ENGENHARIA REVERSA

www.mundohacker.com

Obs: recomendo saber programar em Assembly!

O que é a Engenharia Reversa

Como o próprio nome indica, a Engenharia Reversa é uma engenharia "ao contrário", portanto, é uma atividade que trabalha com um produto existente (um software, uma peça mecânica, uma placa de computador, etc.) e tenta entender como este produto funciona e o que ele faz exatamente (todas as suas propriedades em quaisquer circunstâncias). Fazemos engenharia reversa quando queremos trocar ou modificar uma peça (ou um software) por outro, com as mesmas características, mas não temos todas as informações sobre essa peça. Utilizando mecanismos de busca na Internet como Google, Altavista, Lycos, etc, as referências sobre Engenharia Reversa são surpreendentemente abundantes. Encontra-se desde programas de cursos universitários até ensaios e teses, além dos tradicionais sites "relâmpago" (hoje está no ar, amanhã foi banido do ISP) dos crackers de plantão.

Porque aprender Engenharia Reversa

Crackers são os que aplicam conhecimentos de engenharia reversa para "liberar" programas que exijam algum tipo de registro ou para habilitar funções que estejam bloqueadas em demos, trials, etc. É natural que os autores de software queiram proteger seus programas e o fazem razoavelmente bem... até encontrar um cracker que se dispõe a "liberá-los". O que os programadores esquecem é que precisam se interessar pela engenharia reversa, dando uma atenção especial à programação de segurança. Se você não conhece o inimigo, como é que pretende se defender ? Aprenda as técnicas utilizadas pelos estudiosos da matéria e pelos crackers mais famosos e aplique seus conhecimentos (não existe proteção 100%, mas 99% já é um bom resultado) - ou então parta para o software livre e não pense mais no assunto !

Como aprender Engenharia Reversa

Como foi dito acima, existem muitos sites dedicados ao assunto, basta dar uma olhada no Google. Encontra-se desde conceitos básicos até cursos na web (geralmente em inglês, como Hellforge) e tutoriais de crackers com exemplos muito elucidativos. De qualquer modo, é necessário um conhecimento básico da linguagem assembly além de um domínio razoável de uma outra linguagem de programação (preferencialmente C/C++). Se você programa para Windows rodando em processadores Intel, é claro que também é necessário um conhecimento básico das API do Windows, da arquitetura Intel e do conjunto de instruções desses processadores. O mesmo se aplica para a família Unix/Linux, Sparc, etc. Além disso, entre outras coisas, procure se informar ao máximo sobre padrões: comece com os padrões de PE (portable executable), compactação, encriptação de arquivos e instaladores.

Onde aprender Engenharia Reversa

Na universidade ou... em casa :-). No recesso do lar, alguns livros, cabeça fria e uma conexão com a web. Pesquise na Internet, faça alguns bookmarks e prepare-se para penosos downloads: você vai precisar de algumas ferramentas, paciência para aguentar janelinhas popups de sites de crackers, perseverança para encontrar tutoriais de qualidade, etc, etc, etc.

⇨ **A TEORIA DA ENGENHARIA REVERSA**

O que é a Engenharia Reversa
Como o próprio nome indica, a Engenharia Reversa é uma engenharia "ao contrário", portanto, é uma atividade que trabalha com um produto existente (um software, uma peça mecânica, uma placa de computador, etc.) e tenta entender como este produto funciona e o que ele faz exatamente (todas as suas propriedades em qualquer circunstâncias). Fazemos engenharia reversa quando queremos trocar ou modificar uma peça (ou um software) por outro, com as mesmas características, mas não temos todas as informações sobre essa peça.

Engenharia Reversa de Software
Utiliza-se a Engenharia Reversa de Software nos seguintes casos:
Para adaptar o software a novos computadores.
Para atualizar o softwares (novas bibliotecas, novas linguagem de programação, novas ferramentas).
Para adaptar o software a novas regras (troca de moeda).
Para disponibilizar novas funcionalidades.
Para corrigir bugs.
Os crackers utilizam as técnicas da Engenharia Reversa para disponibilizar funcionalidades "escondidas" em programas que normalmente oferecem estas funcionalidades após serem legalmente adquiridos e registrados. Um cracker "bobinho" é chamado de lammer. Um cracker expert costuma dizer que faz Engenharia Reversa e que o software existente é apenas material de trabalho... (vê lá, nada de crackear programas, hem ;-) Existem alguns experts que chegam até ao ponto de fazer re-engenharia de software, ou seja, analisam e modificam um sistema para recriá-lo e reimplementá-lo com uma nova estrutura.

Etapas da Engenharia Reversa
O trabalho de Engenharia Reversa é feito em etapas bem definidas:

Extração de fatos do sistema a analisar
Tratamento dos fatos e
Visualização dos resultados

Extração de Fatos de um Sistema

A Extração de Fatos é feita através de:

Análise Estática do Código
Análise Dinâmica do Código
Dados
Documentação
Outras fontes de informação

Análise Estática do Código - Parsing
Na Análise Estática do Código procura-se:

Determinar quais são os componentes básicos do sistema, como arquivos, rotinas, variáveis, etc.
Relações de Definição: diretório de determinado arquivo, arquivo onde se encontram determinadas variáveis, etc
Relações de Referência: qual arquivo depende de outro, qual rotina depende de outra, etc
Existem ferramentas para fazer este tipo de análise - são os "Parsers" - e a análise é chamada de "parsing". Algumas linguagens de programação permitem facilmente um parsing (como o Delphi - Pascal, por exemplo), outras já oferecem dificuldades maiores.

Análise Dinâmica do Código - Debugging
Na Análise Dinâmica executa-se o programa e se monitora os valores das variáveis, quais funções são chamadas, etc. As ferramentas utilizadas são denominadas de "Debuggers", sendo o mais conhecido deles o SoftIce da Numega.

Dados
Quando um sistema possuir um banco de dados, este pode servir de fonte de informação sobre o próprio sistema.

Documentação
Documentação é tudo o que não está usado pelo computador para fazer funcionar o sistema. Podem ser textos, diagramas, helps, etc.

Outras fontes de informação
A linguagem de programação que foi utilizada, o sistema operacional, o tipo de processador, etc.

Tratamento dos Fatos

Uma vez que se tenha informações suficientemente claras e precisas, é possível alterar o código e até a estrutura de um sistema (ou programa) utilizando várias técnicas. Nem é preciso dizer que a documentação do tratamento efetuado é fundamental !

Visualização dos Resultados

;-) ;-)

⇨ CONCEITOS BÁSICOS DE PROGRAMAÇÃO

Processo de Desenvolvimento de Programas
O processo de desenvolvimento de qualquer programa envolve quatro etapas principais:

Criação ou edição de um arquivos-fonte na linguagem da sua escolha
Compilação
Linkedição
Execução
(Teste de Segurança caso o programa seja protegido)

1. Criação de arquivos-fonte

A criação de arquivos-fonte é o processo de criar os componentes do programa, definir suas características, suas propriedades e dependências, além de

convencionar seu comportamento. Isto é feito utilizando-se a linguagem da sua escolha, obviamente seguindo rigidamente a sintaxe da linguagem escolhida. Nada mais é do que "escolher os atores", atribuir-lhes um "papel" e exigir que ajam de acordo com o "script": o arquivo-fonte vai ser transformado numa grande peça de teatro que o usuário final vai assistir.

2. Compilação de arquivos-fonte

Os compiladores são programas que funcionam como o diretor da peça de teatro: garantem que o arquivo-fonte (atores, script, etc) atuem de acordo com as regras, ou seja, garantem que os arquivos-fonte não estejam violando nenhuma regra de linguagem. Os compiladores produzem arquivos intermediários, chamados arquivos objeto, que são utilizados na etapa seguinte: a linkedição.

3. Linkedição de arquivos-objeto

O linkeditor prepara a peça teatral para a estréia: combina um ou mais arquivos compilados com arquivos de biblioteca específicos para produzir um programa executável. A linkedição obedece a padrões de estrutura de combinação que, no caso de sistemas de 32 bits do windows, é o formato de arquivos Portable Executable (PE). Os arquivos PE, produzidos pelos linkeditores nada mais são do que a concatenação de arquivos compilados em dados armazenados em seções de acordo com um padrão conhecido.

(Na linguagem corrente do "computês", costuma-se denominar os arquivos compilados E linkeditados apenas como executáveis ou, genericamente, como arquivos compilados).

4. Execução do programa

Na fase de execução, o programa compilado e linkeditado em executável é testado para se verificar se corresponde ao projeto ou se ainda deve sofrer alterações e/ou correções até chegar à sua forma final.

5. Teste de Segurança

É preciso conhecer a Engenharia Reversa tão bem (senão melhor) quanto a Engenharia de Software.

É óbvio que programas protegidos devem possuir no seu projeto um planejamento detalhado do sistema de segurança. Para avaliar a eficácia deste sistema existe a necessidade de se incluir mais uma etapa: o Teste de Segurança.

Dificilmente os programadores se dão ao trabalho de dar uma olhada num arquivo compilado ou linkeditado. Para efetuar uma análise consistente, é óbvio que se deve conhecer a contrapartida da Engenharia de Software, ou seja, a Engenharia Reversa.

Se quisermos analisar um programa a partir do seu código de máquina (compilado/linkeditado), precisamos lançar mão de "tradutores reversos" que transformam código de máquina em alguma linguagem que possa ser entendida. Os mais conhecidos e utilizados são o W32Dasm e o IDA, que "traduzem" para a linguagem Assembly. Os arquivos texto contendo o código "traduzido" são chamados de dead listings.

Se quisermos analisar nosso código dinamicamente, é preciso conhecer, entre outras coisas, a organização da Memória RAM e saber operar debugadores, programas monitores de funções importadas e do registry.

Dead Listing
Dead Listing Dead listing é uma listagem desassemblada de um programa em texto raso e que, na forma de Linguagem Assembly, descreve cada instrução que o programa possa usar ou executar. Geralmente não se dá a devida importância a uma dead listing, seja por não dominar o Assembly, seja por considerar que a análise estática do código não leva aos resultados esperados. Ledo engano... Conhecimentos básicos de Assembly são suficientes e os resultados surpreendentes !

Organização da Memória RAM
RAM Um programa, para poder ser executado, precisa inicialmente ser carregado na memória do computador, ou seja, todo o seu bloco de código referente a rotinas, funções, resources, dados, etc, precisa estar disponível. Este conceito fica um tanto vago se não tivermos pelo menos noções do que é e de como funciona a memória do computador.

É espantoso quantos programadores desconhecem a memória do computador: não sabem o que é e não sabem como funciona e, apesar disso, dependem essencialmente da memória para que seus programas possam ser executados. Se já existe esta dificuldade em relação a conhecimentos essenciais, o que dizer da segurança do programa ?

⇨ FORMATO PE

1. O Formato PE
Imagine o formato PE como sendo simplesmente um padrão de armazenamento de dados que o Windows entenda. São regras para guardar "cada coisa em seu lugar" de modo que o sistema possa achá-las quando necessário. Os programas possuem código, dados inicializados, dados não inicializados, etc, etc, etc. Tudo deve ser guardado de acordo com o modelo estabelecido e o resultado é uma imensa fileira de bytes. O objetivo deste texto (e dos subsequentes) é ir desvendando cada pedaço dessa tripa de bytes.

PE vem de Portable Executable e é um formato de binários executáveis (DLLs e programas) para windows NT, windows 9x e win32s. Também pode ser utilizado para arquivos de objetos e bibliotecas (libraries).

Este formato foi projetado pela Microsoft e padronizado pelo Comitê do TIS (tool interface standart) - Microsoft, Intel, Borland, Watcom, IBM e outros - em 1993. Aparentemente foi baseado no COFF, o "common object file format", usado para arquivos de objetos e executáveis nos vários sabores UNIX e no VMS.

O SDK do win32 inclui um arquivo header <winnt.h> que contém os #defines e typedefs para o formato PE. Estes serão mencionados no decorrer do texto.

A DLL "imagehelp.dll" também poderá ser útil. Ela faz parte do windows NT, porém a documentação é escassa. Algumas de suas funções são descritas no "Developer Network".

2. Layout Geral do formato PE
Cabeçalho MZ do DOS
Fragmento (stub) do DOS
Cabeçalho do Arquivo

Cabeçalho Opcional

Diretório de Dados
Cabeçalhos das Seções
Seção 1
Seção 2
...
Seção n
Logo no início de qualquer arquivo no formato PE encontra-se o cabeçalho MZ do DOS seguido por um fragmento (stub) executável MS-DOS. Este stub transforma qualquer arquivo PE num executável MS-DOS válido (depois tio Bill insiste em afirmar que o windows "baniu" o DOS...).

Após o stub do DOS existe uma assinatura de 32 bits contendo o número mágico (magic number - é assim mesmo que o pessoal o batizou) de valor 00004550h e identificado como IMAGE_NT_SIGNATURE.

Depois segue o cabeçalho do arquivo (file header) no formato COFF que indica em qual máquina o executável deve rodar, o número de seções que contém, a hora em que foi linkado, se é um executável ou uma DLL e assim por diante. (Neste contexto, a diferença entre um executável e uma DLL é a seguinte: uma DLL não pode ser iniciada, somente pode ser utilizada por outro binário e um binário não pode ser linkado a um executável).

Depois do cabeçalho do arquivo vem um cabeçalho opcional ("optional header"). Está sempre presente mas, mesmo assim, é chamado de opcional. É que o COFF utiliza um cabeçalho para bibliotecas, mas não para objetos, que é chamado de opcional. Este cabeçalho indica mais alguns detalhes de como o binário deve ser carregado: o endereço inicial, a quantidade reservada para a pilha (stack), o tamanho do segmento de dados etc.

Uma parte interessante do cabeçalho opcional é o array indicativo dos diretórios de dados (data directories). Estes diretórios contém ponteiros para dados residentes nas seções (sections). Se, por exemplo, um binário tiver um diretório de exportação (export directory), existe um ponteiro para este diretório no array, sob a denominação IMAGE_DIRECTORY_ENTRY_EXPORT, que apontará para uma das seções.

Após os cabeçalhos ficam as seções, precedidas pelos cabeçalhos de seções (section headers). Em última análise, o conteúdo das seções é o que realmente é necessário para executar um programa e todos os cabeçalhos e diretórios servem apenas para localizar este conteúdo.

Cada seção possui algumas flags sobre alinhamento, o tipo de dados que contém, se pode ser compartilhada, etc, além dos dados propriamente ditos. A maioria das seções, mas não todas, contém um ou mais diretórios referenciados através de entradas no array diretório de dados (data directory) do cabeçalho opcional. É o caso do diretório de funções exportadas ou do diretório de base de remanejamento (base relocations). Tipos de conteúdo sem diretório são, por exemplo, código executável ou dados inicializados.

Já COMPLICOU ?
A coisa já complicou? Está querendo desistir por aqui? Não se apavore. Cada um dos tópicos será explicado em detalhes e, se você tiver noções de assembly, vai ter muita coisa boa com que se divertir...

Se o pavor bateu, tente pelo menos o próximo texto - quem sabe você não desiste tão fácil. Se você está tranquilo, basta seguir a sequência.

3. Fragmento do DOS e Assinatura PE

Você já sabe que, logo no início de qualquer arquivo no formato PE encontra-se o cabeçalho MZ do DOS seguido por um fragmento (stub) executável MS-DOS.

Então preste atenção: é um executável DOS completo dentro do executável win32. O stub pode simplesmente mostrar uma string do tipo "This program cannot be run in DOS mode" ou ser um programa DOS completo. Dependendo da vontade do programador, muita coisa pode rolar antes que o executável win32 comece a rodar.

Stub significa fragmento. O conceito do fragmento do DOS vem desde o tempo dos executáveis de 16 bits do windows (os quais estão no formato NE). O mesmo fragmento é usado em executáveis OS/2, arquivos "self-extracting" e outros aplicativos de 32 bits. Para arquivos PE, o fragmento é um executável compatível com o MS-DOS 2.0, quase sempre constituído por cerca de 100 bytes, cuja função é dar mensagens de erro do tipo "este programa precisa do windows...".

Cabeçalho MZ do DOS

Os primeiros dois bytes de qualquer executável em formato PE constituem a assinatura do DOS. Você sabe que dois bytes formam uma palavra (WORD). O word da assinatura SEMPRE é a sequência "MZ", ou seja, 4D 5A em hexadecimal. Portanto, pode-se reconhecer o fragmento do DOS pela sua assinatura. Este reconhecimento é chamado de validação do cabeçalho DOS.

Se abrirmos nosso executável exemplo (o tutNB03.exe) - ou qualquer outro executável em formato PE - num editor hexadecimal, os primeiros 32 bytes serão os seguintes:

```
Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
0000 0000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....ÿÿ..
0000 0010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ¸.......@.......
```

Agora dê uma olhada nos bytes das posições 18 a 1B, marcados em verde. 4 bytes são 2 word, e 2 word são um DWORD. Lemos 40 00 00 00. Acontece que os processadores Intel (e compatíveis) guardam os bytes em ordem inversa. Então, lendo da direita para a esquerda, o valor encontrado é 0000 0040. Este valor indica o quanto devemos nos deslocar (offset) para encontrar o stub do DOS.

Fragmento (stub) do DOS

Seguindo a primeira pista, encontramos o executável DOS. A área destacada em azul é o stub do DOS. Uma parte dos valores tem o correspondente em ASCII de "This program cannot be run in DOS mode", que é a string que será mostrada caso se tente executar este programa à partir do DOS. Geralmente o código usa o serviço 9 da interrupção 21 do DOS para imprimir uma string e o serviço 4C da interrupção 21 para voltar ao ambiente DOS. A instrução de interrupção é CD 21 e as instruções estão destacadas em azul mais claro.

```
Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
0000 0000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....ÿÿ..
0000 0010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ¸.......@.......
0000 0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000 0030 00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 ............À...
0000 0040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..º..´.Í!Th
0000 0050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
0000 0060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
0000 0070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.......
```

No offset 3C encontra-se a segunda pista: C0 00 00 00. Já sabemos que corresponde a 0000 00C0. É onde se encontra a assinatura PE.

A assinatura PE

Seguindo a segunda pista, encontramos a assinatura PE que indica o início do cabeçalho do arquivo. Pressupõem-se que todo arquivo que contenha uma assinatura PE seja um arquivo PE válido - pelo menos o Windows "pensa" assim.

```
Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
0000 0000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.........ÿÿ..
0000 0010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ¸.......@.......
0000 0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000 0030 00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 ............À...
0000 0040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..º..´.Í!Th
0000 0050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
0000 0060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
0000 0070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.......
0000 0080 E3 E2 11 DB A7 83 7F 88 A7 83 7F 88 A7 83 7F 88 ãâ.Û§ƒ.ˆ§ƒ.ˆ§ƒ.ˆ
0000 0090 A7 83 7F 88 B4 83 7f 88 5B A3 6D 88 A6 83 7F 88 §ƒ.ˆ´ƒ.ˆ[£mˆ¦ƒ.ˆ
0000 00A0 60 85 79 88 A6 83 7F 88 52 69 63 68 A7 83 7F 88 `…yˆ¦ƒ.ˆRich§ƒ.ˆ
0000 00B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00 PE..L...yxU<....
```

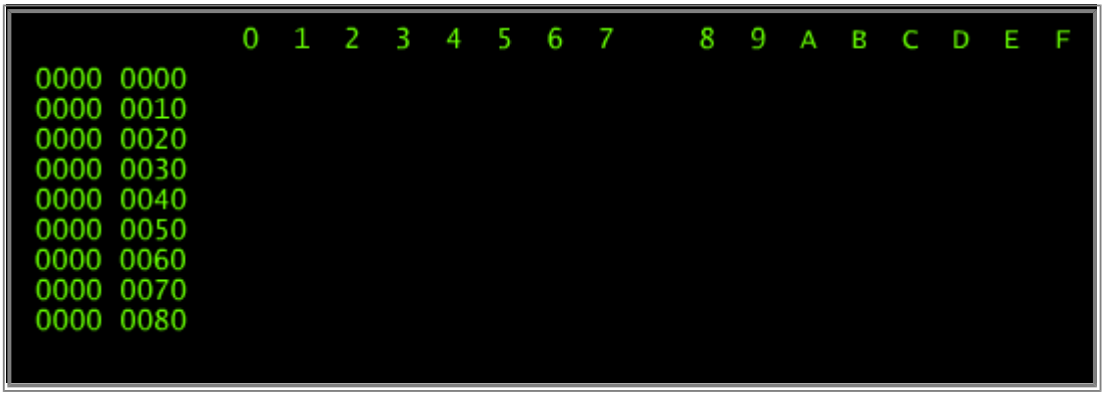
Para tirar a cisma, vamos usar nosso visualizador on line para dar uma olhada num executável que é um velho conhecido: o bloco de notas (notepad.exe). O visualizador é um objeto flash. Caso seu browser não esteja habilitado para rodar flash, habilite-o. Se você não tiver o plugin do flash instalado, faça o download na Macromedia.

Como dito acima, os primeiros dois bytes correspondem à assinatura do DOS. Os valores 4D e 5A são "M" e "Z" em ASCII. Os bytes seguintes, até o offset 78, compõem o stub.

1. Clique na linha 0000 0000, espere os bytes correspondentes entrarem e mova o cursor do mouse sobre o conteúdo: os words 4D e 5A são transformados nos ASCII correspondentes.

O que nos interessa particularmente é o DWORD no offset 3C. Este campo nos indica a posição da assinatura do PE. Encontramos os bytes 80 00 00 00 que, anotados no sentido inverso, indicam um offset de 0000 0080. É onde se encontra a assinatura PE.

2. Clique na linha 0000 0030, espere os bytes correspondentes entrarem e mova o cursor do mouse sobre o conteúdo. Aguarde enquanto os bytes do offset 3C são invertidos e o offset correspondente é destacado. Se você clicar na linha indicada, a 0000 0080, entram novamente os bytes correspondentes e, movendo o cursor sobre o conteúdo, os bytes 50 e 45 são transformados nos ASCII correspondentes "P" e "E".



É evidente que a mensagem que o DOS dá em caso de incompatibilidade precisa estar dentro do fragmento do DOS. Se você tiver curiosidade, siga o passo 3:

3. Clique sobre a linha 0000 0040 (também na 50, 60 e 70 se você quiser), espere os bytes entrarem e descanse o cursor sobre eles. Os valores hexadecimais serão transformados nos ASCII da mensagem.

4. Cabeçalho do Arquivo

Para obter o IMAGE_FILE_HEADER é preciso validar o "MZ" do cabeçalho do DOS (os primeiros 2 bytes), depois encontrar o membro 'e_lfanew' do cabeçalho do fragmento do DOS (offset 3C) e avançar o número indicado de bytes a partir do início do arquivo. Resumindo: na posição 003C encontra-se o 'e_lfanew' cujo valor, de 32 bits, nos indica a posição do início do cabeçalho do arquivo, ou seja, a IMAGE_NT_SIGNATURE, cujo valor é sempre 00004550 (veja no texto anterior se tiver dúvidas).

Os componentes do cabeçalho do arquivo são os seguintes:

4a. Assinatura do cabeçalho PE
4b. Tipo de máquina previsto para rodar o executável
4c. Número de seções
4d. TimeDateStamp
4e. Ponteiro para Tabela de Símbolos e Número de Símbolos
4f. Tamanho do Cabeçalho Opcional
4g. Características

4a. Assinatura PE e 4b. Tipo de Máquina

O cabeçalho do arquivo, uma estrutura do tipo IMAGE_FILE_HEADER, tem como primeiro componente a assinatura PE e, logo a seguir, um dos seguintes elementos:

Nome #define em hexa Significado
IMAGE_FILE_MACHINE_I386 014C processador Intel 80386 ou melhor
014D processador Intel 80486 ou melhor
014E processador Intel Pentium ou melhor
0160 E3000 (MIPS), big endian
IMAGE_FILE_MACHINE_R3000 0162 R3000 (MIPS), little endian
IMAGE_FILE_MACHINE_R4000 0166 R4000 (MIPS), little endian
IMAGE_FILE_MACHINE_R10000 0168 R10000 (MIPS), little endian
IMAGE_FILE_MACHINE_ALPHA 0184 DEC Alpha AXP
IMAGE_FILE_MACHINE_POWERPC 01F0 IBM Power PC, little endian

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
...
0000 00C0 50 45 00 00 4C 01

No nosso exemplo, ficou determinado que a assinatura do PE se encontra no offset 00C0, ocupando 4 bytes. Os dois bytes seguintes, nas posições 00C4 e 00C5, indicam que o executável foi previsto para rodar num processador Intel 80386 ou melhor: IMAGE_FILE_MACHINE_I386 de valor 014C (não esqueça de inverter os bytes, conforme explicado anteriormente).

4c. Número de seções

O terceiro componente do cabeçalho do arquivo é um valor de 16 bits que indica o número de seções após o cabeçalho. As seções serão discutidas em detalhe adiante. No nosso exemplo, são 4 (04 00 invertido = 00 04).

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
...
0000 00C0 50 45 00 00 4C 01 04 00

4d. Carimbo de Data e Hora
Logo após o número de seções encontra-se um valor de 32 bits, o "TimeDateStamp" (carimbo de data e hora), referente ao momento da criação do arquivo. Pode-se distinguir as diversas versões de um mesmo arquivo através deste valor, mesmo que o valor "oficial" da versão não tenha sido alterado. O formato deste carimbo não está documentado, exceto de que deveria ser único entre as versões do mesmo arquivo. Aparentemente corresponde ao número de segundos decorridos a partir de 1 de Janeiro de 1970 00:00:00, em UTC - o formato utilizado pela maioria dos compiladores C para time_t. Este carimbo é utilizado para a união de diretórios de importação, os quais serão abordados adiante.

OBS: alguns linkers costumam atribuir valores absurdos ao carimbo, fora dos padrões time_t descritos acima.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
...
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C

Invertendo os bytes, o TimeDateStamp do nosso executável exemplo mostra um valor hexadecimal de 3C55 77A3 que corresponde a 1.012.234.147 decimal. Sabendo que este valor corresponde ao número de segundos decorridos a partir de 01.01.70 e que cada dia possui 86 400 segundos, podemos calcular que o binário em questão foi criado 11.716 dias (um pouquinho menos) após o início de 1970. Isto corresponde a mais ou menos 32 anos ... UAU ! O executável foi criado no ano de 2002 (1970 + 32). Realmente, se você observar a data do arquivo exemplo (o tutNB03.exe), verá que o executável foi criado em 28 de Janeiro de 2002 às 14:12 horas.

4e. PointerToSymbolTable e NumberOfSymbols
Os componentes "PointerToSymbolTable" (ponteiro para a tabela de símbolos) e "NumberOfSymbols" (número de símbolos), ambos de 32 bits, são utilizados para fins de debug. Não sei como decifrá-los e, geralmente, estão zerados.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
...
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00

4f. SizeOfOptionalHeader
O "SizeOfOptionalHeader" (tamanho do cabeçalho opcional), de 16 bits, é simplesmente o tamanho do IMAGE_OPTIONAL_HEADER. Pode ser utilizado para verificar se a estrutura do arquivo PE está correta. No nosso exemplo, o tamanho do cabeçalho opcional é 224 (E000 invertido = 00E0 hexa = 224 decimal).

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
...
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00

4g. Características
"Characteristics" (características) tem 16 bits e consiste num conjunto de flags, a maioria válida apenas para arquivos objeto e bibliotecas:

Bit Nome Setado (valor 1)
0 IMAGE_FILE_RELOCS_STRIPPED Se não houver informações de remanejamento no arquivo. Isto se refere a informações de remanejamento por seção nas próprias seções. Não é utilizado em executáveis, os quais possuem informações de remanejamento no diretório "base relocation" descrito abaixo.

1 IMAGE_FILE_EXECUTABLE_IMAGE Se o arquivo for um executável, isto é, não é nem um arquivo objeto nem uma biblioteca. Esta flag também pode estar setada se a tentativa do linker em criar um executável tenha falhado por algum motivo - a imagem é mantida para facilitar uma linkagem incremental numa próxima tentativa.

2 IMAGE_FILE_LINE_NUMS_STRIPPED Se a informação do número de linha tiver sido eliminada. Não é usado em executáveis.

3 IMAGE_FILE_LOCAL_SYMS_STRIPPED Se não houver informação sobre símbolos locais. Não é usado em executáveis.
4 IMAGE_FILE_AGGRESIVE_WS_TRIM Se o sistema operacional deve cortar agressivamente o conjunto de trabalho do processo em andamento (a quantidade de RAM que o processo utiliza) através de um 'paging out'. Este bit deve ser setado apenas em aplicativos tipo demon que, na maior parte do tempo, ficam em estado de espera.

7 IMAGE_FILE_BYTES_REVERSED_LO Assim como seu par, o bit 15, está setado se o endian do arquivo não corresponder ao esperado pela máquina, de modo que precisa haver uma troca de bytes antes que uma leitura seja efetuada. Esta flag não é confiável para arquivos executáveis - o sistema operacional conta com os bytes devidamente ordenados nos executáveis.

8 IMAGE_FILE_32BIT_MACHINE Se for para rodar numa máquina de 32 bits.

9 IMAGE_FILE_DEBUG_STRIPPED Se não houver informação de debug no arquivo. Não é utilizado para executáveis. De acordo com outras informações ([6]), este bit é denominado de "fixo" e é setado se a imagem só puder rodar se estiver mapeada no endereço preferido (ou seja, não permite remanejamento).

10 IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP Se o aplicativo não puder rodar a partir de um meio removível, como um disquete ou CD-ROM. Neste caso, o sistema operacional é informado para copiar o arquivo para um arquivo temporário ("swapfile") e executá-lo a partir da cópia.

11 IMAGE_FILE_NET_RUN_FROM_SWAP Se o aplicativo não puder ser executado em rede. Neste caso, o sistema operacional é informado para copiar o arquivo para um arquivo temporário local ("swapfile") e executá-lo a partir da cópia.

12 IMAGE_FILE_SYSTEM Se o arquivo for um arquivo de sistema, por exemplo, um driver. Não é utilizado em executáveis. Também não é usado em todos os drivers NT.

13 IMAGE_FILE_DLL Se o arquivo for uma DLL.

14 IMAGE_FILE_UP_SYSTEM_ONLY Se o arquivo não tiver sido projetado para rodar em sistemas multiprocessados, ou seja, travará porque depende, de algum modo, de um único processador.

15 IMAGE_FILE_BYTES_REVERSED_HI Veja Bit 7.

Agora a coisa complicou um pouquinho. Precisamos "abrir" o hexadecimal de 16 bits destacado em branco para sua notação binária e analisar cada um dos bits:

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
...
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01

Transformando os valores hexadecimais em binários, temos o que precisamos. Observe que tanto os bytes quanto a numeração dos bits, para variar, precisa ser considerada no sentido inverso.

Hexa 01 0F
Binário 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1
Bits 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Agora, de posse dos bits, podemos analisar a informação transmitida por cada um deles:

bit binário característica
0 1 Há informações de remanejamento
1 1 É um arquivo executável
2 1 A numeração de linhas foi eliminada
3 1 Há informações sobre símbolos locais
4 0
5 0
6 0
7 0 O endian corresponde
8 1 É para rodar numa máquina de 32 bits
9 0
10 0
11 0
12 0
13 0
14 0
15 0 O endian corresponde

Algumas considerações sobre RVA
RVA é o endereço virtual relativo. O formato PE faz uso intensivo dos assim chamados RVAs. Um RVA ("relative virtual address") é utilizado para definir um endereço de MEMÓRIA caso se desconheça o endereço base ("base address"). É o valor que, adicionado ao endereço base, fornece o endereço linear.

O endereço base é o endereço a partir do qual a imagem PE é carregada na memória e pode mudar de uma execução para outra. Por exemplo: rodando duas instâncias de um mesmo executável, cada um deles terá um endereço base diferente, ou seja, foram mapeados para localizações diferentes na memória. Outro exemplo: imagine um executável mapeado para o endereço 0x400000 e que a execução se inicie no RVA 0x1560. O endereço de memória que contém o início efetivo será 0x401560. Se o executável tivesse sido mapeado a partir de 0x100000, o início de execução estaria em 0x101560.

As coisas começam a ficar um pouco mais complexas porque algumas partes do arquivo PE (as seções) não estão necessariamente alinhadas da mesma forma que a imagem mapeada. Por exemplo, as seções do arquivo em disco geralmente estão alinhadas em limites de 512 bytes enquanto que a imagem mapeada na memória provavelmente esteja alinhada em limites de 4096 bytes. Veja 'SectionAlignment' e 'FileAlignment' adiante.

Deste modo, para localizar um bloco de informações num arquivo PE para um RVA específico, há a necessidade de se calcular os deslocamentos (offsets) como se o arquivo estivesse mapeado porém, saltar de acordo com os deslocamentos do arquivo. Como exemplo, suponha que você saiba que a execução se inicia no RVA 0x1560 e pretende desassemblar o código a partir deste ponto. Para achar o endereço no arquivo (disco), você precisará descobrir que as seções na RAM estão alinhadas em 4096 bytes, que a seção ".code" começa no RVA 0x1000 da RAM e tem o comprimento de 16384 bytes. Só então você pode determinar que o RVA 0x1560 está deslocado (offset) 0x560 nesta seção. Agora, verificando também que as seções do arquivo em disco estão alinhadas no limite de 512 bytes e que a seção

".code" começa no deslocamento 0x800, com alguns cálculos, torna-se possível localizar o início da execução em disco: 0x800 + 0x560 = 0xD60.

Desassemblando, você encontra uma variável no endereço linear 0x1051D0. O endereço linear será remanejado após o mapeamento do executável e, supostamente, o endereço preferencial será usado. Você consegue determinar que o endereço preferencial é 0x100000, portanto, o RVA é 0x51D0. Isto ocorre na seção de dados que começa no RVA 0x5000 e tem 2048 bytes de comprimento. Ela tem seu início no deslocamento 0x4800 do arquivo em disco. Portanto, a variável pode ser encontrada no deslocamento 0x4800 + 0x51D0 - 0x5000 = 0x49D0.

5. Cabeçalho Opcional
Imediatamente após o cabeçalho do arquivo vem o cabeçalho opcional que, apesar do nome, está sempre presente. Este cabeçalho contém informações de como o arquivo PE deve ser tratado.

Os componentes do cabeçalho opcional são os seguintes:

5a. Magic
5b. MajorLinkerVersion e MinorLinkerVersion
5c. Tamanho do Código, Segmento de Dados e Segmento BSS
5d. AddressOfEntryPoint - O Ponto de Entrada do Código do Executável
5e. Base do Código e Base dos Dados
5f. Base da Imagem
5g. Alinhamento
5h. Versão do Sistema Operacional
5i. Versão do Binário
5j. Versão do Subsistema
5k. Versão do Win32
5l. Tamanho da Imagem
5m. Tamanho dos Cabeçalhos
5n. CheckSum
5o. Subsistema NT
5p. Características de DLL
5q. Tamanho da Reserva de Pilha (StackReserve)
5r. Loader Flags
5s. Número e Tamanho dos RVA

5a. Magic
O primeiro word de 16 bits do cabeçalho opcional é o 'Magic'. Em todos os arquivos PE que analisei até hoje, o valor encontrado sempre foi 010B.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01

5b. MajorLinkerVersion e MinorLinkerVersion
O próximo componente do cabeçalho opcional, composto de 2 bytes, reflete as versões Maior e Menor do linker utilizado. Estes valores, novamente, não são confiáveis e nem sempre refletem apropriadamente a versão do linker. Muitos linkers nem mesmo utilizam estes campos. Aliás, se não se tem a mínima idéia de qual linker tenha sido utilizado, qual é a vantagem de conhecer a versão?

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C

5c. Tamanho do Código, Segmento de Dados e Segmento BSS

Os 3 valores de 32 bits seguintes referem-se ao Tamanho do Código Executável ('SizeOfCode'), ao Tamanho dos Dados Inicializados ('SizeOfInitializedCode') e ao Tamanho dos Dados não Inicializados ('SizeOfUninitializedCode'). O tamanho dos dados inicializados também é conhecido como Segmento de Dados (Data Segment) e o tamanho dos dados não inicializados é conhecido como Segmento BSS (BSS Segment). Estes dados, uma vez mais, não são confiáveis! Por exemplo: o segmento de dados pode estar dividido em vários segmentos por ação do compilador ou do linker. O melhor que se tem a fazer é inspecionar as seções para ter uma idéia mais aproximada dos tamanhos.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00

5d. Ponto de Entrada do Código do Executável
O próximo valor de 32 bits é um RVA. Se você tiver dúvidas sobre RVAs, NÃO CONTINUE. Volte para o texto anterior e leia com atenção as Considerações sobre RVAs. A partir deste ponto, se você não estiver familiarizado com as ditas cujas... vai perder o passo!

Este RVA é o offset para o Ponto de Entrada do Código ('AddressOfEntryPoint'), ou seja, é onde a execução do nosso binário, MAPEADO NA MEMÓRIA, realmente começa.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00

Invertendo os bytes, obtemos o RVA do Endereço do Ponto de Entrada: 00 10 00 00 -> 00 00 10 00. Este valor (1000) é o que será adicionado ao endereço base quando nosso programa for mapeado na MEMÓRIA. Imagine que, ao ser executado, o executável tenha sido mapeado na memória a partir do endereço 40000. Qual será o ponto de entrada do código? Simples: 40000 + 1000 = 41000. Entendeu agora porque o valor deste campo é um RVA?

5e. Base do Código e Base dos Dados
Logo após o importantíssimo Ponto de Entrada encontram-se dois valores de 32 bits, o 'BaseOfCode' (base do código) e o 'BaseOfData' (base dos dados), ambos também RVAs. Infelizmente os dois também perdem importância (como tantos outros campos) porque a informação obtida através da análise das seções é muito mais confiável.

Não existe uma base de dados não inicializados porque, por não serem inicializados, não há a necessidade de incluir esta informação na imagem.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00

5f. Base da Imagem
Segue uma entrada de um valor de 32 bits que indica o endereço de mapeamento preferencial, chamado de endereço linear e correspondendo à 'BaseImage'. No momento da execução, se este endereço de memória estiver vago, o binário inteiro (incluindo os cabeçalhos) será transferido para lá. Este é o endereço, sempre um

múltiplo de 64, para onde o binário é remanejado pelo linker. Se o endereço estiver disponível, o carregador (loader) não precisará remanejar o arquivo, o que representa um ganho no tempo de carregamento.

O endereço preferido de mapeamento não pode ser utilizado se outra imagem já tiver sido mapeada para este endereço (uma colisão de endereços, a qual ocorre com alguma frequência quando se carrega várias DLLs que são remanejadas para o default do linker) ou se a memória em questão estiver sendo usada para outros fins (stack, malloc(), dados não inicializados, etc). Nestes casos, a imagem precisa ser transferida para algum outro endereço (veja 'diretório de remanejamento' logo a seguir). Este fato gera consequências posteriores se a imagem pertencer a uma DLL porque, neste caso, as importações casadas ("bound imports") deixam de ser válidas e há a necessidade de efetuar correções nos binários que utilizam estas DLLs - veja também em 'diretório de remanejamento' a seguir.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00

5g. Alinhamento
Os dois valores de 32 bits seguintes são os alinhamentos das seções do arquivo PE na RAM ('SectionAlignment', quando a imagem estiver carregada na memória) e no arquivo em disco ('FileAlignment'). Geralmente ambos valores são 32, ou então FileAlignment (alinhamento de arquivo) é 512 e SectionAlignment (alinhamento de seções) é 4096. As seções serão vistas posteriormente.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00

No nosso exemplo, o alinhamento de seções é 4096 (0010 0000 -> inverso 0000 1000 -> 4096 decimal) e o alinhamento de arquivo é 512 (0002 0000 -> inverso 0000 0200 -> 512 decimal).

5h. Versão do Sistema Operacional
Os dois valores seguintes são de 16 bits e referem-se à versão esperada do sistema operacional ('MajorOperatingSystemVersion' e 'MinorOperatingSystemVersion'). Esta informação da versão é apenas para o sistema operaconal, por exemplo NT ou Win98, ao contrário da versão do sub-sistema, por exemplo Win32. Geralmente esta informação não é fornecida ou está errada. Aparentemente o carregador (loader) não faz uso da mesma, portanto...

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
0000 0100 04 00 00 00

5i. Versão do Binário
Os dois valores de 16 bits seguintes fornecem a versão do binário ('MajorImageVersion' e 'MinorImageVersion'). Muitos linkers não fornecem dados corretos e uma grande parte dos programadores nem se dá ao trabalho de fornece-los. O melhor é se fiar na versão dos recursos (resource), contanto que exista.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
0000 0100 04 00 00 00 00 00 00 00

5j. Versão do Sub-sistema
Os próximos 2 words de 16 bits são para a versão do sub-sistema esperado ('MajorSubsystemVersion' e 'MinorSubsystemVersion'). Esta versão deveria ser Win32 ou POSIX porque os programas de 16 bits ou os do OS/2 obviamente não estão em formato PE.

Esta versão de subsistema deve ser fornecida corretamente porque ela É checada e usada:

Se o aplicativo for um do tipo Win32-GUI, tiver que rodar em NT4 e a versão do sub-sistema não for 4.0, as caixas de diálogo não terão o estilo 3D e alguns outros aspectos terão a aparência do "estilo antigo". Isto porque o aplicativo acaba sendo rodado no NT 3.51, o qual possui o program manager ao invés do explorer, etc, e o NT 4.0 tentará imitar o 3.51 da melhor maneira possível.
Idem para Win98 e WinMe. O aplicativo indica Win98 e o sistema da máquina é WinMe, então o WinMe tenta de tudo para imitar o Win98...
Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
0000 0100 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00

5k. Versão do Win32
Só Deus sabe para é que serve este próximo valor de 32 bits. Está sempre zerado (veja acima).

5l. Tamanho da Imagem
Este valor de 32 bits indica a quantidade de memória necessária para abrigar a imagem, em bytes ('SizeOfImage'). É a soma do comprimento de todos os cabeçalhos e seções, se estiverem alinhados de acordo com o 'SectionAlignement'. Indica para o carregador quantas páginas serão necessárias para carregar completamente a imagem.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
0000 0100 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
0000 0110 00 50 00 00

No nosso exemplo, obedecendo o alinhamento de seções de 4096 (veja acima), são requeridos 20.480 bytes para abrigar a imagem do executável na memória. Basta calcular: 0050 0000 -> inverso 0000 5000 -> 20.480 decimal.

5m. Tamanho dos Cabeçalhos

O próximo valor de 32 bits é o tamanho de todos os cabeçalhos, incluindo os diretórios de dados e os cabeçalhos das seções ('SizeOfHeaders'). Representa o offset do início do arquivo até os dados (raw data) da primeira seção.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
0000 0100 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
0000 0110 00 50 00 00 00 04 00 00 00 00 00 00

No nosso exemplo, o offset do início do arquivo até os dados propriamente ditos é de 0000 0400 que, em decimal, corresponde a 1024 bytes.

5n. CheckSum
Segue-se o valor de 32 bits do 'CheckSum'. O valor do checksum, para as versões atuais do NT, só é checado se a imagem for um driver NT (o driver não carregará se o checksum não estiver correto). Para outros tipos de binários o checksum não precisa ser fornecido e pode ser 0.

O algoritmo para calcular o checksum é propriedade da Microsoft e o pessoal da MS não entrega o ouro. No entanto, diversas ferramentas do Win32 SDK calculam e/ou inserem um checksum válido. Além disso, a função CheckSumMappedFile(), que faz parte da imagehelp.dll, também faz o serviço completo.

A função do checksum é a de evitar que binários "bichados", que vão dar pau de qualquer forma, sejam carregados - e um driver com pau acaba em BSOD, portanto, é melhor nem carregar.

No nosso exemplo, que não é para NT, o valor está zerado (veja acima).

5o. Subsistema NT
O próximo valor de 16 bits, o 'Subsystem', indica em qual subsistema do NT a imagem deve rodar:

Nome Valor Significado
IMAGE_SUBSYSTEM_NATIVE 1 O binário não precisa de um subsistema. É usado para drivers.
IMAGE_SUBSYSTEM_WINDOWS_GUI 2 A imagem é um binário Win32 gráfico. Ainda pode abrir um console com AllocConsole(), porém não abre automaticamente no startup.
IMAGE_SUBSYSTEM_WINDOWS_CUI 3 O binário é um Win32 de console. Receberá um console no startup (default) ou herda um console (parent's console).
IMAGE_SUBSYSTEM_OS2_CUI 5 O binário é um OS/2 de console. Os binários OS/2 estarão em formato OS/2, portanto, este valor raramente será encontrado num arquivo PE.
IMAGE_SUBSYSTEM_POSIX_CUI 7 O binário usa um subsistema de console POSIX.

Binários do Windows 9x sempre usarão o subsistema Win32, portanto, os únicos valores aceitáveis para estes binários são 2 e 3. Desconheço se binários "nativos" do windows 9x são aceitos.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
0000 0100 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
0000 0110 00 50 00 00 00 04 00 00 00 00 00 00 02 00 00 00

5p. Características de DLL
Este próximo valor de 16 bits indica quando o ponto de entrada deve ser chamado, SE a imagem for de uma DLL. No nosso exemplo, este valor está logicamente zerado (veja acima). Este é mais um campo que parece não ter uso: aparentemente, as DLL recebem notificações de tudo e prescindem deste campo. Novamente os bits são usados para guardar informações:

Bit Setado (valor 1)
0 Notifica uma anexação de processo (isto é, DLL load)
1 Notifica um desligamento de thread (isto é, termina um thread ativo)
2 Notifica uma anexação de thread (isto é, cria um thread novo)
3 Notifica um desligamento de processo (isto é, DLL unload)

5q. Tamanho da Reserva de Pilha (StackReserve)
Os próximos 4 valores de 32 bits são o tamanho da reserva de pilha ('SizeOfStackReserve'), o tamanho do commit inicial da pilha ('SizeOfStackCommit'), o tamanho da reserva de heap ('SizeOfHeapReserve') e o tamanho do commit do heap ('SizeOfHeapCommit').

As quantidades 'reservadas' são espaços endereçados (não RAM real) que são reservadas para um propósito específico. No início do programa, a quantidade "committada" é alocada na RAM. O valor "committado" é também o quanto a pilha ou o heap "committados" irão crescer caso for necessário. Alguns autores alegam que a pilha cresce em páginas, independentemente do valor do 'SizeOfStackCommit'.

Vamos a um exemplo: se o programa possui uma reserva de heap de 1 MB e um commit de heap de 64 Kb, o heap começa com 64 Kb e pode ser expandido até 1 MB. O heap irá crescer de 64 em 64 Kb.

O 'heap' neste contexto é o heap primário (default). Um processo pode criar mais heaps se houver necessidade.

Como as DLLs não possuem pilha ou heap próprios, estes valores são ignorados nas suas imagens.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
0000 00D0 00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00
0000 00E0 00 0E 00 00 00 00 00 00 00 10 00 00 00 10 00 00
0000 00F0 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
0000 0100 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
0000 0110 00 50 00 00 00 04 00 00 00 00 00 00 02 00 00 00
0000 0120 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00
0000 0130 00 00 00 00 10 00 00 00

5r. Loader Flags
Os próximos 32 bits são das 'LoaderFlags' (flags do carregador) para os quais não há uma descrição adequada. No nosso exemplo, de qualquer maneira, todas as flags estão zeradas (veja acima).

5s. Número e Tamanho dos RVA
O número e tamanho dos RVA ('NumberOfRvaAndSizes') se encontram nos 32 bits seguintes e revelam o número de entradas válidas nos diretórios que vêm logo a seguir. Este número parece não ser muito confiável. No nosso exemplo, veja também acima, são 16 (1000 0000 -> invertendo 0000 0010 -> 16 decimal).

6. Diretórios de Dados
Imediatamente após o cabeçalho do opcional vêm os diretórios de dados. É um array de IMAGE_NUMBEROF_DIRECTORY_ENTRIES (16) IMAGE_DATA_DIRECTORY. Cada um destes diretórios descrevem a localização (um RVA de 32 bits denominado 'VirtualAddress') e o tamanho (também de 32 bits, chamado 'Size') de uma peça de informação que está localizada em uma das seções que seguem as entradas de diretório.

Por exemplo, o diretório de segurança (security directory) se encontra no RVA e tem o tamanho indicados no índice 4. Os índices definidos para os diretórios são:

Nome Índice Diretório
IMAGE_DIRECTORY_ENTRY_EXPORT 0 É o diretório de funções exportadas, usado principalmente para DLLs.
IMAGE_DIRECTORY_ENTRY_IMPORT 1 Diretório de símbolos importados.
IMAGE_DIRECTORY_ENTRY_RESOURCE 2 Diretório de recursos (resources).
IMAGE_DIRECTORY_ENTRY_EXCEPTION 3 Diretório de exceções - estrutura e aplicação ignorada.
IMAGE_DIRECTORY_ENTRY_SECURITY 4 Diretório de segurança - estrutura e aplicação ignorada.
IMAGE_DIRECTORY_ENTRY_BASERELOC 5 Tabela da base de remanejamento.
IMAGE_DIRECTORY_ENTRY_DEBUG 6 Diretório de debug, cujo conteúdo depende do compilador. De qualquer forma, muitos compiladores colocam as informações de debug na seção de código e não criam uma seção separada.
IMAGE_DIRECTORY_ENTRY_COPYRIGHT 7 String de descrição com alguns comentários de copyright ou coisa parecida.
IMAGE_DIRECTORY_ENTRY_GLOBALPTR 8 Valor de Máquina (MIPS GP) - estrutura e aplicação ignorada.
IMAGE_DIRECTORY_ENTRY_TLS 9 Diretório de armazenamento local de thread - estrutura desconhecida. Contém variáveis que são declaradas "__declspec(thread)", isto é, variáveis globais per-thread.
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG 10 Diretório de configuração de carregamento - estrutura e aplicação ignorada.
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT 11 Diretório de importação casada (bound import).
IMAGE_DIRECTORY_ENTRY_IAT 12 Tabela de endereços de importação (IAT - Import Address Table).

Como exemplo, se encontrarmos 2 words longos no índice 7, cujos valores sejam 12000 e 33, e o endereço de carregamento for 10000, sabemos que os dados de copyright estão no endereço 10000 + 12000 (independentemente da seção em que possam estar) e que o comentário de copyright tem 33 bytes de comprimento.

Se algum diretório de um tipo em particular não for usado no binário, o tamanho (Size) e o endereço virtual (VirtualAddress) são zero.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 00C0 50 45 00 00 4C 01 03 00 A3 77 55 3C 00 00 00 00
...
0000 0130 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00
0000 0140 40 20 00 00 3C 00 00 00 00 40 00 00 60 09 00 00
0000 0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000 0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000 0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000 0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000 0190 00 00 00 00 00 00 00 00

No nosso exemplo, no array de 12 elementos destacados em amarelo e laranja, apenas os diretórios de índice 1 e 2 possuem referências.

O diretório de índice 1 refere-se aos símbolos importados: seu RVA é 0000 2040 e seu tamanho é 0000 003C. Portanto, os dados referentes aos símbolos importados estarão deslocados em 8256 bytes (2040h = 8256d) e ocupam 60 bytes (3Ch = 60d).

O diretório de índice 2 refere-se aos recursos: seu RVA é 0000 4000 e seu tamanho é 0000 0960. Portanto, os dados referentes aos recursos estarão deslocados 16384 bytes (4000h = 16384d) e ocupam 2400 bytes (0960h = 2400d).

7. Cabeçalhos das Seções

As seções são compostas por duas partes principais: primeiro, a descrição da seção (do tipo IMAGE_SECTION_HEADER) e depois os dados propriamente ditos. Desta forma, logo após os diretórios de dados, encontramos um array de cabeçalhos de seções do tipo número de seções ('NumberOfSections'), ordenado pelos RVAs das seções.

Um cabeçalho de seção contém:

7a. Um array de Nomes das Seções
7b. Endereço Físico e do Tamanho Virtual
7c. Endereço Virtual
7d. Tamanho dos Dados
7e. Ponteiro para os Dados
7f. Ponteiro para Remanejamento
7g. Características

7a. Nomes das Seções

O primeiro componente é um array de IMAGE_SIZEOF_SHORT_NAME de 8 bytes para guardar o nome (ASCII) da seção. Se todos os 8 bytes forem usados não existe um terminador 0 (zero) para a string! O nome é tipicamente algo como ".data" ou ".text" ou mesmo ".bss". Não há a necessidade do nome ser precedido por um ponto '.' e não existem nome predefinidos (qualquer nome é aceito).

Os nomes também não têm qualquer relação com o conteúdo da seção. Uma seção de nome ".code" pode ou não conter código executável: pode perfeitamente conter a tabela de endereços de importação, pode conter código executável E a tabela de endereços de importação e até os dados inicializados.

Para achar informações nas seções, é preciso buscá-las nos diretórios de dados do cabeçalho opcional. Não se deixe influenciar pelo nome da seção e não assuma que os dados da seção estejam logo no início da mesma.

```
Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
...
0000 01B0                  2E 74 65 78 74 00 00 00 .text...
0000 01C0 94 01 00 00 00 10 00 00 00 02 00 00 00 04 00 00 ”...............
0000 01D0 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 ............ ...
0000 01E0 2E 72 64 61 74 61 00 00 C2 01 00 00 00 20 00 00 .rdata..Å.... ..
0000 01F0 00 02 00 00 00 06 00 00 00 00 00 00 00 00 00 00 ................
0000 0200 00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00 ....@..@.data...
0000 0210 24 00 00 00 00 30 00 00 00 02 00 00 00 08 00 00 $....0..........
0000 0220 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0 ............@..À
0000 0230 2E 72 73 72 63 00 00 00 60 09 00 00 00 40 00 00 .rsrc...`....@..
0000 0240 00 0A 00 00 00 0A 00 00 00 00 00 00 00 00 00 00 ................
0000 0250 00 00 00 00 40 00 00 C0                  ....@..À........
```

7b. Endereço Físico e do Tamanho Virtual

O próximo membro da IMAGE_SECTION_HEADER é a união de 32 bits do Endereço Físico ('PhysicalAddress') e do Tamanho Virtual ('VirtualSize'). Num arquivo objeto,

este é o endereço para o qual o conteúdo é remanejado; num executável, é o tamanho do conteúdo. Mais uma vez, este campo não é utilizado! Há linkadores que o preenchem com o tamanho, outros com o endereço e outros ainda que o preenchem com 0. Apesar disso, os executáveis não apresentam problemas.

7c. Endereço Virtual
Logo a seguir vem o Endereço Virtual ('VirtualAddress'), um valor de 32 bits que contém o RVA para os dados da seção quando esta estiver mapeada na RAM.

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
...
0000 01B0 2E 74 65 78 74 00 00 00 .text...
0000 01C0 94 01 00 00 00 10 00 00 00 02 00 00 00 04 00 00 ”.....
0000 01D0 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 ....

No nosso exemplo, o valor encontrado é 0000 1000, ou seja, o RVA será de 4096 bytes (destacado em amarelo).

7d. Tamanho dos Dados
Após o endereço virtual vêm 32 bits para os Tamanho dos Dados ('SizeOfRawData'), que nada mais é do que o tamanho dos dados da seção arredondado para cima para o próximo múltiplo de 'FileAlignment' (alinhamento de arquivo).

No nosso exemplo, o valor encontrado é 0000 0200, ou seja, o tamanho dos dados da seção é de 512 bytes (veja acima, destacado em laranja).

7e. Ponteiro para os Dados
Segue-se o Ponteiro para os Dados ('PointerToRawData'), também de 32 bits. Este ponteiro é extremamente útil porque é o offset do início do arquivo em disco até os dados da seção. Se for 0, os dados da seção não estão contidos no arquivo e serão carregados arbitrariamente no momento da carga do programa.

No exemplo, encontramos 0000 0400, destacado acima em vermelho. Observe o endereço 0400 deste arquivo armazenado em disco:

Offset 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
0000 0400 6A 00 E8 87 01 00 00 A3 1C 30 40 00 E8 77 01 00 j.è‡...£.0@.èw..

Após uma longa sucessão de zeros em endereços anteriores, em 0400 inicia-se a sucessão de dados da seção .text.

7f. Ponteiro para Remanejamento
A seguir vem o Ponteiro para Remanejamento ('PointerToRelocations') de 32 bits e o Ponteiro para Números de Linha ('PointerToLinenumbers'), também de 32 bits, o Número de Remanejamentos ('NumberOfRelocations') de 16 bits e o Número de Números de Linha ('NumberOfLinenumbers'), também de 16 bits. Todas estas informações somente são utilizadas para arquivos objeto. Os executáveis não possuem um diretório de remanejamento base especial e a informação de número de linha, se é que está presente, geralmente está localizada num segmento especial para debugging ou em qualquer outro lugar.

No exemplo, todas estas posições estão preenchidas com zeros (observe a linha 01D0 acima).

7g. Características
O último membro dos cabeçalhos das seções é o valor de 32 bits com as Características. São um punhado de flags que descrevem como a memória das seções deve ser tratada:

Nome Bit Setado (valor 1)
IMAGE_SCN_CNT_CODE 5 A seção contém código executável.
IMAGE_SCN_CNT_INITIALIZED_DATA 6 A seção contém dados que recebem um valor definido antes que a execução se inicie. Em outras palavras: os dados da seção são significativos.
IMAGE_SCN_CNT_UNINITIALIZED_DATA 7 A seção contém dados não inicializados que terão todos os bytes zerados antes que a execução se inicie. Este, geralmente, é o BSS.
IMAGE_SCN_LNK_INFO 9 A seção não contém dados de imagem e sim comentários, descrições ou outra documentação qualquer. Esta informação faz parte de arquivos objeto e pode ser a informação para o linker, como, por exemplo, as bibliotecas necessárias.
IMAGE_SCN_LNK_REMOVE 11 Os dados fazem parte de uma seção de um arquivo objeto que deve ser deixado de fora quando o arquivo executável for linkado. Com frequência este bit está combinado com o bit 9.
IMAGE_SCN_LNK_COMDAT 12 A seção contém o "common block data", que são funções de pacotes.
IMAGE_SCN_MEM_FARDATA 15 Existe 'far data' - significado incerto.
IMAGE_SCN_MEM_PURGEABLE 17 Os dados da seção podem sofrer um 'purge' - não é o mesmo que descartáveis, pois há um bit para este fim (veja abaixo). O mesmo bit, aparentemente, é usado para indicar informações de 16 bits - significado incerto.
IMAGE_SCN_MEM_LOCKED 18 Significado incerto - a seção não pode ser deslocada na memória? - não há informação de remanejamento?
IMAGE_SCN_MEM_PRELOAD 19 Significado incerto - a seção deve ser "paginada" antes do início da execução?
20 a 23 Especificam um alinhamento. Não há informações disponíveis. Existe um #define IMAGE_SCN_ALIGN_16BYTES e parecidos...
IMAGE_SCN_LNK_NRELOC_OVFL 24 A seção contém alguns remanejamentos extendidos - significado incerto.
IMAGE_SCN_MEM_DISCARDABLE 25 Os dados da seção não são necessários após o início do processo. É o caso, por exemplo, das informações de remanejamento. São encontradas também para rotinas de startup de drivers e serviços que são executados apenas uma vez e para diretórios de importação.
IMAGE_SCN_MEM_NOT_CACHED 26 Os dados da seção não devem ir para cache. (Será que significa desligar o cache de segundo nível?)
IMAGE_SCN_MEM_NOT_PAGED 27 Os dados da seção não devem sair da página. Isto é interessante para drivers.
IMAGE_SCN_MEM_SHARED 28 Os dados da seção são compartilhados entre todas as instâncias das imagens que estiverem sendo executadas. Se forem os dados inicializados de uma DLL, por exemplo, todos os conteúdos das mesmas variáveis serão os mesmos em todas as instâncias da DLL. Note que apenas a seção da primeira instância é inicializada. Seções contendo código são sempre compartilhadas copy-on-write, isto é, o compartilhamento não funciona se houver a necessidade de fazer remanejamentos.
IMAGE_SCN_MEM_EXECUTE 29 O processo recebe acesso de 'execução' na memória da seção.
IMAGE_SCN_MEM_READ 30 O processo recebe acesso de 'leitura' na memória da seção.
IMAGE_SCN_MEM_WRITE 31 O processo recebe acesso de 'escrita' na memória da seção.

Analisando os últimos 32 bits do cabeçalho da seção .text, destacados em azul claro logo acima, obtemos o hexadecimal 6000 0020. Abrindo-o em bits para poder analisar este valor, obtemos o seguinte:

Hexa 60 00 00 20
Binário 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
Bits 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Relevante na seção .text são os seguintes bits:

Bit 5 = 0 A seção NÃO contém código executável.
Bit 7 = 1 A seção contém dados não inicializados.
Bit 29 = 0 O processo NÃO recebe acesso de 'execução'.
Bit 30 = 1 O processo recebe acesso de 'leitura'.
Bit 31 = 1 O processo recebe acesso de 'escrita'.

⇨ PROTEÇÃO EFICIENTE

Nunca subestime a capacidade de um cracker reverter um programa para que ele não subestime sua capacidade de proteger seu código !
Se a proteção 100% de um software parece um sonho impossível, prepare-se para uma proteção 99%. A maioria dos mortais são apenas usuários comuns, sem condições de crackear um programa. O máximo que podem fazer é procurar algum crack nos assim chamados sites de warez. Quantos realmente podem se considerar crackers ? 1% ou 2% dos usuários ? Se desses 1% a 2% você conseguir afastar 99%, faça as contas ! Tá pra lá de bão, né não ? ;-)

Se não é para subestimar a capacidade dos crackers... então vamos ouvir o que eles têm a dizer !

Os famosos 14 mandamentos de Mark
1 Nunca nomeie arquivos ou procedimentos com nomes que façam sentido, do tipo IsValidSerialNum ou CodRegOK (dããããã!!!). Se você usar funções para checagens, pelo menos coloque um trecho de código vital para o programa dentro de funções deste tipo. Se o cracker desabilitar a função, o programa gerará resultados incorretos.

2 Não avise o usuário assim que ocorrer uma violação. Faça com que o programa espere, talvez um dia ou dois (crackers odeiam estas surpresas).

3 Use checksums em DLLs e EXEs. Faça com que se chequem entre si. Não é perfeito mas dificulta muito o crack.

4 Introduza uma pausa de 1 a 2 segundos após a entrada de uma senha para que um cracking usando força bruta seja impraticável. Simples de ser feito, raramente usado.

5 Use a correção automática no seu software. Você sabe, como a correção de erros que os modems e os HDs usam. A tecnologia já existe há anos e ninguém a usa nos próprios softwares ? O melhor dessa história é que se o cracker usou um decompilador, ele pode estar olhando para uma listagem que perdeu a validade.

6 Faça um patch no seu próprio software. Mude seu código para que cada vez chame rotinas de validação diferentes. Vença-nos no nosso próprio jogo.

7 Guarde números seriais em locais improváveis, por exemplo como uma propriedade de um campo de uma base de dados.

8 Guarde números seriais em vários locais diferentes.

9 Não dependa da data do sistema. Obtenha a data de diversos arquivos, como SYSTEM.DAT, SYSTEM.DA0 e BOOTLOG.TXT e compare-as com a data do sistema. Exija que a data seja maior que a da última execução.

10 Não utilize strings literais que informem o usuário que tempo de uso expirou. Estas são as primeiras coisas procuradas. Gere strings dinâmicas ou use encriptação.

11 Inunde o cracker com falsas chamadas e strings "hard coded". Armadilhas são divertidas.

12 Não use uma função de validação. Cada vez que for necessário validar, escreva o código de validação dentro do processo atual. Isto apenas vai dar mais trabalho ao cracker.

13 Se usar chaves ou senhas "hard coded", faça com que tenham a aparência de código de programa ou de chamada de função (por exemplo, "73AF" ou "GetWindowText"). Isto funciona muito bem e causa confusão em alguns decompiladores.

14 E, finalmente, nunca revele seus melhores segredos de proteção :-)