Laboratório 3 – Felipe Melo – Thalles Nonato

DRE Felipe: 119093752

DRE Thalles: 119058809

Primeiro desafio:

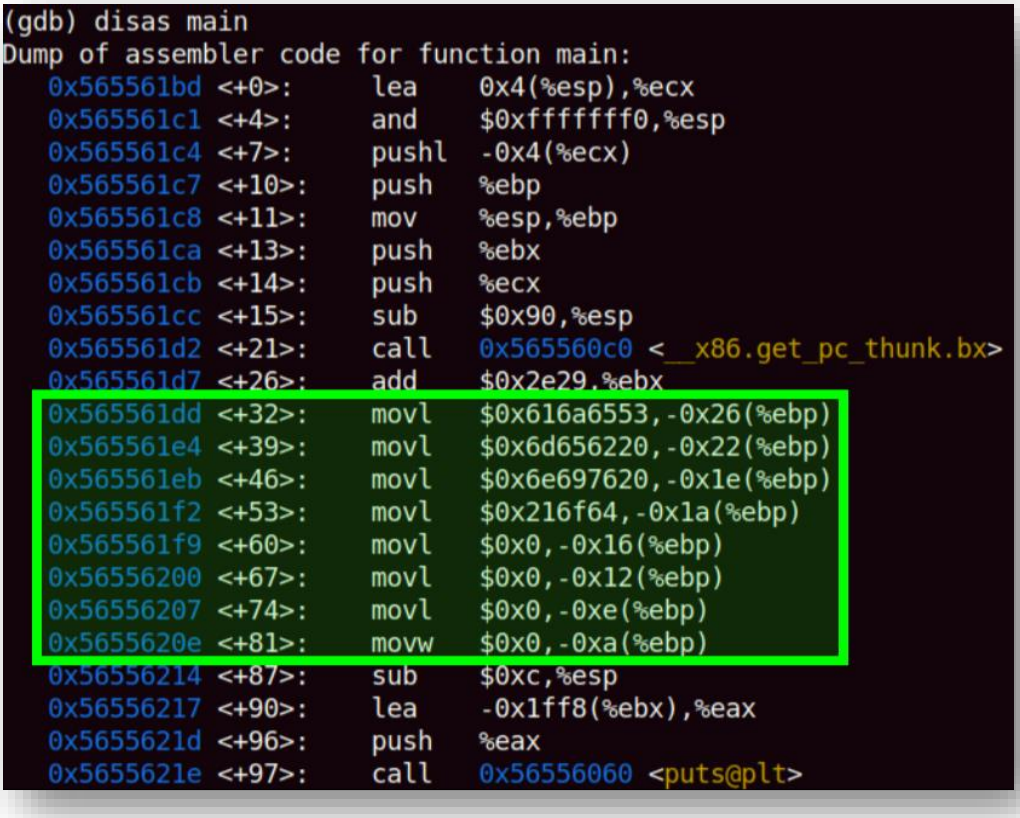
1) Analisando as instruções após o `gets`, vimos que o tamanho do buffer poderia estar na instrução `lea` em destaque. Daí, verificamos que 0x8A = 138_{10}. Ao executar o programa, notamos que se ultrapassássemos o limite de 138 caracteres, o programa apresentava o erro segmentation fault, comprovando o que desconfiávamos. No entanto, essa quantidade de bytes não era necessariamente o que o programa esperava que utilizássemos para digitar o nome. Isso pois descobrimos que o programa sempre imprime "Olá, !" (ocupando 8 bytes, já que o caractere "á" equivale a 2 bytes por ter acento e ao final de toda string há um "\0") e a string "Seja bem vindo!" ocupa 30 bytes do nosso programa. Portanto, o tamanho do buffer responsável por ler o nome do usuário tem 100 bytes.

```
0x56556217 <+90>:    lea    -0x1ff8(%ebx),%eax
0x5655621d <+96>:    push   %eax
0x5655621e <+97>:    call   0x56556060 <puts@plt>
0x56556223 <+102>:   add    $0x10,%esp
0x56556226 <+105>:   sub    $0xc,%esp
0x56556229 <+108>:   lea    -0x8a(%ebp),%eax
0x5655622f <+114>:   push   %eax
0x56556230 <+115>:   call   0x56556050 <gets@plt>
0x56556235 <+120>:   add    $0x10,%esp
0x56556238 <+123>:   sub    $0x8,%esp
0x5655623b <+126>:   lea    -0x8a(%ebp),%eax
0x56556241 <+132>:   push   %eax
0x56556242 <+133>:   lea    -0x1fe3(%ebx),%eax
0x56556248 <+139>:   push   %eax
```

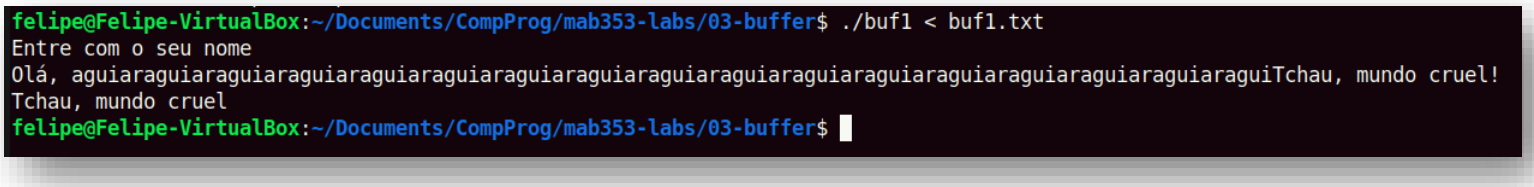
2) O tamanho do buffer de “Seja bem vindo!”, como constatado acima, é 30 bytes. Chegamos a essa constatação devido às instruções `mov` em destaque na imagem abaixo. Da mesma forma como foi cobrado no laboratório 2, elas estão dispostas de modo a receber os bytes correspondentes em ASCII da string “Seja bem vindo!” da seguinte forma:

ajeS meb niv !od

Após essa string, há uma sequência de caracteres nulos, identificados pelos `%0x0`. Portanto, como são 7 instruções `movl` e uma instrução `movw`, que representam, respectivamente, a cópia de dados de 4 e 2 bytes. Dessa forma, [(7 * 4) + 2] = 30 bytes.



3) O `printf` só para depois de “Tchau, mundo cruel” pois está sempre buscando o final do input, caracterizado pelo byte “\0”. Além disso, a função `gets` lê a entrada e imediatamente após o último byte, insere o caractere “\0”. Portanto, não importa o tamanho do nome que colocamos (desde que não passe dos 138 bytes), o `printf` lê até o final da string que passamos pelo `gets`.



Segundo desafio:

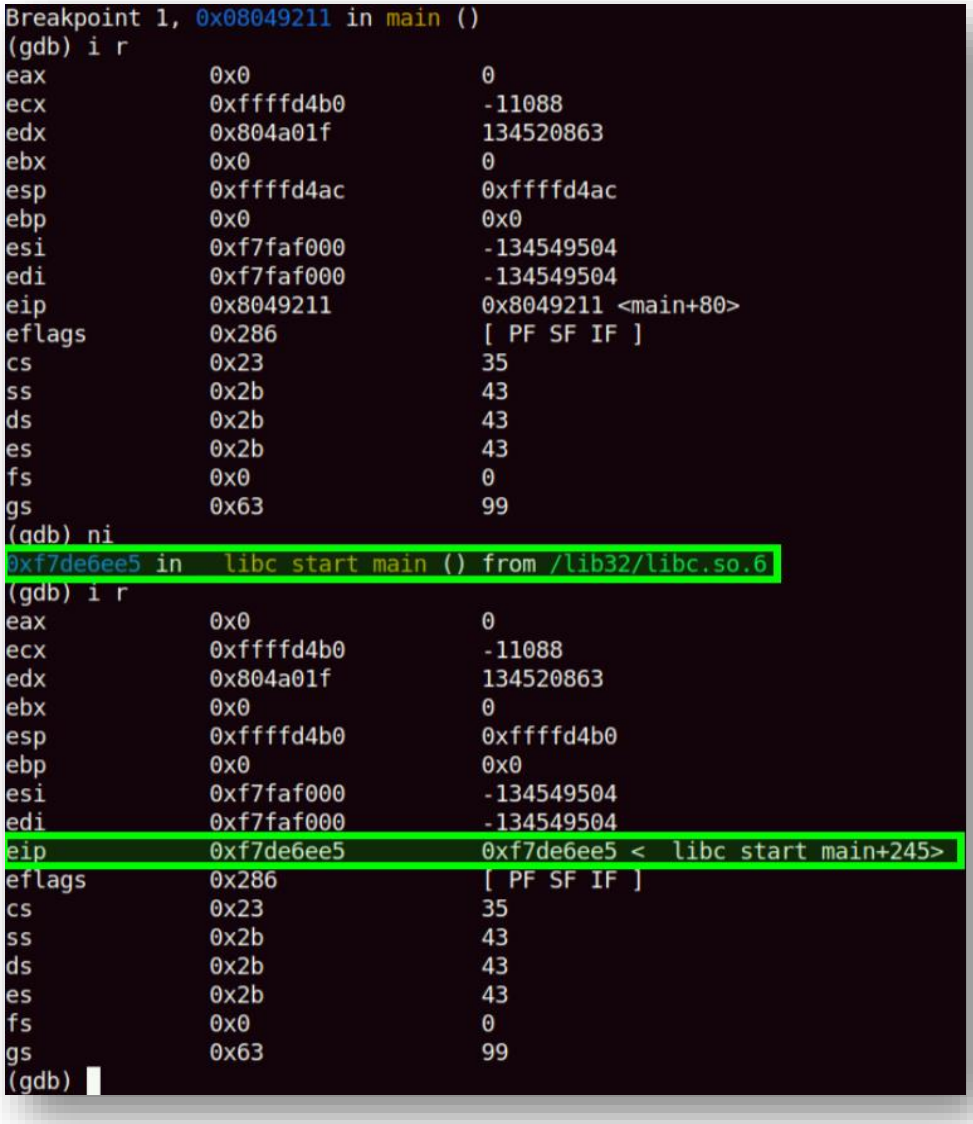
1) O endereço do código morto é 0x08049196. Esse endereço é o da primeira instrução da função e descobrimos utilizando o comando `objdump -d buf2`

```
08049196 <codigo_morto>:
 8049196:       55                      push   %ebp
 8049197:       89 e5                   mov    %esp,%ebp
 8049199:       53                      push   %ebx
 804919a:       83 ec 04                sub    $0x4,%esp
 804919d:       e8 70 00 00 00          call   8049212 <__x86.get_pc_thunk.ax>
 80491a2:       05 5e 2e 00 00          add    $0x2e5e,%eax
 80491a7:       83 ec 0c                sub    $0xc,%esp
 80491aa:       8d 90 08 e0 ff ff       lea    -0x1ff8(%eax),%edx
 80491b0:       52                      push   %edx
 80491b1:       89 c3                   mov    %eax,%ebx
 80491b3:       e8 a8 fe ff ff          call   8049060 <puts@plt>
 80491b8:       83 c4 10                add    $0x10,%esp
 80491bb:       90                      nop
 80491bc:       8b 5d fc                mov    -0x4(%ebp),%ebx
 80491bf:       c9                      leave
 80491c0:       c3                      ret
```

2) Para descobrir o tamanho do buffer de leitura, bastou reparar na instrução `sub $0x40, esp`, responsável por abrir 64 bytes na pilha, uma vez que 0x40 = 64_{10}. Dessa maneira, o buffer possui 64 bytes de tamanho.

```
(gdb) disas main
Dump of assembler code for function main:
   0x080491c1 <+0>:     lea    0x4(%esp),%ecx
   0x080491c5 <+4>:     and    $0xfffffff0,%esp
   0x080491c8 <+7>:     pushl  -0x4(%ecx)
   0x080491cb <+10>:    push   %ebp
   0x080491cc <+11>:    mov    %esp,%ebp
   0x080491ce <+13>:    push   %ebx
   0x080491cf <+14>:    push   %ecx
   0x080491d0 <+15>:    sub    $0x40,%esp
   0x080491d3 <+18>:    call   0x80490d0 <__x86.get_pc_thunk.bx>
   0x080491d8 <+23>:    add    $0x2e28,%ebx
   0x080491de <+29>:    sub    $0xc,%esp
   0x080491e1 <+32>:    lea    -0x48(%ebp),%eax
   0x080491e4 <+35>:    push   %eax
   0x080491e5 <+36>:    call   0x8049050 <gets@plt>
```

3) `%ebp` aponta para OFP. Já para descobrir o `eip` de retorno de main, criamos um breakpoint utilizando o comando `break *08049211` (endereço da instrução `ret` em main). Após executar o programa com `run` e digitar qualquer coisa quando é solicitado uma entrada, pudemos finalmente parar no endereço imediatamente anterior à função de retorno. Com isso, executamos a instrução com `ni` e em seguida digitamos `i r` para olhar os conteúdos dos registradores e, assim, bastou verificar no campo `eip` em destaque na imagem para concluir que o endereço da função `__libc_start_main ()` era para onde ele apontava.



4) 80 bytes

5) 0x08049196, mas devemos escrever do byte menos significativo do endereço para o mais significativo, visto que é uma arquitetura little endian.

6) Pois existem alguns procedimentos que devem ser realizados após o término da função main, tais como desanexação de recursos compartilhados e liberação de quaisquer recursos não limpos automaticamente quando o processo termina

Terceiro desafio:

O processo para inserir shellcode na pilha consiste em estourar o buffer de leitura, de modo a sobrescrever o endereço apontado pelo frame pointer. Podemos fazer isso inserindo uma sequência de instruções No-Op, identificadas por "\x90", após a sequência de bytes que usamos para sobrecarregar o buffer. A quantidade dessas instruções vai depender diretamente do número de bytes do shellcode que queremos inserir. Além disso, devemos alterar o endereço de retorno e para isso, temos de escolher um endereço no meio dos que estão preenchidos com a sequência de No-Op, de modo a finalmente levar o fluxo de execução para o shellcode.

No exemplo do `buf3` há um buffer de tamanho igual a 32 bytes (que identificamos no GDB ao realizar o disassemble da main, pela instrução `sub $0x20, %esp`), portanto, utilizamos um script em Python para sobrecarregá-lo:

```
python2 -c "print('A' * 32)"
```

Com os No-Ops, o script ficaria:

```
python2 -c "print('\x90' * 32 + shellcode + "\x41\x41\x41\x41" * 10)"
```

Adicionamos a sequência de \x41 para assegurar que haverá conteúdo entre o shellcode e a pilha. Porém, se apenas esse script for executado, ele excederá a pilha. Portanto, precisamos retirar os bytes do shellcode e de endereço de retorno, como explicado acima. Dessa maneira, nosso script tem a seguinte forma:

```
python2 -c "print('\x90' * (32 - bytes(shellcode) - bytes(retorno)) + shellcode +
                        "\x41\x41\x41\x41" * 10)"
```