
# Nemesiz Security Group

The Underground For Brazilian Hacking

www.nemesiz.forum.st

# Tutorial Basico de Assembly Para Linux/i386 (AT&T)

Title: Tutorial Basico de Linguagem Assembly Para Linux/i386 na sintaxe AT&T
Author: Felix_Poison
A.k.A: Felix Costa D. S.
E-Mail: elodia_superstar[at]hotmail[dot]com
From Brasil, Jan de 2010

ATENÇÃO:

esse texto foi retirado do 1o projeto de programação do Nemesiz Security Group, do curso basico de Assembly do Felix_Poison
esse pdf foi criado para que os alunos possam baixar o curso e para aqueles que não estavam presentes no projeto, para que possam desfrutar do artigo aprensentado. O tutorial pode ser distribuido livremente  para qualquer lugar desde que sejam mantidos essa nota, sua fonte, creditos e seu objetivo inicial que é distribuir informação para todos

Felix_Poison, Jan de 2010

## ~ Table Of Contents



1. **Preludim**:

bem, amigos, aqui está criado o PDF do curso de Assembly. Só gostaria de releembrar que todo curso de programação organizado pela Nemesiz Security Group será criado um PDF para que todos possam baixar e ler o mesmo futuramente. Gostaria de agradecer mais uma vez, a todos que fizeram desse projeto uma realidade como eu já disse demais no curso, vamos apenas ao PDF ;)

## 2. Introdução

bem, eu nao vou explicar como surgiu, quem inventou e tals, essas coisas menos importantes.
eu adoro trabalhar com asm, apesar de as vezes ser muuuito chato. é uma linguagem muito poderosa e que com pequenas linhas, podem fazer muitos feitos.
eu, posso dizer que asm é essencia para quem quer mecher com vulns-devs, Fuzzing, Engenharia reversa e outros assuntos relacionados a explorações.
claro que a utilidade de asm nao é apenas essa, dã. e pode ter certeza, que essa linguagem será muito util para voce algum dia, amigo.

assembly tem dois tipos de sintaxes, a INTEL e a AT&T. nesse curso, aprenderemos a trabalhar com asm AT&T para Linux/i386. mas porque até especificamos o modelo do processador? porque a coisa muda de processador para processador. como a maioria, creio que use i386, esse curso é voltado para ele. mas pera ae, vamos explicar direito para quem nao entendeu:
Cada arquitectura do pc tem a sua própria linguagem de máquina e, logo, sua propia forma de assembly. Essas linguagens de asm diferentes diferem no número e tipo de operações que suportam. Também têm diferentes tamanhos e números de registros, e diferentes representações dos tipos de dados armazenados, sacou?

agora, como eu disse, asm tem duas sintaxes diferentes. vamos ver um pouco sobre suas diferenças mais a frente quando começarmos.

Nesse tutorial, usaremos um slackware based, mas os exemplos servem para qualquer Unix Like. Usaremos o programa “as” para criarmos o object file de nossos codigos assembly e o “ld” para likarmos o obj file para ser criado nosso executavel . Entao se voce não os tem, baixe vamos tentar entender seus modos de uso:
vamos usar um editor qualquer para editar nossos codigos fontes e iremos salva-lo com a extensao .s exemplo: code.s

depois teremos que criar um object file desse arquivo com o as:

*bt / # as -o code.o code.s*

devemos criar o object file assim, especificando a extesao que nosso arquivo deverá ter agora, que é code.o
agora vamos linka-lo com o ld:

*bt / # ld -o code code.o*

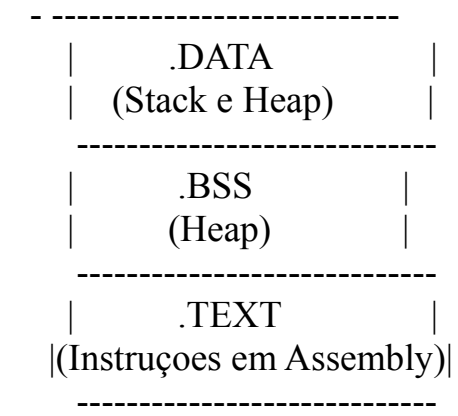
depois disso, teremos nosso executavel  é só executa-lo:

*bt / # ./code*

agora, vamos a um pouco de teoria, que será essensial para podermos programar em ASM.

## 3. Modelo de Memoria

um programa qualquer é dividido em tres sessões 'imaginarias':
.DATA, .BSS e .TEXT




onde:
.DATA -> ela é responsavel por salvar o endereço de retorno da subrotinas do programa, passar parametros para funçoes,criar variaveis locais, ou seja, ela guarda os dados do programa.

.BSS -> é para dados indefinidos, responsavel por armazenar dados alocados dinamicamente
e endereços de variaveis estaticas.

.TEXT -> aqui é onde ficam os codigos do programas em Assembly,
 sao instruçoes que seram passadas para o processador que o executará.

a memoria de qualquer PC é dividida em segmentos de 64k de tamanho cada, que são

separados por 16 bits cada.Os enderecos de memoria trabalham com a base hexadecimal, logo,A memoria comeca a ser escrita a partir do endereco 0 em cada segmento, assim fazendo o '0' ser o 1 e o F ser o 16.

## 4. O Stack

A região denominada Stack ou pilha (mesma coisa), é responsavel por salvar o endereço de subrotinas, passar argumentos (dados) para funçoes, armazenar variaveis locais e etc. Como o propio nome sugere, o stack (pilha)
funciona como uma pilha, onde voce vai pondo dados lá.
logo, o esquema seria o mesmo de uma pilha de livros, por exemplo.
o ultimo livro que voce por nessa pilha, será o primeiro a sair, o primeiro
a ser pego, correto?. fugindo um pouco dos termos tecnicos,
imagine a stack como um clube de festa (meu deus..), e as pessoas
que entraram nessa festa serão os dados.. caso, o cara que chegar por ultimo
na festa quiser sair, ele terá que ser o primeiro a sair..
esse esquema é chamado de LIFO - Last in, first out (ultimo a entrar,
primeiro a sair). dois comandos em asm sao usados com relaçao a stack :
PUSH (empurra) e POP (retira) dados, veremos isso melhor no capitulo
sobre intrunçoes em assembly

## 5. Introdução ao Assembly

Bem, vamos aparti desse capitulo começar a estudar sobre asm.
Vamos esclarecer algumas coisinhas rapidinho:
um comentario em asm AT&T é criado usando o '#' (cerquinha). Ou seja, do lado de cada linhas de codigo voce pode adicionar uma # e adicionar seu comentario.
Ex: movl $0x1, %eax  #  essa linha move o valor 1 para o registrador EAX

para indicar um registrador como tal, deve-se usar o ' % ' (por cento)
antes do nome, exemplo: %EAX, %ah, %ebp..
em caso de voce indicar um endereço de memoria, voce só precissa por
o valor do endereço, mas se for um valor normal, voce deve por o ' $ '
antes, exemplo: para mover o endereço 0x0047abbc para o register %ah,
basta por o esse valor mesmo, sem caracter adicional.
agora, se voce quer mover um valor normal, exemplo: 0xcd para %dh, deve-se
por o '$', ficando assim: $0xcd. Isso é regra!
so lembrando que essa sintaxe usada é a AT&T (Unix). logo, é diferente da
sintaxe INTEL (DOS).

## 6. Registradores

Registradores sao blocos de memoria do processador que sao usadas
para receber e guardar variaveis.
antes dos processadores i386, os registers eram de 16 bits apenas
já nos modelos i386 pra cima, eles possuem 32 bits.
vejamos alguns registers de uso geral, no qual podemos armazenar qualquer valor.

EAX = Extended Acumullator (Registrador aculador extendido)
EBX = Extended Base (Registrador de base extendido)
ECX = Extended Couter (registrador contador extendido)
EDX = Extended Data (registrador de dados extendido)

como eu disse, nos processadores atuais, os registers sao de 32 bits,
que é o caso desse ae de cima. para isso, foi adicionado o 'E'
de extendido no começo do nome original dos registers.
os registers de 16 bits tinham seus nomes apenas : ax,bx,cx,dx, mas como
eles sao de 32 bits agora, ganhanram o "E".
cada register desse possui 32 bits. eles possuem duas partes de 16 bits cada
esses 16 bits são os 16 bits 'altos' e os 16 bits 'baixos'.
os 16 bits mais altos possuem tambem, duas partes de 8 bits cada:
a parte alta (high) e a parte baixa (low). para representar esse tipo de
register, usamos o seu nome seguido de H para alto ou do L de low para baixo.
ficando assim : al e ah, bl e bh, cl e ch, dl e dh.
vamos ver uma imagem pra ficar melhor de entender, vamos usar o register
EAX como exemplo, mas todos os outros citados anteriormente seguem
o mesmo modelo :


acaso alterarmos o valor de al, por exemplo, tambem seria mudado o valor
de ax e eax. exemplo: se movessmos o valor 0x00000000 para EBX, ficaria:
EBX = 0x00000000
BX(parte alta de ebx) = 0x0000
BH e BL (parte alta e baixa de BX) = 0x00

e vice-versa, se movessemos o valor 0x00 para bl, bx teria o valor de
0x0000 e ebx seria 0x00000000. no caso, o mesmo valor 🙂
Vamos ver mais sobre alguns tipos de registers:

***Registradores Gerais:***

EAX: como vimos, ele é um registrador acomulador. Ele é usado mais em operações de entrada e saida de dados, acomulando-os. Ex:

```
mov $0x4c, %EAX   #   isso move (guarda) o valor 4c em EAX
```

EBX: registrador de Base. Bastante usado como um registrador de offset. Assim como eax, ele guarda argumentos: Ex:

```
mov $0x7, %EBX   #  move o valor 7 para EBX
```

ECX: registrador contador. Bastante usado em loops e desvios incondicionais. Ex:

```
movl $0xa, %ECX
```

EDX: Registrador de dados. Mais usado para receber e guardar dados, claro. Pode ser usado tambem em operações aritmeticas como MUL, DIV, ADD e etc.Ex:

```
mul $0x3, %EDX   #  faz a multiplicação de EDX por 3
```

ESI: Registrador de indice fonte. Bastante usado em movimentações de blocos de instruções apontando para a proxima instrução.

EDI: registrador de indice destino (index destination). Pode ser usado como registrador de offset's.

EIP: registrador apontador de instrução (instruction pointer). Ele é o offset do segmento da proxima instrução a ser executada. Esse é um dos registers especiais e não podem ser acessados por qualquer instrução.

***Registradores de Segmento:***

CS: registrador de segmento de codigo (code segment). Os codigos são armazenados no CS, que corresponde a uma parte da memoria, apontando sempre para a proxima instrução. Assim como todos os registradores de segmento, ele não pode ser acessado diretamente, sendo usado mais como origem.

DS: registrador de segmento de dados (data segment). Ele representa a parte na memoria onde os dados são armazenados. Todos os dados antes passam por ele, pois todos os dados lidos pela CPU são do segmento que aponta para DS. Pode ser usado tambem para operações envolvendo o mov, lods e etc, mas sendo usado como origem, já que não podemos acessar registers de segmento diretamente.

ES: registrador de segmento extra (extra segment). Muito usado como segmento destino em operações de transferencia e movimentação de dados.

SS: registrador de segmento de pilha (stack segment). É o segmento onde está o stack. A CPU usa ele para armazenar endereços de retorno de sub-rotinas.

***Registradores de Indexação do Stack***

EBP: registrador apontador base (base pointer). Ele pode ser usado como um apontador para a base pilha ou até como um registrador comum. Mas ele é usado para se acessar o stack

ESP: registrador apontador do stack (stack pointer). Ele aponta para posição de memoria atual do stack. Ele é o offset de SS

existem alguns outros registradores, mas não irei tratar deles aqui, já que particulamente nem eu nunca os usei mesmo. a parti que formos avançando, iremos ver coisas mais avançadas.

## 7. Instruções Assembly

intrucoes em assembly são responsaveis por uma determinada acao.Essas instrucoes sao as
responsaveis pela execucao de nosso programa. Existem diversos tipos de instruções para cada ação, como movimentação de dados, comparação, saltos e etc. Vamos ve-los:

***Intrunçoes De Transferencia***

sao instunçoes usadas para transferir dados, logicamente.
vamos ver apenas alguns dos tais:

MOV: usada para mover valores de um lugar para outro na memoria.
sintaxe: MOV origem, destino. exemplo: MOV $0xc1, %EAX
oque esse exemplo faz é mover o valor 0xc1 para %eax, que agora vale 0xc1.
outro exemplo:

```
MOV $0x0cf84c00, %ECX
MOV $0x42, (%ECX)
```

nesse exemplo, o primeiro comando move o valor $0x0cf84c00 para %ECX
e o segundo move o valor $0x42 para o endereço que está armazenado em %ECX
para isso, usamos os parenteses, se nao, o valor 0x42 seria movido para ECX
fazendo com que ele perdesse o seu valor original.

LEA: Parecido com o mov, só que ele é capaz de efetuar calculos
mais rapidamente antes da transferencia: sintaxe: LEA origem, destino
exemplo: LEA 4 (%EBX), %EAX
isso faz o seguinte: adiciona o valor 4 para EBX e logo depois 'enfia'
tudo para EAX, sem o LEA, nosso codigo ficaria assim :

```
add $0x4, %EBX
mov %EBX, %EAX
```

viu? o lea economiza tempo

existem algumas variações dos comandos MOV e LEA. elas fazem a mesma coisa que eles a diferença é que faz a movimentação de dados com uma opção a mais. vamos ver alguns:

MOVL: movimenta os dados para a parte baixa do registrador. percebam o 'l' no final de LOW(baixo). ex:

```
movl $0x4, %eax    #move o valor 4 para a parte baixa do registrador %eax que é %al
                   # eh a mesma coisa que: mov $0x4, %al
```

MOVSB e MOVSW: MOVSB move 1 byte e MOVSW move 1 word (16bits)

LEAL: mesma coisa de MOVL. move tudo para a parte baixa dos registradores. Ex:

```
leal $0xa, %eax     #     mesma coisa de: lea $0xa, %al
```

LDS: é usado para carregar dois registradores de uma só vez. ex: CS:EBX
LES: é usado tambem para carregar dois registers de uma só vez, mas carregando-os com um Dword(32bits) para o endereço apontado por DS:SI

XCHG: instrução usada para trocar o valor de um registrador pelo outro. Ex:

```
mov $0x2, %eax      # eax vale 2
mov $0x4, %ebx     # ebx vale 4
xchg %eax, %ebx   # trocamos o conteudo de eax por ebx
                   #agora, eax vale 4 e ebx vale 2
```

existem outros tipos que instruções de movimentação, que sao usados para movimentar dados para o stack.
vamos ver-los:

PUSH: Empurra os dados sobre a pilha (stack). Sintaxe : PUSH valor.
exemplo: PUSH $0x43. Nesse exemplo, a instrução PUSH, empurra
o valor $0x43 para o stack.

POP: ao contrario de push, pop retira os dados do stack
lembrando que o esquema do stack é o LIFO(ultimo a entrar, primeiro a sair)
sendo assim, o pop irá retirar primeiro o ultimo dado inserido no stack

```
MOV %EAX, %EDX
PUSH %EDX
PUSH $0x0
POP $0x0
```

***Instrunçoes Logicas E Aritmeticas***

intrunçoes aritmeticas sao usadas para realizaçao de calculos
(operaçoes aritmeticas) nos operadores. sao eles:

ADD: adiciona dois operadores. ex:
add $0xa, %ebx
ae, fizemos a soma de a(10) sobre o register %ebx
e guarda o resultado em %ebx mesmo

temos ainda a instrução ADDL, que adiciona/soma dados na parte baixa
do registrador. ex:
addl $0x1, %edx  #  soma 1 a parte baixa de %edx que é %dl

SUB: usado para subtração de operadores e tambem pode ser usado para alocar
espaço no stack para receber dados. ex:
sub 0x2, %eax

ae, subtraimos 2 do registrador %eax. EX2 :
sub $0x5, %esp
esse ultimo exemplo, reserva 5 bytes no stack para receber dados

tambem temos o SUBL que subtrai da parte baixa dos registradores. ex:
subl $0x7, %ebx  #  subtrai 7 da parte baida de ebx.

MUL: usado para fazer multiplicação de dois operandos. Ex:
mul $0x3, %edx  #  faz a multiplicação de %edx por 3

Teremos ainda a instrucao IMUL, que tambem serve para
multiplicacao de operandos, só que essa instrucao leva em
consideraçao o sinal do operando, enquanto MUL nao.

DIV: usado para fazer divisão de dois operadores. Ex:
div $0x4, %eax  #  faz a divisão de %eax por 4

assim como MUL, temos a instrução IDIV que considera o sinal do operador.

INC: incrementa um operando. ex:
mov $0x04, %ah
inc %ah
ae, %ah agora valerá 5, que antes era 4.

DEC: inverso de INC, dec decrementa um determinado operando. ex:
mov $0xa, %ah
dec %ah
ae, %ah agora valerá 9.

agora vamos ver as istrunçoes logicas, que fazem operações logicas (nao me diga :O )
entre os operadores que podem ser registradores ou endereços de memoria
vamos ver algumas instruçoes logicas do C e seus correspondentes em ASM:

| C | ASM |
|---|---|
| & | AND |
| \| | OR |
| ~ | NOT |
| ^ | XOR |

AND: como o propio nome diz, ele é o "E" das operaçoes logicas. voces devem
ficar confusos com essa tal 'tabela verdade', entao vou explicar melhor isso:

S = A & B

A B - S

0 0 = 0
0 1 = 0

1 0 = 0
1 1 = 1

observem na tabela verdade, que o resultado de "S" só será 1 (verdadeiro)
se A "and"(e) A forem 1. podemos usar o AND para um bit de algum lugar
sem afetar os outros. ex:
mov $0xa, %ecx # ecx vale 10
and $4, %ecx  # resetamos 4o bit do valor de ecx

OR: é o "OU" logico. tipo: "guria, voce quer ficar comigo "OR" com ele? rs
tabela verdade:

S = A | B

A B - S

0 0 = 0
0 1 = 1
1 0 = 1
1 1 = 1

vemos que S só será 1(verdadeiro) se A "OR" (ou) B forem 1. ainda nao sacou?
S só será 1 se A 'ou' B forem 1. no caso do AND, S só seria verdeiro se A 'e' B fossem
1. huuum.. estão sacando, ahn?!
pode-se usar o OR para setar um determinado bit sem afetar os outros. ex:
or $0x0f, %ebx

NOT: é o 'NÃO' logico. pode ser usado para inverter bits de um byte.

S = ~A
A - S
0 = 1
1 = 0

caso tivessemos um byte com esses bits: (00010001), com o NOT podemos inverte-
los fazendo nosso byte ficar assim: (10001000) ex:
not %eax  # inverte todos os bits de eax

XOR: funciona da mesma forma que OR, mas ele é 'vip', só é usado
se uma variavel tiver o resultado esperado.

S = A ^ B
A B - S

0 0 = 0
0 1 = 1
1 0 = 1
1 1 = 0

igual o OR, só que ele só será 1(verdadeiro) se A e B forem diferentes.
tambem podemos usar ele mudar um bit sem afetar os outros. ex:
XOR %ecx

***Instruçoes De Comparaçao E Repetiçao***

logicamente, as instruçoes de comparaçao sao usadas para comparar algo. veja:

CMP: compara dados sendo usados. ex: CMP $0x5, %eax

TEST: Usada em comparaçoes logicas, fazendo uma chegagem
no register, para ver se tal esta tudo certo; ex: TEST %EDX, %EDX

XORL: usado para comparar dois operandos. caso sejam iguais, o primeiro é zerado
ex: xorl %eax, %ebx # se %eax for igual a %ebx, %EAX será zerado

agora de repetiçao:

LOOP: verefica se o register %ECX é diferente de zero. se for,
ele decrementa '1' (%ecx - 1) e salta para o endereço passado como parametro
se for zero, ele continua normal..

***Instruçoes De Desvio De Programa (Salto)***

sao usados para alterar a execucção normal de um programa. sao eles:

JMP: Abreviação de jump (salto,pular), essa instruçao faz saltos
incondicionais para um determinado endereço de memoria. ex: JMP 0xc
é importante lembrar que JMP nao possui nennhum tipo de retorno, logo,
se voce saltar para algum endereço de memoria, nao tem forma simples
de voce voltar a execucção do programa de onde parou.Ex2:
(...)

jmp LABEL1 # Pula para o label "LABEL1"

```
LABEL1: # Declaracao e inicio do LABEL.
movl $0x4, %eax # Inicio das instrucoes dentro
movl $0x1, %ebx # da rotina/label LABEL1.
(...)
```

existem varias variações de jmp, algumas delas são:

JAE: (Jump if Above or Equal). Ela Salta se parametro for acima ou igual aos dados.
JNE: (Jump if Not Equal). Esta Instrucao Salta se parametro nao for igual.
JGE: (Jump if Greater than or Equal). Esta Instrucao Salta se parametro for maior ou igual a...
JLE: (Jump if Less or Equal). Esta Instrucao Salta se parametro for menor ou igual a...
JE: (Jump if Equal). Salta se parametro for igual a...
JB: (Jump if Below). Ela Salta se parametro abaixo.
JG: (Jump if Greater than). Ela Salta se for maior que...
JL: (Jump if Less than). Ela Salta se for menor que...
JZ: (jump if zero). Ela Salta se o parametro acima for igual ao outro, salta
se parametros iguais, ou seja, se uma comparacao com jz
for verdadeira pula para... Ex.:

```
mov $0x1, %eax # Move "1" para %eax
mov $0x1, %ebx # Move "1" para %ebx
cmp %eax, %ebx # Compara %eax e %ebx. como eles sao iguais, é verdadeira
jz IGUAL # Entao JZ pula para IGUAL
jmp DEFERENTE # Se a comparacao fosse falsa, esta instrucao
# seria executada, ao inves de jz.

IGUAL:
# instruções (...)
DEFERENTE:
# instruções (...)
```

Cada uma dessas Intrucoes possuem como parametro, um Label
de alguma rotina pre-definida ou uma posicao de memoria. Ex.:

```
jle 0x404890a1 ou jz LABEL1.
```

CALL: Usada para chamar uma subrotina e para saltos como JMP,
só que com CALL é possivel voltarmos a execução normal do prog.
ex: CALL 0x081400cd . no caso, o endereço da proxima execução apos o salto
é salvo no topo do stack.

***Outras Instruções***

INT: é usado para fazer uma interrupção no sistema, fechando assim
o agrupamento de instruções que serão executadas. na sintaxe AT&T
usamos o 'int 0x80' para passar todas as instruções ao kernel
fazendo assim com que elas sejam executadas. Ex:

```
movl $0x4, %eax # move 4 para %eax
movl $0x1, %ebx # move 1 para %ebx

int $0x80 # executa todas as instruções acima
# chamando o kernel.
```

ps: para cada syscall que formos usar, temos que usar o int 0x80 no final
de cada uma. voces vao entender melhor mais pra frente.

RET: (return adress) endereço de retorno. quando a instrução
RET é encontrada na sub-rotina, a CPU salta de volta para a proxima
instrução que terá um JMP ou um CALL. ele irá voltar a executar
as instruções depois do pulo que fizemos com JMP ou CALL. Ex:

```
call LABEL # "call" pula para a sub-rotina "LABEL"
(...)

LABEL: # inicio da sub-rotina LABEL
movl $0x04, %eax #move 4 para %eax
(...)

RET # retorna a execução normal das instruções
# depois do pulo (call LABEL)
```

NOP: (no operation) nenhuma operação. NOP faz uma coisa muito importante:
porra nenhuma rs. ela nao realiza nada, passando assim para a proxima
instrução. o NOP é largamente usado em exploits de overflows, fazendo
o buffer encher de 'nada' até chegar no shellcode 🙂 é importante que voce saiba
que a representação de NOP em hexa é 0x90. Ex:

```
movl $0x2, %eax # move 2 para %eax
movl $0x2 %ebx # move 2 para %ebx
```

nop # nao faz nada, passando assim para a proxima instrução
cmp %eax, %ebx # compara %eax e %ebx pra ver se são iguais
(...)

existem varias outras instruções assembly, inclusive varias variações de muitas instruções que aprendemos aqui, mas por hora nao trataremos dela visto tambem que nao a usaremos muito, sinceramente, eu nunca usei elas mais a frente, conforme formos avançando e escovando mais bits, iremos tratar delas.

## 8. Seções de Codigo

programas asm são divididos em seções, onde assim, há uma certa organização do codigo, armazenamento de dados e etc.

As seções são:

***Headers:**

assim como na linguagem C, o asm tambem dispoem de cabeçalhos pode ser qualquer arquivo no sistema, sub-rotinas e

etc.
Exemplo de Header em C: “#include <stdio.h>”. exemplo em asm: “.include “define.h”.

***Seção de dados**

a primeira coisa que deve vir em um code em asm, é a seção de dados.
É nela onde ficará declarado as variaveis do programa. Iniciamos a seção de dados assim:
“.section .data” (seção de dados). O .section que é para seção e o .data que declará que será uma seção de dados.

Vamos ver como declarar agora nossas variaveis:

a declaração das variaveis em asm segue da seguite forma:
NOME: .TIPO VALOR ou NOME = VALOR.
Onde nome é o nome da variavel; TIPO é o tipo da variavel, e VALOR é o seu

conteudo armazenado. Os tipos de

variaveis em ASM podem ser: .long, .string e .ascii.
Existem outros tipos, mas usaremos apenas esses por hora.

.long: é o tipo usado para numeros (variavel numerica) de até 32bits. Ex:
NUM: .long 10  #  declaramos NUM como variavel numera e seu valor é 10

podemos fazer calculos com isso tambem, exemplo: caso voce queira que uma segunda variavel tenha o mesmo tamanho da

primeira variavel, fazemos assim:
TAMVAR: .long . - VARL
onde VARL seria uma variavel texto que declaramos e TAMVAR é o seu tamanho.

Há outro maneira de declararmos uma variavel numerica, como vimos, que é:
NOMEVAR = VALOR. Onde nomevar é o nome da variavel e valor é o seu valor que pode ser qualquer numero em decimal,

hexadecimal, um endereço de memoria e etc. Ex:
NUM = 0x7
ps: essa forma de declaração só vale para variaveis numericas

.string: é o tipo usado para variaveis texto ou strings. Podemos usar caracteres de tabulação “\t”, de newline “\n”

e outros desse tipo. Ex:
STR: .string “Nemesiz Security Group\n”

como mencionei, podemos criar uma variavel numerica que conterá o tamanho da nossa string já criada. Ex:
TAM: .long . - STR
agora, TAM tem a quantidade de caracteres de STR. Temos que fazer isso sempre que formos usar variaveis .strings, já

que temos que passar o seu tamanho para o programa futuramente.

.ascii: funciona da mesma forma de .string. É usado para declarar textos ou strings e tambem aceita caracteres como

de tabulação e newline. Ex:
STR2: .ascii “Calangos do sertão\n”

Bem, terminamos de entender a seção de dados. Vamos começar a construir nosso programa e vamos construindo de acordo

com oque aprendemos:

- - -<code.s>- - - -

```
.section .data   #inicio da seção de dados

# vamos a declaração das variaveis

NUM: .long 7 # Variavel numerica
STR: .string “isso eh uma string\n”       # variavel texto
TAM: .long . - STR                        # tamanho da variavel STR
STR2: .ascii “isso tbm eh uma variavel\n”
TAM2: .long . - STR2

# fim da seção de dados.
```

***Seção de Codigo:***

após a declaração da seção de dados, vem a seção de codigo. É nela onde fica o codigo de programa, informações de

symbols e etc. Ele armazena o codigo e o binario lá.
Ela é declarada por “.section .text” (seção texto). Onde como vimos .section diz que ali será uma seção e .text

declara essa seção como seção de texto (codigo).
É obrigatorio ter essa seção em qualquer programa assembly.

Função Principal do ASM

assim como no C, o asm tambem possui uma função principal. E assim como no C, ela é o inicio do corpo do codigo que

será executado.
Em C, a função principal é main(), em asm é “_start”.
Declaramos a função principal (_start) com o “.globl” ou “.global” que é a mesma coisa.
Entao ficaria: “ .global _start”.
Feita a sua declaração, temos que inicia-la com “_start:” e abaixo disso viriam todas as nossas instruções de codigos do programa
é importante lembrar que a função _start está dentro da seção de texto, claro. E ela é

obrigatoria para o nosso

programa funcionar. Vamos ver entao como está ficando o nosso programa começando com todas as seções:

```
.section .data                    # seção de dados
STR: .string “Felix_Poison\n” #variaveis
TAM: .long . - STR

.section .text                    #inicio da seção de codigo
.globl _start                     #declaração da função principal
_start:                           # inicio da função principal

movl $0x5, %eax             # agora abaixo, segueriam todas nossas instruções ASM
(...)
```

***Labels***

Labels são digamos que um lugar no codigo onde poderemos referenciar como instrunções, declaração de variaveis, inicio de funções, como vimos mais atrás. Ex:

```
LABEL1:

LABEL2: .long 0 # label variavel

LABEL3:
movl $0x4, %eax #label como função
movl $0x0, %ebx
(…)
```

## 9. System Calls

As system calls (chamadas de sistema) é uma forma usada pelo programa para requisitar funções do kernel. deixa eu ver se consigo explicar melhor.. quando fazemos um programa imprimir algo na tela, usamos varias system calls para isso, por exemplo a system call write (escrever). essa system call requisita esse serviço do kernel, sendo assim executado com sucesso.
ele chama essa função do kernel. em suma, uma syscall é uma chamada que é feita a uma
determinada funçao diretamente ao kernel.
elas sao utilizadas em todos os codigos funcionais ASM.

cada system call é identificada por um numero. para voce entender melhor
aconselho que der uma olhada na tabela de syscalls de seu sistema
o seu local muda de distro pra distro, mas voce pode encontra-lo em:
/usr/include/asm-i386/unistd.h ou qualquer outro local parecido. procure esse
arquivo!

como eu disse no capitulo de instruções, a instrução int 0x80 é usada
para executar as syscalls. devemos guardar o valor da syscall que iremos usar em
%eax. para registradores com até 6 argumentos, eles irao nos registers seguinte
nesa ordem: %ebx, %ecx, %edx, %esi, %edi. bem, agora que voce já abriu sua tabela
de syscalls, vamos começar fazendo um exemplo de uso delas.
vamos usar de inicio a syscall write, que é usada para escrever na tela.
tambem teremos que usar a syscall exit. teremos que usar a syscall exit em todos
nossos codigos com syscalls, para eles poderem sair limpamente. entao, vamos lá:

*bt / # cat /usr/include/asm-i386/unistd.h*

de acordo com a tabela, a syscall write vale 4 e exit vale 1
huum.. voce nao conseguiu achar a tabela de syscalls do seu sistema?
nao conseguiu entender? puts, deixa isso pra lá e veja nesse link a tabela completa:
http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

eu aconselho voce a procurar porque será melhor. de qualquer forma,
temos o valor da syscalls que iremos usar, agora precissamos saber quais
seus argumentos.

*bt / # man 2 write*
*write:*
*ssize_t write(int fd, const void *buf, size_t count);*

como vemos, esses sao os argumentos da syscall write, e precissmos guarda-los
tambem no nosso codigo. antes vamos entender oque ele realmente pede:

int fd = Onde ele ira escrever
*buff = o que ele ira escrever
count = o tamanho do que ele ira escrever

vamos ver agora os argumentos de exit:

*exit:*
*void _exit(int status);*

como vemos, ele só tem um argumento, que é sua saida
para uma saida normal sem erros, usamos exit(0);

com base nisso, já podemos começar nosso primeiro codigo que irá imprimir algo na tela vamos ver e observa-lo com atenção:

- - - imprime.s - - -

```
# begin

.section .data # seção de dados

STR1: .string "nosso primeiro code em ASM\n"
TAM: .long . - STR1

.section .text    # seção de codigo

.globl _start     # declaração da função principal

_start:             # inicio da função principal


movl $0x4, %eax   # Move o numero do syscall write(4) para o %eax
movl $0x1, %ebx   # Onde ele irá escrever, no caso Saida Padrao

leal STR1, %ecx    # O que ele ira escrever, no caso nossa variavel STR1.
movl TAM, %edx   # O Tamanho da nossa variavel STR1
int $0x80            # executa todas as intruções da syscall acima

movl $0x1, %eax    # Move o numero do syscall exit(1) para %eax
movl $0x0, %ebx    # Status de exit, no caso 0, ou seja, saida sem erro
int $0x80            # executa as instruções da syscall acima

# end
```

- - - imprime.s - - -

para compilar e executar, já sabe, né? 🙂

*bt / # as -o <imprime.o> <imprime.s> <- cria o object file do nosso code*
*bt / # ld -o <imprime> <imprime.o> <- Faz a linkagem do do obj file fazendo se tornar executavel*
*bt / # ./imprime <- executa nosso primeiro code* 🙂

*bt / # ./imprime*
*nosso primeiro code em ASM*

*bt / #*

vimos que funcionou perfeitamente.

vamos usar agora a syscall read, que como o propio nome diz
é usada para ler. faremos um programa que pedirá para o usuario digitar
algo no console (syscall read) e imprimirá oque ele digitou na tela (syscall write)
e depois sairá limpamente sem erros (syscall exit).
vamos agora fazer os esquemas para podermos começarmos:

bt / # cat /usr/include/asm-i386/unistd.h

será aprensentada a tabela de syscalls. vemos que a syscall read vale 3
vamos saber seus argumentos:

bt / # man 2 read
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

apesar de ter 3 argumentos, só vamos precissar passar
qual o tamanho do que será lido pelo programa.
vamos ver num code e uma boa explicação do mesmo:

--- leiaporra.s ---

```
# begin

.section .data

STR1: .string "digita algo ae, mano:\n"
TAM1: .long . - STR1

.section .text

.globl _start

_start:

# Syscall write(4)
movl $0x4, %eax
movl $0x1, %ebx
leal MSG1, %ecx
movl TAM1, %edx
int $0x80
```

```
# Salva o Stack Pointer em %ecx
movl %esp, %ecx

# Reserva 10bytes para o usuario digitar no stack
subl $0xa, %esp

# Syscall read (3)
movl $0x3, %eax     # valor da syscall read (3)
                       # o que for escrito tambem estará em %eax
movl $0x9, %edx    # Tamanho do que vai ser lido para %edx (10 bytes)
int $0x80

movl %eax, %edx   # Move o que foi digitado para %edx.

# Syscall write (4)
movl $0x4, %eax
movl $0x1, %ebx
int $0x80

# Syscall exit(1)
movl $0x1, %eax
movl $0x0, %ebx
int $0x80

# end
```

--- leiaporra.s ---

vamos compilar e rodar para ver o resultado:

*bt / # as -o <leiaporra.o> <leiaporra.s>*
*bt / # ld -o <leiaporra> <leiaporra.o>*
*bt / #./leiaporra*

*digita algo ae, mano: Nemesiz*
*Nemesiz*

*bt / #*

vamos começar a complicar agora..
vamos usar 4 syscalls. iremos abrir um arquivo com a
syscall open, iremos escrever nele com a syscall write
iremos fechar o arquivo com a syscall close e usaremos tambem, obvio,
a syscall exit para sair do programas sem erros.
vamos procurar na tabela o valor de Open e Close:

vemos que o valor de open é 5 e close é 6

ah, antes de começarmos, é necessario que o arquivo que
usaremos para abrir e escrever nele já exista
aqui eu criei ele e ele se encontra em: /testes/open.txt
mas voce pode criar oque quiser e tals. por via de duvidas, deixe
o arquivo limpo, sem nada escrito ainda. ae ficará melhor de vermos o resultado.

--- abracadabra.s ---

```
# begin

.section .data

# apenas string de boas vindas rs
STR1: .string "vamos abrir um arquivo e escrever nele \n"
TAM1: .long . - STR1

STR2: .string "Nemesiz Security Group!!\n" # oque será escrito no arquivo
TAM2: .long . - STR2

FILE: .string "/testes/open.txt"    #path do nosso arquivo
MODO: .string "O_RDWR"        # modo do arquivo, no caso ele está
                                    # leitura-escrita

.section .text

.globl _start

_start:

# Syscall write (4)
movl $0x4, %eax
movl $0x1, %ebx
leal MSG1, %ecx
movl TAM1, %edx
int $0x80

# Syscall open(5)
movl $0x5, %eax        # valor da syscall open
movl $FILE, %ebx     # O arquivo que iremos abrir
movl $MODO, %ecx # como eu disse, o modo do arquivo
movl $0x0, %edx       # Permisao 0 !
int $0x80

movl %eax, %esi        # Move o retorno da funcao open para %esi

# Syscall write(4) irá agorar escrever no arquivo
movl $0x4, %eax
movl %esi, %ebx       # Onde ele irá escrever, no caso o open.txt
leal MSG2, %ecx       # Irá escrever o conteudo da variavel STR2
movl TAM2, %edx    # O tamanho da variavel
int $0x80
```

```
# Syscall close(6)
movl $0x6, %eax
movl %esi, %ebx     # Fecha o arquivo open.txt
int $0x80

# Syscall exit.
movl $0x1, %eax
movl $0x0, %ebx
int $0x80

# end
```

--- abracadabra.s ---

compile e execute e abra o arquivo agora para ver o seu resultado
nossa, que emoção, hein amigo?! quando fiz isso pela primeira vez
fiquei muito feliz rs. mas nada de descanço, temos muita coisa para vermos
ainda aqui no curso. e depois do final dele tambem, claro!

voces podem estar se perguntando agora: " e se a syscall
que eu quero usar tiver mais de 6 argumentos? onde guardarei eles?"
é simples! O numero da syscall continua em %eax, Mas seus argumentos,
devem ser armazenados e organizados na memoria e o endereco
do primeiro argumento armazenado em %ebx.
Empurramos os argumentos no "stack" de traz pra frente, do ultimo argumento para
o primeiro e copiamos o ponteiro para o stack em %ebx. podemos tambemr copiar
os argumentos para um espaco alocado de memoria e armazenar o endereco do
primeiro argumento em %ebx.

amigos, acredito que depois desse capitulo, saibamos usar
qualquer syscall que queiramos. o esquema teoricamente é como eu disse:
basta saber o valor da syscall e seus argumentos para guardar nos registers
se tivermos duvidas quanto algumas syscalls podemos dar uma consultada na tabela
de system calls, nas man pages do seu sistema. um bom link que eu até passei ele
na aula passada, pode ser bem util para voces:

http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

ae ele tem todas as syscalls e seus valores e seus argumentos
e se for o caso, em quais registradores eles podem ir :)

## 10.Instruções C em ASM

bem, achei interesante adicionar um capitulo falando sobre isso
para esclarecer algumas duvidas quanto a uso de programas
até para mostrarmos como ficaria um codigo C em asm
mostrar que existem instruções que podem fazer o mesmo serviço
de instruções Assembly. aqui trataremos de apenas alguns para
dar uma ideia.

**** USANDO UM 'IF' EM ASM:***

bem, nesse capitulo aprenderemos a usar algumas instruções que
usamos constamente na linguagem C como o comando if e as instruções
de loops do C como while, for e etc.

vamos analisar um code com explicação para saber como podemos usar
um 'if' em linguagem assembly.
esse codigo fará checagens e tals. irá fazer uma função condicional
a galera que meche com C sabe do que eu to falando. quem nao sabe vai sacar agora:

--- if-asm.s ---

```
# begin

.section .data

STR1: .string "programa irá comparar o valor dos registers\n"
TAM1: .long . - STR1

STR2: .string "Os registradores sao iguais! %eax = %ebx!\n"
TAM2: .long . - STR2

STR3: .string "Os registradores sao diferentes! %eax != %ebx!\n"
TAM3: .long . - STR3

.section .text

.globl _start

_start:

# Syscall write (4)
# string de boas vindas rs
movl $0x4, %eax
movl $0x1, %ebx
```

```
leal STR1, %ecx
movl TAM1, %edx
int $0x80


movl $0xa, %eax  # %eax vale 10 agora.
movl $0x8, %ebx  # %ebx vale 8 agora.

cmp %eax, %ebx  # Compara os 2 registradores

jz IGUAL   # Se eles forem iguais
# jz pula para IGUAL
# se esqueceu de JZ releia a o capitulo de instruções

jmp DIFERENTE    # ou, se eles forem diferentes
# jmp pula para DIFERENTE.

# Label IGUAL
IGUAL:

# Syscall write
movl $0x4, %eax
movl $0x1, %ebx
leal STR2, %ecx
movl TAM2, %edx
int $0x80

# Pula para EXIT
jmp EXIT

# Label DIFERENTE
DIFERENTE:

# Syscall write
movl $0x4, %eax
movl $0x1, %ebx
leal STR3, %ecx
movl TAM3, %edx
int $0x80

# Label EXIT
EXIT:

# Syscall exit(1)
movl $0x1, %eax
movl $0x0, %ebx
int $0x80

# end
```

--- if-asm.s ---

compile, execute e veja seu resultado. altere o valor dos
registradores, acompanhe, fuçe, refuçe!!

*C***riando Loops em ASM:***

vamos criar um loop agora em asm, semelhante a função
while e for da linguagem C

vamos ver um codigo que realiza loops
este programa vai atribuir 0 a um registrador e irá entrar
em uma funcao que vai incrementar de 1 o registrador e compara-lo com 10. assim
ele vai repetir isso ate que o registrador tenha o valor 10
entrando realmente em um loop:

--- loop-asm.s ---

```
# begin

.section .data

STR1: .string "exemplo de Loops em ASM\n"
TAM1: .long . - STR1

STR2: .string "Exemplo de Funcao que realiza Loops!\n"
TAM2: .long . - STR2

valor: .long 0

.section .text

.globl _start

_start:

# Syscall write (4)
movl $0x4, %eax
movl $0x1, %ebx
leal STR1, %ecx
movl TAM1, %edx
int $0x80

# Atribui valor de 0 para nosso registrador
movl $valor, %esi

# Label INICIO
INICIO:
```

```
# Syscall write.
movl $0x4, %eax
movl $0x1, %ebx
leal STR2, %ecx
movl TAM2, %edx
int $0x80

# Incrementa de 1 o registrador %esi.
inc %esi
int $0x80

# Compara %esi com 10
cmp $0xa, %esi
jz FIM          # Se igual a 10 pula para FIM.
jmp INICIO   # Senao volta ao INICIO.
# Ele vai incrementar ate que %esi seja 10 e assim pular
# para label FIM


# Label FIM
FIM:

# syscall exit (1)
movl $0x1, %eax
movl $0x0, %ebx
int $0x80

# end
```

--- loop-asm.s ---

viu como nao é nada complicado?
é só pegar a manha e estudar, que as coisas ficam claras

## 11. GCC INLINE

Com o gcc existe a possibilidade de inserçao de instrucoes em assembly em codigos 'C',
utilizando a função __asm ();, essa instrucao determina a inicializacao de instrucoes assembly no codigo C.
uma das utilizações do GCC INLINE é ajudar na escrita de
shellcodes e exploits. mas podemos usar para diversos fins, obvio
o GCC INLINE pode ser usado assim:

Exemplo I

```
main (){

__asm (" NOP;NOP;NOP \n");

}
```

Exemplo II

```
main (){

__asm (

"NOP \n"
"NOP ");
}
```

ou

```
asm("nop;nop;nop");
```

ou

```
__asm__(
"movl $0x4, %eax \n"
"int $0x80 \n"
);
```

todas as declarações tem o mesmo significado. só muda
a forma de declararmos. vamos ver um exemplo de uso do
GCC INLINE. esse codigo contem um shellcode que executa
/bin/bash

--- shlpoison.c ---

```
#include <stdio.h>

char shlpoison[] = "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
"\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
"\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
"\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42";
__asm("jmp shlpoison"); /* irá pular para shlpoison, que é noiss
```

shellcode 🙂 */

}

--- shlpoison.c ---

ah, esqueci de comentar:
quando for compilar com o gcc, use o parametro -o

*bt / # gcc shlpoison.c -o shlpoison*

era só isso. espero que tenha ficado claro que o GCC INLINE
é usado usar codigos asm dentro de um codigo em C
caso voce precisse dependendo da sua necessidade e tals
a maioria usa mais em shellcodes e exploits mesmo, mas voce pode usar
para qualquer coisa. basta por as instruções asm dentro do code com a função
__asm(); 🙂

agora se voce encontrar jmp's e tals num codigo C nao vai ficar se perguntando
como faz isso rs..

## 12.Finalizando

the end, but its not the end

meus amigos, é meio triste que anuncio o termino do nosso
pequeno e basico curso de assembly :////
mas como o titulo do capitulo fala, esse é o fim
mas nao é o fim rs.. ou seja, claro que teremos continuação!
seja em outro curso de asm ou em algum outro projeto, nao sei..
mas eu continuarei escrevendo sobre ASM
nem tenho muito oque falar aqui. só tenho a agradeçer a todos os meus amigos
da NSG que fizeram esse curso possivel. nao só o curso, mas todo o projeto
de de programação. voces nao imaginam o quanto eu estou feliz!!

quero agradeçer a todos os meus alunos, que estiverem presente em todo o curso!
quero dizer que se tiverem duvidas, ja sabem.. só falarem aqui ou no meu msn que eu
ajudo sempre com maior prazer

quero agradecer a todos os users do forum que fazem
a NSG crescer. a todos os meus alunos do curso.
a todos os envolvidos do nosso projeto. a todos os nossos

leitores, visitantes, nao importa! a todos que acessam a NSG!

espero que o curso tenha sido util para voces, amigos
que nao tenha ficado nada vago. que tenham gostado..
claro, o assunto sobre ASM nao acaba ae!
isso foi só um basico, como eu disse.. há sempre o que correr atrás
para sempre evoluirmos, entao estude, amigo! nao pare pois ninguem pode
te segurar! duvidas, criticas, sugestões e qualquer tipo de manifestação
me contate pelo forum!

um forte abraço, e nos encontraremos
novamente no proximo curso de ASM :D

**Felix_Poison**

**~ Nemesiz Security Group.** 2009 - 2010