Assembly do IA-32 em ambiente Linux

Exercícios de Programação 3

(Adaptados do livro de Randal E. Bryant and David R. O'Hallaron)
Alberto J. Proença e António M. Pina

Prazos

Entrega **impreterível**: terça 03-Dez-01, no Lab. da disciplina (não serão aceites trabalhos entregues depois deste prazo).

Para mais informações, ler metodologia definida no enunciado dos Exercícios de Programação 1.

Introdução

A lista de exercícios que se apresenta estão directamente relacionados com a codificação de funções em *assembly* do IA-32 (segue o material apresentado nas aulas teóricas e práticas da semana 9 (ver sumários na página da disciplina na Web), requerendo os conceitos básicos adquiridos em aulas anteriores.

Exercício 3.1 (*Transferência de controlo na invocação de funções*):

O seguinte fragmento de código aparece frequentemente em versões compiladas de rotinas de biblioteca:

```
1    call   next
2 next:
3    popl   %eax
```

a) Que valor é colocado no registo `%eax`?

b) Explique a não existência da correspondente instrução `ret` nesta rotina.

c) Qual a finalidade (e utilidade) deste fragmento de código?

Exercício 3.2 (*Convenção*):

A seguinte sequência de código aparece quase no início do código *assembly* gerado por GCC para uma função C:

```
1    pushl  %edi
2    pushl  %esi
3    pushl  %ebx
4    movl   24(%ebp),%eax
5    imull  16(%ebp),%eax
6    movl   24(%ebp),%ebx
7    leal   0(,%eax,4),%ecx
8    addl   8(%ebp),%ecx
9    movl   %ebx,%edx
```

Pode-se ver que 3 registos (`%edi, %esi`, e `%ebx`) são salvaguardados na *stack*. O programa altera no seu corpo não apenas estes, mas outros 3 registos (`%eax, %ecx`, e `%edx`). No fim da função, o valor dos registos `%edi, %esi`, e `%ebx` são recuperados usando a instrução `popl`, enquanto os outros 3 são deixados com os seus valores entretanto modificados.
Explique esta aparente inconsistência na salvaguarda e recuperação de registos na codificação de funções.

Exercício 3.3 (*Funções*):

Dada a seguinte função C:

```
1 int proc(void)
2 {
3    int x,y;
4    scanf("%x %x", &y, &x);
5    return x-y;
6 }
```

GCC gera o seguinte código *assembly*

```
1 proc:
2    pushl  %ebp
3    movl   %esp,%ebp
4    subl   $24,%esp
5    addl   $-4,%esp
6    leal   -4(%ebp),%eax
7    pushl  %eax
8    leal   -8(%ebp),%eax
9    pushl  %eax
10   pushl  $.LC0                Pointer to string "%x %x"
11   call   scanf
```

Faça o diagrama da stack frame nesta altura

```
12   movl   -8(%ebp),%eax
13   movl   -4(%ebp),%edx
14   subl   %eax,%edx
15   movl   %edx,%eax
16   movl   %ebp,%esp
17   opl    %ebp
18   ret
```

Considere que a função `proc` começa a sua execução com os seguintes valores nos registos:

Register	Value
`%esp`	`0x800040`
`%ebp`	`0x800060`

Considere que a função `proc` invoca `scanf` (linha 12), e que `scanf` lê os valores `0x46` e `0x53` da *standard input*. Considere ainda que a *string* "%x %x" se encontra armazenada na memória em `0x300070`.

a) Que valor é colocado no registo `%ebp` na linha 3?

b) Em que endereços estão as variáveis locais `x` e `y` armazenadas?

c) Qual é o valor de `%esp` na linha 11?

d) Desenhe um diagrama da *stack frame* para `proc` logo após o regresso de `scanf`. Inclua o máximo de informação que possua sobre endereços e conteúdos dos elementos da *stack frame*.

e) Indique as regiões da *stack frame* que não são usadas por `proc` (estas áreas desperdiçadas são usadas para melhorar o desempenho da *cache*).

Exercício 3.4 (*Buffer overflow*):

O seguinte código C mostra uma implementação (de baixa qualidade) de uma função que lê uma linha da *standard input*, copia a *string* lida para uma novo local de memória, e devolve um apontador para o resultado.

```
1 /* This is very low quality code.
2    It is intended to illustrate bad programming practices. */
3 char *getline()
4 {
5    char buf[8];
6    char *result;
7    gets(buf);
8    result = malloc(strlen(buf));
9    strcpy(result, buf);
10   return(result);
11 }
```

O programa executável "desmontado" (*disassembly*) até à chamada da função `gets` é o seguinte:

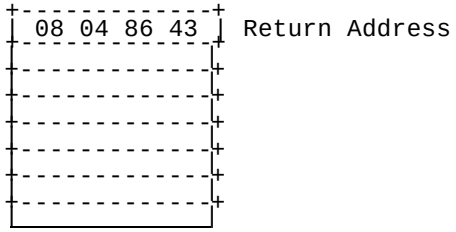
```
1   08048524 <getline>:
2    8048524: 55 push %ebp
3    8048525: 89 e5 mov %esp,%ebp
4    8048527: 83 ec 10 sub $0x10,%esp
5    804852a: 56 push %esi
6    804852b: 53 push %ebx
```

Faça o diagrama da stack frame nesta altura

```
7    804852c: 83 c4 f4 add $0xfffffff4,%esp
8    804852f: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
9    8048532: 53 push %ebx
10   8048533: e8 74 fe ff ff call 80483ac <_init+0x50>          gets
```

Considere o seguinte cenário: a função `getline` é invocada com o endereço de retorno `0x8048643`, o registo `%ebp` com o valor `0xbffffc94`, o registo `%esi` com `0x1`, e o registo `%ebx` com `0x2`; introduz a *string* "`012345678901`" .O programa termina com *segmentation fault*.
Utilize o *debugger* (GDB) para detectar o local onde ocorreu a anomalia (Nota: deverá chegar à conclusão que tal aconteceu na execução da instrução `ret` da função `getline`).

a) Considerando que a *stack* "cresce para baixo", preencha o diagrama (da *stack frame*) com o máximo de indicações, logo após execução da instrução da linha 6 (no código desmontado). Coloque em cada caixa (que representa 4 *bytes*) o respectivo valor em hexadecimal (se conhecido), e à direita uma etiqueta que ajude a esclarecer o conteúdo da *stack* (e.g., "Return Address"). Indique a posição de `%ebp`.



b) Modifique o diagrama para mostrar os valores após a invocação da função `gets` (linha 10).

c) Para que endereço tenta o programa regressar?

d) Que registo(s) foi(oram) corrompido(s) no regresso da função `getline` e como?

e) Para além do problema de *buffer overflow*, que duas outras coisas estão erradas no código de `getline`?

Nº _____ Nome _____

Exercício 3.1 (*Transferência de controlo na invocação de funções)***:**

a)

b)

c)

Exercício 3.2 (*Convenção*):

Exercício 3.3 (*Funções*):

a)

b)

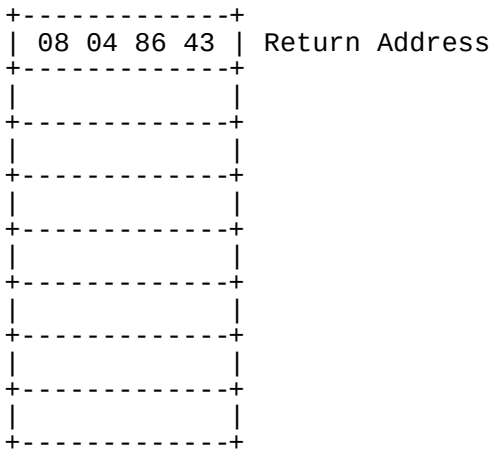
c)

d)

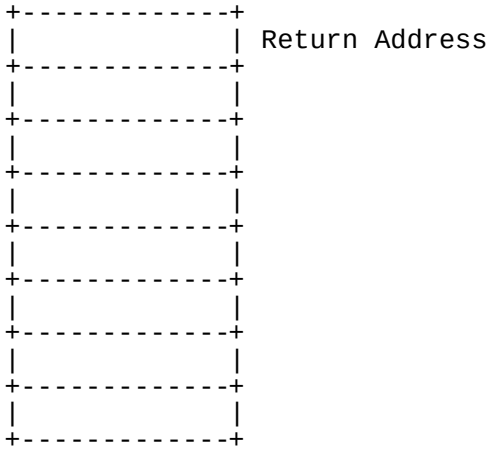
Nº _____ Nome _____

Exercício 3.4 (*Buffer overflow*):

a)



b) **c)**



d)

e)