Guião VIII - Resolução

Exercício 1

A função `getline` lê uma linha da entrada, faz uma cópia da sequência de carateres para o espaço reservado na pilha e retorna um apontador para o espaço reservado.

```
char *getline() {
  char buf[8];
  char *result;
  gets(buf);
  result = malloc(strlen(buf));
  strcpy(result, buf);
  return result;
}
```

Considere que `getline` é chamada com o endereço de retorno igual a `0x8048643`, com `%ebp=0xbffffc94`, `%ebx=0x1`, `%edi=0x2`, e `%esi=0x3`. Após a entrada de dados correspondentes à sequência "0123456789012345678901234" o programa termina com uma falha de segmentação. Usando o `gdb` pôde determinar-se que o erro ocorreu durante a execução da instrução `ret` de `getline`.

```
1 080485c0 <getline>:
2 080485c0: 55          push %ebp
3 080485c1: 89 e5       mov  %esp,%ebp
4 080485c3: 83 ec 28    sub  $0x28,%esp
5 080485c6: 89 5d f4    mov  %ebx,-0xc(%ebp)
6 080485c9: 89 75 f8    mov  %esi,-0x8(%ebp)
7 080485cc: 89 7d fc    mov  %edi,-0x4(%ebp)
```

Diagrama da pilha neste ponto

```
8  080485cf: 8d 75 ec         lea  -0x14(%ebp),%esi ; %ebp-0x14 <=> buf
9  080485d2: 89 34 24         mov  %esi,(%esp)     ; arg. de gets <=> buf
10 080485d5: e8 a3 ff ff ff   call 804857d <gets>
```

Modificar o diagrama da pilha neste ponto

Nota: O compilador `gcc` reserva uma área na pilha para cada uma das funções do programa que inclui: as variáveis locais (escalares ou estruturadas), a informação de estado, valores de registos preservados e o endereço de retorno. No caso especial dos vetores, nem a linguagem nem o compilador faz qualquer controlo sobre a gama de variação dos índices de acesso aos elementos dos vetores. Nesta circunstância, se forem feitas leituras/escritas fora dos limites de espaço podem criar-se situações estranhas que levem à produção de erros de segmentação. É esta a situação criada durante a execução da função `gets`, se o número de elementos fornecidos à entrada ultrapassar os limites definidos para a variável passada como argumento.

a) Preencha o diagrama que se segue da pilha (cada linha representa 4 bytes), indicando a posição do `%ebp` e toda a informação disponível após a execução da instrução na linha 7 no código de montagem: use valores hexadecimais (se conhecidos) dentro das caixas e identifique os mesmos (por exemplo, "endereço de retorno") do lado direito.

```
+-------------+
|             |
+-------------+
|             |
+-------------+
|             | (%ebp-0x28) (%esp após linha 4)
+-------------+
|             | (%ebp-0x24)
+-------------+
|             | (%ebp-0x20)
+-------------+
|             | (%ebp-0x1C)
+-------------+
|             | (%ebp-0x18)
+-------------+
|             | (%ebp-0x14)
+-------------+
|             | (%ebp-0x10)
+-------------+
| 00 00 00 01 | ⇔ %ebx guardado       (%ebp-0xC)
+-------------+
| 00 00 00 03 | ⇔ %esi guardado       (%ebp-8)
+-------------+
| 00 00 00 02 | ⇔ %edi guardado       (%ebp-4)
+-------------+
| bf ff fc 94 | ⇔ %ebp guardado       (%ebp)
+-------------|
| 08 04 86 43 | ⇔ endereço de retorno (%ebp+4)
+-------------+
```

b) Modificar o diagrama para mostrar o efeito da chamada a `gets` (linha 10).

Como a função `gets` vai ler 24 carateres + 1 (carater de terminação '\0'), será excedida a capacidade inicialmente alocada para o `buf`: 8 bytes em vez de 25, o que irá corromper 17 bytes da pilha (valores a vermelho).

```
+-------------+
| 08 04 85 da | ⇔ endereço de retorno do gets      (%ebp-0x2C)
+-------------+
|    buf      | ⇔ arg. do gets (%esp após linha 4) (%ebp-0x28)
+-------------+
|             |                          (%ebp-0x24)
+-------------+
|             |                          (%ebp-0x20)
+-------------+
|             |                          (%ebp-0x1C)
+-------------+
|             |                          (%ebp-0x18)
+-------------+
| 33 32 31 30 | ⇔ buf[3:0]               (%ebp-0x14)
+-------------+
| 37 36 35 34 | ⇔ buf[7:4]
+-------------+
| 31 30 39 38 | ⇔ %ebx guardado          (%ebp-0xc)
+-------------+
| 35 34 33 32 | ⇔ %esi guardado          (%ebp-0x8)
+-------------+
| 39 38 37 36 | ⇔ %edi guardado          (%ebp-0x4)
+-------------+
| 33 32 31 30 | ⇔ %ebp guardado          (%ebp)
+-------------|
| 08 04 86 00 | ⇔ endereço de retorno    (%ebp+0x4)
+-------------+

+---------------------------+
| M[A+3] M[A+2] M[A+1] M[A] | EXPLICAÇÃO DUMA LINHA
+---------------------------+
```

A azul e vermelho está a string "01234567890123456789012 3" com um '\0' a terminar

De notar que o código ASCII 0x30 equivale a '0', 0x31 a '1' ... 0x39 a '9' e 0x00 é o código do carater terminador '\0' (NULL).

c) Aquando da falha de segmentação para que endereço o programa tenta retornar?

O programa está a tentar retornar para o endereço **0x08048600**. O byte menos significativo foi reescrito com o carater terminador '\0' (00).

d) Que registo(s) têm o valor(s) corrompido) quando `getline` retorna ?

Registo	Valor
`%ebp`	0x33323130
`%edi`	0x39383736
`%esi`	0x35343332
`%ebx`	0x31303938

Estes valores serão carregados para registos antes do retorno da função `getline`.

e) Além do potencial de "*buffer overflow*", que outras duas coisas estão erradas no código C de `getline`?

O argumento da função *`malloc`* deveria ter sido "`strlen(buf)+1`".
Deveria ter sido feito uma verificação para saber se o valor devolvido pelo `malloc` é NULL.

Exercício 2

No trecho de código de montagem abaixo, resultante da compilação da função `loop_while`, o `gcc` faz uma transformação interessante que na prática introduz uma nova variável no programa.

```
int loop_while(int a, int b)
{
      int result = 1;
      while (a < b) {
            result *= (a+b);
            a++;}
      return result;
}
```

```
1  movl  8(%ebp), %ecx
2  movl  12(%ebp), %ebx
3  movl  $1, %eax
4  cmpl  %ebx, %ecx
5  jge   .L11
6  leal  (%ebx,%ecx), %edx
7  movl  $1, %eax
8  .L12:
9  imull %edx, %eax
10 addl  $1, %ecx
11 addl  $1, %edx
12 cmpl  %ebx, %ecx
13 jl    .L12
14 .L11:
```

a) Considerando que o registo `%edx` é iniciado na linha 6 e atualizado na linha 11 como se fosse uma nova variável do programa, mostre como esta se relaciona com as variáveis no código C original.

Como podemos verificar o registo %edx é inicializado com a expressão "a+b" (na linha 6) e incrementado em cada iteração do ciclo (linha 11), tal como, o valor da variável 'a' (armazenada no registo `%ecx`). Podemos, portanto, concluir que o valor no registo `%edx` será sempre igual à expressão "a + b". Denominamos tal valor de **amb** (a mais b).

b) Crie uma tabela de uso de registos para esta função.

Registo	Variável	Atribuição inicial
`%ecx`	a	*a*
`%ebx`	b	*b*
`%eax`	result	1
`%edx`	"amb"	*a + b*

c) Anote o código de montagem para explicar o seu funcionamento.

Argumentos: **a** em %ebp+8 e **b** em %ebp+12.

```
1  movl  8(%ebp), %ecx      ; ecx = 1º argumento = a
2  movl  12(%ebp), %ebx     ; ebx = 2º argumento = b
3  movl  $1, %eax           ; eax (= result) = 1
4  cmpl  %ebx, %ecx         ; a-b -> afeta flags
5  jge   .L11               ; se a>=b salta para fim ciclo (.L11)
6  leal  (%ebx,%ecx), %edx  ; edx=a+b ⇔ edx=amb
7  movl  $1, %eax           ; eax (= result) =1
8  .L12:
9  imull %edx, %eax         ; result *= amb
10 addl  $1, %ecx           ; a++
11 addl  $1, %edx           ; amb++
12 cmpl  %ebx, %ecx         ; a-b -> afeta flags
13 jl    .L12               ; se a<b salta para início ciclo (.L12)
14 .L11:
```

d) Usando o servidor *sc.di.uminho.pt* compile com nível de otimização **-O2** a mesma função. Compare o código produzido com o apresentado acima.

```
$ gcc -O2 -S loop_while.c
```

```
loop_while:
    pushl   %ebp
    movl    %esp, %ebp
1   movl    8(%ebp), %edx       ; edx = a
2   movl    12(%ebp), %ecx      ; ecx = b
3   cmpl    %ecx, %edx          ; a-b -> afeta as flags
4   pushl   %ebx                ; salvaguarda ebx na pilha
5   movl    $1, %ebx            ; ebx <=> result = 1
6   jge     .L7                 ; if a >= b salta para .L7
7   .L5:                        ; início do ciclo
8   leal    (%ecx, %edx), %eax  ; %eax = a+b (amb = a+b)
10  incl    %edx                ; a++
11  imull   %eax, %ebx          ; result *= amb
12  cmpl    %ecx, %edx          ; a-b -> afeta as flags
13  jl      .L5                 ; se a<b salta para início do ciclo
14  .L7:
    movl    %ebx, %eax          ; coloca retorno em eax
    popl    %ebx                ; repõe ebx
    leave
    ret
```

Ao analisar o código máquina compilado com -O2, verificamos que, para além dos registos guardarem valores diferentes:

i) A expressão a + b é calculada em todas as iterações do ciclo (*leal (%ecx,%edx), %eax*), em vez de ser calculada antes do ciclo e mantida em registo. Por esta razão, no código máquina gerado só existe uma instrução que realiza o incremento, enquanto antes existiam duas (*addl $1, %ecx e addl $1, %edx*).

ii) O valor do registo %ebx é guardado em memória (*pushl %ebx*) para que mais tarde possa ser recuperado, esta opção está relacionada com a convenção de salvaguarda de registos (*callee-saved/caller-saved*). %ebx é um registo *callee-saved*.

Exercício 3

Nos excertos de código binário desmontado, alguma informação foi substituída por **Xs**.

a) Qual é o alvo da instrução **je** abaixo (não é necessário conhecer nada acerca da instrução **call**)

```
804828f: 74 05             je   XXXXXXX
8048291: e8 1e 00 00 00    call 80482b4
```

A instrução **je** tem como endereço-alvo um valor relativo ao EIP (depois de ele ter sido incrementado para apontar para a próxima instrução):

```
XXXXXXX = 0x804828f + 2 + 0x05 = 0x8048296
```

```
    804828f:  74 05              je     8048296
    8048291:  e8 1e 00 00 00     call   80482b4
```

b) Qual é o alvo da instrução **jb** abaixo?

```
8048357: 72 e7                  jb    XXXXXXX
8048359: c6 05 10 a0 04 08 01   movb $0x1,0x804a010
```

O endereço alvo da instrução **jb** pode ser calculado de 2 modos:

- 0x8048359 – 0x19 (complemento para 2 de 0xE7) ou
- 0x8048359 + 0xfffffe7 (extensão do byte 0xe7 para 32 bits, e desprezando no resultado da adição, o bit de transporte).

```
            1111 1111 1111 1111 1111 1111 1111 111
  08048359 =  0000 1000 0000 0100 1000 0011 0101 1001
+ fffffe7  = +1111 1111 1111 1111 1111 1111 1110 0111
----------    ---------------------------------------
            0000 1000 0000 0100 1000 0011 0100 0000 = 08048340
```

```
      8048357:   72 e7                   jb    8048340
      8048359:   c6 05 10 a0 04 08 01    movb  $0x1,0x804a010
```

c) Qual é o endereço da instrução **mov**?

```
XXXXXXX1: 74 12              je 8048391
XXXXXXX2: b8 00 00 00 00     mov $ 0x0,% eax
```

Podemos verificar pelo código de montagem produzido, que o alvo da instrução **je** é o endereço absoluto 0x8048391. Sabendo o que o alvo é calculado utilizando o endereço da instrução seguinte, podemos deduzir o mesmo subtraindo ao endereço absoluto o número de bytes presente no byte de codificação (0x12 bytes). Portanto, 0x8048391 – 0x12 = 0x804837f. Por fim deduzimos o tamanho da instrução **jb** e temos o seu endereço 0x804837f – 0x2 = 0x804837d.

XXXXXXX2+0x12=0x8048391 <=> XXXXXXX2 = 0x8048391-0x12 = **0x804837F**

```
      804837d:      74 12               je 8048391
      804837f:      b8 00 00 00 00      mov $ 0x0,%eax
```

d) Qual é o endereço alvo do salto?

```
80482bf: e9 e0 ff ff ff      jmp XXXXXXX
80482c4: 90                  nop
```

Lendo por ordem inversa (*little endian*), podemos ver que o endereço alvo do salto é 0xfffffe0. Somando isto a 0x80482c4 (o endereço da instrução `nop`) dá o endereço 0x80482a4

0x80482c4 + 0xfffffe0 = 0x80482c4 – 0x20= **0x80482a4**

```
      80482bf:      e9 e0 ff ff ff    jmp 80482a4
      80482c4:      90                nop
```

e) Explique a relação entre o código de montagem (à direita) e o código máquina (à esquerda).

```
80482aa: ff 25 fc 9f 04 08    jmp * 0x8049ffc
```

Um salto indireto (indicado pelo '*') é codificado com **`ff 25`**. O endereço alvo do **`jmp`** a ser lido é codificado explicitamente pelos 4 bytes seguintes. Uma vez que a arquitetura IA32 é *little endian*, os 4 bytes do endereço 0x8049ffc surgem por ordem inversa, ou seja, **`fc 9f 04 08`**.

Exercício 4

Considere o código C abaixo, onde **M** e **N** são constantes declaradas com `#define`.

```
#define M ??
#define N ??
int mat1[M][N];
int mat2[N][M];
int sum_element(int i, int j){
     return mat1[i][j] + mat2[j][i];
}
```

Use engenharia reversa para determinar os valores de **M** e **N** com base no código de montagem gerado pelo gcc:

```
1 movl 8(%ebp), %ecx
2 leal 0(,%ecx,8), %edx
3 movl 12(%ebp), %eax
4 subl %ecx, %edx
5 addl %eax, %edx
6 leal (%eax,%eax,4), %eax
7 addl %ecx, %eax
8 movl mat2(,%eax,4), %eax
9 addl mat1(,%edx,4), %eax
```

Para aceder ao elemento `(i, j)` de um vetor multidimensional **`tipoV V[R][C]`**, em que **`R`**=número de linhas e **`C`**=número de colunas, utilizamos a seguinte expressão:

`&V[i][j] = xV + L*(C * i + j)`

onde **`L`** é o tamanho do tipo de dados **`tipoV`** em bytes e **x_V** é o endereço base do vetor.

```
1 movl  8(%ebp), %ecx          ; copia 1º arg para ecx <=> ecx=i
2 leal  0(,%ecx,8),%edx        ; edx = 8*i
3 movl  12(%ebp), %eax         ; copia 2º arg para eax <=> eax=j
4 subl  %ecx, %edx             ; edx = 8*i - i = 7*i
5 addl  %eax, %edx             ; edx = 7*i + j
6 leal  (%eax,%eax,4), %eax    ; eax = 5*j
7 addl  %ecx, %eax             ; eax = 5*j + i
8 movl  mat2(,%eax,4), %eax    ; mat2+4*(5*j+i) = mat2[5*j+i]
9 addl  mat1(,%edx,4), %eax    ; mat1+4*(7*i+j) = mat1[7*i+j]
```

Podemos ver que o deslocamento em bytes do elemento (i,j) da matriz **mat1** é 4*(7*i+j), enquanto o deslocamento em bytes do elemento (j,i) da matriz **mat2** é 4*(5*j+i). Podemos determinar, portanto, que a matriz **mat1** tem 7 colunas (**N=7**) e a matriz **mat2** tem 5 colunas (**M = 5**).