Instruções do NASM

Um pouco sobre o uso do NASM

Quando programamos em Assembly estamos escrevendo diretamente as instruções do arquivo binário, mas não apenas isso como também estamos de certa forma o formatando e escrevendo todo o seu conteúdo manualmente.

Felizmente o assembler faz várias formatações que dizem respeito ao formato do arquivo automaticamente, e cabe a nós meramente saber usar suas diretivas e pseudo-instruções. O objetivo desse tópico é aprender o principal para se poder usar o NASM de maneira apropriada.

Seções

Antes de mais nada vamos aprender a dividir nosso código em seções. Não adianta de nada usarmos um *linker* se não trabalharmos com ele, não é mesmo?

A sintaxe para se definir uma seção é bem simples. Basta usar a diretiva `section` seguido do nome que você quer dar para a seção e os atributos que você quer definir para ela. As seções `.text`, `.data`, `.rodata` e `.bss` já tem seus atributos padrões definidos e por isso não precisamos defini-los.

Por padrão o NASM joga todo o conteúdo do arquivo fonte na seção `.text` e por isso nós não a definimos na nossa PoC. Mas poderíamos reescrever nossa PoC desta vez especificando a seção:

assembly.asm | main.c

```
bits 64

section .text

global assembly
assembly:
  mov eax, 777
  ret
```

A partir da diretiva na linha 3 todo o código é organizado no arquivo objeto dentro da seção `.text`, que é destinada ao código executável do programa e por padrão tem o atributo de execução (*exec*) habilitado pelo NASM.

Símbolos

Como já vimos na nossa PoC os símbolos internos podem ser exportados para serem acessados a partir de outros arquivos objetos usando a diretiva `global`. Podemos exportar mais de um símbolo de uma vez separando cada nome de rótulo por vírgula, exemplo:

```
global assembly, anotherFunction, gVariable
```

Dessa forma um endereço especificado por um rótulo no nosso código fonte em Assembly pode ser acessado por código fonte compilado em outro arquivo objeto, tudo graças ao *linker*.

Mas as vezes também teremos a necessidade de acessar um símbolo externo, isto é, pertencente a outro arquivo objeto. Para podermos fazer isso existe a diretiva `extern` que serve para declarar no arquivo objeto que estamos acessando um símbolo que está em outro arquivo objeto.

Já vimos que no arquivo objeto **main.o** havia na *symbol table* a declaração do uso do símbolo `assembly` que estava em um arquivo externo. A diretiva `extern` serve para inserir essa informação na tabela de símbolos do arquivo objeto de saída. A diretiva `extern` segue a mesma sintaxe de `global`:

```
extern symbol1, symbol2, symbol3
```

Veja um exemplo de uso com nossa PoC:

main.c | assembly.asm

```
#include <stdio.h>

int assembly(void);

int main(void)
{
  printf("Resultado: %d\n", assembly());
  return 0;
}

int number(void)
{
  return 777;
}
```

Declaramos na linha 11 do arquivo **main.c** a função `number` e no arquivo **assembly.asm** usamos a diretiva `extern` na linha 2 para declarar o acesso ao símbolo `number`, que chamamos na linha 8.

> Para o NASM não faz diferença alguma aonde você coloca as diretivas **extern** e **global** porém por questões de legibilidade do código eu recomendo que use **extern** logo no começo do arquivo fonte e **global** logo antes da declaração do rótulo.
>
> Isso irá facilitar a leitura do seu código já que ao ver o rótulo imediatamente se sabe que ele foi exportado e ao abrir o arquivo fonte, imediatamente nas primeiras linhas, já se sabe quais símbolos externos estão sendo acessados.

Variáveis?

Em Assembly não existe a declaração de uma variável porém assim como funções existem como conceito e podem ser implementadas em Assembly, variáveis também são dessa forma.

Em um código em C variáveis globais ficam na seção `.data` ou `.bss`. A seção `.data` do executável nada mais é que uma cópia dos dados contidos na seção `.data` do arquivo binário. Ou seja o que despejarmos de dados em `.data` será copiado para a memória RAM e será acessível em tempo de execução e com permissão de escrita.

Para despejar dados no arquivo binário existe a pseudo-instrução `db` e semelhantes. Cada uma despejando um tamanho diferente de dados mas todas tendo a mesma sintaxe de separar cada valor numérico por vírgula. Veja a tabela:

Pseudo-instrução	Tamanho dos dados	Bytes
db	byte	1
dw	word	2
dd	double word	4
dq	quad word	8
dt	ten word	10
do		16
dy		32
dz		64

> ⚠ As quatro últimas `dt, do, dy e dz` não suportam que seja passado uma *string* como valor.

Podemos por exemplo guardar uma variável global na seção `.data` e acessar ela a partir do código fonte em C, bem como também no próprio código em Assembly. Exemplo:

assembly.asm | main.c

```
bits 64

global myVar
section .data
  myVar: dd 777

section .text

global assembly
assembly:
  add dword [myVar], 3
  ret
```

> ℹ Repare que em C usamos a *keyword* `extern` para especificar que a variável global `myVar` estaria em outro arquivo objeto, comportamento muito parecido com a diretiva `extern` do NASM.

Variáveis não-inicializadas

A seção `.bss` é usada para armazenar variáveis não-inicializadas, isto é, que não tem um valor inicial definido. Basicamente essa seção no arquivo objeto tem um tamanho definido para ser alocada pelo sistema operacional em memória mas não um conteúdo explícito copiado do arquivo binário.

Existem pseudo-instruções do NASM que permitem alocar espaço na seção sem de fato despejar nada ali. É a `resb` e suas semelhantes que seguem a mesma premissa de `db`. Os tamanhos disponíveis de dados são os mesmos de `db` por isso não vou repetir a tabela aqui. Só ressaltando que a última letra da pseudo-instrução indica o tamanho do dado. A sintaxe da pseudo-instrução é:

```
resb número_de_dados
```

Onde como operando ela recebe o número de dados que serão alocados, onde o tamanho de cada dado depende de qual variante da instrução foi utilizada. Por exemplo:

```
resd 6  ; Aloca o espaço de 6 double-words, ao todo 24 bytes.
```

A ideia de usar essa pseudo-instrução é poder declarar um rótulo/símbolo que irá apontar para o endereço dos dados alocados em memória. Veja mais um exemplo na nossa PoC:

assembly.asm | main.c

```
bits 64

global myVar
section .bss
  myVar: resd 1

section .text

global assembly
assembly:
  mov dword [myVar], 777
  ret
```

Constantes

Uma constante nada mais é que um apelido para representar um valor no código afim de facilitar a modificação daquele valor posteriormente ou então evitar um *magic number*. Podemos declarar uma constante usando a pseudo-instrução `equ`:

```
NOME_DA_CONSTANTE equ expressão
```

Por convenção é interessante usar nomes de constantes totalmente em letras maiúsculas para facilitar a sua identificação no código fonte em contraste com o nome de um rótulo. Seja lá aonde a constante for usada no código fonte ela irá expandir para o seu valor definido. Exemplo:

```
EXAMPLE equ 34
mov eax, EXAMPLE
```

A instrução na linha 2 alteraria o valor de EAX para 34.

Constantes em memória

Constantes em memória nada mais são do que valores despejados na seção `.rodata`. Essa seção é muito parecida com `.data` com a diferença de não ter permissão de escrita. Exemplo:

```
section .rodata
  const_value: dd 777
```

Expressões

O NASM aceita que você escreva expressões matemáticas seguindo a mesma sintaxe de expressão da linguagem C e seus operadores. Essas expressões serão calculadas pelo próprio NASM e não em tempo de execução. Por isso é necessário usar na expressão somente rótulos, constantes ou qualquer outro valor que exista em tempo de compilação e não em tempo de execução.

Podemos usar expressão matemática em qualquer pseudo-instrução ou instrução que aceita um valor numérico como operando. Exemplos:

```
CONST equ (5 + 2*5) / 3       ; Correto!
mov eax, 4 << 2               ; Correto!
mov eax, [(2341 >> 6) % 10]   ; Correto!
mov eax, CONST + 4            ; Correto!

mov eax, ebx + 2   ; ERRADO!
```

O NASM também permite o uso de dois símbolos especiais nas expressões que expandem para endereços relacionados a posição da instrução atual:

Símbolo	Valor
$	Endereço da instrução atual
$$	Endereço do início da seção atual

Onde o uso do `$` serve como um atalho para se referir ao endereço da linha de código atual, algo equivalente a declarar um rótulo como abaixo:

```
here: jmp here
```

Usando o cifrão fica:

```
jmp $
```

Enquanto o uso de `$$` seria equivalente a declarar um rótulo no início da seção, como em:

```
section .text
text_start:
  nop
  call exemplo
  jmp text_start
```

E esse seria o equivalente com `$$`:

```
section .text
  nop
  call exemplo
  jmp $$
```

No caso de você usar o NASM para um formato de arquivo binário puro (*raw binary*), onde não existem seções, o `$$` é equivalente ao endereço do início do binário.

← Previous
Instruções assembly x86

Next
Pré-processador do NASM →