essa técnica:

### PEFILE.C

```

int WINAPI GetExportFunctionNames (
    LPVOID lpFile,
    HANDLE hHeap,
    char ** pszFunctions)
{
    IMAGE_SECTION_HEADER sh;
    PIMAGE_EXPORT_DIRECTORY ped;
    char * pNames, * pCnt;
    int i, nCnt;

    /* Obtém o cabeçalho da seção e o ponteiro para o diretório de dados
       para a seção .edata. */
    if ((ped = (PIMAGE_EXPORT_DIRECTORY) ImageDirectoryOffset
        (lpFile, IMAGE_DIRECTORY_ENTRY_EXPORT)) == NULL)

```

```

        return 0;
GetSectionHdrByName (lpFile, & sh, ".edata");

/* Determina o deslocamento dos nomes das funções de exportação. */
pNames = (char *) (* (int *) ((int) ped-> AddressOfNames -
                               (int) sh.VirtualAddress +
                               (int) sh.PointerToRawData +
                               (int) lpFile) -
                           (int) sh.VirtualAddress +
                           (int) sh.PointerToRawData +
                           (int) lpFile);

/* Descobrir quanta memória para alocar para todas as seqüências de caracteres. */
pCnt = pNames;
para (i = 0; i <(int) ped-> NumberOfNames; i++)
    while (* pCnt++);
nCnt = (int) (pCnt. pNames);

/* Alocar memória fora da pilha para nomes de função. */
* pszFunctions = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nCnt);

/* Copie todas as strings para o buffer. */
CopiarMemory ((LPVOID) * pszFunctions, (LPVOID) pNames, nCnt);

return nCnt;
}

```

Observe que, nessa função, a variável *pNames* é atribuída determinando primeiro o endereço do deslocamento e, em seguida, o local de deslocamento real. O endereço do deslocamento e o próprio deslocamento são endereços virtuais relativos e devem ser traduzidos antes de serem usados, como a função demonstra. Você poderia escrever uma função semelhante para determinar os valores ordinais ou pontos de entrada das funções, mas por que se incomodar quando eu já fiz isso para você? As **funções GetNumberOfExportedFunctions**, **GetExportFunctionEntryPoints** e **GetExportFunctionOrdinals** também existem no PEFILE.DLL.

## Importar seção de dados, .idata

A seção .idata é dados de importação, incluindo o diretório de importação e a tabela de nomes de endereços de importação. Embora um diretório IMAGE\_DIRECTORY\_ENTRY\_IMPORT esteja definido, nenhuma estrutura de diretório de importação correspondente é incluída no arquivo WINNT.H. Em vez disso, existem várias outras estruturas chamadas IMAGE\_IMPORT\_BY\_NAME, IMAGE\_THUNK\_DATA e IMAGE\_IMPORT\_DESCRIPTOR. Pessoalmente, eu não conseguia entender como essas estruturas deveriam se correlacionar com a seção .idata, então passei várias horas decifrando o corpo da seção .idata e encontrei uma estrutura muito mais simples. Eu nomeei essa estrutura como **IMAGE\_IMPORT\_MODULE\_DIRECTORY**.

## PEFILE.H

```

typedef struct tagImportDirectory
{
    DWORD dwRVAFunctionNameList;
    DWORD dwUseless1;
    DWORD dwUseless2;
    DWORD dwRVAModuleName;
    DWORD dwRVAFunctionAddressList;
} IMAGE_IMPORT_MODULE_DIRECTORY,
* PIMAGE_IMPORT_MODULE_DIRECTORY;

```

Ao contrário dos diretórios de dados de outras seções, este se repete um após o outro para cada módulo importado no arquivo. Pense nisso como uma entrada em uma lista de diretórios de dados do módulo, em vez de um diretório de dados para toda a seção de dados. Cada entrada é um diretório para as informações de importação de um módulo específico.

Um dos campos na estrutura **IMAGE\_IMPORT\_MODULE\_DIRECTORY** é *dwRVAModuleName*, um endereço virtual relativo que aponta para o nome do módulo. Há também dois parâmetros *dwUseless* na estrutura que servem como preenchimento para manter a estrutura alinhada corretamente dentro da seção. A especificação de formato de arquivo PE menciona algo sobre sinalizadores de importação, um registro de data / hora e versões principais / secundárias, mas esses dois campos permaneceram vazios durante toda a minha experimentação, então ainda considero-os inúteis.

Com base na definição dessa estrutura, você pode recuperar os nomes dos módulos e todas as funções em cada módulo que são importados por um arquivo executável. A função a seguir demonstra como recuperar todos os nomes de módulo importados por um arquivo PE particular:

## PEFILE.C

```

int WINAPI GetImportModuleNames (
    LPVOID lpFile,

```

```

HANDLE hHeap,
char ** pszModules)
{
    PIMAGE_IMPORT_MODULE_DIRECTORY pid;
    IMAGE_SECTION_HEADER idsh;
    BYTE * pData;
    int nCnt = 0, nSize = 0, i;
    char * pModule [1024];
    char * psz;

    pid = (PIMAGE_IMPORT_MODULE_DIRECTORY) ImageDirectoryOffset
        (lpFile, IMAGE_DIRECTORY_ENTRY_IMPORT);
    pData = (BYTE *) pid;

    /* Localize o cabeçalho da seção ".idata". */
    if (!GetSectionHdrByName (lpFile, & idsh, ".idata"))
        return 0;

    /* Extraí todos os módulos de importação. */
    while (pid-> dwRVAModuleName)
    {
        /* Alocar buffer para deslocamentos de seqüência de caracteres absolutos. */
        pModule [nCnt] = (char *) (pData +
            (pid-> dwRVAModuleName-idsh.VirtualAddress));
        nSize += strlen (pMódulo [nCnt]) + 1;

        /* Incrementar para a próxima entrada do diretório de importação. */
        pid++;
        nCnt++;
    }

    /* Copie todas as seqüências de caracteres para um bloco de memória heap. */
    * pszModules = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nSize);
    psz = * pszModules;
    para (i = 0; i <nCnt; i++)
    {
        strcpy (psz, pMódulo [i]);
        psz += strlen (psz) + 1;
    }

    return nCnt;
}

```

A função é bem direta. No entanto, vale a pena destacar uma coisa - observe o loop while. Esse loop é finalizado quando *pid-> dwRVAModuleName* é 0. Implícito aqui é que no final da lista de **IMAGE\_IMPORT\_MODULE\_DIRECTORY** estruturas é uma estrutura nula que possui um valor de 0 pelo menos o campo *dwRVAModuleName*. Esse é o comportamento que observei na minha experimentação com o arquivo e depois confirmado na especificação do formato de arquivo PE.

O primeiro campo na estrutura, *dwRVAFunctionNameList*, é um endereço virtual relativo a uma lista de endereços virtuais relativos que apontam para os nomes das funções no arquivo. Conforme mostrado nos dados a seguir, os nomes dos módulos e funções de todos os módulos importados são listados nos dados da seção .idata:

```

E6A7 0000 F6A7 0000 08A8 0000 1AA8 0000 .....
28A8 0000 3CA8 0000 4CA8 0000 0000 0000 (... <... L .....
0000 4765 744F 7065 6E46 696C 654E 616D .. GetOpenFileName
6541 0000 636F 6D64 6C67 3332 2E64 6C6C eA..comdlg32.dll
0000 2500 4372 6561 7465 466F 6E74 496E ... CreateFontIn
6469 7265 6374 4100 4744 4933 322E 646C directA.GDI32.dl
6C00 A000 4765 7444 6576 6963 6543 6170 l ... GetDeviceCap
7300 C600 4765 7453 746F 636B 4F62 6A65 s ... GetStockObject
6374 0000 D500 4765 7454 6578 744D 6574 ct .... GetTextMetrics
7269 6373 4100 1001 5365 6C65 6374 4F62 ricsA ... SelectObject
6A65 6374 0000 1601 5365 7442 6B43 6F6C ject .... SetBkColor
6F72 0000 3501 5365 7454 6578 7443 6F6C ou..5.SetTextCol
6F72 0000 4501 5465 7874 4F75 7441 0000 ou..E.TextOutA ..

```

Os dados acima são uma parte da seção .idata do aplicativo de exemplo EXEVIEW.EXE. Essa seção específica representa o início da lista de nomes de módulos e funções de importação. Se você começar a examinar a parte da seção direita dos dados, deverá reconhecer os nomes das funções familiares da API do Win32 e os nomes dos módulos encontrados. Lendo de cima para baixo, você obtém **GetOpenFileNameA**, seguido pelo nome do módulo COMDLG32.DLL. Logo depois disso, você obtém **CreateFontIndirectA**, seguido pelo módulo GDI32.DLL e, em seguida, as funções **GetDeviceCaps**, **GetStockObject**, **GetTextMetrics** e assim por diante.

Esse padrão se repete em toda a seção .idata. O primeiro nome do módulo é COMDLG32.DLL e o segundo é GDI32.DLL. Observe que apenas uma função é importada do primeiro módulo, enquanto muitas funções são

importadas do segundo módulo. Em ambos os casos, os nomes de função e o nome do módulo ao qual pertencem são ordenados de tal forma que um nome de função apareça primeiro, seguido pelo nome do módulo e depois pelos demais nomes de função, se houver.

A seguinte função demonstra como recuperar os nomes de função para um módulo específico:

### PEFILE.C

```

int WINAPI GetImportFunctionNamesByModule (
    LPVOID lpFile,
    HANDLE hHeap,
    char * pszModule,
    char ** pszFunctions)
{
    PIMAGE_IMPORT_MODULE_DIRECTORY pid;
    IMAGE_SECTION_HEADER idsh;
    DWORD dwBase;
    int nCnt = 0, nSize = 0;
    DWORD dwFunction;
    char * psz;

    /* Localize o cabeçalho da seção ".idata". */
    if (!GetSectionHdrByName (lpFile, & idsh, ".idata"))
        return 0;

    pid = (PIMAGE_IMPORT_MODULE_DIRECTORY) ImageDirectoryOffset
        (lpFile, IMAGE_DIRECTORY_ENTRY_IMPORT);

    dwBase = ((DWORD) pid. idsh.VirtualAddress);

    /* Encontre o pid do módulo. */
    while (pid-> dwRVAModuleName &&
        strcmp (pszModule,
            (char *) (pid-> dwRVAModuleName + dwBase)) )
        pid ++;

    /* Sair se o módulo não for encontrado. */
    if (! pid-> dwRVAModuleName)
        return 0;

    /* Número de contagem de nomes de função e comprimento de seqüências de caracteres. */
    dwFunction = pid-> dwRVAFunctionNameList;
    while (dwFunction &&
        * (DWORD *) (dwFunction + dwBase) &&
        * (char *) ((* (DWORD *) (dwFunction + dwBase)) +
            dwBase + 2))
    {
        nSize += strlen ((char *) ((* (DWORD *) (dwFuncao +
            dwBase)) + dwBase + 2)) + 1;
        dwFunction += 4;
        nCnt++;
    }

    /* Alocar memória fora da pilha para nomes de função. */
    * pszFunctions = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nSize);
    psz = * pszFunctions;

    /* Copie os nomes das funções para o ponteiro da memória. */
    dwFunction = pid-> dwRVAFunctionNameList;
    while (dwFunction &&
        * (DWORD *) (dwFunction + dwBase) &&
        * ((char *) ((* (DWORD *) (dwFunction + dwBase)) +
            dwBase + 2)))
    {
        strcpy (psz, (char *) ((* (DWORD *) (funão + dwBase)) +
            dwBase + 2));
        psz += strlen ((char *) ((* (DWORD *) (dwFunção + dwBase)) +
            dwBase + 2)) + 1;
        dwFunction += 4;
    }

    return nCnt;
}

```

Como a função **GetImportModuleNames** , essa função depende do final de cada lista de informações para ter uma entrada zerada. Nesse caso, a lista de nomes de funções termina com um que é zero.

O campo final, *dwRVAFunctionAddressList* , é um endereço virtual relativo a uma lista de endereços virtuais que serão colocados nos dados da seção pelo carregador quando o arquivo é carregado. Antes que o arquivo seja carregado, no entanto, esses endereços virtuais são substituídos por endereços virtuais relativos que correspondem exatamente à lista de nomes de funções. Portanto, antes que o arquivo seja carregado, há duas listas idênticas de endereços virtuais relativos apontando para nomes de função importados.

## Seção de informações de depuração, .debug

Informações de depuração são inicialmente colocadas na seção .debug. O formato de arquivo PE também suporta arquivos de depuração separados (normalmente identificados com uma extensão .DBG) como um meio de coletar informações de depuração em um local central. A seção de depuração contém as informações de depuração, mas os diretórios de depuração estão na seção .rdata mencionada anteriormente. Cada um desses diretórios faz referência a informações de depuração na seção .debug. A estrutura do diretório de depuração é definida como um

**IMAGE\_DEBUG\_DIRECTORY** , da seguinte maneira:

### WINNT.H

```
typedef struct _IMAGE_DEBUG_DIRECTORY {
    ULONG Características;
    ULONG TimeStamp;
    USHORT MajorVersion;
    USHORT MinorVersion;
    Tipo ULONG;
    ULONG SizeOfData;
    ULONG AddressOfRawData;
    ULONG PointerToRawData;
} IMAGE_DEBUG_DIRECTORY, * PIMAGE_DEBUG_DIRECTORY;
```

A seção é dividida em partes separadas de dados que representam diferentes tipos de informações de depuração. Para cada um, existe um diretório de depuração descrito acima. Os diferentes tipos de informações de depuração estão listados abaixo:

### WINNT.H

```
#define IMAGE_DEBUG_TYPE_UNKNOWN 0
#define IMAGE_DEBUG_TYPE_COFF 1
#define IMAGE_DEBUG_TYPE_CODEVIEW 2
#define IMAGE_DEBUG_TYPE_FPO 3
#define IMAGE_DEBUG_TYPE_MISC 4
```

O campo *Tipo* em cada diretório indica qual tipo de informação de depuração o diretório representa. Como você pode ver na lista acima, o formato de arquivo PE suporta muitos tipos diferentes de informações de depuração, bem como alguns outros campos informativos. Dessas, as informações de **IMAGE\_DEBUG\_TYPE\_MISC** são exclusivas. Essas informações foram adicionadas para representar diversas informações sobre a imagem executável que não puderam ser adicionadas a nenhuma das seções de dados mais estruturadas no formato de arquivo PE. Esse é o único local no arquivo de imagem em que o nome da imagem aparecerá com certeza. Se uma imagem exportar informações, a seção de dados de exportação também incluirá o nome da imagem.

Cada tipo de informação de depuração possui sua própria estrutura de cabeçalho que define seus dados. Cada um deles está listado no arquivo WINNT.H. Uma coisa legal sobre a estrutura **IMAGE\_DEBUG\_DIRECTORY** é que ela inclui dois campos que identificam as informações de depuração. O primeiro deles, *AddressOfRawData* , é o endereço virtual relativo dos dados quando o arquivo é carregado. O outro, *PointerToRawData* , é um deslocamento real dentro do arquivo PE, onde os dados estão localizados. Isso facilita a localização de informações específicas de depuração.

Como último exemplo, considere a seguinte função, que extrai o nome da imagem da estrutura **IMAGE\_DEBUG\_MISC** :

### PEFILE.C

```
int WINAPI RetrieveModuleName (
    LPVOID lpFile,
    HANDLE hHeap,
    char ** pszModule)
{

    PIMAGE_DEBUG_DIRECTORY pdd;
    PIMAGE_DEBUG_MISC pdm = NULL;
    int nCnt;

    if (! (pdd = (PIMAGE_DEBUG_DIRECTORY) ImageDirectoryOffset
        (lpFile, IMAGE_DIRECTORY_ENTRY_DEBUG)))
        return 0;

    pdm = (PIMAGE_DEBUG_MISC) pdd->PointerToRawData;
```

```

while (pdd-> SizeOfData)
{
    if (pdd-> Type == IMAGE_DEBUG_TYPE_MISC)
    {
        pdm = (PIMAGE_DEBUG_MISC)
            ((DWORD) pdd-> PointerToRawData + (DWORD) lpFile);

        nCnt = lstrlen (pdm-> Data) * (pdm-> Unicode? 2: 1);
        *pszModule = (char *) HeapAlloc (hHeap,
                                         HEAP_ZERO_MEMORY,
                                         nCnt + 1);
        CopyMemory (*pszModule, pdm-> Data, nCnt);

        pausa;
    }

    pdd++;
}

if (pdm! = NULL)
    return nCnt;
outro
    return 0;
}

```

Como você pode ver, a estrutura do diretório de depuração torna relativamente fácil localizar um tipo específico de informações de depuração. Uma vez localizada a estrutura **IMAGE\_DEBUG\_MISC**, extrair o nome da imagem é tão simples quanto invocar a função **CopyMemory**.

Como mencionado acima, as informações de depuração podem ser separadas em arquivos .DBG separados. O SDK do Windows NT inclui um utilitário chamado REBASE.EXE que serve essa finalidade. Por exemplo, na seguinte declaração, uma imagem executável denominada TEST.EXE está sendo removida das informações de depuração:

```
rebase -b 40000 -xc: \ samples \ testdir test.exe
```

As informações de depuração são colocadas em um novo arquivo chamado TEST.DBG e localizado no caminho especificado, neste caso, c: \ samples \ testdir. O arquivo começa com uma única estrutura **IMAGE\_SEPARATE\_DEBUG\_HEADER**, seguida por uma cópia dos cabeçalhos de seção existentes na imagem executável removida. Em seguida, os dados da seção .debug seguem os cabeçalhos da seção. Então, logo após os cabeçalhos de seção, estão as séries de estruturas **IMAGE\_DEBUG\_DIRECTORY** e seus dados associados. A informação de depuração em si mantém a mesma estrutura descrita acima para as informações de depuração do arquivo de imagem normal.

## Resumo do formato de arquivo PE

O formato de arquivo PE para o Windows NT introduz uma estrutura completamente nova para desenvolvedores familiarizados com os ambientes Windows e MS-DOS. No entanto, desenvolvedores familiarizados com o ambiente UNIX descobrirão que o formato de arquivo PE é similar, se não baseado na especificação COFF.

Todo o formato consiste em um cabeçalho MS-DOS MZ, seguido por um programa stub de modo real, a assinatura de arquivo PE, o cabeçalho de arquivo PE, o cabeçalho opcional PE, todos os cabeçalhos de seção e, finalmente, todos os corpos de seção.

O cabeçalho opcional termina com uma matriz de entradas do diretório de dados que são endereços virtuais relativos aos diretórios de dados contidos nos corpos de seção. Cada diretório de dados indica como os dados de um corpo de seção específico são estruturados.

O formato de arquivo PE possui onze seções predefinidas, como é comum em aplicativos para o Windows NT, mas cada aplicativo pode definir suas próprias seções exclusivas para código e dados.

A seção pré-definida .debug também tem a capacidade de ser removida do arquivo para um arquivo de depuração separado. Nesse caso, um cabeçalho de depuração especial é usado para analisar o arquivo de depuração e um sinalizador é especificado no cabeçalho do arquivo PE para indicar que os dados de depuração foram eliminados.

## Descrições da Função PEFILE.DLL

PEFILE.DLL consiste principalmente de funções que recuperam um deslocamento em um determinado arquivo PE ou copiam uma parte dos dados do arquivo para uma estrutura específica. Cada função tem um único requisito - o primeiro parâmetro é um ponteiro para o início do arquivo PE. Ou seja, o arquivo deve primeiro ser mapeado na memória para o espaço de endereço do seu processo, e o local base do mapeamento de arquivo é o valor *lpFile* que você passa como o primeiro parâmetro para cada função.

Os nomes das funções devem ser auto-explicativos, e cada função é listada com um breve comentário descrevendo seu propósito. Se, depois de ler a lista de funções, você não puder determinar para que serve uma função, consulte o aplicativo de exemplo EXEVIEW.EXE para encontrar um exemplo de como a função é usada. A seguinte lista de protótipos de função também pode ser encontrada em PEFILE.H:

## PEFILE.H

```

/* Recupere um deslocamento de ponteiro para o cabeçalho MS-DOS MZ. */
BOOL WINAPI GetDosHeader (LPVOID, PIMAGE_DOS_HEADER);

/* Determine o tipo de um arquivo .exe. */
DWORD WINAPI ImageFileType (LPVOID);

/* Recupera um deslocamento de ponteiro para o cabeçalho do arquivo PE. */
BOOL WINAPI GetPEFileHeader (LPVOID, PIMAGE_FILE_HEADER);

/* Recupera um deslocamento de ponteiro para o cabeçalho opcional do PE. */
BOOL WINAPI GetPEOptionalHeader (LPVOID,
                                 PIMAGE_OPTIONAL_HEADER);

/* Retorna o endereço do ponto de entrada do módulo. */
LPVOID WINAPI GetModuleEntryPoint (LPVOID);

/* Retorna uma contagem do número de seções no arquivo. */
int WINAPI NumOfSections (LPVOID);

/* Retorna o endereço base desejado do executável quando
   Ele é carregado no espaço de endereço de um processo. */
LPVOID WINAPI GetImageBase (LPVOID);

/* Determine o local dentro do arquivo de um determinado
   diretório de dados de imagem. */
LPVOID WINAPI ImageDirectoryOffset (LPVOID, DWORD);

/* Função recupera nomes de todas as seções no arquivo. */
int WINAPI GetSectionNames (LPVOID, HANDLE, char **);

/* Copie as informações do cabeçalho da seção para uma seção específica. */
BOOL WINAPI GetSectionHdrByName (LPVOID,
                                 PIMAGE_SECTION_HEADER, char *);

/* Obter lista separada por núm de nomes de módulos de importação. */
int WINAPI GetImportModuleNames (LPVOID, HANDLE, char **);

/* Obter lista de funções de importação separadas por nulo para um módulo. */
int WINAPI GetImportFunctionNamesByModule (LPVOID, HANDLE,
                                             char *, char **);

/* Obter lista separada por NULL de nomes de função exportados. */
int WINAPI GetExportFunctionNames (LPVOID, HANDLE, char **);

/* Obtém o número de funções exportadas. */
int WINAPI GetNumberOfExportedFunctions (LPVOID);

/* Obter lista de pontos de entrada de endereço virtual de função exportados. */
LPVOID WINAPI GetExportFunctionEntryPoints (LPVOID);

/* Obter lista de valores ordinais de função exportados. */
LPVOID WINAPI GetExportFunctionOrdinals (LPVOID);

/* Determinar o número total de objetos de recursos. */
int WINAPI GetNumberOfResources (LPVOID);

/* Lista de retorno de todos os tipos de objetos de recursos usados no arquivo. */
int WINAPI GetListOfResourceTypes (LPVOID, HANDLE, char **);

/* Determine se as informações de depuração foram removidas do arquivo. */
BOOL WINAPI IsDebugInfoStripped (LPVOID);

/* Obtém o nome do arquivo de imagem. */
int WINAPI RetrieveModuleName (LPVOID, HANDLE, char **);

/* Função determina se o arquivo é um arquivo de depuração válido. */
BOOL WINAPI IsDebugFile (LPVOID);

/* Função retorna o cabeçalho de depuração do arquivo de depuração. */

```

```
BOOL WINAPI GetSeparateDebugHeader (LPVOID,
                                     PIMAGE_SEPARATE_DEBUG_HEADER);
```

Além das funções listadas acima, as macros mencionadas anteriormente neste artigo também são definidas no arquivo PEFILE.H. A lista completa é a seguinte:

```
/ * Deslocamento para assinatura de arquivo PE * /
#define NTSIGNATURE (a) ((LPVOID) ((BYTE *) a + \
                           ((PIMAGE_DOS_HEADER) a) -> e_lfanew))

/ * O cabeçalho MS-OS identifica o dword de assinatura do NT PEfile;
   o cabeçalho PEFILE existe logo após esse dword. * /
#define PEFHDOFFSET (a) ((LPVOID) ((BYTE *) a + \
                           ((PIMAGE_DOS_HEADER) a) -> e_lfanew + \
                           SIZE_OF_NT_SIGNATURE))

/ * O cabeçalho opcional PE é imediatamente após o cabeçalho PEfile. * /
#define OPTHDOFFSET (a) ((LPVOID) ((BYTE *) a + \
                           ((PIMAGE_DOS_HEADER) a) -> e_lfanew + \
                           SIZE_OF_NT_SIGNATURE + \
                           sizeof (IMAGE_FILE_HEADER)))

/ * Os cabeçalhos de seção são imediatamente após o cabeçalho opcional do PE. * /
#define SECHDOFFSET (a) ((LPVOID) ((BYTE *) a + \
                           ((PIMAGE_DOS_HEADER) a) -> e_lfanew + \
                           SIZE_OF_NT_SIGNATURE + \
                           sizeof (IMAGE_FILE_HEADER) + \
                           sizeof (IMAGE_OPTIONAL_HEADER)))
```

Para usar o PEFILE.DLL, simplesmente inclua o arquivo de cabeçalho PEFILE.H e vincule a DLL ao seu aplicativo. Todas as funções são mutuamente exclusivas, mas algumas foram escritas tanto para apoiar outras quanto para as informações que elas fornecem. Por exemplo, a função **GetSectionNames** é útil para obter os nomes exatos de todas as seções. No entanto, para poder recuperar o cabeçalho de seção para um nome de seção exclusivo (um definido pelo desenvolvedor de aplicativos durante a compilação), você primeiro teria que obter a lista de nomes e, em seguida, chamar a função **GetSectionHeaderByName** com o nome exato da seção. Apreciar!

© 1997 Microsoft Corporation. Todos os direitos reservados. [Notícias legais](#)