

# **Fundamentos de Engenharia Reversa**

# Apresentação

## O que é este livro

Este livro é mais um projeto da [Mente Binária](#). Ele consiste o resultado de um extenso trabalho de organização de conteúdo e escrita após centenas de horas de aulas ministradas sobre engenharia reversa de software para iniciantes. A intenção é documentar os conhecimentos fundamentais necessários para a formação dos novos engenheiros reversos.

---

## A quem se destina este livro

É esperado que o leitor seja iniciante ou mesmo já iniciado em engenharia reversa, mas não em computação. Lógica de programação é essencial para o máximo aproveitamento deste conteúdo. Conhecer o básico da organização de computadores também é essencial para o aproveitamento do livro.

Tendo estes conhecimentos, qualquer pessoa que almeje se tornar um engenheiro reverso para atuar na área de segurança, construção e desconstrução de software ou simplesmente reverter programas como um *hobby*. Também pode ser utilizado por quem quer entender um pouco melhor como um computador moderno funciona.

---

## Apoie este livro

**Este livro está em constante desenvolvimento** e é colocado disponível para você e outros milhões de falantes da língua portuguesa mediante nenhum custo. No entanto, para continuar, precisamos de doações. Se este conteúdo te ajudar, por favor considere tornar-se um apoiador clicando [aqui](#). Isso vai ajudar a garantir que mantenhamos esse livro atualizado e cada vez mais abrangente.

---

## Erros, comentários e sugestões

Se encontrou algum erro, quer fazer um comentário, sugestão ou pergunta, por favor utilize o [tópico criado especialmente para este livro](#) no fórum do portal Mente Binária.

---

## Colaboradores

### Conteúdo

Fernando Mercês ([@mer0x36](#))

### Revisão

Carlos Cabral ([@kbralx](#))

# Antes de começar

Todo escritor quer que sua mensagem seja lida e compreendida, isso não é diferente no meu caso. Então, estabeleci umas regras em meu processo de escrita para facilitar o seu processo de compreensão da disciplina de Engenharia Reversa.

---

## Terminologia

Utilizo do *itálico* para neologismos, palavras em Inglês ou em outro idioma, como *crypter* (encriptador). Em geral, prefiro não utilizar termos "aportuguesados" como baite (para *byte*) ou linkeditor (para *linker*). Acho que isso confunde o leitor e por isso mantenho-os em seus originais em Inglês, mas em itálico.

Já **negrito** utilizo para dar destaque ou para me referir à um comando.

Algumas vezes utilizo o termo **GNU/Linux** ao invés de somente Linux. O projeto GNU é o principal projeto da FSF (*Free Software Foundation*), que criou o conceito de software livre. Suas ferramentas são parte essencial de qualquer sistema operacional baseado no kernel Linux e por isso faz bem lembrá-la de vez em quando.

Após a introdução, **engenharia reversa** (ou simplesmente **ER**) passa a ser utilizado como forma curta de engenharia reversa de *software*.

Nas **operações bit-a-bit** (*bitwise*), utilizo os símbolos da programação para representar as operações E, OU, OU EXCLUSIVO, etc. No texto, adoto seus mnemônicos em Inglês: AND, OR e XOR.

---

## Arquitetura de software

Cada frase deste livro, a não ser que expressado diferente, considera a arquitetura Intel x86 (IA-32), visto que esta é documentada o suficiente para começar o estudo de engenharia

reversa e moderna o suficiente para criar exemplos funcionais de código e analisar programas atuais.

---

## Exercícios

Este livro é recheado de trechos de código. É recomendável que o leitor pratique escrevendo os códigos no ambiente específico cada vez que encontrar blocos como os abaixo.

Exemplos de código em Python como a seguir devem ser digitados no ambiente do Python da sua máquina virtual (ou real) Linux:

Python

```
>>> print 'Execute isto no console do Python!'
```

Vários exemplos são no *shell* do Linux, que é o Bash por padrão:

Bash

```
$ echo 'Este vai no Bash'
```

O leitor também encontrará códigos em linguagem C como este:

exemplo.c

```
1  #include <stdio.h>
2
3  int main(void) {
4      printf("Compilar com o gcc e executar!\n");
5
6      return 0;
7  }
```

Neste caso, exceto quando especificado diferente, é preciso salvar o código em C num arquivo de texto e assim compilar e executar no ambiente GNU/Linux, assim:

```
1  $ gcc -o exemplo exemplo.c
2  $ ./exemplo
```

Caso o código em C inclua a *windows.h*, então deve ser compilado em ambiente Windows utilizando de preferência o [Orwell DevC++](#).

Há ainda outros tipos de blocos, mas tenha em mente que é **necessário** para o aprendizado que você os escreva, execute e analise seus resultados. ;-)

---

## Preparação do ambiente

Este é um guia prático. Sendo assim, é recomendável que você seja capaz de reproduzir o que é sugerido neste livro em seu próprio ambiente. Precisaremos de:

- Uma máquina (virtual ou real) com Windows 7 ou 10.
- Uma máquina (virtual ou real) com Linux. Usamos o Debian, mas qualquer outra serve, desde que você saiba instalar pacotes nela.

Na máquina com Windows, os programas necessários são:

- [x64dbg](#)
- [Detect It Easy 2.05](#)
- [wxHexEditor 0.24](#)
- [pev 0.80](#)
- [Orwell Dev-C++](#)
- [Python 3.7.7](#)


# Introdução

## O que é engenharia reversa

Em termos amplos, engenharia reversa é o processo de entender como algo funciona através da análise de sua estrutura e de seu comportamento. É uma arte que permite conhecer como um sistema foi pensado ou desenhado sem ter contato com o projeto original.

Não restrinjam, portanto, a engenharia reversa à tecnologia. Qualquer um que se disponha a analisar algo de forma minuciosa com o objetivo de entender seu funcionamento a partir de seus efeitos ou estrutura está fazendo engenharia reversa. Podemos dizer, por exemplo, que Carl G. Jung (conhecido como o pai da psicologia analítica) foi um grande engenheiro reverso ao introduzir conceitos como o de inconsciente coletivo através da análise do comportamento humano.

No campo militar e em época de guerras é muito comum assegurar que inimigos não tenham acesso às armas avançadas: aviões, tanques e outros dispositivos, pois é importante que adversários não desmontem esses equipamentos, não entendam seu funcionamento e, conseqüentemente, que não criem versões superiores deles ou encontrem falhas que permitam inutilizá-los com mais facilidade. Na prática, é evitar a engenharia reversa.

 Vale a pena assistir ao filme Jogo da Imitação (*Imitation Game*) que conta a história do criptoanalista inglês Alan Turing, conhecido como o pai da ciência de computação teórica, que quebrou a criptografia da máquina nazista Enigma utilizando engenharia reversa.

---

## Engenharia reversa de software

Este livro foca na engenharia reversa de software, ou seja, no processo de entender como uma ou mais partes de um programa funcionam, sem ter acesso a seu código-fonte.



Focaremos inicialmente em programas para a plataforma x86 (de 32-bits), rodando sobre o sistema operacional Windows, da Microsoft, mas vários dos conhecimentos expressos aqui podem ser úteis para engenharia reversa de software em outros sistemas operacionais, como o GNU/Linux e até mesmo em outras plataformas, como ARM.

---

## Por que a engenharia reversa de software é possível

Assim como o *hardware*, o *software* também pode ser desmontado. De fato, existe uma categoria especial de *softwares* com esta função chamados de *disassemblers*, ou desmontadores. Para explicar como isso é possível, primeiro é preciso entender como um programa de computador é criado atualmente. Farei um resumo aqui, mas entenderemos mais a fundo em breve.

A parte do computador que de fato executa os programas é o chamado processador. Nos computadores de mesa (*desktops*) e *laptops* atuais, normalmente é possível encontrar processadores fabricados pela Intel ou AMD. Para ser compreendido por um processador, um programa precisa falar sua língua: a **linguagem (ou código) de máquina**.

Os humanos, em teoria, não falam em linguagem de máquina. Bem, alguns falam, mas isso é outra história. Acontece que para facilitar a criação de programas, algumas boas almas começaram a escrever programas onde humanos escreviam código (instruções para o processador) numa linguagem mais próxima da falada por eles (Inglês no caso). Assim nasceram os primeiros **compiladores**, que podemos entender como programas que "traduzem" códigos em linguagens como **Assembly** ou **C** para código de máquina.

Um programa é então uma série de instruções em código de máquina. Quem consegue olhar pra ele desta forma, consegue entender sua lógica, mesmo sem ter acesso ao código-fonte que o gerou. Isso vale para praticamente qualquer tipo de programa, seja ele criado em linguagens onde a compilação ocorre separada da execução, como C, C++, Pascal, Delphi, Visual Basic, D, Go e até mesmo em linguagens onde a compilação ocorre junto à execução, como Python, Ruby, Perl ou PHP. Lembre-se que o processador só entende código de máquina e para ele não importa qual é o código fonte, ou se a linguagem é compilada(análise e execução separadas) ou interpretada(análise e execução juntas). Então é importante notar que, para o processador poder executar, "tudo tem que acabar em linguagem de máquina". Qualquer um que a conheça será capaz de inferir qual lógica o

programa possui. Aliado ao conhecimento do ambiente de execução, é possível inclusive descrever exatamente o que um programa faz e nisto está a arte da engenharia reversa de software que você vai aprender neste livro. ;-)

---

## Áreas de aplicação da engenharia reversa de software

### Análise de malware

Naturalmente, os criadores de programas maliciosos não costumam compartilhar seus códigos-fonte com as empresas de segurança de informação. Sendo assim, analistas que trabalham nessas empresas ou mesmo pesquisadores independentes podem lançar mão da engenharia reversa afim de entender como essas ameaças digitais funcionam e então poder criar suas defesas.

### Análise de vulnerabilidade

Alguns *bugs* encontrados em *software* podem ser exploráveis por outros programas. Por exemplo, uma falha no componente SMB do Windows permitiu que a NSA desenvolvesse um programa que dava acesso a qualquer computador com o componente exposto na Internet. Para encontrar tal vulnerabilidade, especialistas precisam conhecer sobre engenharia reversa, dentre outras áreas.

### Correção de *bugs*

Às vezes um *software* tem um problema e por algum motivo você ou sua empresa não possui mais o código-fonte para repará-lo ou o contrato com o fornecedor que desenvolveu a aplicação foi encerrado. Com engenharia reversa, pode ser possível corrigir tal problema.

### Mudança e adição de recursos

Mesmo sem ter o código-fonte, é possível também alterar a maneira como um programa se comporta. Por exemplo, um programa que salva suas configurações num diretório específico pode ser instruído a salvá-las num compartilhamento de rede.

Adicionar um recurso é, em geral, trabalhoso, mas possível.

## (Anti-)pirataria

*Software* proprietário costuma vir protegido contra pirataria. Você já deve ter visto programas que pedem número de série, chave de registro, etc. Com engenharia reversa, os chamados *crackers* são capazes de quebrar essas proteções. Por outro lado, saber como isso é feito é útil para programadores protegerem melhor seus programas. ;-)

## Reimplementação de software e protocolos

Um bom exemplo de uso da engenharia reversa é o caso da equipe que desenvolve o LibreOffice: mesmo sem ter acesso ao código fonte, eles precisam entender como o Microsoft Office funciona, a fim de que os documentos criados nos dois produtos sejam compatíveis. Outros bons exemplos incluem:


- o **Wine**, capaz de rodar programas feitos para Windows no GNU/Linux;
- o **Samba** que permite que o GNU/Linux apareça e interaja em redes Windows;
- o **Pidgin** que conecta numa série de protocolos de mensagem instantânea;
- e até um sistema operacional inteiro chamado **ReactOS**, que lhe permite executar seus aplicativos e drivers favoritos do Windows em um ambiente de código aberto e gratuito.

Todos estes são exemplos de implementações em software livre, que tiveram de ser criadas a partir da engenharia reversa feita em programas e/ou protocolos de rede proprietários.

# Números

## Tudo é número (Pitágoras)

Costumo dizer para meus alunos que um computador é basicamente uma calculadora gigante. Claro que esta é uma afirmação muito simplista, mas a verdade é que a ideia pitagórica de que "tudo é número" cabe muito bem aqui. Não é à toa que em textos sobre a origem da computação você encontra a foto de um ábaco, a primeira máquina de calcular, datando-se aproximadamente de mais de 2000 anos antes de Cristo e que é feita de pedras. De fato, *calculus* em Latim significa pedrinha (agora você entende a expressão "cálculo renal"!), porque era a maneira que o povo tinha para contar na antiguidade.

 Um fato interessante é que a Google detém uma **patente** de um ábaco hexadecimal datada de 1988!

Neste capítulo vamos focar nos números. Em breve veremos como o processador trabalha com eles também.

Pois então, o que é um número? Segundo [artigo na Wikipédia](#), um número é um **objeto matemático utilizado para contar, medir ou descrever uma quantidade**. Na prática também utilizamos números para outros fins, como um número de telefone ou número de série de um equipamento.

O processador de um computador moderno consegue realizar muitos cálculos num intervalo de tempo muito curto. Mas, considerando o computador como dispositivo eletrônico que ele é, já parou para pensar como é que um número "entra" no processador? Para entender isso com precisão, seria necessário falar de eletricidade, física, química e talvez quântica, mas vou resumir: os elétrons que caminham pelos circuitos de um computador e chegam até o processador são **interpretados** de modo que uma baixa tensão elétrica é interpretada como o número 0 e uma mais alta, como 1. É através de um componente eletrônico chamado transístor que se consegue representar 0 e 1 dentro do processador. Você pode aprender mais sobre isso nas [referências](#) deste livro. Parece pouco, mas nas próximas seções você verá como que, a partir de somente dois números é possível obter-se todos os outros.



# Sistemas de numeração

Conhecemos bem os dez símbolos latinos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9 utilizados no sistema de numeração decimal. Neste sistema, o símbolo 0 (zero) é utilizado para descrever uma quantidade nula, enquanto o símbolo 1 (um) descreve uma quantidade, o 2 (dois) duas quantidades e assim sucessivamente, até que atinjam a quantidade máxima com apenas um dígito, que é 9 (nove). Para representar uma quantidade a mais que essa, a regra é: pegamos o símbolo que representa uma quantidade e colocamos à sua direita o que representa uma quantidade nula formando, assim, 10 (dez). O mesmo processo ocorre com este zero à direita, até que os dígitos "acabem" novamente e aí incrementamos o 1 da esquerda em uma unidade, até que chegamos ao 20. Estudos futuros definiram este conjunto como **números naturais** e adicionaram outros: números inteiros (que contemplam os negativos), fracionários, complexos, etc.

Mas este não é o único - nem é o primeiro - sistema para representação de quantidades. Ou seja, não é o único sistema de numeração possível. Os computadores, diferente dos humanos, são máquinas elétricas. Sendo assim, a maneira mais fácil de números fluírem por eles seria com um sistema que pudesse ser **interpretado** a partir de dois estados: ligado e desligado.

---

## Binário

O sistema binário surgiu há muito tempo e não vou arriscar precisar quando ou onde, mas em 1703 o alemão Leibniz publicou um [trabalho refinado](#) baseado na dualidade taoísta chinesa do *yin* e *yan* a qual descrevia o sistema binário moderno com dois símbolos: 0 (nulo) e 1 (uma unidade). Por ter somente dois símbolos, ficou conhecido como sistema binário, ou de base 2. A contagem segue a regra: depois de 0 e 1, pega-se o símbolo que representa uma unidade e se insere, à sua direita, o que representa nulo, formando o número que representa duas unidades neste sistema: 10.



Daí vem a piada nerd que diz haver apenas 10 tipos de pessoas no mundo: as que entendem linguagem binária e as que não entendem.

Assim sendo, se formos contar até dez unidades, teremos: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001 e 1010.

Perceba que a lógica de organização dos símbolos no sistema binário é a mesma do sistema de numeração decimal. No entanto, em binário, como o próprio nome sugere, só temos dois símbolos disponíveis para representar todas as quantidades.

Por utilizar dois símbolos que são idênticos aos do sistema decimal, num contexto genérico, números binários são normalmente precedidos com 0b para não haver confusão. Então para expressar dez quantidades faríamos 0b1010. Por exemplo, a seguinte linha na console do Python imprime o valor 10:

Python

```
print(0b1010)
```

Bash

```
echo $((0b1010))
```


C


```
printf("%d\n", 0b1010);
```

---

## Octal

Como o próprio nome sugere, o sistema octal possui oito símbolos: 0, 1, 2, 3, 4, 5, 6 e 7. À esta altura já dá pra sacar que para representar oito quantidades em octal o número é 10. Nove é 11, dez é 12 e assim sucessivamente.

 O sistema octal é utilizado para as permissões de arquivo pelo comando `chmod` nos sistemas baseados em Linux e BSD. Os números 1, 2 e 4 representam permissão de execução, escrita e leitura, respectivamente. Para combiná-las, basta somar seus números correspondentes. Sendo assim, uma permissão 7 significa que se pode-se tudo (`rwX`) enquanto uma permissão 6 somente escrita e leitura (`rw-`). Tais números foram escolhidos para não haver confusão. Se fossem os números 1, 2 e 3 a permissão 3 poderia significar tanto ela mesma quanto 1+2 (execução + escrita). Usando 1, 2 e 4 não há brechas para dúvida. ;)

 Na programação normalmente um número octal é precedido de um algarismo 0 para diferenciá-lo do decimal. Por exemplo, 12 seria doze (decimal), enquanto 012 é dez (octal). Portanto, cuidado ao ignorar o zero à esquerda!

Veja o exemplo:

Python

```
1 >>> 012
2 10
```



## Bash

```
1 echo "$((8#012))"  
2 10  
3 # Usando o bc:  
4 echo "obase=10; ibase=8; 012" | bc  
5 10
```

## Hexadecimal

Finalmente o queridinho hexa (para os íntimos); o sistema de numeração que mais vamos utilizar durante todo o livro. O hexadecimal apresenta várias vantagens sobre seus colegas, a começar pelo número de símbolos: 16. São eles: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F. Os números que eles formam são normalmente prefixados com 0x.

Perceba que os sistemas utilizam os mesmos símbolos latinos. Isso é só para facilitar mesmo. :)

Aqui cabe uma tabela comparativa, só para exercitar:

Hexadecimal	Decimal	Octal	Binário
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110

7	7	7	111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

Existem algumas propriedades interessantes que cabe ressaltar quando relacionamos os diferentes sistemas de numeração vistos aqui. São elas:

- Quanto mais símbolos (dígitos) existem no sistema, menos dígitos utilizamos para representar mais quantidades.
- 0xF = 0b1111 assim como 0xFF = 0b11111111 e 0xFE = 0b11111110.
- 0x10 = 16. Então, 0x20 = 32 e 0x40 = 64.
- Em hexadecimal, 9 + 1 = A, então 19 + 1 = 1A.
- Na arquitetura x86 os endereços são de 32-bits. Ao analisar a pilha de memória, por exemplo, é bom se acostumar com decrementos de 4 bytes, tipo: 12FF90, 12FF8C, 12FF88, 12FF84, 12FF80, 12FF7C...
- Cada dígito hexa pode ser compreendido como 4 dígitos binários, então 0xB0B0CA = 0b101100001011000011001010. Fica fácil se aplicarmos dessa maneira, veja:

```

1  B  0  B  0  C  A
2  1011 0000 1011 0000 1100 1010

```

Teste no terminal:

## Python

```
1 >>> 0xa
2 10
3 >>> 0x0A
4 10
5 >>> 0x000000000000000000000000a
6 10
7 >>> 0xA
8 10
```

## Bash

```
1 echo "$((16#a))"
2 10
3 echo "$((16#0A))"
4 10
5 echo "$((16#000000000000000000000000a))"
6 10
7 echo "$((16#A))"
8 10
```



Em Python, C e outras linguagens, não importa se escrevemos números hexadecimais com letras maiúsculas ou minúsculas (mais comum), desde que os prefixemos com 0x. Os zeros à esquerda (imediatamente após o 0x) também não importam.

Falaremos bastante em endereços de memória no conteúdo de engenharia reversa e todos estão em hexadecimal, por isso é importante "pensar em hexa" daqui pra frente. Mas não se preocupe, se precisar calcular algo, sempre poderá recorrer à calculadora ou ao Python mesmo. Uma alternativa no shell do Linux é o comando `bc`, que o vídeo a seguir explica bem:



<https://www.youtube.com/watch?v=vLhABLeb11o>

Cálculo no shell com o bc

## Crie seu próprio sistema de numeração


Um bom exercício é criar o seu sistema de numeração, com símbolos diferentes dos habituais. Pode ser qualquer coisa. Digamos que farei um sistema ternário chamado Lulip's que possui os seguintes símbolos para representar zero, uma e duas quantidades respectivamente: @, # e \$. Olha só como ficaria a comparação com decimal:

Decimal	Lulip's
0	@
1	#
2	\$
3	#@
4	##
5	#\$
6	\$@
7	\$#
8	\$\$
9	#@@
10	#@#
11	#@\$

É importante que o leitor perceba a lógica utilizada para contar no sistema Lulip's. Ele não existe, mas criar o próprio sistema é um bom exercício para compreender como qualquer um pode ser convertido entre si.

# O byte

Agora que já temos outro olhar sobre os números, é necessário entender como o computador trabalha com eles. Acontece que é preciso armazenar estes números em algum lugar antes de usá-los em qualquer operação. Para isso foi criado o *byte*, a unidade de medida da computação. Consiste em um **espaço** para armazenar *bits* (8 na arquitetura Intel, também chamado de **octeto**). Então, neste livro, sempre que falarmos em 1 *byte*, leia-se, "um espaço onde cabem 8 *bits*". Sendo assim, o primeiro número possível num *byte* é 0b00000000, ou simplesmente 0 (já que zero à esquerda não vale nada). E o maior número possível é 0b11111111 que é igual a 0xff ou 255, em decimal.

 Uma maneira rápida de calcular o maior número positivo que pode ser representado num espaço de  $x$  *bits* é usando a fórmula  $2^x - 1$ . Por exemplo, para os 8 *bits* que mencionamos, basta elevar 2 à oitava potência (que resulta em 256) e diminuir uma unidade:  $2^8 - 1 = 255$ . Se você se perguntou por que diminuir 1, lembre-se de que o zero precisa ser representado também. Se podemos representar 256 números diferentes e o zero é um deles, ficamos com 0 à 255. ;-)

Agora que você já sabe o que é um *byte*, podemos apresentar seus primos *nibble* (metade dele), *word*, etc. Veja a tabela:

Medida	Tamanho (Intel)	Nomenclatura Intel
<i>nibble</i>	4 bits	
<i>byte</i>	8 bits	BYTE
<i>word</i>	16 bits	WORD
<i>double word</i>	32 bits	DWORD
<i>quad word</i>	64 bits	QWORD

Fica claro que o maior valor que cabe, por exemplo, numa variável, depende de seu tamanho (quantidade de espaço para armazenar algum dado). Normalmente um tipo inteiro tem 32 bits, portanto, podemos calcular 2 elevado a 32 menos 1, que dá 4294967295. O inteiro de 32 *bits* ou 4 *bytes* é muito comum na arquitetura Intel x86.

# Números negativos

Já vimos que um *byte* pode armazenar números de 0 a 255 por conta de seus 8 *bits*. Mas como fazemos quando um número é negativo? Não temos sinal (-), só *bits*. E agora? Não é possível ter números negativos então? Claro que sim, do contrário você não poderia fazer contas com números negativos e o código abaixo falharia:

Python

```
1 >>> -3 + 1
2 -2
```

Bash

```
1 # Usando o bc:
2 echo "-3 + 1" | bc
3 -2
```

Mas não falhou! Isso acontece porque na computação dividimos as possibilidades quase que "ao meio". Por exemplo, sabendo que 1 *byte* pode representar 256 possibilidades (sendo o 0 e mais 255 de números positivos), podemos dividir tais possibilidades, de modo a representar de -128 até +127. Continuamos com 256 possibilidades diferentes (incluindo o zero), reduzimos o máximo e aumentamos o mínimo. :-)

O *bit* mais significativo (mais à esquerda) é utilizado para representar o sinal. Se for 0, é um número positivo. Se for 1, é um número negativo.

Há ainda a técnica chamada de **complemento de dois**, necessária para calcular um valor negativo. Para explicá-la, vamos ao exemplo de obter o valor negativo -10 a partir do valor positivo 10. Os passos são:



1. Converter 10 para binário, que resulta em 0b1010.
2. Acrescentar à esquerda do valor binário os zeros para formar o *byte* completo (8 *bits*): 0b00001010.
3. Inverter todos os *bits*: 0b11110101 (essa operação é chamada de complementação ou complemento de um).
4. Somar 1 ao resultado final, quando finalmente chegamos ao complemento de dois: 0b11110110.

Sendo assim, vamos checar em Python:

Python

```
1 >>> 0b11110110
2 246
```

Bash

```
1 echo "$((2#11110110))"
2 246
3 # Usando o bc:
4 echo "obase=10; ibase=2; 11110110" | bc
5 246
```

O que aconteceu? Bem, realmente 0b11110110 dá 246 (em decimal), se interpretado como número sem sinal. Acontece que temos que dizer explicitamente que vamos interpretar um número x como número com sinal (que pode ser positivo ou negativo). Em Python, um jeito é usando a biblioteca *ctypes*:

```
1 >>> import ctypes
2 >>> ctypes.c_byte(0b11110110).value
3 -10
```

Já em C, é preciso especificar se uma variável é *signed* ou *unsigned*. O jeito como o processador reconhece isso será visto mais à frente, mas por hora entenda que não há mágica: 0b11110110 (ou 0xf6) pode ser tanto 246 quanto -10. Depende de como é interpretado, com ou sem sinal.


Por fim, é importante notar que a mesma regra se aplica para números de outros tamanhos (4 *bytes* por exemplo). Analise a tabela abaixo, que considera números de 32 *bits*:

Binário	Hexa	Com sinal	Sem sinal
10000000000000000000000000000000	80000000	-2147483648	2147483648
11111111111111111111111111111111	FFFFFFFF	-1	4294967295
00000000000000000000000000000000	00000000	0	0
01111111111111111111111111111111	7FFFFFFF	2147483647	2147483647

Perceba que o número 0x7fffffff tem seu primeiro *bit* zerado, portanto nunca será negativo, independente de como seja interpretado. Para ser um número negativo, é necessário que o primeiro *bit* do número esteja *setado*, ou seja, igual a 1.

# Cálculos com binários

Nesta seção faremos alguns cálculos com números binários, considerando cada um de seus dígitos, também chamados de **bits**. Além das clássicas como adição, subtração, multiplicação e divisão, estudaremos aqui a conjunção, disjunção, negação e disjunção exclusiva. Também incluiremos outras operações bit-a-bit que fogem da álgebra tradicional, como deslocamento e rotação de bits. Todas são importantes pois existem no contexto do Assembly, que estudaremos no futuro, e são aplicadas a números de qualquer espécie, como endereços de memória.

 Você pode encontrar mais sobre este assunto pesquisando por álgebra booleana e operações bit-a-bit (*bitwise*).

## Conjunção (AND)

Dados dois bits  $x$  e  $y$ , a conjunção deles resulta em 1 se ambos forem iguais a 1. Na programação o seu símbolo é normalmente um  $\&$ . Sendo assim, a chamada **tabela verdade** desta operação é:

$x$	$y$	$x \& y$
0	0	0
0	1	0
1	0	0
1	1	1

Então suponha que queiramos calcular a conjunção do número 0xa com 12. Sim, estamos misturando dois sistemas de numeração (hexadecimal e decimal) na mesma operação. E por que não? O segredo é converter para binário e fazer o cálculo para cada bit, respeitando a tabela verdade da conjunção. Mãos à obra:

```
1  0xa = 1010
2  12  = 1100
3      -----
4      1000
```

O resultado é 0b1000, ou 8 em decimal. Sendo assim, as linhas abaixo farão o mesmo cálculo, mesmo utilizando sistemas de numeração (bases) diferentes:

```
1  >>> 0b1010 & 12
2  8
3  >>> 10 & 12
4  8
5  >>> 0xa & 0b1100
6  8
7  >>> 012 & 0xc
8  8
9  >>> 0xa & 0xc
10 8
```

Por que utilizei tantas bases diferentes? Quero com isso por na sua cabeça que um número é só um número, independente da base na qual ele está sendo representado ou visualizado.

---

## Disjunção (OR)

O resultado da disjunção entre dois bits  $x$  e  $y$  é 1 se pelo menos um deles for 1. Sendo assim, segue a tabela verdade:

$x$	$y$	$x   y$
0	0	0
0	1	1
1	0	1

---

---

1

1

1

---

Na programação, o símbolo normalmente é a barra em pé: |. Por exemplo, vamos calcular a disjunção entre 8 e 5:

```
1  8 = 1000
2  5 = 0101 (perceba o zero à esquerda, para facilitar)
3      -----
4      1101
```

O resultado é 0b1101, que é 13 em decimal.

Aí você pode questionar:

- Opa, então a disjunção é tipo a soma?

Te respondo:

Mais ou menos.

Veja que o resultado da disjunção entre 9 e 5, também é 13:

```
1  9 = 1001
2  5 = 0101
3      -----
4      1101
```

Isso porque numa soma entre 1 e 1 o resultado seria 10 (2 em decimal), já na operação OU o resultado é 1.

---

## Disjunção exclusiva (XOR)

A disjunção exclusiva entre x e y resulta em 1 se **somente um** deles for 1. Sendo assim:

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

Assim como a disjunção é normalmente chamada de "OU", a disjunção exclusiva é chamada de "OU exclusivo", ou simplesmente XOR. Já virou até verbo e é comum ouvir pessoas falando que "XORream" um dado.

Algumas propriedades importantes desta operação são:

1. Você pode aplicá-la em qualquer ordem. Então,  $a \wedge (b \wedge c) = (a \wedge b) \wedge c$  por exemplo.
2. Um número "XORado" com ele mesmo é sempre zero.
3. Um número "XORado" com zero é sempre ele mesmo.

A operação XOR tem vários usos em computação. Alguns exemplos:

## Detecção de diferenças

É possível saber se um número é diferente de outro com XOR. Se os números forem diferentes, o resultado é diferente de zero. Por exemplo, tomemos um XOR entre 8 e 5 e outro entre 5 e 5:

```

1  1000  0101
2  0101  0101
3  -----
4  1101  0000
```

## Zerar variáveis

Fica claro que é possível zerar variáveis bastando fazer uma operação XOR do valor dela com ele mesmo, independentemente de que valor é este:

```
1 >>> x=90
2 >>> x = x ^ x
3 >>> x
4 0
```

## Troca de valores entre duas variáveis

O algoritmo conhecido por *XOR swap* consiste em trocar os valores de duas variáveis somente com operações XOR, sem usar uma terceira variável temporária. Basta fazer:

- Um XOR entre x e y e armazenar o resultado em **x**.
- Outro XOR entre x e y e armazenar o resultado em **y**.
- Outro XOR entre x e y e armazenar o resultado em **x**.

Veja:

```
1 >>> x=8
2 >>> y=5
3 >>> x = x ^ y
4 >>> y = x ^ y
5 >>> x = x ^ y
6 >>> x
7 5
8 >>> y
9 8
```

Analisando em binário:

```
1 x = 0b1000 # 8 em decimal
2 y = 0b0101 # 5 em decimal
3 x = x ^ y   # 0b1101
4 y = x ^ y   # resulta em 0b1000 (já é o valor original de x)
5 x = x ^ y   # resulta em 0b0101 (o valor original de y)
```

## Cifragem

Dado um número  $x$ , é possível calcular o resultado de uma operação XOR com um valor que chamamos de chave. Se usarmos a mesma chave num XOR com este resultado, obtemos novamente o número original:

```
1 >>> x = 2017
2 >>> x = x ^ 0x51
3 >>> x
4 1968
5 >>> x = x ^ 0x51
6 >>> x
7 2017
```

Portanto, para uma cifragem básica, se quiser esconder o valor original de um número antes de enviá-lo numa mensagem, basta *XOReá-lo* com uma chave que só eu e o receptor da mensagem conheça (0x51 no exemplo). Assim eu uso tal chave para fazer a operação XOR com ele e instruo o receptor da mensagem (por outro canal) a usar a mesma chave e operação XOR, afim de obter o número original.

⚠ Em textos matemáticos sobre lógica, o acento circunflexo  $\wedge$  representa conjunção ao invés de disjunção exclusiva. Já em softwares matemáticos, pode significar potência, por exemplo:  $2^{32}$  é dois elevado à trigésima segunda potência.

ℹ Na língua Portuguesa utilizamos a palavra "OU" no sentido de "OU exclusivo". Por exemplo, quando você pede "Pizza de presunto ou pepperoni ou lombo", quer dizer que só quer um dos sabores (exclusividade). Se fosse uma disjunção tradicional "OU", o garçom poderia trazer presunto com pepperoni, ou mesmo todos os três ingredientes e você não poderia reclamar. :-D



## Deslocamento (SHL e SHR)

O deslocamento para a **esquerda** (*shift left*) consiste em deslocar todos os *bits* de um número para a esquerda e completar a posição criada mais à direita com zero. Tomemos por exemplo o número 7 e uma operação SHL com 1 (deslocar uma vez para a esquerda):

```
1  0111  # 7 em decimal
2      1  # Deslocar uma vez para a esquerda (SHL)
3  ----
4  1110  # 14
```

Assim podemos perceber que deslocar à esquerda dá no mesmo que multiplicar por 2. Veja:

```
1  >>> x = 7
2  >>> x = x << 1
3  >>> x
4  14
5  >>> x = x << 1
6  >>> x
7  28
8  >>> x = x << 1
9  >>> x
10 56
```

No exemplo acima deslocamos 1 *bit* do número 7 (0b111) para a esquerda três vezes, que resultou em 56. Seria o mesmo que deslocar 3 *bits* de uma só vez:

```
1  >>> 7 << 3
2  56
```

De forma análoga, o deslocamento para a **direita** (*shift right*), ou simplesmente SHR, consiste em deslocar todos os *bits* de um número para a direita e completar a posição criada à esquerda com zero. Tomando o mesmo 7 (0b111):

```
1 >>> 0b111 >> 1
2 3
```

O resultado é uma divisão inteira (sem considerar o resto) por 2. Assim,  $7/2 = 3$  (e sobra 1, que é desconsiderado neste cálculo). Esta é de fato uma maneira rápida de dobrar ou calcular a metade de um número.

---

## Rotação (ROL e ROR)

Assim como no deslocamento, a rotação envolve deslocar os *bits* de um número para a esquerda (*rotate left*) ou direita (*rotate right*) mas o *bit* mais significativo (mais à esquerda) é posto no final (mais à direita), no lugar de zero. Por isso é necessário considerar o tamanho ... Tomemos o número 5 como exemplo:

```
1 00000101 # 5 em decimal
2         1 # ROL com 1
3 -----
4 00001010 # 10 em decimal
```

O *bit* *\_zero* mais à esquerda "deu a volta" e veio parar ao lado direito do *\_bit* 1 mais à direita do número 0b00000101. Analisa com o número 133 agora:

```
1 10000101 # 133 em decimal
2         1 # ROL com 1
3 -----
4 00001011 # 11 em decimal
```

Desta vez o *bit* 1, que estava mais à esquerda veio parar ao lado direito do *bit* mais à direita, e todos os outros *bits* foram "empurrados" para a esquerda.

Não estamos limitados a fazer operações ROL e ROR somente com 1. O byte 133 ROL 3 por exemplo resulta em 0x2c. Você é capaz de conferir?

## Negação (NOT)

Para negar um bit, basta invertê-lo:

x	~x
0	1
1	0

No entanto, para inverter o número 0b100 é preciso saber seu tamanho:

Tamanho	0b100	~0b100
1 byte	0b00000100	0b11111011
2 bytes (WORD)	0b000000000000000100	0b111111111111111011
4 bytes _(DWORD)	0b0000000000000000000000000000000100	0b11111111111111111111111111111111

Isso é o mesmo que calcular o complemento (ou "complemento de um") de um número. Para obter seu simétrico, é preciso ainda somar uma unidade, como vimos anteriormente. Por isso, um NOT *bit-a-bit* no número 4, por exemplo, resulta em -5. Veja:

```
1  >>> ~4
2  -5
```

Os processadores Intel x86 trabalham com muitas outras operações *bitwise*, mas que não serão discutidas neste livro. Conforme você avançar no estudo de engenharia reversa, vai se deparar com elas. ;-)

# Cadeias de texto

Se o computador só entende números, como podemos trabalhar com texto então? Bem, não se engane, o computador realmente só entende números. O fato de você apertar uma tecla no teclado que tem o desenho de um símbolo do alfabeto utilizado no seu país não garante que é isto que de fato seja enviado para o computador e, naturalmente, não é. Ao invés disso, cada tecla possui um código conhecido como *scan code* ou *make code* que é enviado, entre outras informações, pelo barramento de dados do teclado para a placa-mãe do computador e passa por vários estágios até chegar ao *kernel*, o núcleo do sistema operacional. Por exemplo, de acordo com informação contida no site [Beyond Logic](#), o *scan code* da tecla "A" é o byte 0x1c.

Após receber o *scan code* da tecla e outros dados, o *driver* pode adicionar ou modificar valores. É o que acontece no Linux, por exemplo. Com o comando abaixo é possível por exemplo analisar os *bytes* gerados no dispositivo do teclado quando a tecla "A" é pressionada:

```
1 # hd /dev/input/event0
2 00000000 f1 e7 6e 59 00 00 00 00 6d e6 04 00 00 00 00 00 |..nY....m....
3 00000010 04 00 04 00 1e 00 00 00 f1 e7 6e 59 00 00 00 00 |.....nY.
4 00000020 6d e6 04 00 00 00 00 00 01 00 1e 00 01 00 00 00 |m.....
5 00000030 f1 e7 6e 59 00 00 00 00 6d e6 04 00 00 00 00 00 |..nY....m....
6 00000040 00 00 00 00 00 00 00 00 f1 e7 6e 59 00 00 00 00 |.....nY.
7 00000050 8b 3b 06 00 00 00 00 00 04 00 04 00 1e 00 00 00 |.;.....
8 00000060 f1 e7 6e 59 00 00 00 00 8b 3b 06 00 00 00 00 00 |..nY.....;...
9 00000070 01 00 1e 00 00 00 00 00 f1 e7 6e 59 00 00 00 00 |.....nY.
10 00000080 8b 3b 06 00 00 00 00 00 00 00 00 00 00 00 00 00 |.;.....
11 00000090
```

O simples pressionar de uma tecla gerou 144 (0x90) *bytes* de dados. Esta é uma especificidade do Linux, que define vários tipos de eventos, cada um com 24 *bytes* de tamanho. O utilitário **evtest** interpreta os *bytes* acima e produz uma saída mais amigável ao pressionarmos a mesma tecla "A":

```
1 # evtest /dev/input/event0
2 Event: time 1500442813.544820, type 4 (EV_MSC), code 4 (MSC_SCAN), value 1
3 Event: time 1500442813.544820, type 1 (EV_KEY), code 30 (KEY_A), value 1
```

```
4 Event: time 1500442813.544820, ----- EV_SYN -----
5 Event: time 1500442813.610261, type 4 (EV_MSC), code 4 (MSC_SCAN), value 1
6 Event: time 1500442813.610261, type 1 (EV_KEY), code 30 (KEY_A), value 0
7 Event: time 1500442813.610261, ----- EV_SYN -----
```

Perceba os 6 eventos. Se quiser saber mais sobre como o *kernel* Linux padroniza e transforma os *scan codes* enviados pelo teclado para estes valores, basta ler a [documentação](#), mas este não é nosso assunto alvo aqui.

Assim como na entrada de dados pelo teclado, o tratamento da entrada do *mouse* ou de qualquer outro dispositivo também é numérico e, de maneira geral, o computador necessita entender que ao ler um determinado número, precisa tomar alguma ação, como desenhar o que conhecemos por caractere "a". Para ele é um número, para nós, um símbolo. Veremos nas seções a seguir como essa conversão se dá.

# ASCII

## ASCII 7-bits

Computadores trabalham com números, mas humanos trabalham também com texto. Sendo assim, houve a necessidade de criar um padrão de representação textual - e também de controle, que você entenderá a seguir.

O **American Standard Code for Information Interchange**, ou em português, Código Padrão Americano para o Intercâmbio de Informação, é uma codificação criada nos EUA (como o nome sugere), já que o berço da computação foi basicamente lá.

Na época em que foi definido, lá pela década de 60, foi logo usado em equipamentos de telecomunicações e também nos computadores. Basicamente é uma tabela que relaciona um **número** de 7 *bits* com sinais de controle e caracteres imprimíveis. Por exemplo, o número 97 (0b1100001) representa o caractere 'a', enquanto 98 (0b1100010) é o 'b'. Perceba que tais números não excedem os 7 *bits*, mas como em computação falamos quase sempre em *bytes*, então acaba que um caractere ASCII possui 8 *bits* mas só usa 7. A tabela ASCII vai de 0 a 127 e pode ser encontrada no Apêndice, que você deve consultar agora.

Há vários testes interessantes que você pode fazer para entender melhor as *strings* ASCII. Ao saber que o caractere 'a' é o número 97, você pode usar a função `chr()` no Python, por exemplo:

```
1 >>> print chr(97)
2 a
```

Ou no próprio shell do Linux, o comando `echo`:

```
1 $ echo -ne \\x61 # 0x61 em hexa é 97 em decimal!
2 a
```

Viu? Quando você digita 'a', o computador entende o *byte* 0x61. De forma análoga, quando um programa que exibe um texto na tela encontra o *byte* 0x61, ele exibe 'a'. Como você pode ver na tabela ASCII, todas as letras do alfabeto americano estão lá, então é razoável concluir que uma frase inteira seja na verdade uma sequência de *bytes* onde cada um deles está dentro da faixa da tabela ASCII. Isso fica facilmente visualizável no shell do Linux com o comando *hd*(hexdump):

```
1 $ echo 'Acesse mentebinaria.com.br' | hd
2 00000000 41 63 65 73 73 65 20 6d 65 6e 74 65 62 69 6e 61 |Acesse menteb
3 00000010 72 69 61 2e 63 6f 6d 2e 62 72 0a                |ria.com.br.|
```


É exatamente assim que um texto ASCII vai parar dentro de um programa ou arquivo.

Agora olhe novamente a tabela e perceba o seguinte:

- O primeiro sinal é o NUL, também conhecido como *null* ou nulo. É o *byte* 0.
- Outro *byte* importante é 0x0a, conhecido também por \n, *line feed*, LF ou simplesmente "caractere de nova linha".
- O MS-DOS e o Windows utilizam na verdade **dois** caracteres para delimitar uma nova linha. Antes do 0x0a, temos um 0x0d, conhecido também por \r, *carriage return* ou CR. Essa dupla é também conhecida por **CrLf**.
- O caractere de "espaço em branco" é o 0x20.
- Os dígitos vão de 0x30 a 0x39.
- As letras maiúsculas vão de 0x41 a 0x5a.
- As letras minúsculas vão de 0x61 a 0x7a.

Agora, algumas relações:

- Se somarmos 0x20 ao número ASCII equivalente de um caractere maiúsculo, obtemos o número equivalente do caractere minúsculo em questão.
- Se diminuirmos 0x30 de um dígito, temos o equivalente numérico do dígito. Ex.: 0x35 - 0x30 = 5.

 Sabe quando no Linux você dá um cat num arquivo que não é de texto e vários caracteres "doidos" aparecem na tela enquanto você escuta alguns beeps?



Esses sons são os bytes 0x07 encontrados no arquivo. Experimente! ;-)

## ASCII estendida

A tabela ASCII padrão de 7 *bits* é limitada ao idioma inglês no que diz respeito ao texto. Perceba que uma simples letra 'a' com sinal indicativo de crase é impossível nesta tabela. Sendo assim, ela foi estendida e inteligentemente passou-se a utilizar o último *bit* do *byte* que cada caractere ocupa, tornando-se assim uma tabela de 8 *bits*, que vai de 128 a 255 (em decimal).

Essa extensão da tabela ASCII varia de acordo com a **codificação** utilizada. Isso acontece porque ela foi criada para permitir texto em outros idiomas, mas somente 128 caracteres a mais não são suficientes para representar os caracteres de todos os idiomas existentes. A codificação mais conhecida é a ISO-8859-1, também chamada de Latin-1, que você vê no Apêndice B.



Outro nome para ASCII é US-ASCII. Alguns textos referem-se a texto em ASCII como ANSI strings também.

# UNICODE

A esta altura o leitor já pode imaginar a dificuldade que programadores enfrentam em trabalhar com diferentes codificações de texto. Mas existe um esforço chamado de UNICODE mantido pelo [Unicode Consortium](#) que compreende várias codificações, que estudaremos a seguir. Estas *strings* também são chamadas de **wide strings** (largas, numa tradução livre).

---

## UTF-8

O padrão UTF (*Unicode Transformation Format*) de 8 *bits* foi desenhado originalmente por Ken Thompson (sim, o criador do Unix!) e Rob Pike para abranger todos os caracteres possíveis nos vários idiomas deste planeta.

Os primeiros 128 caracteres da tabela UTF-8 são exatamente os mesmos valores da tabela ASCII padrão e somente necessitam de 1 *byte* para serem representados. Os próximos caracteres utilizam **2 bytes** e compreendem não só o alfabeto latino (como na ASCII estendida com codificação ISO-8859-1) mas também os caracteres gregos, árabes, hebraicos, dentre outros. Já para representar os caracteres de idiomas como o chinês e japonês, **3 bytes** são necessários. Por fim, há os caracteres de antigos manuscritos, símbolos matemáticos e até *emojis* (que lindo!) que utilizam **4 bytes**.

Concluímos que os caracteres UTF-8 **variam** de 1 a 4 bytes. Sendo assim, como ficaria o texto "papobinário" numa sequência de *bytes*? Podemos ver com os comandos **echo** e **hd** no Linux:

```
1 $ echo -n "papobinário" | hd
2 00000000 70 61 70 6f 62 69 6e c3 a1 72 69 6f |papobin..rio|
```

Como dito antes, os caracteres da tabela ASCII são os mesmos, mas o caractere 'á' utiliza 2 bytes (no caso, 0xc3 e 0xa1) para ser representado. Esta é uma *string* UTF-8 válida.

Dizemos que seu tamanho é 11, já que ela contém 11 caracteres, mas em *bytes* seu tamanho é 12.

Você pode confirmar que esta é uma *string* UTF-8 utilizando o comando **file** (presente no Linux e macOS). Veja a diferença:

```
1 $ echo -n "papobinario" | file -  
2 /dev/stdin: ASCII text, with no line terminators  
3  
4 $ echo -n "papobinário" | file -  
5 /dev/stdin: UTF-8 Unicode text, with no line terminators
```

Como os *shells* atuais utilizam UTF-8, ao utilizar um caractere não presente na tabela ASCII padrão, uma *string* UTF-8 é gerada. O traço após o nome do comando **file** o fez ler da entrada padrão (**stdin**). Para saber mais sobre como o comando **file** funciona, assista ao seguinte vídeo:



[https://www.youtube.com/watch?v=D7\\_zPEt5vGs](https://www.youtube.com/watch?v=D7_zPEt5vGs)

Identificando arquivos com o file

## UTF-16

Também conhecido por UCS-2, este tipo de codificação é frequentemente encontrado em programas compilados para Windows, incluindo os escritos em .NET. É de extrema importância que o engenheiro reverso o conheça bem.

Representados em UTF-16, os caracteres equivalentes na tabela ASCII possuem **2 bytes** de tamanho onde o primeiro *byte* é o mesmo da tabela ASCII e o segundo é um zero. Por exemplo, para se escrever "A" em UTF-16, faríamos: 0x41 0x00. Vamos entender melhor com o comando **strings** do Linux, a seguir.

Primeiro vamos exibir o texto em ASCII "papo" mas ao invés de imprimir na tela, vamos passar a saída para o programa **strings**:

```
1 $ echo -ne "\x70\x61\x70\x6f" | strings
2 papo
```

Até aí, nenhuma novidade. O **strings** busca justamente *strings* naquilo que é passado para ele. Mas vamos agora tentar o mesmo texto escrito em UTF-16, em que cada caractere possui dois *bytes* e seu equivalente em ASCII e um zero em sequência:

```
1 $ echo -ne "\x70\x00\x61\x00\x70\x00\x6f\x00" | strings
2 $
```

Nada é retornado porque o comando **strings** só busca por padrão *strings* ASCII e a *string* que passamos é uma UTF-16. Por sorte este comando possui a opção **-e** em sua versão para Linux que permite especificar a codificação:

```
1 $ echo -ne "\x70\x00\x61\x00\x70\x00\x6f\x00" | strings -e l
2 papo
```

Agora sim vemos o texto. O motivo está no manual do comando **strings**. Veja:

```
$ man strings
```

```
1 -e encoding
2 --encoding=encoding
3 Select the character encoding of the strings that are to be found. Possib
4
5 s = single-7-bit-byte characters (ASCII, ISO 8859, etc., default)
6 S = single-8-bit-byte characters
7 b = 16-bit bigendian
```

```
8 l = 16-bit littleendian
9 B = 32-bit bigendian
10 L = 32-bit littleendian.
11
12 Useful for finding wide character strings. (l and b apply to, for example,
13 Unicode UTF-16/UCS-2 encodings).
```


Perceba que há uma opção para ASCII estendido também (-S):

```
1 $ echo binário | strings
2 $
3
4 $ echo binário | strings -e S
5 binário
```

Ainda sobre UTF-16, é importante observar que no Windows o ASCII estendido com codificação ISO-8859-1 é *encodado* em UTF-16. Por exemplo, se criarmos um programa em .NET que contenha a *string* "Papo Binário", ela vai para o executável deste jeito:

```
42 00 69 00 6e 00 e1 00 72 00 69 00 6f 00
```

O "á" é o *byte* 0xe1 ("á" na tabela ASCII estendida, vide X) seguido de um *nullbyte* (*byte* nulo) 0x00. Você vai entender melhor a importância destes conceitos quando buscarmos por texto em programas utilizando *debuggers* e outras ferramentas.

 Notou que o comando **strings** tem opções de *endianess* para caracteres de 16 e 32 *bits*? Acontece que estas codificações suportam tanto *little endian* (padrão), no qual o *byte* ASCII do caractere é **seguido** de um *nullbyte* quanto o *big endian*, no qual ele é **precedido** por um *nullbyte*. Veja:

```
1 $ echo -ne "\x00\x70\x00\x61\x00\x70\x00\x6f" | strings -e b
2 papo
```

Uma boa leitura adicional é o artigo [Viewing strings in executables](#) (em Inglês), do pesquisador Didier Stevens sobre *strings* UTF-16.

## UTF-32

Raramente utilizado no Windows porém existente em alguns programas para Linux e Unix, este padrão utiliza 4 *bytes* para cada caractere. Vamos já analisar a *string* "papo" em UTF-32 com a opção -L do comando **strings**:

```
1 $ echo -ne "\x70\x00\x00\x00\x61\x00\x00\x00\x70\x00\x00\x00\x6f\x00\x00\x00"
2 papo
```

É importante ressaltar que simplesmente dizer que uma *string* é UNICODE não diz exatamente qual codificação ela está utilizando, fato que normalmente depende do sistema operacional, do programador, do compilador, etc. Por exemplo, um programa feito em C no Windows e compilado com Visual Studio tem as *wide strings* em UTF-16 normalmente. Já no Linux, o tamanho do tipo *wchar\_t* é 32 *bits*, resultando em *strings* UTF-32. Escreva o seguinte programa em C para entender:

```
wide.c

1 #include <wchar.h>
2
3 int main(void) {
4     wchar_t *s = L"papo";
5     wprintf(L"%S\n", s);
6
7     return 0;
8 }
```

Salve-o no Linux como *wide.c* e compile utilizando o gcc:

```
$ gcc -o wide wide.c
```

Agora vamos buscar as *strings* dentro deste binário compilado.

Foi dito que o no Linux as *wide strings* são UTF-32, então a opção correta para utilizarmos com o comando **strings** é a "-L":

```
1 $ strings -e L wide
2 papo
```

O mesmo programa compilado em Windows resultaria em *strings* UTF-16 ao invés de UTF-32, portanto, fique esperto. ;-)

Há muito mais sobre codificação de texto para ser dito, mas isso foge ao escopo deste livro. Se o leitor desejar se aprofundar, basta consultar a documentação oficial dos grupos que especificam tais padrões. No entanto, cabe ressaltar que a prática (compilar programas e buscar como as *strings* são codificadas) é a melhor escola.

# C strings

Na linguagem C foi criado um padrão para saber *programaticamente* o fim de uma *string*: ela precisa ser terminada com um **byte** nulo. Sendo assim, a *string* ASCII "fernando", se utilizada num programa escrito em C, fica no binário compilado (no .exe final, ou outro formato) da seguinte forma:

```
66 65 72 6e 61 6e 64 6f 00
```

⚠ É importante não confundir o *nullbyte* com o caractere de nova linha. Este pode ser o *Line Feed* (0x0a), também conhecido por `\n` em sistemas Unix/Linux. Já no DOS/Windows, a nova linha é definida por dois caracteres: o *Carriage Return* (0x0d) seguido do *Line Feed*, sequência conhecida pelo jargão **CrLf** em linguagens como Visual Basic.

Se a *string* for UTF-16, então dois *bytes* nulos serão adicionados ao fim. Se for UTF-32, quatro. :-)

O leitor pode estar se perguntando a razão pela qual este conceito é útil ao engenheiro reverso. Bem, no caso de busca de *strings* num binário compilado, você pode refinar a busca por sequências de *bytes* na tabela ASCII, usando ou não uma codificação UNICODE (já que os valores ASCII são os mesmos) terminados com por um ou mais *nullbytes*. Por exemplo, supondo que você esteja buscando a *string* "Erro" dentro de um programa, o primeiro passo é descobrir quais são os *bytes* equivalentes na tabela ASCII desta *string*. Ao invés de usar a tabela, você pode usar o comando **hd** no Linux:

```
1 $ echo -n Erro | hd
2 00000000 45 72 72 6f
```

|Erro|



Alternativamente, você pode experimentar a função `bh_str2hex()` do projeto [bashacks](#), disponível para Linux, macOS e Windows 10:

```
1 $ bh_str2hex Erro
2 45 72 72 6f
```

Agora sabemos que a sequência de *bytes* a ser procurada vai depender do tipo de *string*.

- Em ASCII:

45 72 72 6f

- Em UTF-16-LE (*Little Endian*, que é o padrão) ou simplesmente UTF-16:

45 00 72 00 72 00 6f 00

Mas para sermos mais assertivos, caso não haja mais nada depois do "o" da palavra "Erro" no programa, podemos adicionar o *nullbyte* na busca:

- Em ASCII:

45 72 72 6f 00

- Em UTF-16:

45 00 72 00 72 00 6f 00 00 00

Claro que os programas feitos para buscarem texto dentro de arquivos já possuem esta inteligência, no entanto, a proposta deste livro é entender a fundo como a engenharia reversa funciona e por isso não poderíamos deixar de cobrir esta valiosa informação. ;-)

# Arquivos

Não há dúvida de que o leitor já se deparou com diversos arquivos, mas será que já pensou numa definição para eles? Defino arquivo como uma sequência de *bytes* armazenada numa mídia digital somados a uma entrada, um registro, no sistema de arquivos (*filesystem*) que o referencie. Por exemplo, vamos no Linux criar um arquivo cujo seu conteúdo é somente a *string* ASCII "mentebinaria.com.br" (sem aspas):

```
$ echo -n 'mentebinaria.com.br' > arquivo
```

Se nosso estudo sobre *strings* estiver correto, este arquivo deve possuir 19 *bytes* de tamanho. Chequemos:

```
1 $ wc -c arquivo
2 19 arquivo
```

O comando **wc** conta caracteres (opção -m), palavras (-w), linhas (-l) ou *bytes* (-c) de um arquivo.

E isto, nos sistemas de arquivos modernos, é tudo que contém um arquivo: seu conteúdo. Qualquer outra informação sobre ele, inclusive seu nome fica numa referência neste *filesystem*. A maneira mais rápida de checar tudo isso no Linux é com o comando **stat**:

```
1 $ stat arquivo
2   File: 'arquivo'
3   Size: 19          Blocks: 8          IO Block: 4096   regular file
4 Device: 801h/2049d Inode: 532417       Links: 1
5 Access: (0644/-rw-r--r--)  Uid: ( 1000/   user)   Gid: ( 1000/   user)
6 Access: 2017-09-08 17:21:32.745159845 -0400
7 Modify: 2017-09-08 17:21:24.097159494 -0400
8 Change: 2017-09-08 17:21:24.097159494 -0400
9  Birth: -
```

A referência é justamente o que o Linux chama de *inode*, mas este assunto foge ao escopo de nosso livro. Se quiser entender melhor sobre este e outros detalhes de implementação do Linux, recomendo o livro [Descobrimdo o Linux](#), do João Eriberto Mota Filho.

A pergunta mais interessante para nós é, no entanto, em relação ao **tipo** de arquivo. Criamos eles sem extensão e aqui é bom lembrar que uma extensão de arquivo nada mais é que parte de seu nome e esta não mantém nenhuma relação com seu tipo real. A única forma de saber um tipo de arquivo é **inferindo-o** através de seu conteúdo. Vamos olhar seu conteúdo com um visualizador hexadecimal, o **hexdump**, invocado por seu atalho mais comum, o **hd**:

```
1 $ hd arquivo
2 00000000 6d 65 6e 74 65 62 69 6e 61 72 69 61 2e 63 6f 6d |mentebinaria.
3 00000010 2e 62 72 |.br|
4 00000013
```

O **hd** também nos entrega o tamanho do arquivo em hexa, que neste caso é 0x13 ou 19 em decimal. Então, sabendo que todos os *bytes* deste arquivo estão na faixa numérica da tabela ASCII, podemos dizer então que este é, de fato, um arquivo de texto. :-)

Uma maneira mais rápida de identificar um tipo de arquivo é trabalhando com o comando **file**, que se utiliza da **libmagic**, uma biblioteca que tem catalogadas várias sequências de *bytes* que representam tipos de arquivos diferentes:

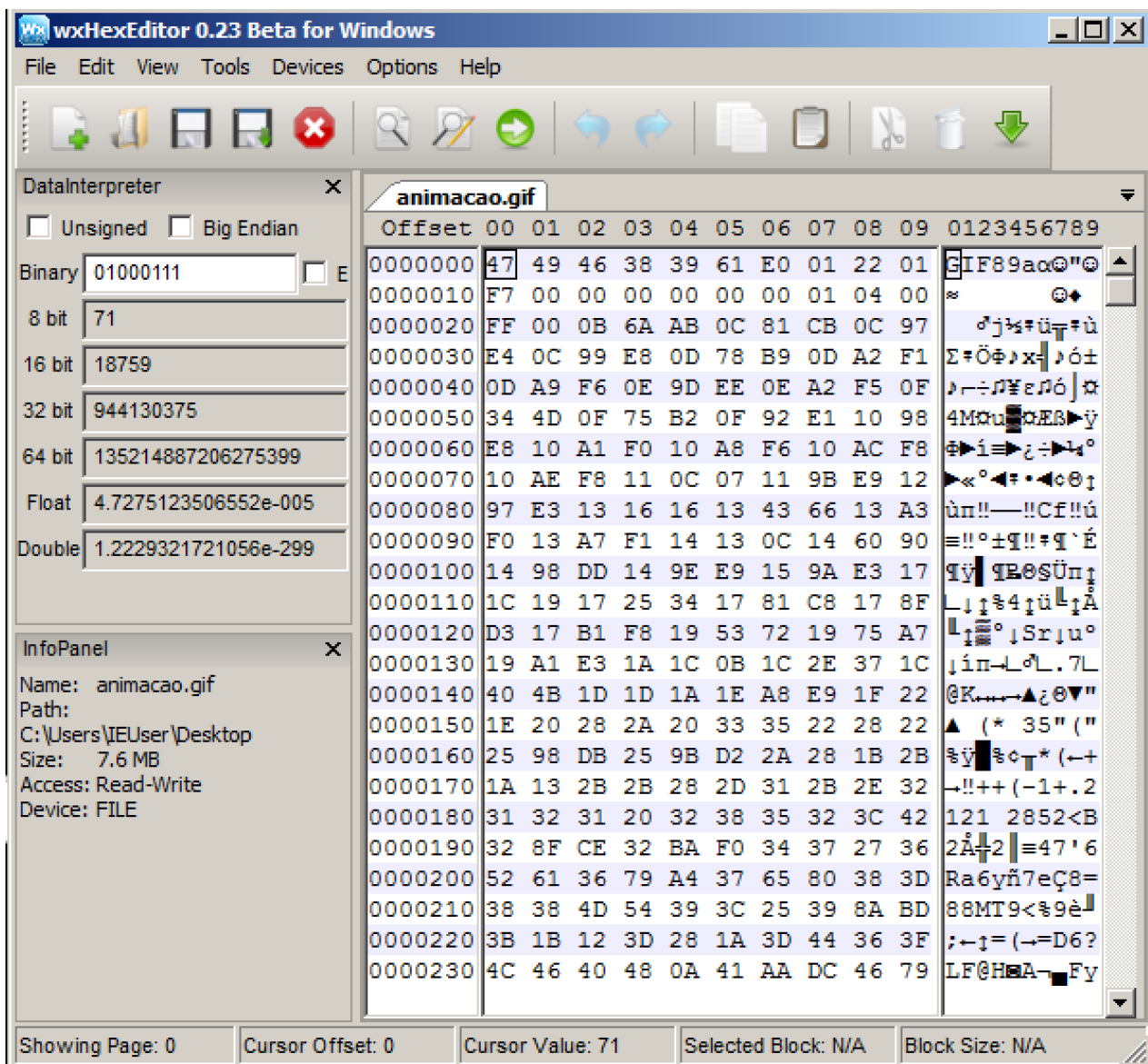
```
1 $ file arquivo
2 arquivo: ASCII text, with no line terminators
3
4 $ file -i arquivo
5 arquivo: text/plain; charset=us-ascii
```

Como o leitor pode notar, a opção **-i** o **file** também nos dá o tipo MIME (do inglês *Multipurpose Internet Mail Extensions*), que é o tipo de arquivo utilizado na Internet para *downloads*, e-mail, etc.

# Formatos

Os arquivos de texto, sejam eles ASCII ou UNICODE são tipos de arquivos bem simples. Começaremos agora a avaliar tipos de arquivos mais complexos. Acontece que para um programa **salvar** ou **abrir** um determinado tipo de arquivo, é essencial que este conheça seu **formato**. Tomemos por exemplo o formato GIF, mas agora ao invés de utilizar o **hd** para inspecionar seu conteúdo, utilizaremos, no Windows, um editor hexadecimal gráfico, o [wxHexEditor](#).

Ao abrir o arquivo GIF no **wxHexEditor**, é exibida uma tela similar a abaixo:



Conteúdo de um arquivo GIF visualizado em hexadecimal

Para entender como este tipo de *software* funciona, vamos começar pelas colunas onde os *bytes* estão organizados:

---

## Offset em disco

Define a **posição** no arquivo, em *bytes*. No exemplo acima, o primeiro *byte* (no *offset* 0) é o 0x47. O segundo é o 0x49 e assim por diante. Por padrão no **wxHexEditor** os *offsets* estão em decimal, não em hexa. Identifique na imagem as afirmações a seguir:

- O *byte* no *offset* 122 é o 0xb1.
- A *word* no *offset* 226 é 0x3d 0x44.
- O *byte* no *offset* 0x11 (17 em decimal) é 1.



No **wxHexEditor** você pode exibir os *offsets* em hexadecimal clicando com o botão direito do mouse sobre eles. Outros editores como o **XVI32** já exibem o *offset* em hexadecimal por padrão.

---

## Conteúdo

A próxima coluna exibe os *bytes* em si. Sem segredos. Por serem editores hexadecimais, programas como o wxHexEditor exibem o conteúdo do arquivo em bytes hexadecimais separados por espaços, mas é importante lembrar que o conteúdo de um arquivo é uma sequência de bits em disco ou em outro dispositivo de armazenamento que, quando aberto num editor, tem seu conteúdo copiado para a memória. A maneira como estes bytes serão visualizados fica a cargo do programa que o abre. Por exemplo, se o editor exibisse os bytes em decimal, os primeiros dois bytes (0x47 e 0x49) seriam 71 e 73. Se fosse em binário, seria 1000111 e 1001001.

A esta altura o leitor já deve ter percebido que um número pode ser expresso de várias maneiras, no entanto, o sistema hexadecimal é bem otimizado para o engenheiro reverso.

---

## Representação textual

Na terceira coluna o **wxHexEditor** nos apresenta a interpretação ASCII de cada *byte* na segunda coluna, ou seja, se o *byte* em questão estiver dentro da faixa de caracteres ASCII imprimíveis (de 0x20 à 0x7e), sua representação é exibida. Para valores fora desta faixa, o **wxHexEditor** exibe um espaço em branco.

Há dezenas de outros editores hexadecimais disponíveis, inclusive alguns visualizadores de linha de comando. Vale consultar o Apêndice C - Ferramentas e testar alguns. Se o leitor tiver curiosidade de saber como funciona um visualizador hexadecimal, recomendo olhar o código do [hdump](#), visualizador para linha de comando que implementei em C para funcionar em Windows, Linux e macOS.

---

## Exemplo do formato GIF

De volta ao formato, é importante ressaltar que tanto o programador do software que salva um determinado tipo de arquivo quanto o programador de um software que visualiza tal tipo precisam conhecê-lo bem. Como exemplificamos com o formato GIF, vamos ver como este formato é definido. Em geral, os formatos são definidos por campos (faixas de *bytes*) de tamanho fixo ou variável, que podem assumir um certo valor. Para entendê-los, precisamos da documentação deste formato (no caso, do GIF). Conforme sua [especificação](#), o formato GIF segue, dentre outras, as seguintes regras:

Byte offset (posição no arquivo)	Tamanho do campo em bytes	Valor em hexadecimal	Descrição
0	6	47 49 46 38 39 61	Cabeçalho
6	2	<variável>	Largura em pixels
8	2	<variável>	Altura em pixels

Seguindo esta tabela fornecida por quem desenhou o formato GIF e olhando o conteúdo do arquivo de exemplo na imagem anterior, podemos verificar que o primeiro campo, de 6 *bytes*, casa exatamente com o que está definido no padrão. Os *bytes* são a sequência 0x47, 0x49, 0x46, 0x38, 0x39 e 0x61 que representam a sequência em ASCII GIF89a. É bem comum ao definir formatos de arquivo que o primeiro campo, normalmente chamado de cabeçalho (*header*) ou número mágico (*magic number*) admita como valor uma representação ASCII que dê alguma indicação de que tipo de arquivo se trata. Por exemplo, os tipos de arquivo ZIP possuem o *magic number* equivalente ao texto **PK**. Já o tipo de arquivo RAR começa com os *bytes* equivalentes ao texto **Rar!**. Não é uma regra, mas é comum.



No exemplo do formato GIF o tamanho do primeiro campo é de 6 *bytes*, mas nem todo *magic number* possui este tamanho. Na verdade, não há regra.

Logo após o primeiro campo, temos o segundo campo, que define a largura em *pixels* da imagem GIF, segundo sua documentação. Este campo possui 2 *bytes* e, na imagem de exemplo, são os *bytes* 0xe0 e 0x01. Aqui cabe explicar um conceito valioso que é o **endianess**. Acontece que na arquitetura Intel os *bytes* de um número inteiro são armazenados de trás para frente (chamado de **little endian**). Sendo assim a leitura correta da largura em *pixels* deste GIF é 0x01e0, ou simplesmente 0x1e0 (já que zero à esquerda não conta), que é 480 em decimal.

O próximo campo, também de 2 *bytes*, diz respeito a altura em *pixels* da imagem GIF e possui o valor 0x122 (já lendo os *bytes* de trás para frente conforme explicado), que é 290 em decimal. É correto dizer então que esta é uma imagem de 480 x 290 *pixels*.

É por isso que alguns sistemas operacionais, com o GNU/Linux, não consideram a extensão de arquivo como sendo algo importante para definir seu tipo. Na verdade, o conteúdo do arquivo o define.

Não seguiremos com toda a interpretação do formato GIF pois este foge ao escopo de estudo de engenharia reversa, mas vamos seguir a mesma lógica para entender o formato de arquivos executáveis do sistema Windows, objeto de estudo do próximo capítulo.

# O formato PE

Como explicado no capítulo anterior, a maioria dos tipos de arquivo que trabalhamos possuem uma especificação. Com os arquivos executáveis no Windows não é diferente: eles seguem a especificação do formato PE (*Portable Executable*) que conheceremos agora.

O formato PE é o formato de arquivo executável atualmente utilizado para os programas no Windows, isso inclui os famosos arquivos EXE mas também DLL, OCX, CPL e SYS. Seu nome deriva do fato de que o formato não está preso a uma arquitetura de *hardware* específica.

Os programas que criam estes programas, chamados compiladores precisam respeitar tal formato e o programa que os interpreta, carrega e inicia sua execução (chamado de *loader*) precisa entendê-lo também.

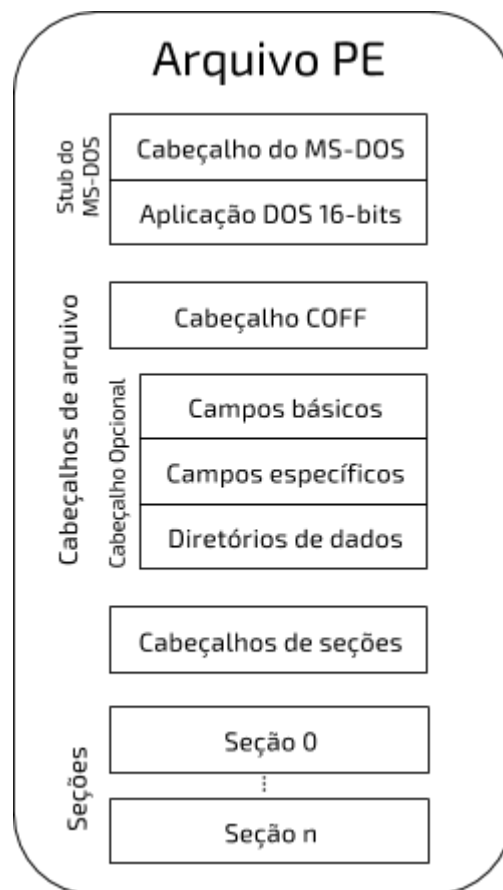


A documentação completa do formato PE é mantida pela própria Microsoft e está disponível [online](#). Uma versão em formato do Word esteve disponível por um tempo e a salvamos [aqui](#).

Assim como o formato GIF e outras especificações de formato de arquivo, o formato PE possui cabeçalhos, que possuem campos e valores possíveis. Outro conceito importante é o de seções.

A estrutura geral de um arquivo PE é apresentada na imagem abaixo:





Estrutura de um arquivo PE

Conheceremos agora os cabeçalhos mais importantes para este primeiro contato com a engenharia reversa e, em seguida, as seções de um arquivo PE.

# Cabeçalhos

Cabeçalhos, como o nome sugere, são áreas de dados no início de um arquivo. Basicamente são definidos por conjuntos de diferentes campos, que admitem valores.

Cada campo possui um **tipo** que também já define seu tamanho. Por exemplo, se dissermos que o primeiro campo de um primeiro cabeçalho é do tipo WORD, estamos afirmando que este tem 2 *bytes* de tamanho, conforme a tabela a seguir:

Nomenclatura Intel	Nome do tipo em C (x86)	Tamanho em <i>bytes</i>
BYTE	char	1
WORD	short int	2
DWORD	int	4
QWORD	long int	8

Há também os campos que possuem o que chamamos de máscara de bits. Neste campos, cada *bit* de seus *bytes* podem significar alguma coisa. Um bom exemplo é o campo "Characteristics" do cabeçalho de seções do arquivo PE, que veremos mais adiante.

# MS-DOS

Não estranhe o nome deste cabeçalho. Ele é parte do que chamamos de **stub do MS-DOS**: um executável completo de MS-DOS presente no início de todo executável PE, para fins de retrocompatibilidade.

Sendo assim, todo arquivo PE começa com este cabeçalho, que é definido pela seguinte estrutura:

```
1  typedef struct {
2      uint16_t e_magic;
3      uint16_t e_cblp;
4      uint16_t e_cp;
5      uint16_t e_crlc;
6      uint16_t e_cparhdr;
7      uint16_t e_minalloc;
8      uint16_t e_maxalloc;
9      uint16_t e_ss;
10     uint16_t e_sp;
11     uint16_t e_csum;
12     uint16_t e_ip;
13     uint16_t e_cs;
14     uint16_t e_lfarlc;
15     uint16_t e_ovno;
16     uint16_t e_res[4];
17     uint16_t e_oemid;
18     uint16_t e_oeminfo;
19     uint16_t e_res2[10];
20     uint32_t e_lfanew;
21 } IMAGE_DOS_HEADER;
```

Este cabeçalho possui 64 *bytes* de tamanho. Para chegar a esta conclusão basta somar o tamanho de cada campo, onde **uint16\_t** é um **tipo** na linguagem C que define uma variável de 16 *bits* ou 2 *bytes*. Os seguintes campos variam deste tamanho:

- `uint16_t e_res[4]` que é um *array* de 4 campos de 16 *bits*, totalizando em 64 *bits* ou 8 *bytes*.
- `uint16_t e_res2[10]` que é um *array* de 10 campos de 16 *bits*, totalizando em 160 *bits* ou 20 *bytes*.
- `uint32_t e_lfanew` que é um campo de 32 *bits* ou 4 *bytes*.

Os outros 16 campos possuem o tamanho de um **uint16\_t** (16 *bits* ou 2 *bytes*). Então somando os tamanhos de todos os campos em *bytes*, temos:

```
1 $ echo 8+20+4+16*2 | bc
2 64
```

Por ser um cabeçalho ainda presente no formato PE somente por questões de compatibilidade com o MS-DOS, não entraremos em muitos detalhes, mas estudaremos alguns de seus campos a seguir.

---

## e\_magic

Este campo de 2 *bytes* sempre contém os valores 0x4d e 0x5a, que são os caracteres 'M' e 'Z' na tabela ASCII. Portanto é comum verificar que todo arquivo executável do Windows que segue o formato PE começa com tais valores, que representam as iniciais de Mark Zbikowski, um dos criadores deste formato para o MS-DOS.

Podemos utilizar um visualizador hexadecimal como o **hexdump** no Linux para verificar tal informação. Vamos pedir, por exemplo, os primeiros 16 *bytes* de um arquivo `putty.exe`:

```
1 $ hd -n16 putty.exe
2 00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ.....
```

Perceba os *bytes* 0x4d e 0x5a logo no início do arquivo.



O **hexdump** exibe um caractere de ponto (.) na terceira coluna quando o *byte* não está na faixa ASCII imprimível, ao contrário do **wxHexEditor** que exibe um caractere de espaço em branco.

## e\_lfanew


O próximo campo importante para nós é o *e\_lfanew*, um campo de 4 *bytes* cujo valor é a posição no arquivo do que é conhecido por **assinatura PE**, uma sequência fixa dos seguintes 4 *bytes*: 50 45 00 00.

Como o cabeçalho do DOS possui um tamanho fixo, seus campos estão sempre numa **posição** fixa no arquivo. Isso se refere aos campos e não a seus valores que, naturalmente, podem variar de arquivo para arquivo. No caso do *e\_lfanew*, se fizermos as contas, veremos que **ele sempre estará na posição 0x3c** (ou 60 em decimal), já que ele é o último campo de 4 *bytes* de um cabeçalho de 64 *bytes*.

Para ver o valor deste campo rapidamente podemos pedir ao **hexdump** que pule (opção **-s** de *skip*) então 0x3c *bytes* antes de começar a imprimir o conteúdo do arquivo em hexadecimal. No comando a seguir também limitamos a saída em 16 *bytes* com a opção **-n** (*number*):

```
1 $ hd -s 0x3c -n16 putty.exe
2 0000003c f8 00 00 00 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c |.....!|
```

O número de 32 *bits* na posição 0x3c é então 0x000000f8 ou simplesmente 0xf8 (lembre-se do *little endian*). Este é então o endereço da assinatura PE, que consiste numa sequência dos seguintes 4 *bytes*: 0x50 0x45 0x00 0x00.

 Perceba que os dois primeiros *bytes* na assinatura PE possuem representação ASCII justamente das letras 'P' e 'E' maiúsculas. Sendo assim, essa assinatura pode ser escrita como "PE\0\0", no estilo C string.

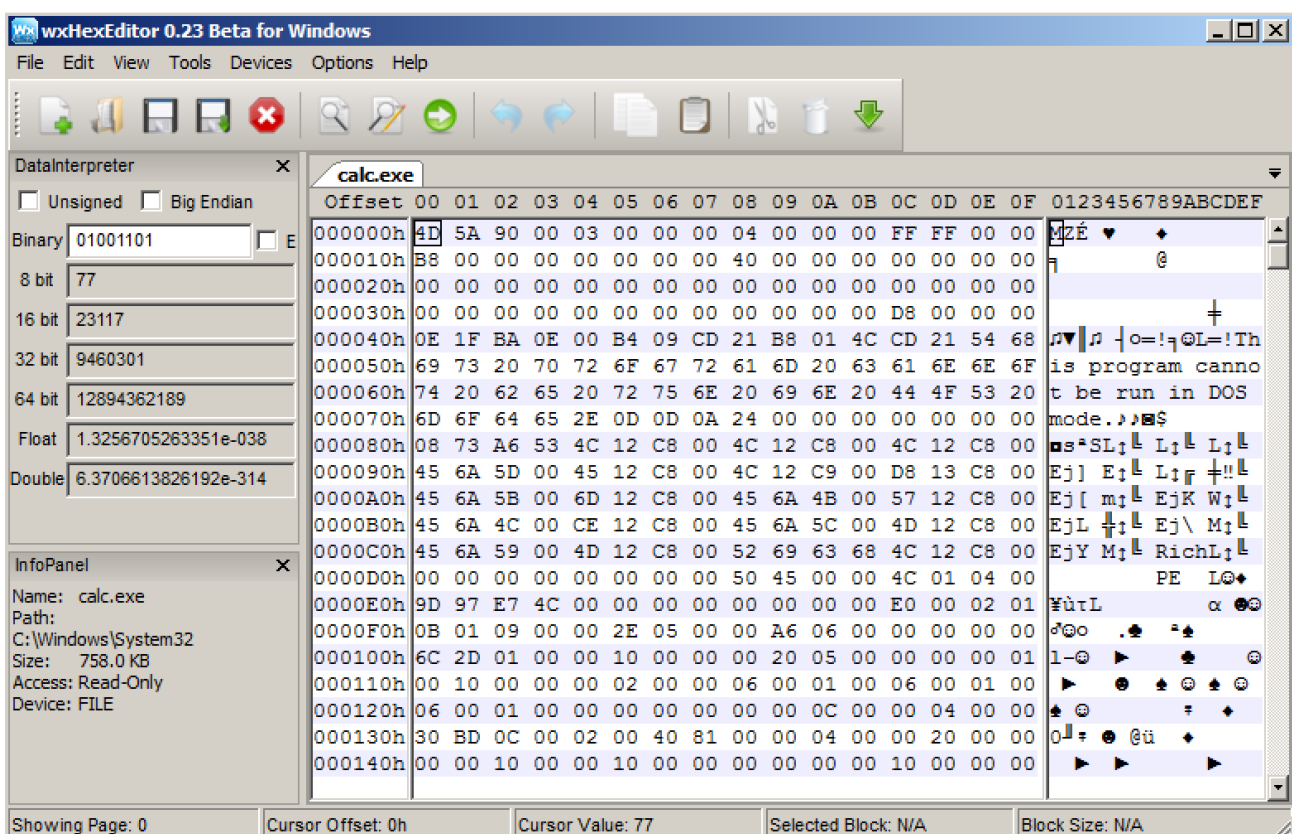
Logo após o cabeçalho há o código do programa que vai imprimir na tela uma mensagem de erro caso um usuário tente rodar este arquivo PE no MS-DOS. Normalmente o texto impresso na tela é:

This program cannot be run in DOS mode.

Depois disso o programa sai. Mas este pequeno programa de MS-DOS é adicionado pelo compilador (pelo *linker* mais especificamente) e seu conteúdo pode variar, pois não há um padrão rígido a ser seguido.

## Exercícios

Para por em prática a análise desta primeira parte do arquivo PE, vamos abrir o executável da calculadora do Windows (normalmente em *C:\Windows\System32\calc.exe*) no **wxHexEditor**, conforme ilustra a imagem abaixo:



Visualização hexadecimal do calc.exe com o wxHexEditor

Analisando o que vemos:

- Logo no início do arquivo, há o número mágico "MZ".
- Na posição 0x3c, ou seja, no campo *e\_lfanew*, há o endereço da assinatura PE (0xd8 no caso deste executável).
- Logo após os 4 *bytes* do campo *e\_lfanew*, começa o código do programa de MS-DOS, no offset 0x40, com uma sequência de *bytes* que não fazem sentido para nós por enquanto (veja que o texto impresso na tela pelo DOS stub é todavia bem visível).
- Finalmente, na posição 0xd8 encontramos a assinatura PE\0\0. Aqui sim, começa o formato PE propriamente dito.

# COFF

Imediatamente após a assinatura PE temos o cabeçalho COFF (*Common Object File Format Specification*) às vezes chamado simplesmente de Cabeçalho do Arquivo (*File Header*) ou mesmo Cabeçalho do Arquivo PE (*PE File Header*). Trata-se de um cabeçalho especificado antes mesmo do formato PE para o sistema operacional VAX/VMS (depois chamado de OpenVMS) da DEC (empresa comprada pela Compaq, que por sua vez, foi comprada pela HP) na década de 70. A razão pela qual a Microsoft teria aproveitado o formato COFF é que boa parte dos engenheiros do time que desenvolveu o Windows NT trabalhavam para a DEC antes.

O cabeçalho COFF possui apenas 20 *bytes* e é representado pela seguinte estrutura:

```
1  typedef struct {
2      uint16_t Machine;
3      uint16_t NumberOfSections;
4      uint32_t TimeDateStamp;
5      uint32_t PointerToSymbolTable;
6      uint32_t NumberOfSymbols;
7      uint16_t SizeOfOptionalHeader;
8      uint16_t Characteristics;
9  } IMAGE_FILE_HEADER, IMAGE_COFF_HEADER;
```

Vamos à definição dos campos importantes para nós:

---

## Machine

Campo de 2 *bytes* que define a arquitetura da máquina para qual o programa foi construído. Valores comuns são 0x14c (Intel i386 ou compatíveis) e 0x8664 (AMD64 ou compatíveis). A tabela completa está disponível na documentação oficial.

---

## NumberOfSections



Também de 2 *bytes*, o valor deste campo é o número de seções que o arquivo PE em questão possui. As seções serão estudadas mais a frente.

---

## TimeStamp

Este é um número de 32 *bits* que define o número de segundos desde à meia-noite do dia 1 de Janeiro de 1970, conhecido também por *Epoch time*. Com este valor é possível saber quando o arquivo foi criado. Para mais informações, veja a dica [Convertendo Epoch no Linux](#).

Vale lembrar que este campo não é utilizado pelo *loader* de arquivos PE no Windows e seu valor pode ser alterado após a compilação, logo, não é 100% confiável, ou seja, você não pode **garantir** que um binário PE foi compilado na data e hora informadas pelo valor neste campo.

---

## SizeOfOptionalHeader

Contém o tamanho do próximo cabeçalho, conhecido como Cabeçalho Opcional, que estudaremos muito em breve.

---

## Characteristics

Campo que define alguns atributos do arquivo. Este campo é uma **máscara de bits**, ou seja, cada *bit* desses 2 *bytes* diz respeito à uma característica específica do binário. Não cabe aqui explicar todos os possíveis valores, mas os mais comuns são:

Bit	Nome	Comentários
2	IMAGE_FILE_EXECUTABLE_IMAGE	Obrigatório para arquivos executáveis
9	IMAGE_FILE_32BIT_MACHINE	Arquivo de 32-bits

---

Analise novamente o (*dump* hexadecimal do executável da calculadora)  
[dos.md#exercicios] considere que:

- Logo após a assinatura PE na posição 0xd8 temos o primeiro campo do cabeçalho COFF que é o **Machine**. Ele é um campo de 2 *bytes* conforme já dito, então os *bytes* 0x4c e 0x01 definem seu valor. Considerando o *endianness*, chegamos ao valor 0x14c, que define que este executável foi criado para máquinas Intel i386 ou compatíveis.
- Em seguida, na posição 0xde, temos o **NumberOfSections** que é 4.
- Depois vem o campo **TimeDateStamp** com o número inteiro de 32 *bits* (4 *bytes*) sem sinal 0x4ce7979d que é 1290246045 em decimal. Podemos usar o comando **date** do Linux para converter para data e hora atuais:

```
1 $ date -ud @1290246045
2 Sat Nov 20 09:40:45 UTC 2010
```

- Pulamos então 8 *bytes* referentes aos campos **PointerToSymbolTable** e **NumberOfSymbols** (normalmente zerados mesmo), encontrando o valor da *word* **SizeOfOptionalHeader** em 0xec de valor 0xe0.
- A próxima *word* é o valor do campo **Characteristics**, que neste arquivo é 0x102. Convertendo para binário temos o valor 100000010 (*bits* 2 e 9 *setados*) significando que o arquivo é um executável de 32-bits.

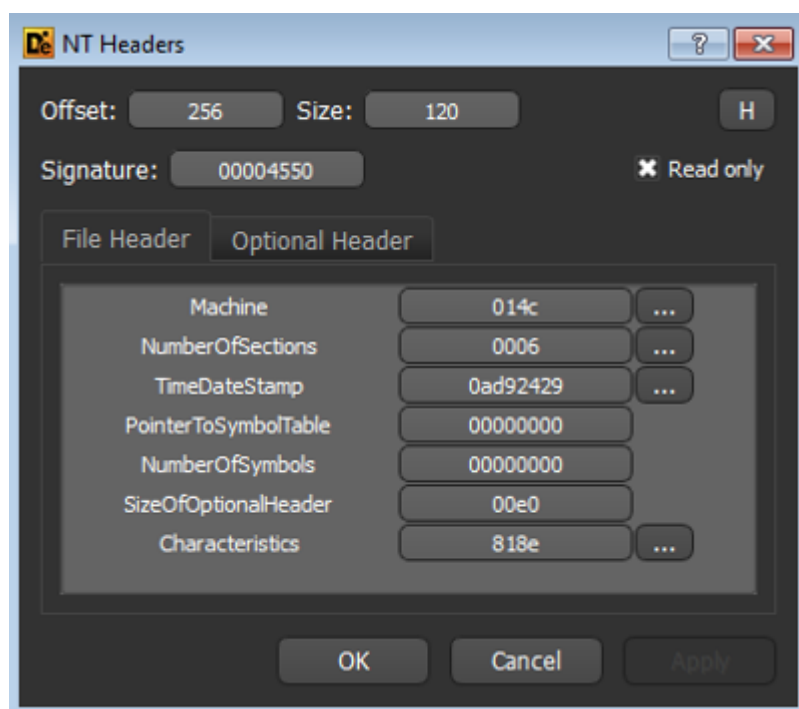
❗ Em algumas referências o leitor encontrará o cabeçalho COFF como parte do cabeçalho NT (IMAGE\_NT\_HEADER), onde o primeiro campo é chamado de *Signature Bytes*, que é onde fica a assinatura PE para binários PE, mas também pode conter os bytes equivalentes das strings NE, LE ou MZ (executáveis puros de MS-DOS). Na verdade o COFF é uma especificação completa para arquivos do tipo "código-objeto", mas não exploraremos seu uso "fora" do formato PE neste livro.

## Exercícios

Usando o comando **readpe** do toolkit do **pev**, exiba o cabeçalho COFF do binário CRACKME.EXE. Você deve ver algo assim:

```
1 C:\Users\Homer\Desktop>readpe --header coff CRACKME.EXE
2
3 COFF/File header
4 Machine:                0x14c IMAGE_FILE_MACHINE_I386
5 Number of sections:      6
6 Date/time stamp:        182002729 (Wed, 08 Oct 1975 12:18:49 UTC)
7 Symbol Table offset:    0
8 Number of symbols:      0
9 Size of optional header: 0xe0
10 Characteristics:       0x818e
11                         IMAGE_FILE_EXECUTABLE_IMAGE
12                         IMAGE_FILE_LINE_NUMS_STRIPPED
13                         IMAGE_FILE_LOCAL_SYMS_STRIPPED
14                         IMAGE_FILE_BYTES_REVERSED_LO
15                         IMAGE_FILE_32BIT_MACHINE
16                         IMAGE_FILE_BYTES_REVERSED_HI
```

Com o **DIE**, é preciso carregar o CRACKME.EXE nele, clicar no botão **PE (Alt+P)**, na aba **NT Headers** e por fim, na aba **File Header**. Você deve ver uma janela como a abaixo:



## Cabeçalho COFF exibido pelo DIE

Os botões com "..." localizados ao lado direito de vários valores de campos provêm informações adicionais sobre tais valores. Não deixe de experimentar. ;)

# Opcional

Não parece, mas este cabeçalho é muito importante. Seu nome deve-se ao fato de que ele é opcional para arquivos objeto, mas é requerido em arquivos executáveis, nosso foco de estudo. Seu tamanho não é fixo. Ao contrário, é definido pelo campo *SizeOfOptionalHeader* do cabeçalho COFF, que vimos anteriormente. Sua estrutura para arquivos PE de 32-bits, também chamados de PE32, é a seguinte:

```
1  typedef struct {
2      uint16_t Magic;
3      uint8_t MajorLinkerVersion;
4      uint8_t MinorLinkerVersion;
5      uint32_t SizeOfCode;
6      uint32_t SizeOfInitializedData;
7      uint32_t SizeOfUninitializedData;
8      uint32_t AddressOfEntryPoint;
9      uint32_t BaseOfCode;
10     uint32_t BaseOfData;
11     uint32_t ImageBase;
12     uint32_t SectionAlignment;
13     uint32_t FileAlignment;
14     uint16_t MajorOperatingSystemVersion;
15     uint16_t MinorOperatingSystemVersion;
16     uint16_t MajorImageVersion;
17     uint16_t MinorImageVersion;
18     uint16_t MajorSubsystemVersion;
19     uint16_t MinorSubsystemVersion;
20     uint32_t Reserved1;
21     uint32_t SizeOfImage;
22     uint32_t SizeOfHeaders;
23     uint32_t CheckSum;
24     uint16_t Subsystem;
25     uint16_t DllCharacteristics;
26     uint32_t SizeOfStackReserve;
27     uint32_t SizeOfStackCommit;
28     uint32_t SizeOfHeapReserve;
29     uint32_t SizeOfHeapCommit;
30     uint32_t LoaderFlags;
31     uint32_t NumberOfRvaAndSizes;
32     IMAGE_DATA_DIRECTORY DataDirectory[MAX_DIRECTORIES];
33 } IMAGE_OPTIONAL_HEADER_32;
```

---

Vamos analisar agora os campos mais importantes no nosso estudo:

## Magic

O primeiro campo, de 2 bytes, é um outro número mágico que identifica o tipo de executável em questão. O valor 0x10b significa que o executável é um PE32 (executável PE de 32-bits), enquanto o valor 0x20b diz que é um PE32+ (executável PE de 64-bits).



Perceba que a Microsoft chama os executáveis de PE de 64-bits de PE32+ e não de PE64.

## AddressOfEntryPoint

Este é talvez o campo mais importante do cabeçalho opcional. Nele está contido o endereço do ponto de entrada (*entrypoint*), abreviado EP, que é onde o código do programa deve começar. Para arquivos executáveis este endereço é relativo à base da imagem (campo que veremos a seguir). Para bibliotecas, ele não é necessário e pode ser zero, já que as funções de biblioteca podem ser chamadas arbitrariamente.

## ImageBase

Imagem é como a Microsoft chama um arquivo executável (para diferenciar de um código-objeto) quando vai para a memória. Neste campo está o endereço de memória que é a base da imagem, ou seja, onde o programa será carregado em memória. Para arquivos executáveis (.EXE) o padrão é 0x400000. Já para bibliotecas (.DLL), o padrão é 0x10000000, embora executáveis do Windows como o *calc.exe* também apresentem este valor no *ImageBase*.

## Subsystem

Este campo define o tipo de subsistema necessário para rodar o programa. Valores interessantes para nós são:

- 0x002 - Windows GUI (*Graphical User Interface*) - para programas gráficos no Windows (que usam janelas, etc).
- 0x003 - Windows CUI (*Character User Interface*) - para programas de linha de comando.

---

## DllCharacteristics

Ao contrário do que possa parecer, este campo não é somente para DLL's. Ele está presente e é utilizado para arquivos executáveis também. Assim como o campo *Characteristics* do cabeçalho COFF visto anteriormente, este campo é uma máscara de *bits* com destaque para os possíveis valores:

Bit	Nome
6	IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
8	IMAGE_DLLCHARACTERISTICS_NX_COMPAT

O estado *bit* 6 nos diz se a randomização de endereços de memória, também conhecida por ASLR (*Address Space Layout Randomization*), está ativada para este binário, enquanto o estado do *bit* 8 diz respeito ao DEP (*Data Execution Prevention*), também conhecido pela sigla NX (*No eXecute*). O estudo aprofundado destes recursos foge do escopo inicial deste livro, mas é importante que saibamos que podemos desabilitar tais recursos simplesmente forçando estes *bits* para zero.

---

## Diretórios de dados

Ainda como parte do cabeçalho opcional, temos os diretórios de dados, ou *Data Directories*. São 16 diretórios ao todo, cada um com uma função. Concentraremos, no entanto, nos mais importantes para este estudo inicial. A estrutura de cada diretório de dados é conhecida por **IMAGE\_DATA\_DIRECTORY** e tem a seguinte definição:

```
1 typedef struct _IMAGE_DATA_DIRECTORY {
2     uint32_t    VirtualAddress;
3     uint32_t    Size;
4 } IMAGE_DATA_DIRECTORY;
```

Vejamos agora alguns diretórios:

---

## Export Table

O primeiro diretório de dados aponta para a tabela de **exports**, ou seja, de funções exportadas pela aplicação. Por isso mesmo sua presença (campos *VirtualAddress* e *Size* diferentes de zero) é muito comum em bibliotecas.

O campo *VirtualAddress* aponta para uma outra estrutura chamada EDT (*Export Directory Table*), que contém os nomes das funções exportadas e seus endereços, além de um ponteiro para uma outra estrutura, preenchida em memória, chamada de EAT (*Export Address Table*). Entenderemos mais sobre estas e outras estruturas em breve.

---

## Import Table

Sendo a contraparte da *Export Table*, a *Import Table* aponta para a tabela de **imports**, ou seja, de funções importadas pela aplicação. O campo *VirtualAddress* aponta para a IDT (*Import Directory Table*), que tem um ponteiro para a IAT (*Import Address Table*), que estudaremos mais à frente.

---

## Resource Table

Aponta para uma estrutura de árvore binária que armazena todos os *resources* num executável (ícones, janelas, strings - principalmente quando o programa suporta vários idiomas), etc. Também estudaremos um pouco mais sobre estes "recursos" no futuro.



# Cabeçalhos das seções

Após o cabeçalho opcional, encontramos os cabeçalhos das seções (estas serão explicadas no próximo capítulo). Neste cabeçalho há um *array* de estruturas como a seguir:

```
1  typedef struct {
2      uint8_t Name[SECTION_NAME_SIZE];
3      uint32_t VirtualSize;
4      uint32_t VirtualAddress;
5      uint32_t SizeOfRawData;
6      uint32_t PointerToRawData;
7      uint32_t PointerToRelocations;
8      uint32_t PointerToLinenumbers; // descontinuado
9      uint16_t NumberOfRelocations;
10     uint16_t NumberOfLinenumbers; // descontinuado
11     uint32_t Characteristics;
12 } IMAGE_SECTION_HEADER;
```

Cada estrutura define uma seção no executável e a quantidade de estrutura (quantidade de elementos neste *array*) é igual ao número de seções no executável, definido no campo *NumberOfSections* do cabeçalho COFF. Vamos aos campos importantes:

---

## Name

Este campo define o nome da seção. Como é um *array* de 8 elementos do tipo **uint8\_t**, este nome está limitado à 8 caracteres. A string **.text** por exemplo ocupa apenas 5 *bytes*, então os outros 3 devem estar zerados. Há suporte à UTF-8 para estes nomes.

---

## VirtualSize

O tamanho em *bytes* da seção depois de ser mapeada (carregada) em memória pelo *loader*. Se este valor for maior que o valor do campo **SizeOfRawData**, os *bytes* restantes

são preenchidos com zeros.

## VirtualAddress

O endereço relativo à base da imagem (campo **ImageBase** do cabeçalho Opcional) quando a seção é carregada em memória. Por exemplo, se para uma seção este valor é 0x1000 e o valor de **ImageBase** é 0x400000, quando carregada em memória esta seção estará no endereço 0x401000. Para chegar nesta conclusão basta somar os dois valores.

---

## SizeOfRawData

Tamanho em *bytes* da seção no arquivo PE, ou seja, antes de ser mapeada em memória. Alguns autores também usam a expressão "tamanho em disco" ou simplesmente "tamanho da seção".

---

## PointerToRawData

O *offset* em disco da seção no arquivo. É correto dizer que aponta para o primeiro *byte* da seção. Por exemplo, se para dada seção este valor é 0x400 e o valor do campo **SizeOfRawData** é 0x1800, para ver somente seu conteúdo em hexadecimal com o **hexdump** poderíamos fazer:


```
$ hd -s 0x400 -n 0x1800 arquivo.exe > secao.txt
```

## Characteristics

Este é um campo define algumas *flags* para a seção, além de as permissões em memória que ela deve ter quando for mapeada pelo *loader*. Ele possui 32-bits, onde alguns

significam conforme a tabela a seguir:

Bit	Nome da flag	Descrição
5	IMAGE_SCN_CNT_CODE	A seção contém código executável
6	IMAGE_SCN_CNT_INITIALIZED_DATA	A seção contém dados inicializados
7	IMAGE_SCN_CNT_UNINITIALIZED_DATA	A seção contém dados não inicializados
29	IMAGE_SCN_MEM_EXECUTE	Terá permissão de execução
30	IMAGE_SCN_MEM_READ	Terá permissão de leitura
31	IMAGE_SCN_MEM_WRITE	Terá permissão de escrita

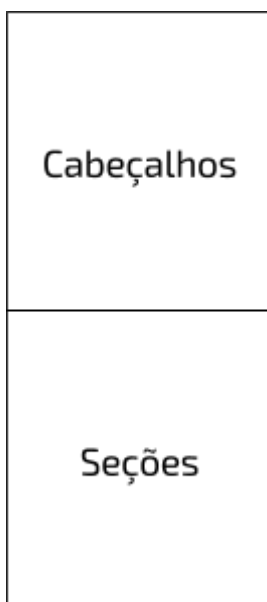
 As *flags* que contém o texto "MEM" no nome dizem respeito às permissões que a seção terá quando mapeada em memória. De acordo com elas o SO vai *setar* as permissões nas páginas de memória nas quais a seção é carregada.

É importante notar que campos como o **Characteristics** são o que chamamos de máscaras de *bits*. Por exemplo, a tabela anterior diz que se o *bit* 30 deste campo está *setado* (seu valor é 1), então esta seção terá permissão de leitura quando em memória. O valor de campo **Characteristics** seria então 01000000000000000000000000000000 em binário, mas você provavelmente vai encontrar este valor representado em hexadecimal (0x80000000) nos analisadores de executáveis que for utilizar. Aliás, agora é uma boa hora para abrir o **DIE** e analisar alguns arquivos executáveis, colocando em prática tudo o que foi visto até aqui. ;-)

# Seções

As seções são divisões num binário PE. Uma analogia que torna o conceito de seções simples de entender é a de comparar o binário PE com uma cômoda: as seções seriam suas gavetas. Cada gaveta da cômoda, em teoria, guarda um tipo de dado distinto, e assim é com as seções, apesar de não ser uma regra muito rígida. Elas são necessárias porque diferentes conteúdos exigem diferentes tratamentos quando carregados em memória pelo Sistema Operacional.

Podemos então dizer que um binário PE é completamente definido por cabeçalhos e seções (com seu conteúdo), como na seguinte ilustração:



Versão simplificada do arquivo PE

Como dito, a principal separação que existe entre as seções é em relação a seu conteúdo, que distinguimos entre **código** ou **dados**. Apesar de terem seus nomes ignorados pelo *loader* do Windows, convencionam-se alguns, normalmente iniciados por um ponto. As seções padrão importantes são discutidas a seguir:

---

**.text**

Também nomeada **CODE** em programas compilados com o Delphi, esta seção contém o código executável do programa. Em seu cabeçalho normalmente encontramos as permissões de leitura e execução.

---

## .data

Também chamada de CODE em programas criados com Delphi, esta seção contém dados inicializados com permissão de leitura e escrita. Estes dados podem ser, por exemplo, uma C string declarada e já inicializada. Por exemplo, considere o programa abaixo:

```
1  #include <stdio.h>
2
3  int main(void) {
4      char s[] = "texto grande para o compilador utilizar a seção de dados";
5
6      s[0] = 'T';
7      puts(s);
8      return 0;
9  }
```

A variável local **s** é um *array* de *char* e pode ser alterada a qualquer momento dentro da função *main()*. De fato, o código na linha 6 a altera. Sendo assim, é bem possível que um compilador coloque seu conteúdo numa seção de dados inicializados com permissão tanto para leitura quanto para escrita. ;-)



Apesar de fazer sentido, os compiladores não precisam respeitar tal lógica. O conteúdo da variável **s** no exemplo apresentado pode ser armazenado na seção **.rdata** (ou mesmo na **.text**) e ser manipulado na pilha de memória para sofrer alterações. Não há uma imposição por parte do formato e cada compilador escolhe fazer do seu jeito.

---

## .rdata

Seção que contém dados inicializados, com permissão somente para leitura. Um bom exemplo seria com o programa abaixo:

```
1  int main(void) {  
2      const char s[] = "texto grande para o compilador utilizar a seção de d  
3  
4      puts(s);  
5      return 0;  
6  }
```

Neste caso declaramos a variável **s** como **const**, o que instrui o compilador a armazená-la numa região de memória somente para leitura, casando perfeitamente com a descrição da seção **.rdata**. ;-)

---

## .idata

Seção para abrigar as tabelas de *imports*, comum em todos os binários que importam funções de outras bibliotecas. Possui permissão tanto para leitura quanto para gravação. Entenderemos o motivo em breve.

---

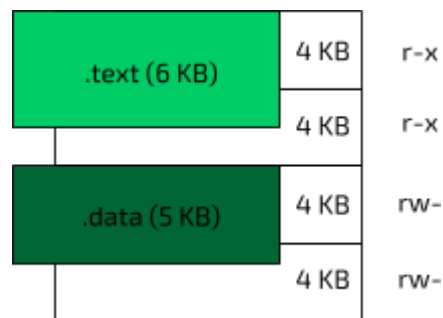
## Alinhamento de seções

O sistema operacional divide a memória RAM em páginas, normalmente de 4 *kilobytes* (ou 4096 *bytes*) nas versões atuais do Windows. Nestas páginas de memória o sistema configura as permissões (leitura, escrita e execução). Os arquivos executáveis precisam ser carregados na memória e cada seção requer um permissionamento diferente. Com isso em mente, considere a seguinte situação hipotética:

- Um executável tem seus cabeçalhos ocupando 2 KB.
- Sua seção `.text` tem 6 KB de tamanho e requer leitura e execução.

- Sua seção `.data` tem 5 KB de tamanho e requer leitura e escrita.
- O tamanho final do executável em disco é 13 KB.

Para mapear este executável em memória e rodá-lo, o SO precisa copiar o conteúdo de suas seções em páginas de memória e *setar* suas permissões de acordo. Analise agora a figura abaixo:



Mapeamento de seções em memória

Perceba na figura que a seção `.text` já ocuparia duas páginas que precisariam ter permissões de leitura e execução. No que sobrou da segunda página, o SO não pode mapear a `.data` pois esta, apesar de compartilhar a permissão de leitura, exige escrita ao invés de execução. Logo, ele precisa mapeá-la na próxima página.

Como consequência, o tamanho total de cada seção em memória é maior que seu tamanho em disco, devido ao que chamamos de **alinhamento de seção**. No cabeçalho opcional existe o campo **SectionAlignment**, que pulei propositalmente. Este campo define qual fator de alinhamento deve ser utilizado para todas as seções do binário quando mapeadas em memória. O padrão é o valor do tamanho da página de memória do sistema.

**i** Como bônus por ter chegado até aqui, deixamos um presente para o leitor: abaixo há um código que, depois de compilado e executado, vai dizer qual o tamanho da página de memória na sua versão do Windows.

pagesize.c

```
1  #include <stdio.h>
2  #include <windows.h>
3
4  int main(void) {
5      SYSTEM_INFO info;
6
7      GetNativeSystemInfo(&info);
8      printf("dwPageSize: %u\n", info.dwPageSize);
9      return 0;
10 }
```



# Import Table

Foi visto que um dos diretórios de dados presentes no cabeçalho opcional é reservado para a *Import Table*, também conhecida por sua sigla IT. Nele há um ponteiro para a IDT (*Import Directory Table*), apontado pelo valor do campo *VirtualAddress*.

A IDT, por sua vez, é um *array* de estruturas do tipo `IMAGE_IMPORT_DESCRIPTOR` definidas a seguir:

```
1  typedef struct {
2      union {
3          uint32_t Characteristics;
4          uint32_t OriginalFirstThunk; // Endereço da ILT
5      } u1;
6      uint32_t TimeDateStamp;
7      uint32_t ForwarderChain;
8      uint32_t Name; // Endereço do nome da DLL
9      uint32_t FirstThunk; // Endereço da IAT
10 } IMAGE_IMPORT_DESCRIPTOR;
```



Não se deve confundir a IDT (*Import Descriptor Table*) com a IDT (*Interrupt Descriptor Table*). Esta última é uma estrutura que mapeia interrupções para seus devidos *handlers*, assunto que não é coberto neste livro.

O número de elementos do *array* de estruturas `IMAGE_IMPORT_DESCRIPTOR` é igual ao número de bibliotecas que o executável PE depende, ou seja, o número de DLL's das quais o executável importa funções. Há ainda um elemento adicional, completamente zero (preenchido com *null bytes*) para indicar o fim do *array*.

---

## OriginalFirstThunk / rvaImportLookupTable

O campo *OriginalFirstThunk* (chamado de *rvaImportLookupTable* em algumas literaturas) aponta para o que chamamos *Import Lookup Table (ILT)*, um *array* de números de 32-bits (64-bits para PE32+), onde seu *bit* mais significativo (*MSB - Most Significant Bit*) define se a função será importada por número ordinal (caso o *bit* seja 1). Já no caso de este *bit* estar zerado, a importação da função dá-se por nome e os outros 31 *bits* do número representam um endereço para uma estrutura que finalmente contém o nome da função.

Sendo assim, o número de elementos do *array* ILT é igual ao número de funções importadas por uma DLL em particular, definida na estrutura `IMAGE_IMPORT_DESCRIPTOR`.

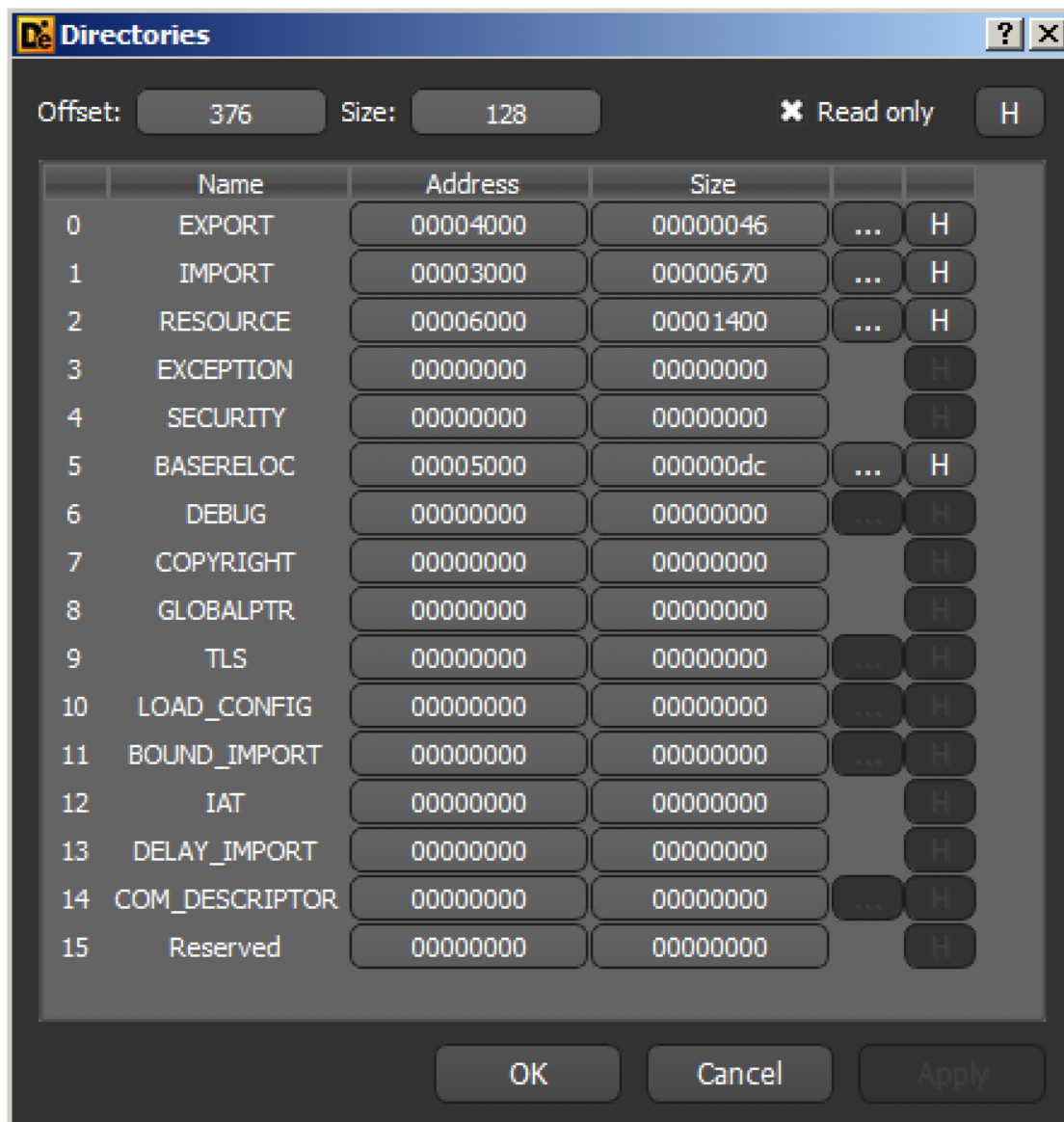
---

## FirstThunk

Este campo aponta finalmente para o que chamamos de *IAT (Import Address Table)*, muito conhecida dos engenheiros reversos. Essa tabela é em princípio idêntica à *Import Lookup Table*, mas no processo de carregamento do executável (*load time*, que estudaremos mais à frente no livro), é preenchida com os endereços reais das funções importadas. Isto porque um executável dinamicamente *linkado* não sabe ainda qual o endereço de cada função de cada DLL que ele precisa chamar.

É importante lembrar o conceito de biblioteca compartilhada aqui. A ideia é ter apenas uma cópia dela carregada em memória e todos os programas que a utilize possam chamar suas funções. Por isso todo este esquema de preenchimento da IAT pelo *loader* é necessário.

Para fixar o conteúdo, é interessante validar tais informações. O exemplo abaixo utiliza o DIE para ver o diretório de dados de um arquivo PE:

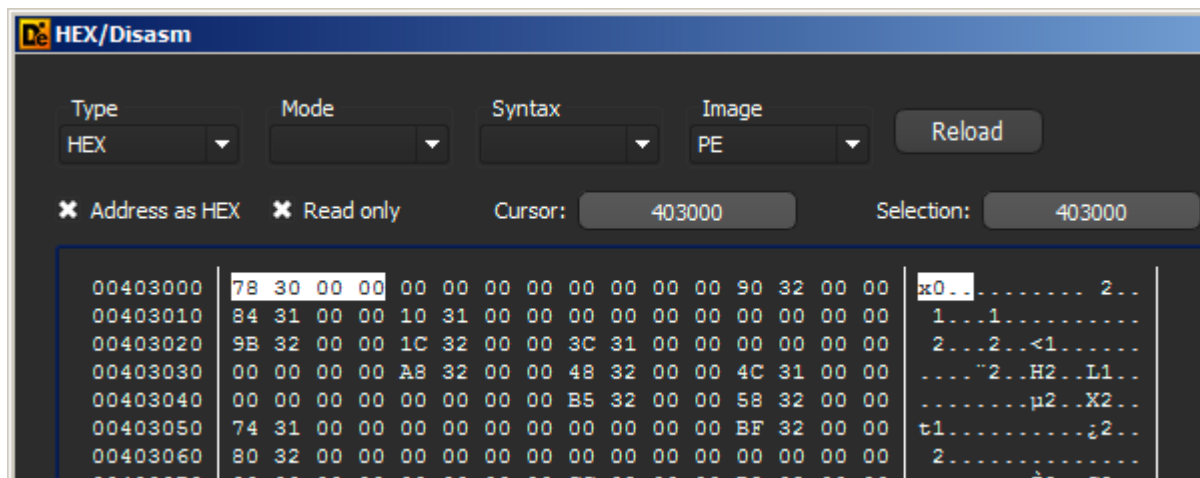


Diretório de dados de um arquivo PE

Neste exemplo o campo *VirtualAddress* do *Import Directory* tem o valor 0x3000. Este é um endereço relativo, que você aprenderá sobre na próxima seção, no entanto, por agora você precisa apenas saber que este endereço é somado ao *ImageBase* para funcionar. No caso, o *ImageBase* deste binário é 0x400000 (muito comum), então o endereço da *Import Descriptor Table*, apontado por este campo *VirtualAddress*, é 0x403000.

⚠ Perceba que o DIE chama o campo *VirtualAddress* dos diretórios apenas de *Address*.

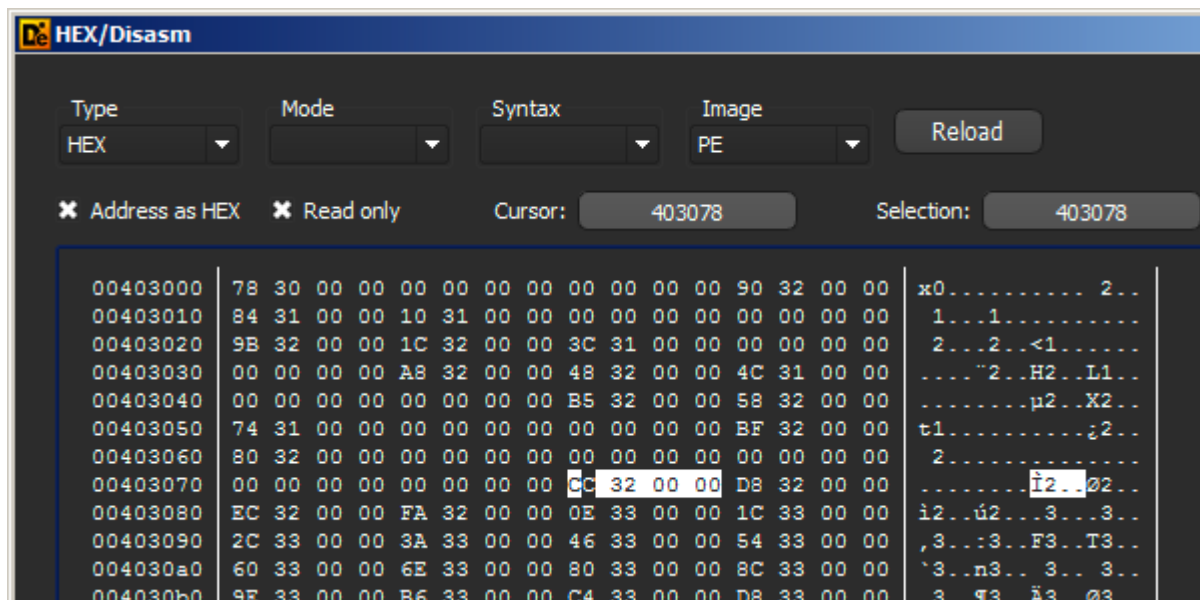
Ao clicar no botão "H" à direita do diretório IMPORT, vemos o conteúdo do primeiro elemento do *array* IDT, que é justamente o campo *OriginalFirstThunk*:



Valor do campo OriginalFirstThunk do primeiro elemento do array IDT

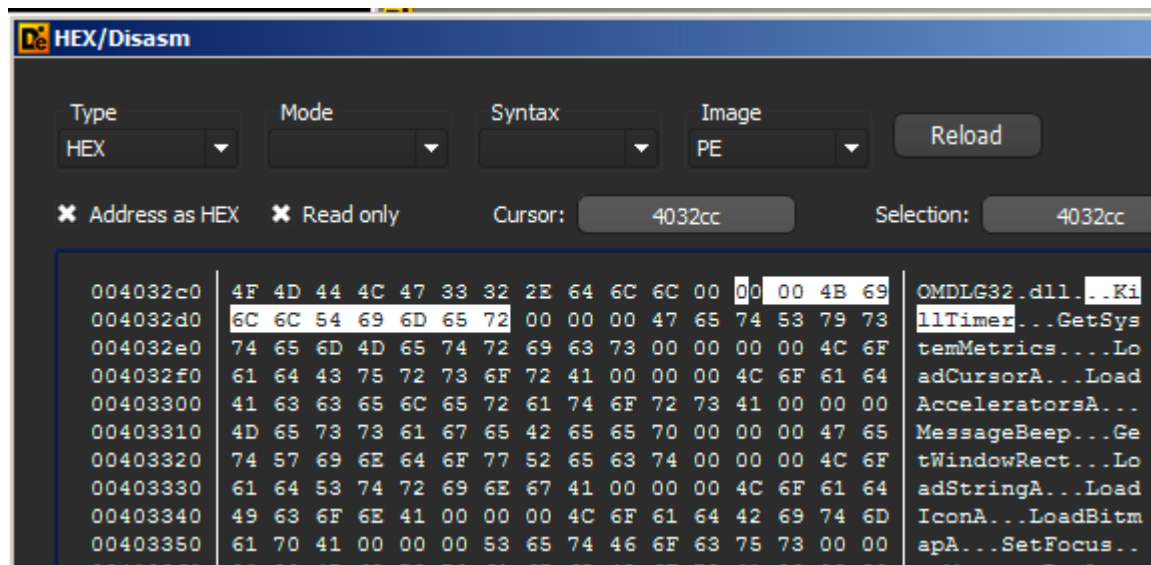
Neste exemplo o valor é 0x00003078 (lembre-se que números são armazenados em *little-endian*). Sendo novamente um endereço relativo, então o endereço da ILT é 0x403078.

Seguindo este endereço, encontramos achamos a ILT, que é um *array* de números de 32-bits como já dissemos:



O primeiro número deste *array* é então 0x000032cc. Como seu MSB está zerado, sabemos que se trata de uma importação por nome (e não por número da função). Se seguirmos

este endereço, novamente somando o *ImageBase*, finalmente chegamos ao nome da função:

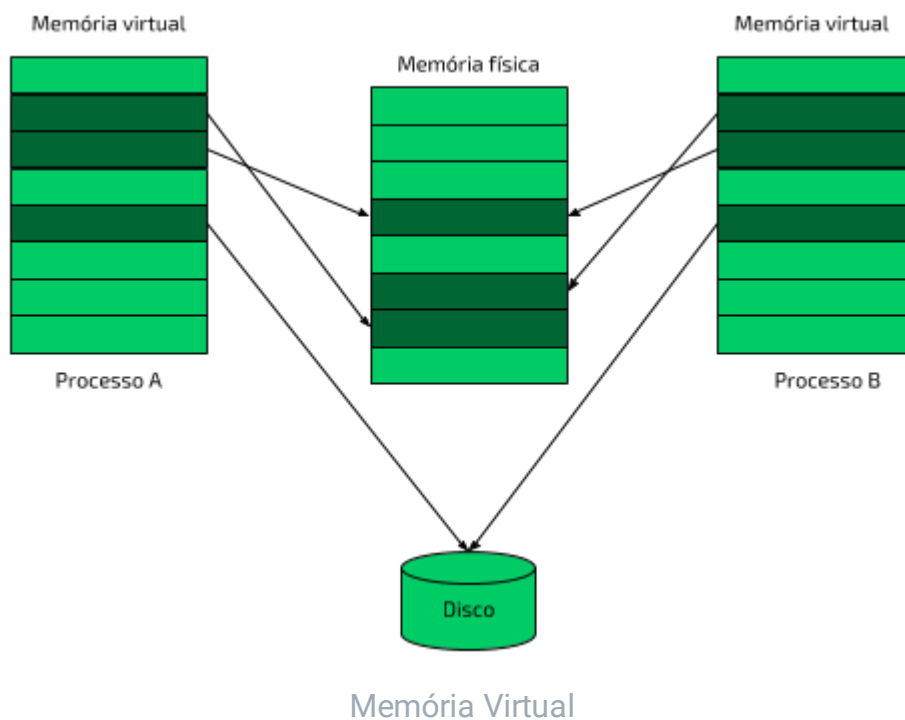


A estrutura que contém o nome da função é chamada de **Hint/Name Table** \*\*onde o nome da função começa no terceiro *byte*, neste caso, em 0x4032ce. O tamanho do nome é variável (naturalmente o tamanho em *bytes* do nome de uma função pode variar).

# Endereçamento

## Memória virtual

Onde já se viu dois executáveis com o **ImageBase** em 0x400000 rodarem ao mesmo tempo se ambos são carregados no mesmo endereço de memória? Bem, a verdade é que não são. Existe um esquema chamado de **memória virtual** que consiste num mapeamento da memória RAM real física para uma memória virtual para cada processo no sistema, dando a eles a ilusão de que estão sozinhos num ambiente monotarefa como era antigamente (vide MS-DOS e outros sistemas antigos). Essa memória virtual também pode ser mapeada para um arquivo em disco, com o *pagefile.sys*. O desenho a seguir ilustra o mecanismo de mapeamento:



Conforme explicado no capítulo sobre as Seções dos arquivos PE, a memória é dividida em páginas, tanto a virtual quanto a física. No desenho, os dois processos possuem páginas mapeadas pelo *kernel* (pelo gerenciador de memória, que é parte deste) em memória física e em disco (sem uso no momento). Perceba que as páginas de memória não precisam ser contíguas (uma imediatamente após a outra) no *layout* de memória física, nem no da virtual. Além disso, dois processos diferentes podem ter regiões virtuais mapeadas para a mesma região da memória física, o que chamamos de páginas compartilhadas.

Em resumo, o sistema gerencia uma tabela que relaciona endereço físico de memória (real) com endereço virtual, para cada processo. Todos "acham" que estão sozinhos no sistema, mas na verdade estão juntos sob controle do *kernel*.

---

## Endereço Virtual

O endereço virtual, em inglês Virtual Address, ou simplesmente VA, é justamente a localização virtual em memória de um dado ou instrução. Por exemplo, quando alguém fazendo engenharia reversa num programa diz que no endereço 0x401000 existe uma função que merece atenção, quer dizer que ela está no VA 0x401000 do binário quando carregado. Para ver a mesma função, você precisa carregar o binário em memória (normalmente feito com um *debugger*, como veremos num capítulo futuro) e verificar o conteúdo de tal endereço.

---

## Endereço Virtual Relativo

Em inglês, *Relative Virtual Address*, é um VA que, ao invés de ser absoluto, é relativo à alguma base. Por exemplo, o valor do campo *entrypoint* no cabeçalho Opcional é um RVA relativo à base da imagem (campo *ImageBase* no mesmo cabeçalho). Com isso em mente, avalie seu valor na saída a seguir:

```
1  Optional/Image header
2      Magic number:          0x10b (PE32)
3      Linker major version:  48
4      Linker minor version:  0
5      Size of .text section:  0x1a00
6      Size of .data section:  0x800
7      Size of .bss section:   0
8      Entrypoint:            0x39c2
9      Address of .text section: 0x2000
10     Address of .data section: 0x4000
11     ImageBase:              0x400000
12     Alignment of sections:  0x2000
13     Alignment factor:       0x200
14     Major version of required OS: 4
15     Minor version of required OS: 0
```

```

16      Major version of image:      0
17      Minor version of image:      0
18      Major version of subsystem:   4
19      Minor version of subsystem:   0
20      Size of image:                0x8000
21      Size of headers:              0x200
22      Checksum:                     0
23      Subsystem required:            0x3 (IMAGE_SUBSYSTEM_WINDOWS_CUI)
24      DLL characteristics:          0x8540
25      DLL characteristics names
26                                     IMAGE_DLLCHARACTERISTICS_DYNAMIC_
27                                     IMAGE_DLLCHARACTERISTICS_NX_COMPAT
28                                     IMAGE_DLLCHARACTERISTICS_NO_SEH
29                                     IMAGE_DLLCHARACTERISTICS_TERMINAL_A
30      Size of stack to reserve:      0x100000
31      Size of stack to commit:       0x1000
32      Size of heap space to reserve: 0x100000
33      Size of heap space to commit:  0x1000


```

No exemplo acima, o campo *entrypoint* tem o valor 0x39c2, que é um RVA. Como este campo é relativo ao *ImageBase*, o VA (endereço virtual) do *entrypoint* é então dado pela sua soma com o valor de *ImageBase*:

```

1  entrypoint_va = entrypoint_rva + imagebase
2  entrypoint_va = 0x39c2 + 0x400000
3  entrypoint_va = 0x4039c2

```

 Os RVAs podem ser relativos à outras bases que não a base da imagem. É preciso consultar na documentação qual a relatividade de um RVA para convertê-lo corretamente para o VA correspondente.



# Execução de programas

## Privilégios de execução

Para impedir que os programas do usuário acessem ou modifiquem dados críticos do sistema operacional, o Windows suporta dois níveis de execução de código: modo de usuário e modo de kernel, mais conhecidos por suas variantes em inglês: *user mode* e *kernel mode*.

Os programas comuns rodam em *user mode*, enquanto os serviços internos do SO e *drivers* rodam em *kernel mode*.

Apesar de o Windows e outros sistemas operacionais modernos trabalharem com somente estes dois níveis de privilégios de execução, os processadores Intel e compatíveis suportam quatro níveis, também chamado de anéis (*rings*), numerados de 0 a 3. Para *kernel mode* é utilizado o *ring 0* e para *user mode*, o *ring 3*.

Programas rodando em *user mode* tampouco possuem acesso ao hardware do computador. Essencialmente, todos estes fatores combinados fazem com que os programas rodando neste privilégio de execução não gerem erros fatais como a famosa "tela azul da morte" (ou BSOD - **B**lue **S**creen **O**f **D**eath).

Passa que toda a parte legal acontece *kernel mode*, sendo assim, um processo (na verdade uma *thread*) rodando em *user mode* pode executar tarefas em *kernel mode* através da API do Windows, que funciona como uma interface para tal.

---

## Dependências

Quando um programador cria um programa, em muitos casos ele utiliza funções de bibliotecas (ou *libraries* em inglês), também chamadas de DLL (*Dynamic-Link Library*). Sendo assim, analise o seguinte simples programa em C:

```
1 #include <stdio.h>
```

```
2
3  int main(void) {
4      printf("Olá, mundo!\n");
5      return 0;
6  }
```

Este programa utiliza a função *printf()*, que não precisou ser implementada pelo programador. Ao contrário, ele simplesmente a chamou, já que esta está definida no arquivo *stdio.h*.

Quando compilado, este programa terá uma dependência da biblioteca de C (arquivo **msvcrt.dll** no Windows) pois o código para a *printf()* está nela.

Tudo isso garante que vários programadores usem tal função, que tenha sempre o mesmo comportamento se usada da mesma forma. Mas já parou para pensar como a função *printf()* de fato escreve na tela? Como ela lidaria com as diferentes placas de vídeo, por exemplo?

O fato é que a *printf()* não escreve diretamente na tela. Ao contrário, a biblioteca de C pede ao *kernel* através de uma função de sua API que seja feito. Sendo assim, temos, neste caso um EXE que chama uma função de uma DLL que chama o *kernel*. Estudaremos mais a frente como isso é feito.

---

## Loader

Quando um programa é executado (por exemplo, com um duplo-clique no Windows), ele é copiado para a memória e um **processo** é criado para ele. Dizemos então que um processo está rodando, mas esta afirmação não é muito precisa: na verdade, todo processo no Windows possui pelo menos uma *thread* e ela sim é que roda. O processo funciona como um "container" que contém várias informações sobre o programa rodando e suas *threads*.

Quem cria esse processo em memória é um componente do sistema operacional chamado de *image loader*, presente na biblioteca **ntdll.dll**.



O código do *loader* roda **antes** do código do programa a ser carregado. É um código comum a todos os processos executados no sistema operacional.

Dentre as funções do *loader* estão:

- Ler os cabeçalhos do arquivo PE a ser executado e alocar a memória necessária para a imagem como um todo, suas seções, etc.
  - As seções são mapeadas para a memória, respeitando-se suas permissões.
- Ler a tabela de importações do arquivo PE a fim de carregar as DLLs requeridas por este e que ainda não foram carregadas em memória. Esse processo também é chamado de **resolução de dependências**.
- Preencher a IAT com os endereços das funções importadas.
- Carregar módulos adicionais em tempo de execução, se assim for pedido pelo executável principal (também chamado de módulo principal).
- Manter uma lista de módulos carregados por um processo.
- Transferir a execução para o *entrypoint* (EP) do programa.

# Executáveis

Normalmente, chamamos de arquivos executáveis os arquivos que, quando clicados duas vezes (no caso do Windows), executam. A extensão mais comum - e foco do nosso estudo aqui - são os arquivos .exe. Para entender como estes arquivos são criados, é preciso notar os passos:

1. Escrita do código-fonte na linguagem escolhida num arquivo de texto.
2. Compilação.
3. *Linking*.

O compilador cria os chamados arquivos **objeto** a partir do código-fonte (em texto). Estes objetos contém instruções de máquina e dados.

O *linker* serve para **juntar** todos os objetos num único arquivo, **realocar** seus endereços e **resolver** seus símbolos (nomes de função importadas, por exemplo).

O processo de compilação (transformação do código-fonte em texto em código de máquina) gera como saída um arquivo chamado de **objeto**.

No que diz respeito ao processo de *linking*, estes executáveis podem ser de dois tipos:

---

## Estáticos

Todo o código das funções externas ao executável principal é compilado junto a ele. O resultado é um executável livre de dependências, porém grande.

---

## Dinâmicos

O executável vai **depender** de bibliotecas externas (DLL's, no caso do Windows) para funcionar, como estudamos na seção Tabela de Importações.

O nosso binário CRACKME.EXE, que usamos em vários momentos deste livro é dinâmico. Nós já vimos isso ao analisar sua *Imports Table* com o DIE, mas agora vamos utilizar a ferramenta **dumpbin**, parte integrante do Visual Studio da Microsoft, para checar suas dependências:

```
1 D:\>dumpbin /dependents CRACKME.EXE
2 Microsoft (R) COFF/PE Dumper Version 14.12.25830.2
3 Copyright (C) Microsoft Corporation. All rights reserved.
4
5
6 Dump of file CRACKME.EXE
7
8 File Type: EXECUTABLE IMAGE
9
10 Image has the following dependencies:
11
12     USER32.dll
13     KERNEL32.dll
14     COMCTL32.DLL
15     GDI32.dll
16     COMDLG32.dll
17
18 Summary
19
20     1000 .edata
21     1000 .idata
22     1000 .reloc
23     2000 .rsrc
24     1000 CODE
25     1000 DATA
```

O mesmo resultado pode ser atingido utilizando a ferramenta **readpe**. A vantagem desta sobre o **dumpbin** é que o **readpe** é multi-plataforma e livre. Para entender como ele funciona, assista ao vídeo:



[https://www.youtube.com/watch?v=7EI\\_wRk3VBU](https://www.youtube.com/watch?v=7EI_wRk3VBU)

# Bibliotecas

As bibliotecas, ou DLL's no Windows, são também arquivos PE, mas sua intenção é ter suas funções utilizadas (importadas e chamadas) por arquivos executáveis. Elas também importam funções de outras bibliotecas, mas além disso, **exportam** funções para serem utilizadas.

Novamente, é possível utilizar o DIE para ver as funções importadas e exportadas por uma DLL, mas no exemplo a seguir, utilizamos novamente o **dumpbin** contra a biblioteca *SHELL32.DLL*, nativa do Windows:

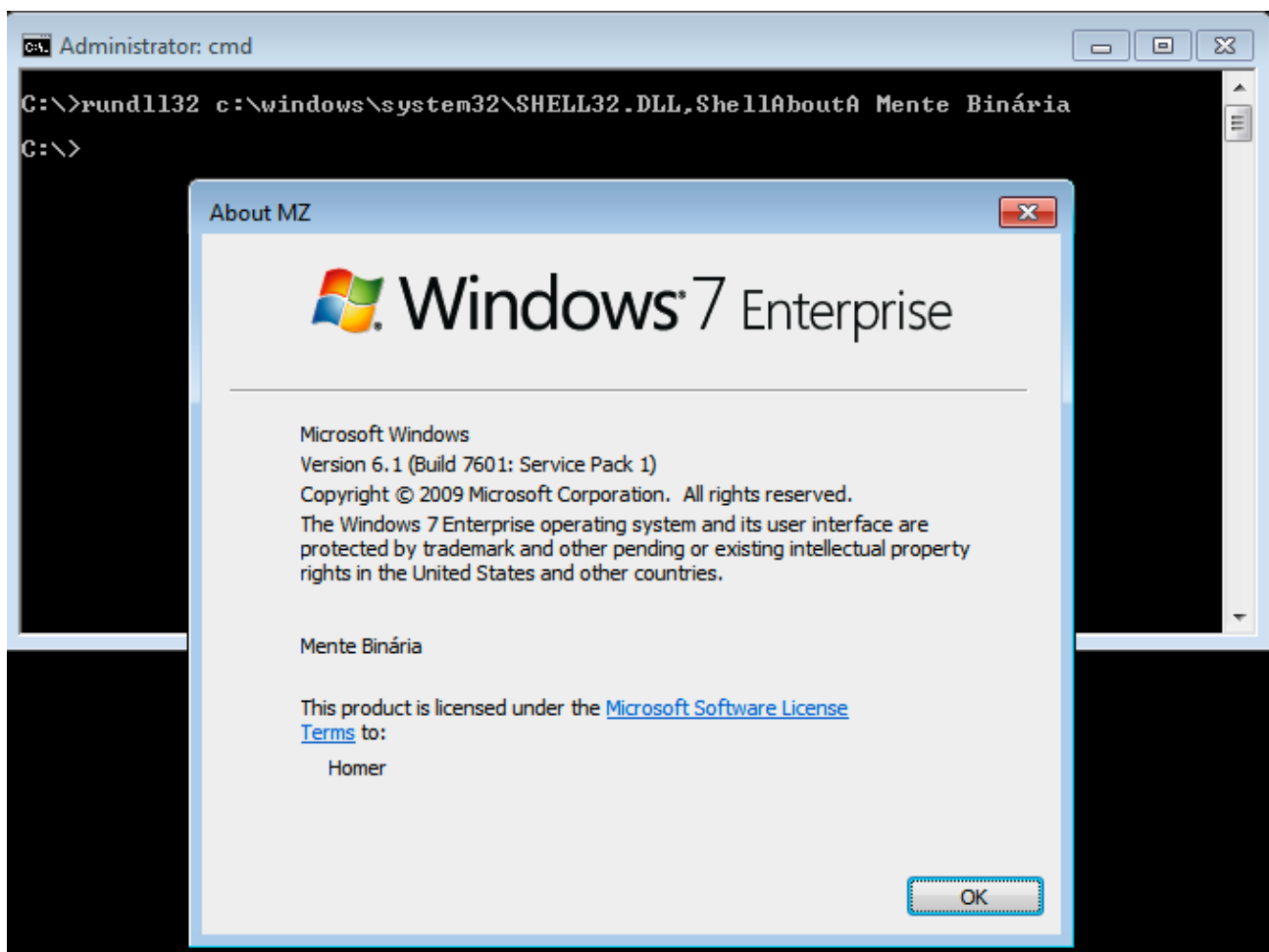
```
1 C:\>dumpbin /exports %windir%\system32\shell32.dll | findstr /i shellabout
2          428  119 0027CC59 ShellAboutA
3          429  11A 000CA129 ShellAboutW
```

Utilizamos o comando **findstr** do Windows para filtrar a saída por funções que criam caixas de mensagem. Este comando é como o **grep** no Linux. A sua opção **/i** faz com que o filtro de texto ignore o case (ou seja, funciona tanto com letras maiúsculas quanto com minúsculas).

Para chamar uma função desta DLL, teríamos que criar um executável que a **importe**. No entanto, o próprio Windows já oferece um utilitário chamado *rundll32.exe*, capaz de chamar funções de uma biblioteca. A maneira via linha de comando é como a seguir:

```
C:\>rundll32.exe <DLL>, <Função> <Parâmetros>
```

Como a função *ShellAboutA()* recebe um texto ASCII para ser exibido na tela "Sobre" do Windows, podemos testá-la da seguinte forma:



Utilizar o `rundll32.exe` para chamar funções de biblioteca não é a maneira mais adequada de fazê-lo e não funciona com todas as funções, principalmente as que precisam de parâmetros que não são do tipo *string*. Somente o utilizamos aqui para fins de compreensão do conteúdo.

Em breve também aprenderemos como *debugar* uma DLL, mas por hora o conhecimento reunido aqui é suficiente.

# Windows API

Uma API (*Application Programming Interface*) é uma interface para programar uma aplicação. No caso do Windows, consiste num conjunto de funções expostas para serem usadas por aplicativos, inclusive em *user mode*.

Para o escopo deste livro, vamos cobrir uma pequena parte da Win32 API, com foco nas funções disponíveis basicamente em duas bibliotecas diferentes: a USER32.DLL e a KERNEL32.DLL.

Considere o seguinte programa:

```
1  #include <windows.h>
2
3  int main() {
4      MessageBox(NULL, "Cash", "Johnny", MB_OK);
5      return 0;
6  }
```


A função *MessageBox()* está definida em *windows.h*. Quando compilado, o código acima gera um executável dependente da USER32.DLL (além de outras bibliotecas), que provê a versão compilada de tal função. A documentação desta e de outras funções da Win32 está disponível na [MSDN](#). Copiamos seu protótipo abaixo para explicar seus parâmetros:

```
1  int WINAPI MessageBox(
2      _In_opt_ HWND    hWnd,
3      _In_opt_ LPCTSTR lpText,
4      _In_opt_ LPCTSTR lpCaption,
5      _In_      UINT    uType
6  );
```

A Microsoft utiliza várias convenções de nome que precisam ser explicadas para o entendimento dos protótipos das funções de sua API. Para entender o protótipo da função *MessageBox*, é preciso conhecer algumas:



WINAPI	Define que a convenção de chamada da função é a <code>__stdcall</code>
_In_	Define que o parâmetro é de entrada
_Out_	Define que o parâmetro é de saída (a função vai escrever nele)
_opt_	O parâmetro é opcional (pode ser NULL)
HANDLE	Um número identificador de um objeto no ambiente Windows
HWND	Um <i>handle</i> (identificador) da janela
LPCTSTR	Long <b>P</b> ointer to a <b>C</b> onst <b>T</b> CHAR <b>S</b> TRing
UINT	<i>unsigned int</i> ou DWORD (32-bits)

 Um *handle* é um número que identifica um objeto (arquivo, chave de registro, diretório, etc) aberto usado por um processo. É um conceito similar ao *file descriptor* em ambiente Unix/Linux. *Handles* só são acessíveis diretamente em *kernel mode*, por isso os programas interagem com eles através de funções da API do Windows. Um bom exemplo é a [CloseHandle] ([https://msdn.microsoft.com/en-us/library/windows/desktop/ms724211\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724211(v=vs.85).aspx)).

Agora vamos explicar os parâmetros da função *MessageBox*:

### **hWnd [entrada, opcional]**

Um *handle* que identifica qual janela é dona da caixa de mensagem. Isso serve para atrelar uma mensagem a uma certa janela (e impedi-la de ser fechada antes da caixa de mensagem, por exemplo). Como é opcional, este parâmetro pode ser NULL, o que faz com que a caixa de mensagem não possua uma janela dona.

### **lpText [entrada, opcional]**

Um ponteiro para um texto (uma *string*) que será exibido na caixa de mensagem. Se for NULL, a mensagem não terá um conteúdo, mas ainda assim aparecerá.

### **lpCaption [entrada, opcional]**

Um ponteiro para o texto que será o título da caixa de mensagem. Se for NULL a caixa de mensagem não terá um título, mas ainda assim aparecerá.

### **uType [entrada]**

Configura o tipo de caixa de mensagem. É um número inteiro que pode ser definido por macros para cada *flag*, definida na [documentação da função](#). Se passada a macro MB\_OKCANCEL (0x00000001L), por exemplo, faz com que a caixa de mensagem tenha dois botões: OK e Cancelar. Se passada a macro MB\_ICONEXCLAMATION (0x00000030L), a janela terá um ícone de exclamação. Se quiséssemos combinar as duas características, precisaríamos passar as duas *flags* utilizando uma operação OU entre elas, assim:

```
MessageBox(NULL, "Cash", "Johnny", MB_OKCANCEL | MB_ICONEXCLAMATION);
```

Como macros e cálculos assim são resolvidos em C numa etapa conhecida por pré-compilação, o resultado da operação OU entre 1 e 0x30 será substituído neste código, antes de ser compilado, ficando assim:

```
MessageBox(0, "Cash", "Johnny", 0x31);
```



Dizer que um parâmetro é opcional não quer dizer que você não precise passá-lo ao chamar a função, mas sim que ele pode ser NULL, ou zero, dependendo do que a documentação da função diz.

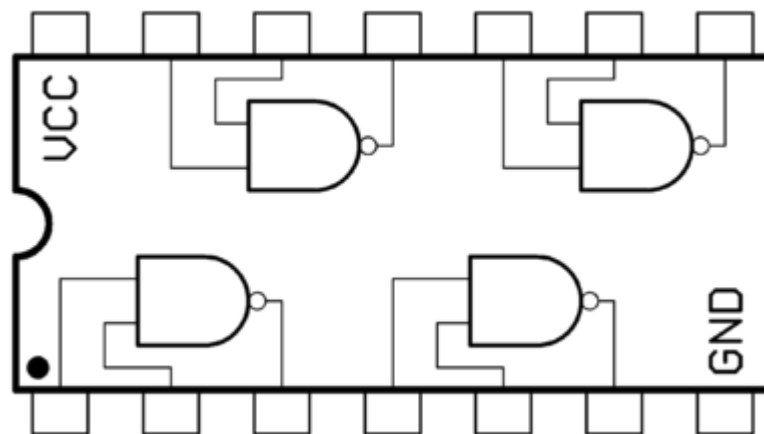
Funções importantes da Win32 incluem CreateFile, DeleteFile, RegOpenKey, RegCreateKey, dentre outras. É altamente recomendado que o leitor crie programas de exemplo utilizando-

as para atestar o funcionamento delas. Você encontrará algumas dessas funções separadas por categoria no apêndice [Funções da API do Windows](#).

# Assembly

Agora que você já sabe como um binário PE é construído, está na hora de entender como o código contido em suas seções de código de fato executa. Acontece que um processador é programado em sua fábrica para entender determinadas sequências de *bytes* como código e executar alguma operação. Para entender isso, vamos fazer uma analogia com um componente muito mais simples que um processador, um **circuito integrado** (popularmente chamado de *chip*).

Um circuito integrado (CI) bastante conhecido no mundo da eletrônica é o 7400 que tem o seguinte diagrama esquemático:



Se você já estudou portas lógicas, vai perceber que este CI tem 4 portas NAND (AND com saída negada). Cada porta possui duas entradas e uma saída, cada uma delas conectada a seu respectivo pino/perna.

Admitindo duas entradas (A e B) e uma saída S, a tabela verdade de cada uma das portas deste CI é a seguinte:

A	B	A & B	S
0	0	0	1
1	0	0	1
0	1	0	1

---

1	1	1	0
---	---	---	---

---

Podemos dizer então que este CI faz uma única operação sempre, com as entradas de dados que recebe.

Se seu projeto precisasse também de portas OR, XOR, AND, etc você precisaria comprar outros circuitos integrados, certo? Bem, uma solução inteligente seria utilizar um *chip* que fosse **programável**. Dessa forma, você o configuraria, via *software*, para atuar em certos pinos como porta NAND, outros como porta OR e por aí vai, de acordo com sua necessidade. Aumentando ainda mais a complexidade, temos os **microcontroladores**, que podem ser **programados** em linguagens de alto nível, contando com recursos como repetições, condicionais, e tudo que uma linguagem de programação completa oferece. No entanto, estes *chips* requerem uma reprogramação a cada mudança no projeto, assim como se faz com o Arduino hoje em dia.

Neste sentido um microprocessador, ou simplesmente **processador** é muito mais poderoso. Ao invés de o usuário gravar um programa nele, o próprio fabricante já o faz, de modo que este microprograma entenda diferentes instruções para realizar diferentes operações de muito alto nível (se comparadas às simples operações *booleanas*). Sua entrada de dados também é muito mais flexível: ao invés de entradas binárias, um processador pode receber números bem maiores. O antigo Intel 8088 já possuía um barramento de 8 *bits*, por exemplo.

Isso significa que se um processador receber em seu barramento um conjunto de *bytes* específico, sabe que vai precisar executar uma operação específica. À estes *bytes* possíveis damos o nome de **opcodes**. Ao conjunto dos *opcodes* + operandos damos o nome de **instrução**.

Supondo que queiramos então fazer uma operação OR entre os valores 0x20 e 0x18 utilizando um processador x86. Na documentação deste processador, constam as seguintes informações:

- Ao receber o *opcode* 0xb8, os próximos quatro *bytes* serão um número salvo em memória (similar àquela memória M+ das calculadoras), acessível através do *byte* identificador 0xc8.

- Ao receber o *opcode* 0x83, seguido de um *byte* 0xc8 (que identifica a memória), seguido de mais um *byte*, o número armazenado na memória identificada pelo segundo *byte* vai sofrer uma operação OR com este terceiro *byte*.

Precisaríamos então utilizar as duas instruções, da seguinte forma:

```
1  B8 20 00 00 00
2  83 C8 18
```

Na primeira, que tem um total de 5 *bytes*, o *opcode* 0xb8 é utilizado para colocar o número de 32-bits (4 *bytes*) na sequência em memória. Como nosso número desejado possui somente 1 *byte*, preenchemos os outros três com zero, respeitando o *endianess*.

A segunda instrução tem 3 *bytes* sendo que o primeiro é o *opcode* dela (OR), o segundo é o identificador da área de memória e o terceiro é o nosso operando 0x18, que deve sofrer o OR com o valor na área de memória indicada por C8.

Temos que concordar que criar um programa assim não é nada fácil. Para resolver este problema foi criada uma **linguagem de programação**, completamente presa à arquitetura do processador (seus *opcodes*, suas instruções), chamada **Assembly**. Com ela, os programadores poderiam escrever o programa acima praticamente em inglês:

```
1  MOV EAX, 20
2  OR  EAX, 18
```

De posse de um compilador de Assembly, chamado na época de **assembler**, o resultado da compilação do código-fonte acima é justamente um arquivo (objeto) que contém os *opcodes* e argumentos corretos para o processador alvo, onde o programa vai rodar.



Agora você sabe o motivo pelo qual um programa compilado não é compatível entre diferentes processadores, de diferentes arquiteturas. Como estes possuem instruções diferentes e *opcodes* diferentes, não há mesmo compatibilidade.

Perceba que Assembly é uma linguagem legível para humanos, diferente da linguagem de máquina que não passa de uma "tripa de *bytes*". Os comandos da linguagem Assembly são chamados de **mnemônicos**. No exemplo de código acima, utilizamos dois: o MOV e o OR. Estudaremos mais mnemônicos em breve.

# Registradores

Os processadores possuem uma área física em seus *chips* para armazenamento de dados (de fato, somente números, já que é só isso que existe neste mundo!) chamadas de **registradores**, justamente porque registram (salvam) um número por um tempo, mesmo este sendo não determinado.

Os registradores possuem normalmente o tamanho da palavra do processador, logo, se este é um processador de 32-bits, seus registradores possuem este tamanho também.

---

## Registradores de uso geral

Um registrador de uso geral, também chamado de GPR (*General Purpose Register*) serve para armazenar temporariamente qualquer tipo de dado, para qualquer função.

Existem 8 registradores de uso geral na arquitetura Intel x86. Apesar de poderem ser utilizados para qualquer coisa, como seu nome sugere, a seguinte convenção é normalmente respeitada:

Registrador	Significado	Uso sugerido
EAX	Accumulator	Usado em operações aritméticas
EBX	Base	Ponteiro para dados
ECX	Counter	Contador em repetições
EDX	Data	Usado em operações de E/S
ESI	Source Index	Ponteiro para uma <i>string</i> de origem
EDI	Destination Index	Ponteiro para uma <i>string</i> de destino
ESP	Stack Pointer	Ponteiro para o topo da pilha
EBP	Base Pointer	Ponteiro para a base do <i>stack frame</i>



Para fixar o assunto, é importante trabalhar um pouco. Vamos escrever o seguinte programa em Assembly no Linux ou macOS:

ou.s

```
1 section .text
2     mov eax, 0x30
3     or  eax, 0x18
```

Para compilar, vamos instalar o [NASM](#), que para o nosso caso é útil por ser multiplataforma:

Linux

```
1 $ sudo apt install nasm
2 $ nasm -felf ou.s
```

macOS

```
1 $ brew install nasm
2 $ nasm -fmacho ou.s
```

Confira como ficou o código **compilado** no arquivo objeto com a ferramenta **objdump**, do pacote [binutils](#):

## Linux

```
1 $ objdump -dM intel ou.o
2
3 ou.o:      file format elf32-i386
4
5
6 Disassembly of section .text:
7
8 00000000 <.text>:
9      0:  b8 30 00 00 00      mov     eax,0x30
10     5:  83 c8 18            or      eax,0x18
```

## macOS

```
1 $ objdump -d -x86-asm-syntax=intel -print-imm-hex ou.o
2
3 ou.o:      file format Mach-O 32-bit i386
4
5 Disassembly of section __TEXT,__text:
6 __text:
7      0:  b8 30 00 00 00      mov     eax, 0x30
8      5:  83 c8 18            or      eax, 0x18
```

Percebe os *opcodes* e argumentos idênticos aos exemplificados na introdução deste capítulo? Prova que não mentimos. :)



O NASM compilou corretamente a linguagem Assembly para código de máquina. O `objdump` mostrou tanto o código de máquina quanto o equivalente em Assembly, processo conhecido como **desmontagem** ou **disassembly**.

Agora cabe à você fazer mais alguns testes com outros registradores de uso geral. Um detalhe importante é que os primeiros quatro registradores de uso geral podem ter sua

parte baixa manipulada diretamente. Por exemplo, é possível trabalhar com o registrador de 16 bits AX, a parte baixa de EAX. Já o AX pode ter tanto sua parte alta (AH) quanto sua parte baixa (AL) manipulada diretamente também, onde ambas possuem 8 bits de tamanho. O mesmo vale para EBX, ECX e EDX.

Para entender, analise as seguintes instruções:

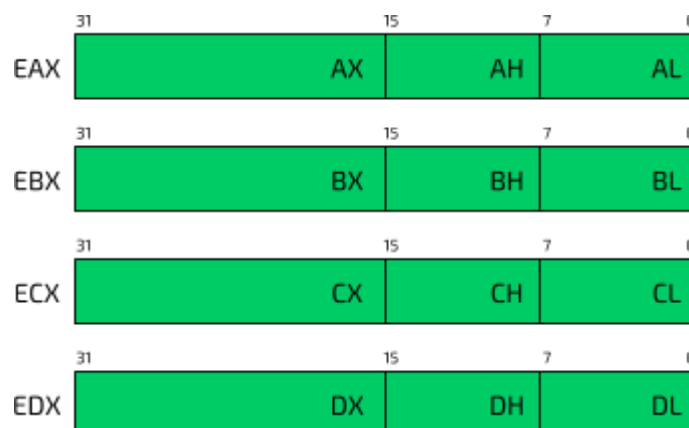
```
1  mov eax, 0xaabbccdd
2  mov ax, 0xeeff
```

Após a execução das instruções acima, EAX conterá o valor **0xaabb**eeff****, já que somente sua parte **baixa** foi modificada pela segunda instrução. Agora analise o seguinte trecho:

```
1  mov eax, 0xaabbccdd
2  mov ax, 0xeeff
3  mov ah, 0xcc
4  mov al, 0xdd
```

Após a execução das quatro instruções acima, EAX volta para o valor **0xaabbccdd**.

A imagem a seguir ilustra os ditos vários registradores dentro dos quatro primeiros registradores de uso geral.



Sub-divisões de EAX, EBX, ECX e EDX

Os registradores EBP, ESI, EDI e ESP também podem ser utilizados como registradores de 16-bits BP, SI, DI e SP, respectivamente. Note porém que estes últimos não são subdivididos em partes alta (*high*) e baixa (*low*).

## Ponteiro de instrução

Existe um registrador chamado de EIP (*Extended Instruction Pointer*), também de PC (*Program Counter*) em algumas literaturas que aponta para a próxima instrução a ser executada. Não é possível copiar um valor literal para este registrador. Portanto, uma instrução `mov eip, 0x401000` **não** é válida.

Outra propriedade importante deste registrador é que ele é incrementado com o número de *bytes* da última instrução executada. Para fixar, analise o exemplo do *disassembly* a seguir, gerado com a ferramenta **objdump**:

```
1 08049000 <_start>:
2 8049000: b8 01 00 00 00      mov     eax,0x1
3 8049005: bb 00 00 00 00      mov     ebx,0x0
4 804900a: cd 80               int     0x80
5 804900c: b8 02 00 00 00      mov     eax,0x2
```

Quando a primeira instrução do trecho acima estiver prestes à ser executada, o registrador EIP conterá o valor 0x8049000. Após a execução desta primeira instrução MOV, o EIP será incrementado em 5 unidades, já que tal instrução possui 5 *bytes*. Por isso o **objdump** já mostra o endereço correto da instrução seguinte. Perceba, no entanto, que a instrução no endereço 804900a possui apenas 2 *bytes*, o que vai fazer com o que o registrador EIP seja incrementado em 2 unidades para apontar para a próxima instrução MOV, no endereço 804900c.

## Registradores de segmento


Estes registradores armazenam o que chamamos de seletores, ponteiros que identificam segmentos na memória, essenciais para operação em modo real. Em modo protegido, que é o modo de operação do processador que os sistemas operacionais modernos utilizam, a função de cada registrador de segmento fica a cargo do SO. Abaixo a lista dos registradores de segmento e sua função em modo protegido:

Registrador	Significado
CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Data Segment
FS	Data Segment (no Windows em x86, aponta para o o <b>TEB</b> ( <i>Thread Environment Block</i> ) da <i>thread</i> atual do processo em execução)
GS	Data Segment

Não entraremos em mais detalhes sobre estes registradores por fugirem do escopo deste livro.

## Registrador de flags

O registrador de *flags* EFLAGS é um registrador de 32-bits usado para *flags* de estado, de controle e de sistema.

 *Flag* é um termo genérico para um dado, normalmente "verdadeiro ou falso". Dizemos que uma *flag* está *setada* quando seu valor é verdadeiro, ou seja, é igual a 1.

Existem 10 *flags* de sistema, uma de controle e 6 de estado. As *flags* de estado são utilizadas pelo processador para reportar o estado da última operação (pense numa comparação, por exemplo - você pede para o processador comparar dois valores e a resposta vem através de uma *flag* de estado). As mais comuns são:

Bit	Nome	Sigla	Descrição
0	Carry	CF	Setada quando o resultado estourou o limite do tamanho do dado. É o famoso "vai-um" na matemática para números sem sinal ( <i>unsigned</i> ).
6	Zero	ZF	Setada quando o resultado de uma operação é zero. Do contrário, é zerada. Muito usada em comparações.
7	Sign	SF	Setada de acordo com o MSB ( <i>Most Significant Bit</i> ) do resultado, que é justamente o <i>bit</i> que define se um inteiro com sinal é positivo (0) ou negativo (1), conforme visto na seção Números negativos.
11	Overflow	OF	Estouro para números com sinal.

Além das outras *flags*, há ainda os registradores da FPU (*Float Point Unit*), de debug, de controle, XMM (parte da extensão SSE), MMX, 3DNow!, MSR (*Model-Specific Registers*), e possivelmente outros que não abordaremos neste livro em prol da brevidade.

Agora que já reunimos bastante informação sobre os registradores, é hora de treinarmos um pouco com as instruções básicas do Assembly.

# Instruções básicas

Uma instrução é um conjunto definido por um código de operação (*opcode*) mais seus operandos, se houver. Ao receber *bytes* específicos em seu barramento, o processador realiza determinada operação. O formato geral de uma instrução é:

```
opcode operando1, operando2, operando3
```

Onde *opcode* representa um código de operação definido no [manual da Intel](#). O número de operandos, que podem variar de 0 a 3 na IA-32 (Intel Architecture de 32-bits), consistem em números literais, registradores ou endereços de memória necessários para a instrução funcionar. Por exemplo, considere a seguinte instrução, que coloca o valor 2018 no registrador EAX:

```
B8 E2 07 00 00
```

O primeiro *byte* é o *opcode*. Os outros 4 *bytes* representam o primeiro e único argumento dessa instrução. Sabemos então que 0xB8 faz com que um valor seja colocado em EAX. Como este registrador tem 32-bits, nada mais natural que o argumento dessa instrução ser também de 32-bits ou 4 *bytes*. Considerando o *endianess*, como já explicado anteriormente neste livro, o valor literal 2018 (0x7E2 ou, em sua forma completa de 32-bits, 0x000007E2) é escrito em *little-endian* com seus *bytes* na ordem inversa, resultando em E2 07 00 00.

Na arquitetura Intel IA-32, uma instrução (considerando o *opcode* e seus argumentos) pode ter de 1 à 15 *bytes* de tamanho.

---

## Copiando valores

Uma instrução muito comum é a MOV, forma curta de "move" (do Inglês, "mover"). Apesar do nome, o que a instrução faz é copiar o segundo operando (origem) para o primeiro

(destino). O operando de origem pode ser um valor literal, um registrador ou um endereço de memória. O operando de destino funciona de forma similar, com exceção de não poder ser um valor literal, pois não faria sentido mesmo. Ambos os operandos precisam ter o mesmo tamanho, que pode ser de um *byte*, uma *word* ou uma *doubleword*, na IA-32.

Analise o exemplo a seguir:

```
MOV EBX, B0B0CA
```

A instrução acima copia um valor literal 0xB0B0CA para o registrador EBX. A versão compilada desta instrução resulta nos seguintes *bytes*:

```
BB CA B0 B0 00
```

---

## Aritimética

Naturalmente, processadores fazem muitos cálculos matemáticos. Veremos agora algumas dessas instruções, começando pela instrução ADD, que soma valores. Analise:

```
1 MOV ECX, 7
2 ADD ECX, 1
```

No código acima, a instrução ADD soma 1 ao valor de ECX (que no nosso caso é 7, conforme instrução anterior). O resultado desta soma é **armazenado no operando de destino**, ou seja, no próprio registrador ECX, que passa a ter o valor 8.

Uma outra forma de atingir este resultado seria utilizar a instrução INC, que incrementa seu operando em uma unidade, dessa forma:



```
1  MOV ECX, 7
2  INC ECX
```

A instrução INC recebe um único operando que pode ser um registrador ou um endereço de memória. O resultado do incremento é armazenado no próprio operando, que em nosso caso é o registrador ECX.

O leitor pode se perguntar por que existe uma instrução INC se é possível incrementar um operando em uma unidade com a instrução ADD. Para entender, compile o seguinte programa com o NASM:

soma.s

```
1  BITS 32
2
3  global start
4
5  section .text
6  start:
7      mov eax, 7
8      add eax, 1
9      inc eax
```

Após compilar, verifique o código objeto gerado com o **objdump**:

## Linux

```
1 $ objdump -dM intel soma.o
2
3 soma.o:      file format elf32-i386
4
5 Disassembly of section .text:
6
7 00000000 <start>:
8   0:   b8 07 00 00 00      mov     eax,0x7
9   5:   83 c0 01            add     eax,0x1
10  8:   40                  inc     eax
```

## macOS

```
1 $ objdump -d -x86-asm-syntax=intel -print-imm-hex soma.o
2
3 soma.o:      file format Mach-O 32-bit i386
4
5 Disassembly of section __TEXT,__text:
6 start:
7   0:   b8 07 00 00 00      mov     eax, 0x7
8   5:   83 c0 01            add     eax, 0x1
9   8:   40                  inc     eax
```

Há duas diferenças básicas entre as instruções ADD e INC neste caso. A mais óbvia é que a instrução ADD EAX, 1 custou três *bytes* no programa, enquanto a instrução INC EAX utilizou somente um. Isso pode parecer capricho, mas não é: binários compilados possuem normalmente milhares de instruções Assembly e a diferença de tamanho no resultado final pode ser significativa.



Existem sistemas onde cada *byte* economizado num binário é valioso. Alguns exigem que os binários sejam os menores possíveis, tanto em disco (ou memória *flash*) quanto sua imagem na memória RAM. Este consumo de

memória é por vezes chamado de *footprint*, principalmente em literatura sobre sistemas embarcados.

Outra vantagem da INC sobre a ADD é a velocidade de execução, já que a segunda requer que o processador leia os operandos.

A instrução SUB funciona de forma similar e para subtrair somente uma unidade, também existe uma instrução DEC (de decremento). Vamos então estudar um pouco sobre a instrução MUL agora. Esta instrução tem o primeiro operando (o de destino) **implícito**, ou seja, você não precisa fornecê-lo: será sempre EAX ou uma sub-divisão dele, dependendo do tamanho do segundo operando (de origem), que pode ser um outro registrador ou um endereço de memória. Analise:

mul.s

```
1  BITS 32
2
3  global start
4
5  section .text
6  start:
7      mov eax, 5
8      mov ebx, 2
9      mul ebx
```

A instrução MUL EBX vai realizar uma multiplicação sem sinal (sempre positiva) de EBX com EAX e armazenar o resultado em EAX.



Perceba que não se pode fazer diretamente MUL EAX, 2. Foi preciso colocar o valor 2 em outro registrador antes, já que a MUL não aceita um valor literal como operando.

A instrução DIV funciona de forma similar, no entanto, é recomendável que o leitor faça testes e leia sobre estas instruções no manual da Intel caso queira se aprofundar no entendimento delas.

Neste ponto acredito que o leitor esteja confortável com a aritmética em processadores x86, mas caso surjam dúvidas, não deixe de enviá-las em nosso [fórum de discussão](#). □

---

## Operações bit-a-bit

Já explicamos o que são as operações bit-a-bit quando falamos sobre cálculo com binários então vamos dedicar aqui à particularidades de seu uso. Por exemplo, a instrução XOR, que faz a operação OU EXCLUSIVO, pode ser utilizada para zerar um registrador, o que seria equivalente a mover o valor 0 para o registrador, só que muito mais rápido. Analise:

```
1  b9 00 00 00 00      mov    ecx,0x0
2  31 c9                xor    ecx,ecx
```

Além de menor em *bytes*, a versão XOR é também mais rápida. Em ambas as instruções, o resultado é que o registrador ECX terá o valor 0 e *flag* ZF será *setada*, como em qualquer operação que resulte em zero.

Faça você mesmo testes com as instruções AND, OR, SHL, SHR, ROL, ROR e NOT. Todas as suas operações já foram explicadas na seção [Cálculos com binários](#).

---

## Comparando valores

Sendo uma operação indispensável ao funcionamento dos computadores, a comparação precisa ser muito bem compreendida. Instruções chave aqui são a CMP (*Compare*) e [TEST](#). Analise o código a seguir:

```
1  b8 b0 b0 00 00      mov     eax,0xb0b0
2  3d 10 fe 00 00      cmp     eax,0xfe10
```

A instrução CMP neste caso compara o valor de EAX (previamente *setado* para 0xB0B0) com 0xFE10. O leitor tem alguma ideia de como tal comparação é feita matematicamente? Acertou quem pensou em diminuir de EAX o valor a ser comparado. Dependendo do resultado, podemos saber o resultado da comparação da seguinte maneira:

- Se o resultado for **zero**, então os operandos de destino e origem são **iguais**.
- Se o resultado for um número **negativo**, então o operando de destino é **maior** que o de origem.
- Se o resultado for um número **positivo**, então o operando de destino é **menor** que o de origem.



O resultado da comparação é configurado no registrador EFLAGS, o que significa dizer que a instrução CMP **altera** as *flags*, para que instruções futuras tomem decisões baseadas nelas. Por exemplo, para operandos iguais, a CMP faz ZF=1.

A instrução CMP é normalmente precedida de um salto, como veremos a seguir.

---

## Alterando o fluxo do programa


A ideia de fazer uma comparação é tomar uma decisão na sequência. Neste caso, **decisão** significa para onde transferir o fluxo de execução do programa, o que é equivalente a dizer para onde **pular**, **saltar**, ou para onde **apontar o EIP** (o ponteiro de instrução). Uma maneira de fazer isso é com as instruções de saltos (*jumps*).


### Salto incondicional

Existem vários tipos de saltos. O mais simples é o salto **incondicional** produzido pela instrução **JMP**, que possui apenas um operando, podendo ser um valor literal, um registrador ou um endereço de memória. Para entender, analise o programa abaixo:

```
1      0:      b8 01 00 00 00      mov     eax,0x1
2      5:      eb 03                jmp     0xa
3      7:      83 c0 04             add     eax,0x4
4      a:      40                  inc     eax
```

A instrução **ADD EAX, 4** nunca será executada pois o salto faz a execução pular para o endereço **0x0A**, onde temos a instrução **INC EAX**. Portanto, o valor final de **EAX** será **2**.

 Note aqui o *opcode* do salto incondicional **JMP**, que é o **0xEB**. Seu argumento, é o número de *bytes* que serão pulados, que no nosso caso, são **3**. Isso faz a execução pular a instrução **ADD EAX, 4** inteira, já que ela tem exatamente **3 bytes**.

 Você pode entender o salto incondicional **JMP** como um comando **goto** na linguagem de programação **C**, mas não conte a ninguém que eu te falei isso. □

## Saltos condicionais sem sinal

Os saltos condicionais **Jcc** onde *cc* significa *condition code*, podem ser de vários tipos. O mais famoso deles é o **JE (Jump if Equal)**, utilizado para saltar quando os valores da comparação anterior são iguais. Em geral ele vem precedido de uma instrução **CMP**, como no exemplo abaixo:

```
1      0:      b8 01 00 00 00      mov     eax,0x1
2      5:      83 f8 01            cmp     eax,0x1
3      8:      74 03                je      0xd
4      a:      83 c0 03            add     eax,0x3
```

5

d: 40

inc eax

A instrução no endereço 0x5 compara o valor de EAX com 1 e vai sempre resultar em verdadeiro neste caso, o que significa que a *zero flag* será *setada*.

O salto JE ocorre se  $ZF=1$ , ou seja, se a *zero flag* estiver *setada*. Por essa razão, ele também é chamado de **JZ (*Jump if Zero*)**. Abaixo uma tabela com os saltos que são utilizados para comparações entre números sem sinal e as condições para que o salto ocorra:

Instrução	Alternativa	Condição
JZ (Zero)	JE (Equal)	$ZF=1$
JNZ (Not Zero)	JNE (Not Equal)	$ZF=0$
JC (Carry)	JB (Below) ou JNAE (Not Above or Equal)	$CF=1$
JNC (Not Carry)	JNB (Not Below) ou JAE (Above or Equal)	$CF=0$
JA (Above)	JNBE (Not Below or Equal)	$CF=0$ e $ZF=0$
JNA (Not Above)	JBE (Below or Equal)	$CF=1$ ou $ZF=1$
JP (Parity)	JPE (Parity Even)	$PF=1$
JNP (Not Parity)	JPO (Parity Odd)	$PF=0$
JCXZ (CX Zero)		Registrador CX=0
JECXZ (ECX Zero)		Registrador ECX=0

Nem é preciso dizer que vai ser necessário você criar programas em Assembly para treinar a compreensão de cada um dos saltos, é? ☐

## Saltos incondicionais com sinal

Já vimos que comparações são na verdade subtrações, por isso os resultados são diferentes quando utilizados números com e sem sinal. Apesar de a instrução ser a mesma

(CMP), os saltos podem mudar. Eis os saltos para comparações com sinal:

Instrução	Alternativa	Condição
JG (Greater)	JNLE (Not Less or Equal)	
JGE (Greater or Equal)	JNL (Not Less)	
JL (Less)	JNGE (Not Greater or Equal)	
JLE (Less or Equal)	JNG (Not Greater)	
JS (Sign)		SF=1
JNS (Not Sign)		SF=0
JO (Overflow)		OF=1
JNO (Not Overflow)		OF=0

Não se preocupe com a quantidade de diferentes instruções na arquitetura. O segredo é ir estudando-as conforme o necessário. Para avançar, só é preciso que você entenda o conceito do salto. Muitos problemas de engenharia reversa são resolvidos com o entendimento de um simples JE (ZF=1). Se você já entendeu isso, é suficiente para prosseguir. Se não, volte uma casa. ☐☐



# Funções e pilha

Apesar de não estudarmos todos os aspectos da linguagem Assembly, alguns assuntos são de extrema importância, mesmo para os fundamentos da engenharia reversa de software. Um deles é como funcionam as funções criadas em um programa e suas chamadas, que discutiremos agora.

## O que é uma função

Basicamente, uma função é um **bloco de código reutilizável** num programa. Tal bloco faz-se útil quando um determinado conjunto de instruções precisa ser invocado em vários pontos do programa. Por exemplo, suponha um programa que precise converter a temperatura de Fahrenheit para Celsius várias vezes no decorrer de seu código:

fahrenheit2celsius.py

```
1 fahrenheit = 230.4
2 celsius = (fahrenheit - 32) * 5 / 9
3 print(celsius)
4
5 fahrenheit = 130.3
6 celsius = (fahrenheit - 32) * 5 / 9
7 print(celsius)
8
9 fahrenheit = 90.1
10 celsius = (fahrenheit - 32) * 5 / 9
11 print(celsius)
```

O programa acima funciona e a saída é, conforme esperada:

```
1 110.22222222222223
2 54.611111111111114
3 32.277777777777778
```

No entanto, é pouco prático, pois repetimos o mesmo código várias vezes. Além disso, uma versão compilada fica maior em *bytes*. Também prejudica a manutenção do código pois se o programador precisar fazer uma alteração no cálculo, vai ter que alterar em todos eles. É aí que entram as funções. Veja:

fahrenheit2celsius.py


```
1 def fahrenheit2celsius(fahrenheit):
2     return (fahrenheit - 32) * 5 / 9
3
4 celsius = fahrenheit2celsius(230.4)
5 print(celsius)
6
7 celsius = fahrenheit2celsius(130.3)
8 print(celsius)
9
10 celsius = fahrenheit2celsius(90.1)
11 print(celsius)
```

A saída é a mesma, mas agora o programa está utilizando uma função, onde o cálculo só foi definido uma única vez e toda vez que for necessário, o programa a chama.

Uma função tem:

1. Argumentos, também chamados de parâmetros, que são os dados que a função recebe, necessários para cumprir seu propósito.
2. Retorno, que é o resultado da conclusão do seu propósito, seja bem sucedida ou não.
3. Um nome (na visão do programador) ou um endereço de memória (na visão do processador).

Agora cabe a nós estudar como isso tudo funciona em baixo nível. Pronto? ☐ ☐

 Nos primórdios da computação as funções eram chamadas de **procedimentos** (*procedures*). Em algumas linguagens de programação, no entanto, possuem

tanto funções quanto procedimentos. Estes últimos são "funções que não retornam nada". Já no paradigma da programação orientada a objetos (POO), as funções de uma classe são chamadas de **métodos**.

## Funções em Assembly

Em baixo nível, uma função é implementada basicamente num bloco que não será executado até ser chamado por uma instrução CALL. Ao final de uma instrução, encontramos normalmente a instrução RET. Vamos analisar uma função simples de soma para entender:

```
1  #include <stdio.h>
2
3  int soma(int x, int y) {
4      return x+y;
5  }
6
7  int main(void) {
8      printf("%d\n", soma(3,4));
9      return 0;
10 }
```

Olha como ela fica compilada no Linux em 32-bits:

```
1  0804840b <soma>:
2      804840b:    55                push    ebp
3      804840c:    89 e5             mov     ebp,esp
4      804840e:    8b 55 08           mov     edx,DWORD PTR [ebp+0x8]
5      8048411:    8b 45 0c           mov     eax,DWORD PTR [ebp+0xc]
6      8048414:    01 d0             add     eax,edx
7      8048416:    5d                pop     ebp
8      8048417:    c3                ret
9
10 08048418 <main>:
11 ...
12 8048429:    6a 04             push    0x4
13 804842b:    6a 03             push    0x3
```

```
14 804842d: e8 d9 ff ff ff call 804840b <soma>
15 8048432: 83 c4 08 add esp,0x8
```

Removi partes do código intencionalmente, pois o objetivo neste momento é apresentar as instruções que implementam as chamadas de função. Por hora, você só precisa entender que a instrução CALL (no endereço 0x804842d em nosso exemplo) chama a função *soma()* em 0x0804840b e a instrução RET (em 0x8048417) retorna para a instrução imediatamente após a CALL (0x8048432), para que a execução continue.

## A pilha de memória

A memória RAM para um processo é dividida em áreas com diferentes propósitos. Uma delas é a pilha, ou *stack*.

Essa área de memória funciona de forma que o que é colocado lá fique no topo e o último dado colocado na pilha seja o primeiro a ser retirado, como uma pilha de pratos ou de cartas de baralho mesmo. Esse método é conhecido por LIFO (*Last In First Out*).

Seu principal uso é no uso de funções, tanto para passagem de argumentos (parâmetros da função) quanto para alocação de variáveis locais da função (que só existem enquanto a função executa).

Na arquitetura IA-32, a pilha é alinhada em 4 *bytes* (32-bits). Por consequência, todos os seus endereços também o são. Logo, se novos dados são colocados na pilha (empilhados), o endereço do topo é **decrementado** em 4 unidades. Se um dado for desempilhado, o endereço do topo é **incrementado** em 4 unidades. Perceba a lógica invertida, porque a pilha começa num endereço alto e cresce em direção a endereços menores.

Existem dois registradores diretamente associados com a pilha de memória alocada para um processo. São eles:

- O ESP, que aponta para o topo da pilha.
- O EBP, que aponta para a base do *stack frame*.

Veremos agora as instruções de manipulação de pilha. A primeira é a instrução PUSH (do inglês "empurrar") que, como o nome sugere, empilha um dado. Na forma abaixo, essa instrução faz com que o processador copie o conteúdo do registrador EAX para o topo da pilha:

```
push eax
```

Também é possível empilhar um valor literal. Por exemplo, supondo que o programa coloque o valor um na pilha:

```
push 1
```

Além de copiar o valor proposto para o topo da pilha, a instrução PUSH **decrementa** o registrador ESP em 4 unidades, conforme já explicado o motivo. Sempre.

Sua instrução antagônica é a POP, que só precisa de um registrador de destino para copiar lá o valor que está no topo da pilha. Por exemplo:

```
pop edx
```

Seja lá o que estiver no topo da pilha, será copiado para o registrador EDX. Além disso, o registrador ESP será **incrementado** em 4 unidades. Sempre.

Temos também a instrução CALL, que faz duas coisas:

1. Coloca o endereço da próxima instrução na pilha de memória (no caso do exemplo, 0x8048432).
2. Coloca o seu parâmetro, ou seja, o endereço da função a ser chamada, no registrador EIP (no exemplo é o endereço 0x804840b).

Por conta dessa atualização do EIP, o fluxo é desviado para o endereço da função chamada. A ideia de colocar o endereço da próxima instrução na pilha é para o processador saber para onde tem que voltar quando a função terminar. E, falando em terminar, a estrela do fim da festa é a instrução RET (de *RETURN*). Ela faz uma única coisa:

1. Retira um valor do topo da pilha e coloca no EIP.

Isso faz com que o fluxo de execução do programa volte para a instrução imediatamente após a CALL, que chamou a função.

---

## Análise da MessageBox

Vamos agora analisar a pilha de memória num exemplo com a função MessageBox, da API do Windows:

```
1 00401516 | 6A 31 | push 31 |  
2 00401518 | 68 00 | push msgbox.404000 | 404000:"  
3 0040151D | 68 07 | push msgbox.404007 | 404007:"  
4 00401522 | 6A 00 | push 0 |  
5 00401524 | E8 E8 | call <user32.MessageBoxA> |
```

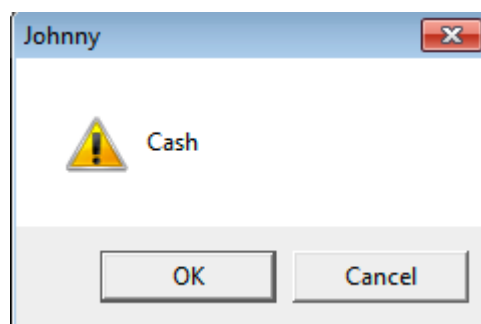
Perceba que quatro parâmetros são empilhados antes da chamada à *MessageBoxA* (versão da função *MessageBox* que recebe *strings* ASCII, por isso o sufixo **A**).

Os parâmetros são empilhados na ordem inversa.

Já estudamos o protótipo desta função no capítulo que apresenta a [Windows API](#) e por isso sabemos que o 0x31, empilhado em 00401516, é o parâmetro *uType* e, se o decompormos, veremos que 0x31 é um OU entre 0x30 (MB\_ICONEXCLAMATION) e 0x1 (MB\_OKCANCEL).

O próximo parâmetro é o número 404000, um ponteiro para a *string* "Johnny", que é o título da mensagem. Depois vem o ponteiro para o texto da mensagem e por fim o zero (NULL), empilhado em 00401522, que é o *handle*.

O resultado é apresentado a seguir:



Resultado da chamada à MessageBox

É importante perceber que, após serem compreendidos, podemos controlar estes parâmetros e alterar a execução do programa conforme quisermos. Este é o assunto do próximo capítulo, sobre depuração.

# Depuração

Chegamos no capítulo onde a engenharia reversa de fato começa. Aqui vamos estudar a depuração, ou *debugging* em inglês. O conceito, como o nome em sugere, é buscar identificar erros (*bugs*) num programa, a fim de corrigi-los. No entanto, os *debuggers* - como são chamados os softwares que servem a este fim - servem para muito mais que isso.

Neste livro usaremos o [x64dbg](#) por ser um *debugger* de código aberto, gratuito e muito atualizado para Windows.

Na próxima seção apresentaremos como baixar e configurar o x64dbg. Também utilizaremos um binário de exemplo durante o livro, que é um desafio chamado de [AnalyseMe00](#). Com ele estudaremos os conceitos de engenharia reversa que precisamos para criar um fundamento sólido para avançar nesta área.



# O debugger

## Instalação

1. Numa máquina virtual com o Windows (aqui utilizamos o Windows 7 mas qualquer um superior a este serve), baixe o [snapshot](#) mais recente do x64dbg. É um arquivo .zip chamado *snapshot\_YYYY-MM-DD\_HH-MM.zip* que vai variar dependendo da data e hora do *release* (quando o software é liberado) pelo autor do projeto.
  2. Ao descompactar o arquivo .zip, execute o arquivo *x96dbg.exe* dentro do diretório *release*. Esse nome deve-se ao fato de que o x64dbg tem suporte tanto a 32 quanto a 64-bits, então o autor resolveu somar 32+64 e nomear o binário assim. :)
  3. O *x96dbg.exe* é o *launcher* do x64dbg e tem três botões. Escolha **Setup** e responda "Sim" para todas as perguntas.
  4. À esta altura você já deve ter o atalho x32dbg na área de trabalho. Ao clicar, você verá a tela inicial do *debugger*. A ideia é que você depure binários (.exe, .dll, etc) portanto, vamos abrir o AnalyseMe-00.exe e seguir.
- 

## Configuração

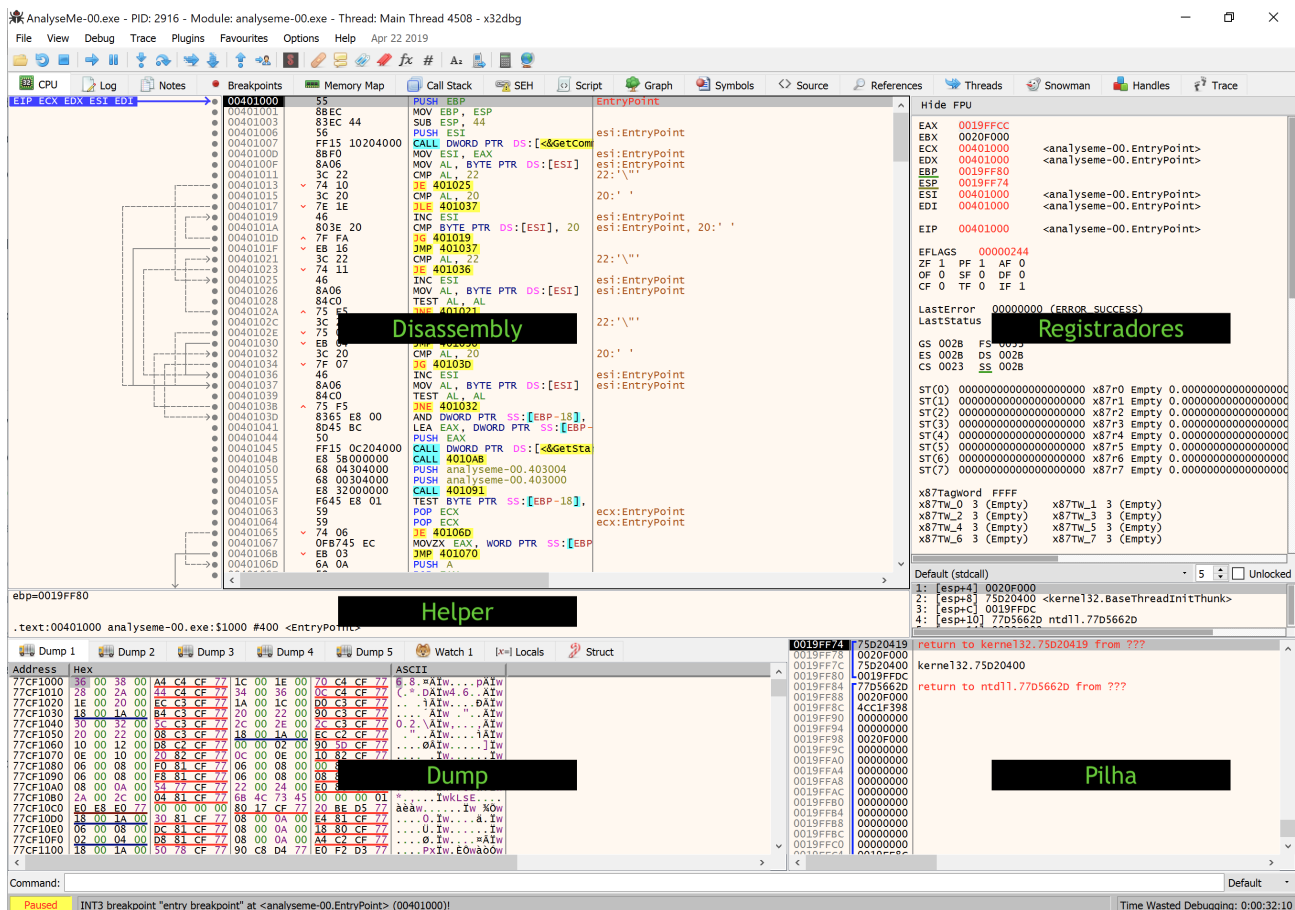
1. Vá em **Options -> Settings** e desmarque a caixa **System breakpoint**. Isso vai fazer com que o *debugger* pare direto no *entrypoint* de um programa ao abrímos. Clique em **Save**.

Existem muitas outras opções de configuração que você pode experimentar, mas para o momento esta basta.

---

## Tela inicial

Se ainda não o fez, faça download do nosso binário de exemplo, o [AnalyseMe-00.exe](#) e abra-o no x32dbg clicando em **File -> Open**. Você deverá ver uma tela como esta:



## AnalyseMe-00.exe aberto no x32dbg

A aba **CPU** é sem dúvida a mais utilizada no processo de *debugging*, por isso fizemos questão de nomear algumas de suas áreas, que descreveremos agora.

## Disassembly

Nesta região são exibidos os endereços (VA's), os *opcodes* e argumentos em *bytes* de cada instrução, seu *disassembly* (ou seja, o que significam em Assembly) e alguns comentários úteis na quarta coluna, como a palavra *EntryPoint*, na primeira instrução do programa a ser executada (em 40104D no nosso exemplo).

## Helper

Tomei a liberdade de nomear essa seção de **Helper**, porque de fato ela ajuda. Por exemplo, quando alguma instrução faz referência à um dado em memória ou em um registrador, ela já mostra que dado é este. Assim você não precisa ir buscar. É basicamente um

economizador de tempo. Supondo que o *debugger* esteja parado na instrução `push esi`, no Helper aparecerá o valor do registrador ESI.

## Dump

O dump é um visualizador de que você pode usar para inspecionar *bytes* em qualquer endereço. Por padrão há cinco abas de dump, mas você pode adicionar mais se precisar.

## Registradores

Como o nome sugere, mostra o valor de cada registrador do processador.

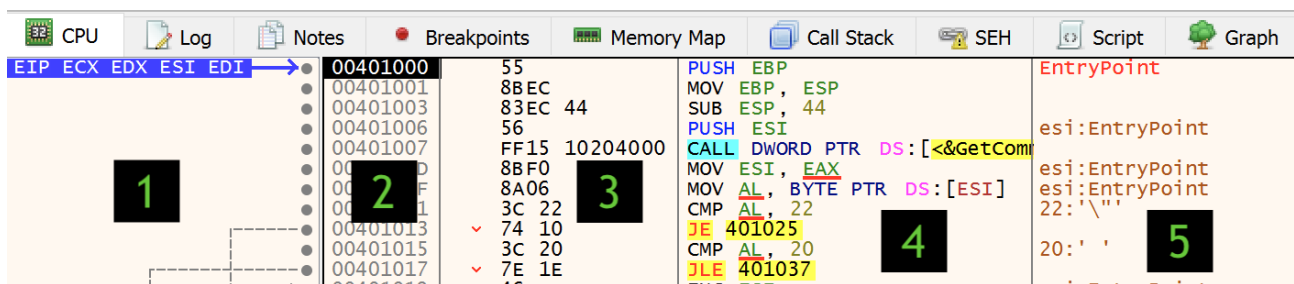
## Pilha

Mostra a pilha de memória, onde o endereço com fundo em preto indica o topo da pilha.

Na próxima seção, iremos depurar o binário de exemplo e devemos nos atentar às informações exibidas em cada uma das regiões da tela do debugger, acima apresentadas. Preparado? :)

# Disassembly

Ao observar a região que chamamos de disassembly, você verá 5 colunas onde a primeira exibe algumas informações e relações entre endereços. A segunda mostra os endereços em si. A terceira mostra os *bytes* do opcode e operandos das instruções. A quarta mostra as os mnemônicos onde podemos ler Assembly e por fim, a quinta mostra alguns comentários, sejam estes gerados automaticamente pelo debugger ou adicionados pelo usuário.



Colunas do disassembly

## Destaque de instruções

Perceba que por padrão o debugger já destaca vários aspectos das instruções, na janela de disassembly. O nome mais comum para este destaque é sua versão em inglês *highlight*.

O *highlight* refere-se principalmente às cores, mas o debugger também dá altas dicas. Na instrução acima, perceba que a operação em si está em azul, enquanto o argumento está em verde.

O endereço para o qual o ponteiro de instrução aponta (EIP, embora o x64dbg chame-o genericamente de CIP, já que em x86-64 seu nome é RIP) é destacado com um fundo preto (na imagem anterior, vemos que é o endereço 401000).

Na coluna de comentários, temos os comentários automáticos em marrom.

É importante lembrar que no arquivo há somente os *bytes* referentes às instruções (terceira coluna). Todo o resto é interpretação do debugger para que a leitura seja mais intuitiva.

## Step into/over

Neste primeiro momento, o debugger está parado e a próxima instrução a ser executada é justamente o que chamamos de OEP (*Original EntryPoint*).

O primeiro comando que aprenderemos é o **Step over**, que pode ser ativado por pelo menos três lugares:

1. Menu **Debug** -> **Step over**.
2. Botão **Step over** na barra de botões (é o sétimo botão).
3. Tecla de atalho F8.
4. Digitando um dos comandos a seguir na barra de comandos: StepOver/step/sto/st

Se você emitir este comando uma vez, verá que o debugger vai executar uma única instrução e parar. Na janela do disassembly, você vai perceber que o cursor (EIP) "pulou uma linha" e a instrução anterior foi executada. No caso de nosso binário de teste, é a instrução PUSH EBP. Após sua execução, perceba que o valor de EBP foi agora colocado no topo da pilha (observe a pilha, abaixo dos registradores).

Você pode seguir teclando F8 até alcançar a primeira instrução CALL em 401007, destacada por um fundo azul claro.

O comando **Step over** sobre uma CALL faz com que o debugger execute a rotina apontada pela instrução e "volte" para o endereço imediatamente após a CALL. Você não verá essa execução, pois o debugger não a instrumentará. Caso queira observar o que foi executado "dentro" da CALL, é necessário utilizar o **Step into** (F7). Vamos fazer dois testes:

1. Com o EIP apontado para a CALL em 401007, tecle F8. Você verá que a execução simplesmente "passa para a linha abaixo da CALL". Isso quer dizer que ela **foi executada**, mas você "não viu no debugger".
  2. Agora reinicie o programa no debugger (F2), vá teclando F8 até chegar sobre a CALL novamente e tecle F7, que é o **Step into**. Perceba que o debugger agora "entrou" na CALL. Neste caso, você vai precisar teclar F8 mais três vezes até voltar ao fluxo de execução original, isso porque esta CALL só possui três instruções.
-

## Run

Um outro comando importante é o **Run** (F9). Ele simplesmente inicia a execução a partir do EIP de todas as instruções do programa. Se você emiti-lo com este binário, vai ver que a execução vai terminar, o que significa que o programa rodou até o final e saiu. Aí basta reiniciar o programa (F2) para recomençar nossos estudos. ;)

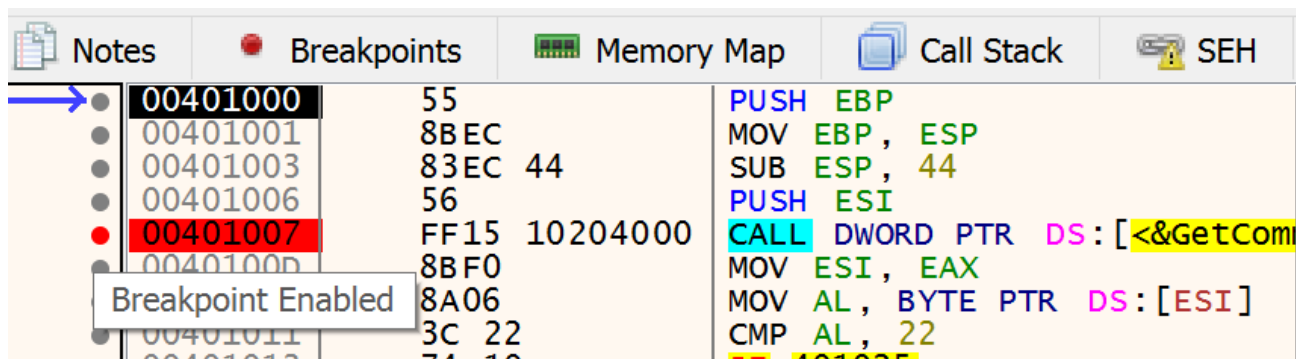
Na próxima seção, vamos entender os pontos de paradas, mais conhecidos como *breakpoints*.

# Breakpoints

Um *breakpoint* nada mais é que um ponto do código onde o debugger vai parar para que você analise o que precisa. É o mesmo conceito dos *breakpoints* presentes nos IDE's como Visual Studio, NetBeans, CodeBlocks, etc. A diferença é que nestas IDE's colocamos *breakpoints* em determinadas linhas do código-fonte. Já nos debuggers destinados à engenharia reversa, colocamos *breakpoints* em **endereços** (RVA's), onde há instruções.

## Seu primeiro *breakpoint*

Há várias maneiras de se colocar um *breakpoint* em um endereço utilizando o x64dbg. Você pode selecionar a instrução e pressionar F2, usar um dos comandos SetBPX/bp/bpx ou simplesmente clicar na pequena bolinha cinza à esquerda do endereço. Ao fazê-lo, este ficará com um fundo vermelho, como sugere a imagem:



Colocando um breakpoint na CALL

Um segundo clique desabilita o *breakpoint*, mas não o exclui da aba **Breakpoints** (Alt+B). O terceiro clique de fato o deleta.

Após colocar o *breakpoint* nesta CALL, rode o programa (F9). O que acontece? O debugger executa todas as linhas anteriores a este *breakpoint* e pára onde você pediu. Simples assim.

## Como *breakpoints* são implementados

Talvez você tenha notado que ao atingir um *breakpoint*, o x64dbg denuncia na barra de status: **INT3 breakpoint at analyseme-00.00401007 (00401007)!**. Este é um tipo de **breakpoint de software**. Existem ainda os de memória e de hardware, mas não trataremos neste curso básico.

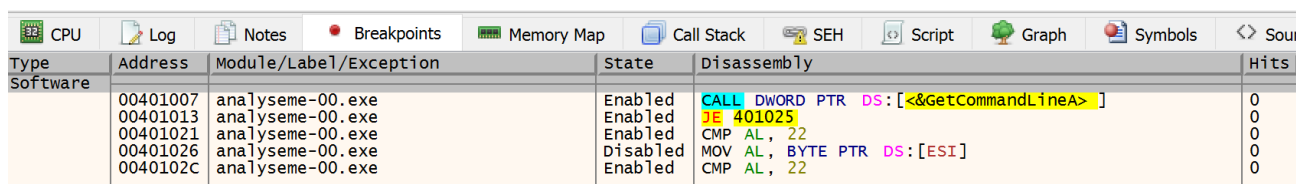
A instrução INT é uma instrução Assembly que gera uma interrupção. A interrupção número 3 é chamada de **Breakpoint Exception (#BP)** no manual da Intel. Seu opcode (0xcc) tem somente um *byte*, o que facilita para que os debuggers implementem-na.

De forma resumida, para parar nesta CALL, o que o x64dbg faz é:

1. Substituir o primeiro *byte* do opcode da CALL (0xff, neste caso) por 0xcc e salvar o original numa memória.
2. Executar o programa.
3. Restaurar o primeiro *byte* do opcode da CALL, substituindo o 0xcc por 0xff (neste caso).

Isso poderia ser feito manualmente, mas os debuggers facilitam o trabalho, bastando você pressionar F2 ou clicar na bolinha para que todo este trabalho seja executado em segundo plano, sem que o usuário perceba. Incrível, não? :)

Você pode adicionar quantos *breakpoints* de software quiser numa sessão de debugging. Todos ficam salvos na aba **Breakpoints**, a não ser que você os exclua. Veja a imagem abaixo:



Type	Address	Module/Label/Exception	State	Disassembly	Hits
Software	00401007	analyseme-00.exe	Enabled	CALL DWORD PTR DS:[<&GetCommandLineA>]	0
	00401013	analyseme-00.exe	Enabled	JE 401025	0
	00401021	analyseme-00.exe	Enabled	CMP AL, 22	0
	00401026	analyseme-00.exe	Disabled	MOV AL, BYTE PTR DS:[ESI]	0
	0040102C	analyseme-00.exe	Enabled	CMP AL, 22	0

Lista de breakpoints

Você também pode assistir a aula a seguir do CERO, que trata sobre este assunto.





**Curso de Engenharia Reversa Online - Aula  
16 - Breakpoints de software**

[https://www.youtube.com/watch?  
v=823KK-FYV9s](https://www.youtube.com/watch?v=823KK-FYV9s)

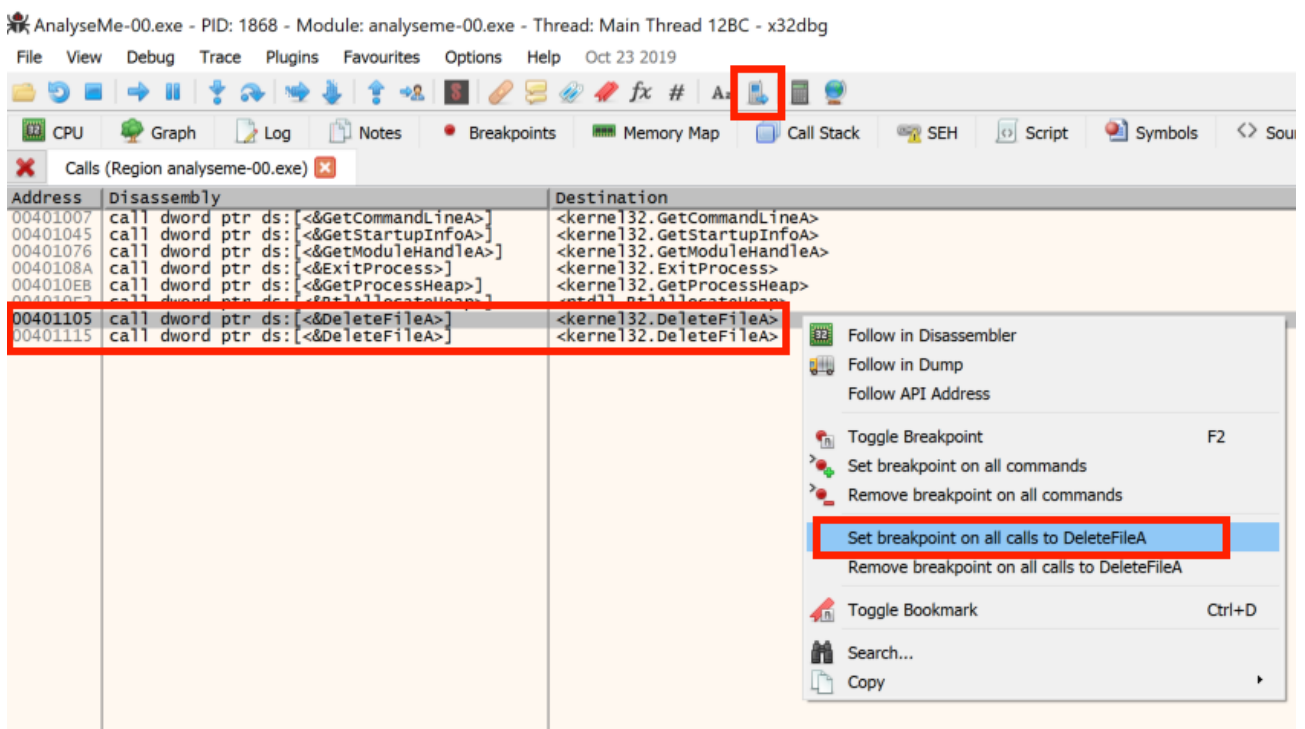
Curso de Engenharia Reversa Online - Aula 16 - Breakpoints de software

# Manipulação do fluxo

Agora que já sabemos o básico do *debugging* e sabemos colocar *breakpoints* de software, vamos começar a manipular o programa da maneira que queremos.

Em geral, quando falamos de manipulação, falamos de alguma alteração no código do programa, para que este execute o que queremos, da forma como queremos.

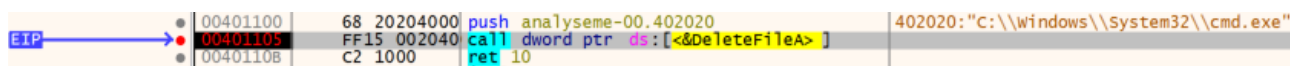
Tomemos como exemplo o AnalyseMe-00 mesmo. Supondo que não sabemos o que o mesmo faz. Um bom início para a engenharia reversa é a busca por chamdas intermodulares (o botão com um celular e uma seta azul). Ao clicar nele, encontrará duas chamdas à função DeleteFileA da KERNEL32.DLL. Colocaremos um breakpoint em todas as chamadas à estas funções, bastando para isso dar um clique com o botão direito do mouse em uma delas e escolher a opção "Set breakpoint on all calls to DeleteFileA", como sugere a imagem abaixo:



Chamadas intermodulares

## Encontrando o local para manipulação

Ao voltar à aba CPU e rodar o programa (F9), paramos aqui:



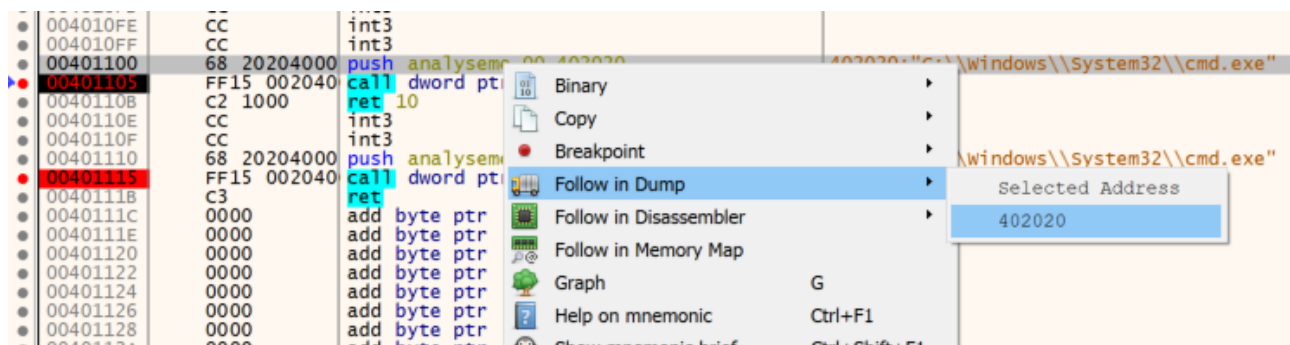
EIP	00401100	68 20204000	push analyseme-00.402020	402020: "C:\windows\System32\cmd.exe"
	00401105	FF15 002040	call dword ptr ds:[402020]	
	00401108	C2 1000	ret 10	

Função que chama a DeleteFileA

Esta função é bem simples. No endereço 00401105 há um PUSH que coloca o endereço 402020 na pilha, depois há a chamada da DeleteFileA em si.

O x64dbg já resolve a referência do endereço e, caso encontre uma string, exibe ao lado (na quarta coluna), como acontece com a string "C:\Windows\System32\cmd.exe". Ora, se este é o argumento passado para a função DeleteFile(), este é o caminho do arquivo que o programa AnalyseMe-00 pretende deletar.

O que a gente vai fazer é mudar esta string, mudando assim o programa que o AnalyseMe-00 tenta deletar. Para isso, clique com botão direito do mouse sobre a instrução PUSH e escolha "Follow in Dump -> 402020".



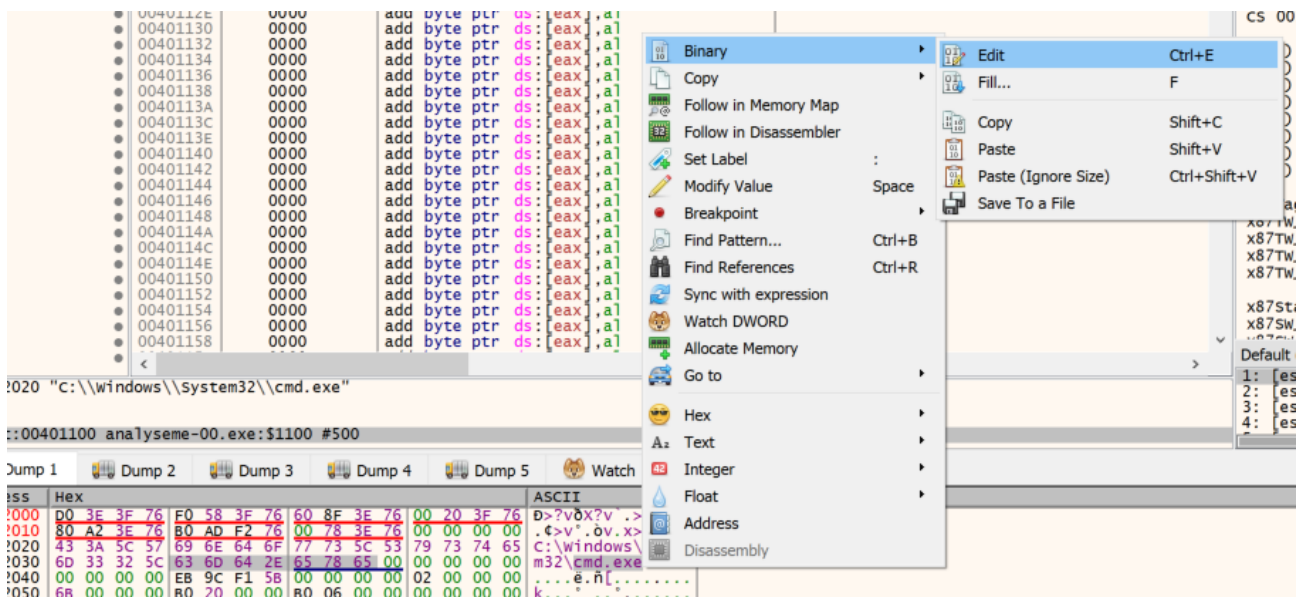
Seguindo o endereço da string no Dump

O endereço em questão é exibido no Dump 1. Outra opção seria ir no Dump 1, teclar Ctrl+G, digitar 402020 e clicar em OK. ☐

## Alterando a string

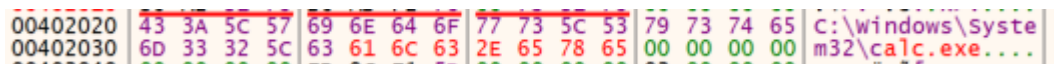
Para alterar a string, você vai precisar selecionar todos os bytes desejados, pois o x64dbg não sabe exatamente onde começa e onde termina cada bloco de dados usado pelo programa. Supondo que queiramos alterar "cmd.exe" para "calc.exe", fazendo assim com

que o programa tente excluir a calculadora do Windows. Para este caso, selecionamos o trecho e pressionamos Ctrl+E, que é o equivalente ao clicar com o botão direito sobre a seleção e escolher "Binary -> Edit".



Editando os bytes da string string

Após fazer a alteração e clicar em OK, perceba que o Dump 1 agora destaca os bytes alterados em vermelho:



Bytes alterados exibidos em vermelho no Dump

Ao seguir com a execução da chamada à DeleteFileA (F8), o programa tenta excluir o calc.exe ao invés de o cmd.exe. No entanto, como em versões modernas do Windows o conteúdo deste diretório é protegido, a função retorna zero (perceba o registrador EAX zerado), que no caso desta função, indica que houve **falha**, e as variáveis **LastError** e **LastStatus** são modificadas para refletir o que aconteceu.

Hide FPU		
EAX	00000000	
EBX	0037F000	
ECX	00650000	
EDX	00650000	
EBP	0019FF70	
ESP	0019FF14	
ESI	00653719	
EDI	00401000	<analyseme-00.EntryPoint>
EIP	0040110B	analyseme-00.0040110B
EFLAGS	00000244	
ZF	1	PF 1 AF 0
OF	0	SF 0 DF 0
CF	0	TF 0 IF 1
LastError	00000005	(ERROR_ACCESS_DENIED)
LastStatus	C0000022	(STATUS_ACCESS_DENIED)

Retorno zero em EAX e variáveis LastError e LastStatus em vermelho

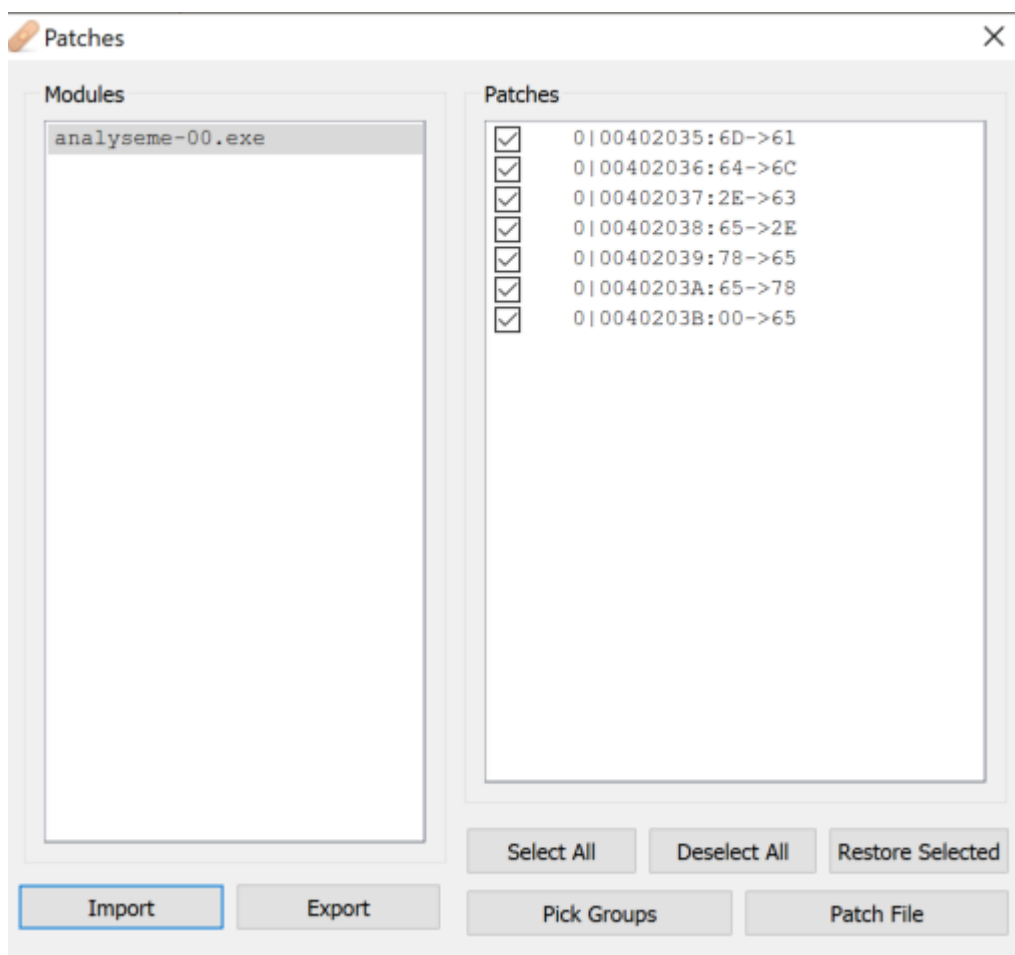
Espero que com esta seção você entenda que, tendo o programa sob o controle de um debugger, é possível modificar praticamente tudo o que queremos. Podemos impedir que funções sejam chamadas, podemos chamar novas funções, alterar dados, modificar parâmetros, enfim, a lista é quase infinita. Na próxima seção vamos ver como salvar as alterações feitas.

# Patches

Um *patch* é qualquer alteração, seja nos dados ou no código de um programa. O que fizemos na seção anterior é justamente isto. No entanto, não salvamos nossas modificações e neste caso elas serão perdidas caso você feche o x64dbg.

É possível salvar as alterações acessando o menu **View -> Patch file...**, clicando no botão com um pequeno curativo (tipo um band-aid) na barra de ferramentas ou pressionando Ctrl+P.

Se você veio da seção anterior e tem ainda as modificações no AnalyseMe-00, sua tela de patches aparecerá assim:



Visualizando as modificações feitas

A partir desta tela, é possível exportá-los para um arquivo (Export), mas também criar um novo arquivo executável com as alterações salvas (Patch File).

Perceba que você pode fazer qualquer tipo de patch, desde uma simples alteração da lógica de um salto até inserir de fato novas funções num programa. Pode dar trabalho, mas é possível. □

# Apêndices

Esses apêndices servem tanto de referência como de material de apoio para estudar assuntos que não foram abordados com profundidade neste livro.

→ **Tabela ASCII**

</apendices/tabela-ascii>

→ **Exemplos de código em Assembly**

</apendices/exemplos-de-codigo-em-assembly>

→ **Ferramentas**

</apendices/ferramentas>

→ **Referências**

</apendices/referencias>



# Tabela ASCII

	Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex
1																							
2	0	00	NUL	16	10	DLE	32	20		48	30	0	64	40	@	80	50	P	96	60	`	112	70
3	1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71
4	2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72
5	3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73
6	4	04	EOT	20	14	DC4	36	24	\$	52	34	4	68	44	D	84	54	T	100	64	d	116	74
7	5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75
8	6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76
9	7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77
10	8	08	BS	24	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78
11	9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79
12	10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	80
13	11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	81
14	12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	82
15	13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	83
16	14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	84
17	15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	85

# Exemplos de código em Assembly

Reuno aqui alguns exemplos de códigos em Assembly, úteis para a compreensão de trechos de binários quando fazemos engenharia reversa.

## Zerar variáveis

Assembly

```
xor eax, eax
```

C

```
int eax=0;
```

## Contagem de 1 a 10

Assembly v1

```
1 xor ecx, ecx
2 loop:
3   inc ecx
4   cmp ecx, 0xa
5   jl loop
```

### Assembly v2

```
1 mov ecx, 0
2 loop:
3   add ecx, 1
4   cmp ecx, 0x9
5   jle loop
```

### C

```
1 int ecx;
2 for (ecx=0; ecx<10; ecx++) {}
```

## Testar se é zero

### Assembly v1

```
1 cmp eax, 0
2 je destino
```

### Assembly v2

```
1 test eax, eax
2 je destino
```

C

```
1  if (eax == 0)
2      // destino
```

## Não fazer nada

Parece bobo, mas "fazer nada" corretamente significa não alterar nenhuma *flag*, nem nenhum registrador. Seguem as instruções que conheço:

Assembly v1

```
xchg eax, eax
```

Assembly v2

```
nop
```

C

```
;
```

Instruções que não fazem nada também podem ser utilizadas como *padding* necessário para o correto alinhamento das seções do binário em memória. Já vi o GCC utilizar XCHG AX, AX neste caso.

# Funções da API do Windows

Segue uma lista de funções interessantes para colocarmos *breakpoints* quando revertendo binários em Windows.

---

## Caixas de diálogo

**MessageBoxA**

**MessageBoxW**

**MessageBoxExA**

**MessageBoxExW**

**DialogBoxParamA**

**GetWindowTextA**

**GetDlgItemTextA**

---

## Janelas

**EnableWindow**

**EnableMenuItem**

---

**EnableWindow**

## **Entrada e saída (I/O)**

**CreateFileA**

**CreateFileW**

**OpenFile**

**OpenFileMappingA**

**OpenFileMappingW**

**OpenMutexA**

**OpenMutexW**

**LoadLibraryA**

**LoadLibraryExA**

**LoadLibraryW**

**LoadLibraryExW**

**CreateFileMappingA**

**CopyFileA**

**CopyFileW**

**CopyFileExA**

**CopyFileExW**

**MoveFileA**

**MoveFileW**

**MoveFileExA**

**MoveFileExW**

**DeleteFileA**

**DeleteFileW**

**LoadCursorFromFileA**

**GetPrivateProfileStringA**

**GetPrivateProfileIntA**

---

## **Registro**

**RegOpenKeyA**

**RegOpenKeyExA**

**RegCloseKey**

**RegQueryValueA**



**RegEnumKeyExA**

**RegSetValueA**

**RegSetValueW**

**RegSetValueExA**

**RegSetValueExW**

---

## **Data e hora**

**SetSystemTime**

**GetLocalTime**

**SetLocalTime**

---

## **Processos e *threads***

**CreateToolhelp32Snapshot**

**Process32First**

**Process32Next**

**Process32FirstW**

**Module32First**

**Module32Next**

**Module32FirstW**

**Module32NextW**

**Toolhelp32ReadProcessMemory**

**Heap32ListFirst**

**Heap32ListNext**

**Heap32First**

**Heap32Next**

**OpenProcess**

**TerminateProcess**

**ExitProcess**

**ExitThread**

**OpenProcessToken**

ADVAPI32.DLL

**OpenThreadToken**

ADVAPI32.DLL

## **ZwQueryInformationProcess**

NTDLL.DLL

## **ZwSetInformationThread**

NTDLL.DLL

## **WriteProcessMemory**

## **CreateThread**

## **CreateRemoteThread**

## **CreateProcessA**

---

## **Anti-debugging**

## **IsDebuggerPresent**

## **CheckRemoteDebuggerPresent**

## **NtSetInformationThread**

NTDLL.DLL

---

## **Strings**

**IstrcatA**

**IstrcmpA**

**IstrcpyA**

**IstrlenA**

---

**Disco**

**GetDiskFreeSpaceA**

**GetDriveTypeA**

# Ferramentas

Esta seção aborda não somente ferramentas utilizadas no livro, mas também outras que vale a pena citar na esperança que o leitor se sinta atraído a baixar, usar e tirar suas próprias conclusões em relação à eficiência delas.

## Editores hexadecimais

Este tipo de ferramenta é útil para editar arquivos binários em geral, não somente executáveis, *dumpar* (copiar) conteúdo de trechos de arquivos, etc. Também é possível editar uma partição ou disco com bons editores hexadecimal a fim de recuperar arquivos, por exemplo.

Nome	Licença	Descrição
010 Editor	Shareware	Multiplataforma, bastante usado.
GNU poke	Livre	Editor REPL genérico (não só para executáveis), ainda em beta, mas muito interessante. Escrevemos um <a href="#">artigo</a> sobre.
fhex	Livre	Editor gráfico multiplataforma capaz de exibir o binário graficamente, além de suportar expressões regulares na busca.
Hex Workshop	Comercial	Pago, somente para Windows, antigo, mas muito bem feito.
HexFiend	Livre	Somente para macOS, com recursos legais como diff e data inspector.
Hiew (Hacker's View)	Comercial	Editor (e disassembler) muito poderoso, principalmente por conta de seus módulos HEM. É muito usado por analistas de malware, mas é pago.
HT Editor	Livre	Interface gráfica baseada em texto, parecido com o Hiew. Feio, mas cumpre seu trabalho.

HxD	Freeware	Bem bom. Possui recursos extras como geração de hashes, suporte a abrir discos.
Reverse Engineer's Hex Editor	Livre	Nova proposta de um editor especificamente para engenharia reversa.
wxHexEditor	Livre	Multiplataforma, recursos interessantes.
XVI32	Freeware	Sem muitos recursos, mas quebra um galho.

## Analísadores de executáveis

Nome	Licença	Descrição
DIE (Detect It Easy)	Freeware	Detecta compilador, linker, packer e protectors em binários. Também edita os arquivos.
DUMPBIN	Freeware	Analísador de PE de linha de comando disponível no SDK do Visual Studio.
Exeinfo PE	Freeware	Detecta compilador, packer, protectors e edita os arquivos, além de suportar vários plugins loucos. Tem versão VIP mediante doação.
Malwoverview	Livre	Mais um nacional pra uma primeira impressão de arquivos suspeitos, URLs e domínios. Checa também APK's. :)
objdump	Livre	Parte do GNU binutils, também analisa PE, além de ELF, a.out, etc.
PE-Bear	Freeware	Analísador gráfico (Qt) multiplataforma que também detecta packers/protectors.
PEdump	Livre	Analísador online muito legal!

<a href="#">pestudio</a>	Freeware	Analizador de PE padrão da indústria, com foco em malware. Tem versão Pro (paga).
<a href="#">pev</a>	Livre	Nosso <a href="#">☐</a> toolkit de ferramentas de linha de comando para análise de PE. Artigo introdutório <a href="#">aqui</a> .
<a href="#">readelf</a>	Livre	Também parte do binutils, analisador de ELF.
<a href="#">Stud_PE</a>	Freeware	Analizador e editor com suporte a plugins, assinaturas do antigo PEiD, editor de recursos e mais.

## Bibliotecas para *parsear* executáveis

Nome	Licença	Descrição
<a href="#">BFD</a>	Livre	<b>B</b> inary <b>F</b> ile <b>D</b> escriptor é a biblioteca usada por programas como readpe e objdump. Tem suporte a muitos tipos de arquivos, incluindo PE e ELF, claro.
<a href="#">libpe</a>	Livre	Nossa <a href="#">☐</a> biblioteca multiplataforma para <i>parsing</i> de arquivos PE.
<a href="#">libPeConv</a>	Livre	Biblioteca em C++ para PE usada pelo PE-Bear.
<a href="#">pefile</a>	Livre	Famosa biblioteca em Python pra fazer qualquer coisa com arquivos PE.
<a href="#">PeNet</a>	Livre	Biblioteca em .Net para PE.
<a href="#">PeParser</a>	Livre	Biblioteca em Java para PE.
<a href="#">pyelftools</a>	Livre	Biblioteca em Python para <i>parsear</i> binários ELF.

## Assemblers



Nome	Licença	Descrição
<a href="#">flat assembler (FASM)</a>	Livre	Assembler bem recente que já vem com vários exemplos de código. Windows e Linux.
<a href="#">GNU Assembler (GAS)</a>	Livre	Também chamado simplesmente de <b>as</b> , é o assembler do projeto GNU e provavelmente já está instalado no seu Linux! ;)
<a href="#">Microsoft Macro Assembler (MASM)</a>	Freeware	Atualmente já vem com o Visual Studio da Microsoft, mesmo na versão Community. Segue um <a href="#">tutorial de como compilar um "Hello, world"</a> .
<a href="#">Netwide Assembler (NASM)</a>	Livre	Multiplataforma, suporte à sintaxe Intel e bem popular. Veja um <a href="#">tutorial de como compilar um "Hello, world" no Linux</a> .
<a href="#">Yasm Modular Assembler (YASM)</a>	Livre	Multiplataforma, escrito com base no NASM pra ser um substituto mas acho que não vingou. hehe

## Disassemblers

Nome	Licença	Descrição
<a href="#">Binary Ninja</a>	Comercial	Novo disassembler que teoricamente compete com o IDA. Possui versão <a href="#">online gratuita</a> mediante registro.
<a href="#">Ghidra</a>	Livre	Disassembler livre lançado pela NSA. Temos um <a href="#">treinamento</a> em vídeo sobre ele.
<a href="#">Hopper</a>	Comercial	Disassembler com foco em binários de Linux e macOS.
<a href="#">IDA</a>	Comercial	Disassembler <b>interativo</b> padrão de mercado. Possui versão freeware.



## Debuggers

Nome	Licença	Descrição
<a href="#">dnSpy</a>	Livre	Somente para .NET, descompila, debuga e (dis)assembla.
<a href="#">edb</a> (Evan's Debugger)	Livre	Debugger gráfico (Qt) com foco em binários ELF no Linux.
<a href="#">GDB (GNU Debugger)</a>	Livre	Super conhecido debugger pra Linux do projeto GNU. Tem uma <a href="#">versão não oficial pra Windows</a> também. É modo texto, mas possui vários front-ends como <a href="#">GDBFrontend</a> , dentre <a href="#">outros</a> .
<a href="#">GEF</a>	Livre	O <b>GDB Enhanced Features</b> estende o GDB com recursos para engenharia reversa.
<a href="#">OllyDbg</a>	Freeware	Poderoso debugger, apesar de não mais mantido.
<a href="#">PEDA</a>	Livre	O <b>P</b> ython <b>E</b> xploit <b>D</b> evelopment <b>A</b> ssistance também estende o GDB, assim como o GEF. A interface é um pouco diferente, no entanto.
<a href="#">WinDbg</a>	Freeware	Debugger ring0/3, parte integrante do SDK do Windows.
<a href="#">x64dbg</a>	Livre	Debugger user-mode para Windows com suporte a 32 e 64-bits.

## Descompiladores

Nome	Licença	Descrição
<a href="#">IDR (Interactive Delphi Reconstructor)</a>	Livre	Para binários compilados em Delphi.
<a href="#">ILSpy</a>	Livre	Descompilador multiplataforma para .NET.

JD-GUI	Livre	Descompilador para Java com GUI.
RetDec (Retargetable Decompiler)	Livre	Descompilador para C/C++ feito pelo time do Avast. Inclui detector de compilador e packer, plugins para IDA e radare2.
Snowman	Livre	Descompilador livre para C/C++. Pode ser usado como plugin no IDA, x64dbg, etc.
VB Decompiler	Comercial	Descompilador para VB5/6 e disassembler para VB .NET.

## Frameworks

Nome	Licença	Descrição
angr	Livre	Framework para análise estática e simbólica de binários.
Frida	Livre	Framework para instrumentação dinâmica de binários.
Radare	Livre	Suíte completa com debugger, disassembler e outras ferramentas para quase todo tipo de binário existente!

## Visualizadores hexadecimais

Nome	Licença	Descrição
hdump	Livre	Clone nosso <code>hexdump</code> , multiplataforma (funciona no Windows!) do <code>hexdump</code> que imita a saída do <code>hd</code> .
heksa	Livre	Multiplataforma, com saída colorida e recursos interessantes como seek negativo (a partir do fim do arquivo).

hexdump/hd	Livre	Padrão no BSD e Linux. Se chamado por "hd" exibe saída em hexa/ASCII.
od	Livre	Padrão no UNIX e Linux. O comando <b>od -tx1</b> produz uma saída similar à do <b>hd</b> .
xxd	Livre	Vem com o vim. Uma saída similar à do <b>hd</b> é obtida com <b>xxd -g1</b> .

Abaixo um comparativo onde dumpamos os primeiros 32 bytes do binário /bin/ls utilizando os visualizadores acima:

```

1  $ hdump -n 32 /bin/ls
2  00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF.....
3  00000010  03 00 28 00 01 00 00 00  21 3e 00 00 34 00 00 00  |..(.....!>..4
4
5  $ hexa -l 32 /bin/ls
6  000000 | 7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00 |.ELF...0 00000000
7  000100 | 03 00 28 00 01 00 00 00  21 3e 00 00 34 00 00 00 |.0(0.0000 !>004000
8
9  $ hd -n32 /bin/ls
10 00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF.....
11 00000010  03 00 28 00 01 00 00 00  21 3e 00 00 34 00 00 00  |..(.....!>..4
12 00000020
13
14 $ od -tx1 -N /bin/ls
15 00000000    7f  45  4c  46  01  01  01  00  00  00  00  00  00  00  00  00
16 00000020    03  00  28  00  01  00  00  00  21  3e  00  00  34  00  00  00
17 00000040
18
19 $ xxd -g1 -l32 /bin/ls
20 00000000: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF.....
21 00000010: 03 00 28 00 01 00 00 00 21 3e 00 00 34 00 00 00  ..(.....!>..4..

```

# Referências

## Artigos

[Qual a diferença entre linguagem compilada para linguagem interpretada?](#), StackOverflow.

[Linguagem compilada](#), Wikipedia.

[Linguagem interpretada](#), Wikipedia.

[Microprocessador](#), Wikipedia.

[Transístor](#), Wikipedia.

[A Crash Course in Everything Cryptographic](#), Leo Whitehead, 2019.

[Understand flags and conditional jumps](#) - Jeremy Gordon, 2002-2003.

---

## Vídeos

[Pluralsight's Windows Internals](#) - Pavel Yosifovich, 2013.

---

## Manuais

Intel® 64 and IA-32 Architectures Software Developer's Manual.

Microsoft Portable Executable and Common Object File Format Specification.

---

## Livros

Crackproof Your Software: Protect Your Software Against Crackers - Pavol Cerven, 2002.

Fundamentos em programação Assembly - José Augusto N. G. Manzano, 2006.

História da Matemática - Carl B. Boyer, 1974.

Reversing: Secrets of Reverse Engineering - Eldad Eilam, 2005.

## Sobre o livro

# Registro de alterações

## Versão 1.0

Lançada em ?

As seguintes seções foram completadas:

- ☒ Tabela de Importações
- ☒ Execução de programas
  - ☒ Executáveis
  - ☒ Bibliotecas
- ☒ Windows API
- ☒ Assembly
  - ☒ Registradores
  - ☒ Instruções básicas
  - ☒ Funções e pilha
- ☒ Depuração
  - ☒ Disassembly
  - ☒ Breakpoints
  - ☒ Manipulação do programa
  - ☒ Patches

Os requisitos foram revistos e agora deixam claro que precisamos de duas máquinas: uma com Windows e uma com Linux. Alguns exemplos também podem ser utilizados no macOS.

O capítulo de registradores foi expandido. Uma subseção sobre o EIP foi incluída.

Os exemplos em Assembly foram melhorados. O *label* **start** foi removido para evitar confusões, já que não *linkamos* os binários compilados gerados para os exemplos.

Além disso, os exemplos do livro foram atualizados para o Python 3, erros foram corrigidos e algumas ferramentas adicionadas.

---

# Versão 0.1

Lançada em 12 de Maio de 2017

Primeira versão pública do livro. As seguintes seções ainda estão sendo trabalhadas:

- Tabela de Importações
- Execução de programas

Além disso, as seguintes seções não foram iniciadas ainda:

- Assembly x86
- Depuração.