Assembly x86-64

Computadores executam código de máquina, codificado em bytes, para executar tarefas em um computador. Como computadores diferentes possuem processadores diferentes, o código de máquina executado nesses computadores é específico para cada processador. Neste caso, examinaremos a arquitetura de conjunto de instruções Intel x86-64, a mais comumente encontrada atualmente. O código de máquina geralmente é representado por uma forma mais legível do código, chamada código assembly. Esse código de máquina geralmente é produzido por um compilador, que pega o código-fonte de um arquivo e, após passar por alguns estágios intermediários, produz código de máquina que pode ser executado por um computador. Sem entrar em muitos detalhes, a Intel começou construindo um conjunto de instruções de 16 bits, seguido por um de 32 bits e, finalmente, criou o de 64 bits. Todos esses conjuntos de instruções foram criados para compatibilidade com versões anteriores, de modo que o código compilado para a arquitetura de 32 bits será executado em máquinas de 64 bits. Como mencionado anteriormente, antes de um arquivo executável ser produzido, o código-fonte é primeiro compilado em assembly (arquivos .s), depois o montador o converte em um programa objeto (arquivos .o), e operações com um vinculador finalmente o tornam um executável.

A melhor maneira de começar a explicar assembly é mergulhando de cabeça. Usaremos o radare2 para isso. radare2 é um framework para engenharia reversa e análise de binários. Ele pode ser usado para desmontar binários (traduzir código de máquina para assembly, que é realmente legível) e depurar esses binários.

O primeiro passo é executar a introdução do programa executando
./file1

```
ashu@ashu-Inspiron-5379 ~/D/t/c/christmas-re> ./file1
the value of a is 4, the value of b is 5 and the value of c is 9
```

O programa acima mostra que há 3 variáveis (a, b, c), onde c é a soma de a e b.

Hora de ver o que está acontecendo nos bastidores! Execute o comando
r2 -d ./file1

Isso abrirá o binário em modo de depuração. Uma vez aberto, uma das primeiras coisas a fazer é pedir ao r2 para analisar o programa, **digitando: aa**

Este é o comando de análise mais comum. Ele analisa todos os símbolos e pontos de entrada no executável.

A análise neste caso envolve a extração de nomes de funções, informações de controle de fluxo e muito mais! As instruções r2 geralmente são baseadas em um único caractere, então é fácil obter mais informações sobre os comandos.

Para obter ajuda geral, execute:
?

Para obter informações mais específicas, por exemplo, sobre análise, execute
a?

Após a conclusão da análise, você vai querer saber por onde começar a análise. A maioria dos programas tem um ponto de entrada definido como ***main***. Para encontrar uma lista das funções, execute:
afl

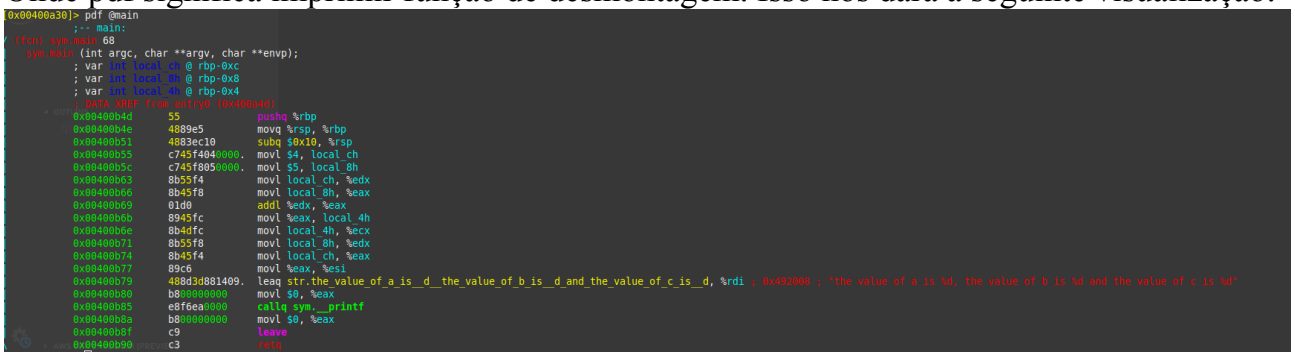
```
[0x00400a30]> afl | grep main


0x00400b4d    1 68           sym.main
0x00400e10  114 1657         sym.__libc_start_main
0x00403870  346 6038 -> 5941 sym._nl_find_domain
0x00415fe0    1 43           sym._IO_switch_to_main_get_area
0x0044cf00    1 8            sym._dl_get_dl_main_map
0x00470520    1 49           sym._IO_switch_to_main_wget_area
0x0048fae0    7 73   -> 69   sym._nl_finddomain_subfreeres
0x0048fb30   16 247  -> 237  sym._nl_unload_domain
```

Observe que os endereços de memória podem ser diferentes no seu computador.

Como visto aqui, na verdade existe uma função em main. Vamos examinar o código assembly em main executando o comando
pdf @main

Onde pdf significa imprimir função de desmontagem. Isso nos dará a seguinte visualização:



O núcleo da linguagem assembly envolve o uso de registradores para fazer o seguinte:

- Transferir dados entre a memória e o registrador e vice-versa
- Executar operações aritméticas em registradores e dados
- Transferir o controle para outras partes do programa

Como a arquitetura é x86-64, os registradores são de 64 bits e a Intel tem uma lista de 16 registradores:

64 bit	32 bit
%rax	%eax
%rbx	%ebx

%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp
%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

Embora os registradores sejam de 64 bits, o que significa que podem armazenar até 64 bits de dados, outras partes dos registradores também podem ser referenciadas. Nesse caso, os registradores também podem ser referenciados como valores de 32 bits, como mostrado. O que não é mostrado é que os registradores podem ser referenciados como de 16 bits e 8 bits (4 bits superiores e 4 bits inferiores).

Os primeiros 6 registradores são conhecidos como registradores de uso geral, enquanto **%rsp** e **%rbp** são de uso especial e seus significados serão explicados posteriormente. Para mover dados usando registradores, a seguinte instrução é usada:
movq origem, destino

Isso envolve:

- Transferir constantes (que são prefixadas com o operador $), por exemplo, ***movq $3 rax***, moveria a constante 3 para o registrador.
- Transferir valores de um registrador, por exemplo, ***movq %rax %rbx***, que envolve mover o valor de ***rax*** para ***rbx***.

- Transferir valores da memória, que é mostrado colocando os registradores entre colchetes. Por exemplo, ***movq %rax (%rbx)***, que significa mover o valor armazenado em ***%rax*** para o local de memória representado por ***%rbx***.

A última letra da instrução **mov** representa o tamanho dos dados:

Intel Data Type	Suffix	Size(bytes)
Byte	b	1
Word	w	2
Double Word	l	4
Quad Word	q	8
Quad Word	q	8
Single Precision	s	4
Double Precision	l	8

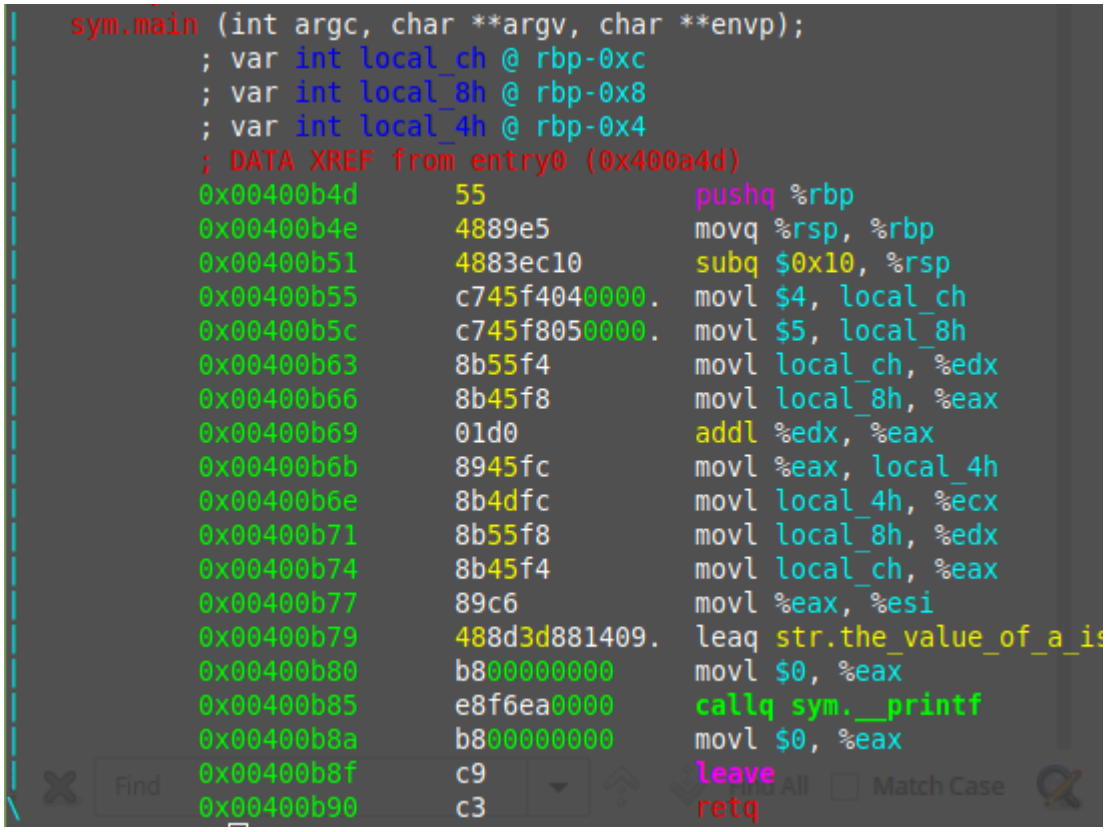
Ao lidar com manipulação de memória usando registradores, há outros casos a serem considerados:

- (Rb, Ri) = Localização da Memória[Rb + Ri]
- D(Rb, Ri) = Localização da Memória[Rb + Ri + D]
- (Rb, Ri, S) = Localização da Memória(Rb + S * Ri]
- D(Rb, Ri, S) = Localização da Memória[Rb + S * Ri + D]

Algumas outras instruções importantes são:

- ***leaq source, destination:*** esta instrução define o destino para o endereço denotado pela expressão em source
- ***addq source, destination:*** destino = destino + fonte
- ***subq* source, destination:** destino = destino - fonte
- ***imulq source, destination:*** destino = destino * fonte
- ***salq source, destination:*** destino = destino << fonte onde << é o operador de deslocamento de bit à esquerda
- ***sarq source, destination*:** destino = destino >> fonte onde >> é o operador de deslocamento de bit correto
- ***xorq source, destination*:** destino = destino OU fonte
- ***andq source, destination*:** destino = destino e fonte
- ***orq source, destination*:** destino = destino | fonte

Agora vamos examinar o código assembly para ver o que as instruções significam quando combinadas.



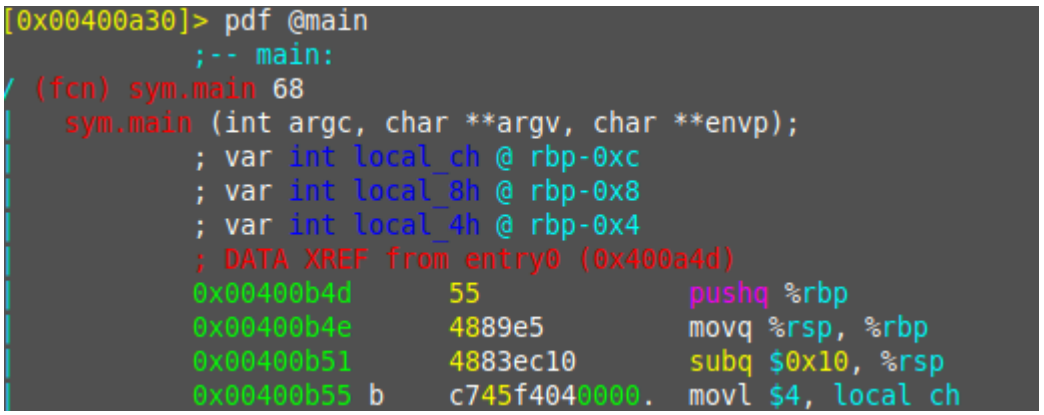
A linha que começa com ***sym.main*** indica que estamos olhando para a função principal. As próximas 3 linhas são usadas para representar as variáveis armazenadas na função. A segunda coluna indica que são inteiros (int), a terceira coluna especifica o nome que r2 usa para referenciá-los e a quarta coluna mostra a localização real da memória.

As 3 primeiras instruções são usadas para alocar espaço na pilha (garantir que haja espaço suficiente para alocar variáveis e mais). Começaremos a analisar o programa a partir da 4ª instrução (***movl $4***).

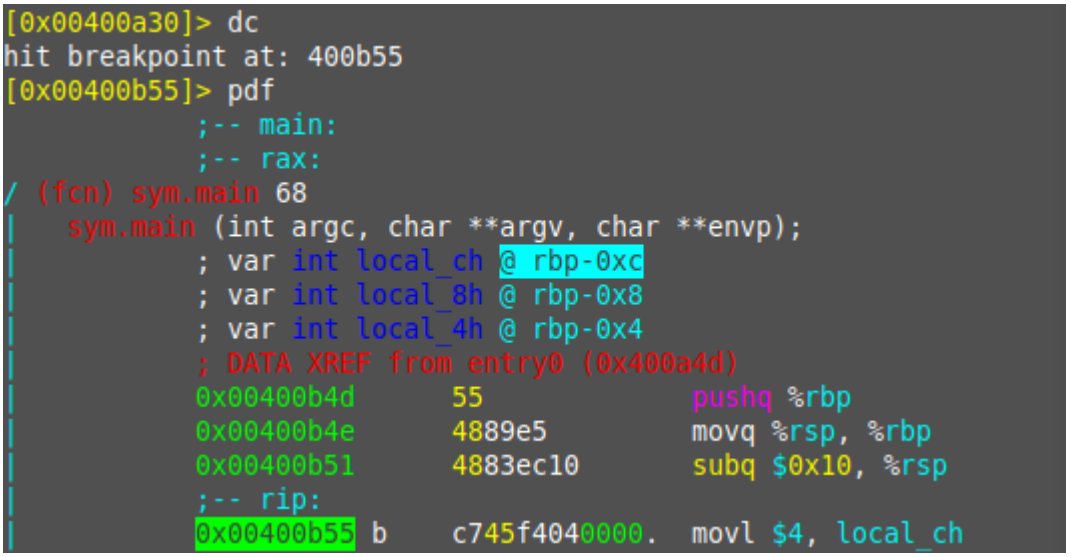
Queremos analisar o programa enquanto ele é executado e a melhor maneira de fazer isso é usando pontos de interrupção. Um ponto de interrupção especifica onde o programa deve parar de executar. Isso é útil, pois nos permite observar o estado do programa naquele ponto específico.

Então, vamos definir um ponto de interrupção usando o comando
db address
neste caso, seria
db 0x00400b55

Para garantir que o ponto de interrupção esteja definido, executamos o comando pdf @main novamente e vemos um pequeno b ao lado da instrução na qual queremos parar



Agora que definimos um ponto de interrupção, vamos executar o programa usando **dc**



Executar ***dc*** executará o programa até atingirmos o ponto de interrupção. Assim que atingirmos o ponto de interrupção e imprimirmos a função principal, o rip, que é a instrução atual, mostra onde a execução parou. Pelas notas acima, sabemos que a instrução ***mov*** é usada para transferir valores. Esta instrução está transferindo o valor 4 para a variável ***local_ch***.

Para visualizar o conteúdo da variável ***local_ch***, usamos a seguinte instrução:
px @memory-address

Neste caso, o endereço de memória correspondente para ***local_ch*** será ***rbp-0xc*** (das primeiras linhas de @pdf main).
Esta instrução imprime os valores de memória em hexadecimal:

```
[0x00400b55]> px @ rbp-0xc
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffc914f7bc4  0000 0000 1890 6b00 0000 0000 7018 4000  ......k.....p.@.
0x7ffc914f7bd4  0000 0000 1911 4000 0000 0000 0000 0000  ......@.........
0x7ffc914f7be4  0000 0000 0000 0000 0100 0000 f87c 4f91  .............|O.
0x7ffc914f7bf4  fc7f 0000 4d0b 4000 0000 0000 0000 0000  ....M.@.........
0x7ffc914f7c04  0000 0000 0600 0000 8e00 0000 8000 0000  ................
0x7ffc914f7c14  0a00 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c24  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c34  0000 0000 0000 0000 0000 0000 0004 4000  ..............@.
0x7ffc914f7c44  0000 0000 52db fe41 3933 915f 1019 4000  ....R..A93._..@.
0x7ffc914f7c54  0000 0000 0000 0000 0000 0000 1890 6b00  ..............k.
0x7ffc914f7c64  0000 0000 0000 0000 0000 0000 52db de86  ............R...
0x7ffc914f7c74  2711 68a0 52db 8a50 3933 915f 0000 0000  '.h.R..P93._....
0x7ffc914f7c84  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c94  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7ca4  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7cb4  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Isso mostra que a variável atualmente não possui nada armazenado (apenas 0000). Vamos executar esta instrução e passar para a próxima usando o seguinte comando (que só leva para a próxima instrução):
ds

Se visualizarmos a localização da memória após executar este comando, obteremos o seguinte:

```
[0x00400b55]> px @ rbp-0xc
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffc914f7bc4  0400 0000 1890 6b00 0000 0000 7018 4000  ......k.....p.@.
0x7ffc914f7bd4  0000 0000 1911 4000 0000 0000 0000 0000  ......@.........
0x7ffc914f7be4  0000 0000 0000 0000 0100 0000 f87c 4f91  .............|O.
0x7ffc914f7bf4  fc7f 0000 4d0b 4000 0000 0000 0000 0000  ....M.@.........
0x7ffc914f7c04  0000 0000 0600 0000 8e00 0000 8000 0000  ................
0x7ffc914f7c14  0a00 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c24  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c34  0000 0000 0000 0000 0000 0000 0004 4000  ..............@.
0x7ffc914f7c44  0000 0000 52db fe41 3933 915f 1019 4000  ....R..A93._..@.
0x7ffc914f7c54  0000 0000 0000 0000 0000 0000 1890 6b00  ..............k.
0x7ffc914f7c64  0000 0000 0000 0000 0000 0000 52db de86  ............R...
0x7ffc914f7c74  2711 68a0 52db 8a50 3933 915f 0000 0000  '.h.R..P93._....
0x7ffc914f7c84  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c94  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7ca4  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7cb4  0000 0000 0000 0000 0000 0000 0000 0000  ................
[0x00400b55]>
```

Podemos ver que os 2 primeiros bytes têm o valor 4! Se fizermos o mesmo processo para a próxima instrução, veremos que a variável ***local_8h*** tem o valor 5.

```
[0x00400b55]> px @ rbp-0x8
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffc914f7bc8  0500 0000 0000 0000 7018 4000 0000 0000  ........p.@.....
0x7ffc914f7bd8  1911 4000 0000 0000 0000 0000 0000 0000  ..@.............
0x7ffc914f7be8  0000 0000 0100 0000 f87c 4f91 fc7f 0000  .........|O.....
0x7ffc914f7bf8  4d0b 4000 0000 0000 0000 0000 0000 0000  M.@.............
0x7ffc914f7c08  0600 0000 8e00 0000 8000 0000 0a00 0000  ................
0x7ffc914f7c18  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c28  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c38  0000 0000 0000 0000 0004 4000 0000 0000  ..........@.....
0x7ffc914f7c48  52db fe41 3933 915f 1019 4000 0000 0000  R..A93._..@.....
0x7ffc914f7c58  0000 0000 0000 0000 1890 6b00 0000 0000  ..........k.....
0x7ffc914f7c68  0000 0000 0000 0000 52db de86 2711 68a0  ........R...'.h.
0x7ffc914f7c78  52db 8a50 3933 915f 0000 0000 0000 0000  R..P93._........
0x7ffc914f7c88  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7c98  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7ca8  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffc914f7cb8  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Se formos para a instrução ***movl local_8h, %eax***, sabemos pelas notas que isso move o valor de ***local_8h*** para o registrador ***%eax***. Para ver o valor do registrador ***%eax***, podemos usar o comando:
dr

```
[0x00400b55]> dr
rax = 0x00400b4d
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
r8 = 0x00000000
r9 = 0x00000007
r10 = 0x00000002
r11 = 0x00000001
r12 = 0x00401910
r13 = 0x00000000
r14 = 0x006b9018
r15 = 0x00000000
rsi = 0x7ffc914f7cf8
rdi = 0x00000001
rsp = 0x7ffc914f7bc0
rbp = 0x7ffc914f7bd0
rip = 0x00400b66
rflags = 0x00000206
orax = 0xffffffffffffffff
```

Se executarmos a instrução e executarmos o comando ***dr*** novamente, obteremos:

```
[0x00400b55]> dr
rax = 0x00000005
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
r8 = 0x00000000
r9 = 0x00000007
```

Tecnicamente, isso ignora a instrução anterior ***movl local_ch, %edx***, mas o mesmo processo pode ser aplicado a ela.

Mostrando o valor de ***rax*** (a versão de 64 bits) como 5. Podemos fazer o mesmo para instruções semelhantes e visualizar a alteração dos valores dos registradores.

Quando chegamos ao ***addl %edx***, ***%eax,*** sabemos que isso adicionará os valores em ***edx*** e ***eax*** e os armazenará em ***eax***. Executar ***dr*** nos mostra que ***rax*** contém 5 e ***rdx*** contém 4, então esperaríamos que ***rax*** contivesse 9 após a execução da instrução.

```
[0x00400b55]> dr
rax = 0x00000005
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
```

Executar ds para passar para a próxima instrução e depois executar dr para visualizar a variável de registro nos mostra que estamos corretos

```
[0x00400b55]> dr
rax = 0x00000009
rbx = 0x00400400
```

As próximas instruções envolvem mover os valores nos registradores para as variáveis e vice-versa

```
;-- rip:
0x00400b6b      8945fc          movl %eax, local_4h
0x00400b6e      8b4dfc          movl local_4h, %ecx
0x00400b71      8b55f8          movl local_8h, %edx
0x00400b74      8b45f4          movl local_ch, %eax
0x00400b77      89c6            movl %eax, %esi
```

```
0x00400b79      488d3d881409.   leaq str.the_value_of_a_is__d__the_value_of_b_is__d_and_the_value_of_c_is__d, %r
0x00400b80      b800000000      movl $0, %eax
0x00400b85      e8f6ea0000      callq sym.__printf
0x00400b8a      b800000000      movl $0, %eax
0x00400b8f      c9              leave
0x00400b90      c3              retq
```

Depois disso, uma string (que é a saída) é carregada em um registrador e a função printf é chamada na terceira linha. A segunda linha limpa o valor de ***eax***, pois ***eax*** às vezes é usado para armazenar resultados de funções. A quarta linha limpa o valor de ***eax***. A quinta e a sexta linhas são usadas para sair da função principal.

A fórmula geral para resolver algo assim é:

- Defina pontos de interrupção apropriados
- Use ds para navegar pelas instruções e verificar os valores do registrador e da memória
- Se cometer um erro, você sempre pode recarregar o programa usando o comando ***ood***

Parte 2: Execução
Comandos

aa
afl | grep main
pdf @main
db 0x00f00b58
dc
px @ rbp-0xc

```
[0×00400b58]> px @ rbp-0×c
- offset -      8485 8687 8889 8A8B 8C8D 8E8F 9091 9293  456789ABCDEF0123
0×7ffe2afae984  0100 0000 1890 6b00 0000 0000 4018 4000  . ... ..k.....@.@.
0×7ffe2afae994  0000 0000 e910 4000 0000 0000 0000 0000  ....  ..@.........
0×7ffe2afae9a4  0000 0000 0000 0000 0100 0000 b8ea fa2a  ........ ... ...*
0×7ffe2afae9b4  fe7f 0000 4d0b 4000 0000 0000 0000 0000  .. ..M.@.........
0×7ffe2afae9c4  0000 0000 0600 0000 9e00 0000 9000 0000  ..... ... . ... . ...
0×7ffe2afae9d4  0a00 0000 0000 0000 0000 0000 0000 0000  ................
0×7ffe2afae9e4  0000 0000 0000 0000 0000 0000 0000 0000  ................
0×7ffe2afae9f4  0000 0000 0000 0000 0000 0000 0004 4000  ..............@.
0×7ffe2afaea04  0000 0000 870d bb07 f719 a47c e018 4000  ...........|..@.
```

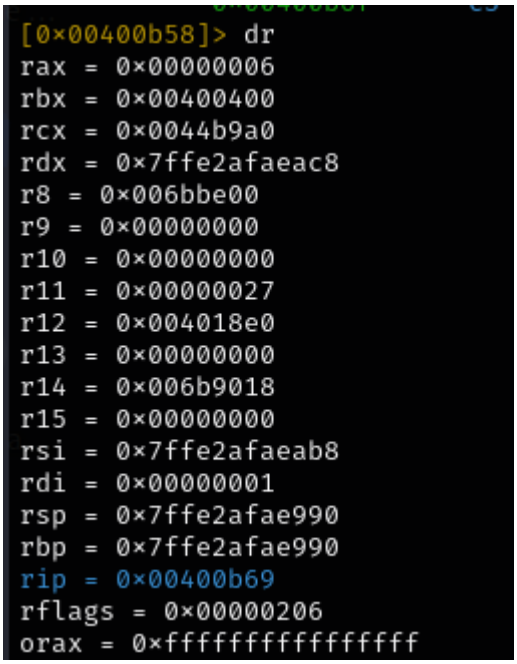
valor = 1

ds
ds
ds
dr

```
[0×00400b58]> dr
rax = 0×00000006
rbx = 0×00400400
rcx = 0×0044b9a0
rdx = 0×7ffe2afaeac8
r8 = 0×006bbe00
r9 = 0×00000000
r10 = 0×00000000
r11 = 0×00000027
r12 = 0×004018e0
r13 = 0×00000000
r14 = 0×006b9018
r15 = 0×00000000
rsi = 0×7ffe2afaeab8
rdi = 0×00000001
rsp = 0×7ffe2afae990
rbp = 0×7ffe2afae990
rip = 0×00400b66
rflags = 0×00000206
orax = 0×ffffffffffffffff
[0×00400b58]>
```

valor = 6

ds
dr


[0×00400b58]> dr
rax = 0×00000006
rbx = 0×00400400
rcx = 0×0044b9a0
rdx = 0×7ffe2afaeac8
r8 = 0×006bbe00
r9 = 0×00000000
r10 = 0×00000000
r11 = 0×00000027
r12 = 0×004018e0
r13 = 0×00000000
r14 = 0×006b9018
r15 = 0×00000000
rsi = 0×7ffe2afaeab8
rdi = 0×00000001
rsp = 0×7ffe2afae990
rbp = 0×7ffe2afae990
rip = 0×00400b69
rflags = 0×00000206
orax = 0×ffffffffffffffff

valor = 6 (rax)