

Guião: G-VIII

Exercícios adaptados do livro CSPP
Randal E. Bryant e David R. O'Hallaron

Apresentação

Este guião tem em vista abordar os temas relacionadas com a geração e execução de código de montagem produzido pelo compilador gcc, para a arquitectura IA32.

Exercício 1.

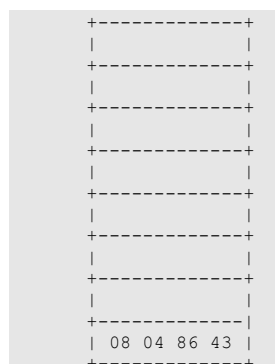
A função `getline` lê uma linha da entrada, faz uma cópia da sequência de caracteres para o espaço reservado na pilha e retorna um apontador para o resultado.

```
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 correspondente à sequência "012345678901234567890123" 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 80485c0: 55                push    %ebp
3 80485c1: 89 e5            mov     %esp,%ebp
4 80485c3: 83 ec 28         sub     $0x28,%esp
5 80485c6: 89 5d f4         mov     %ebx,-0xc(%ebp)
6 80485c9: 89 75 f8         mov     %esi,-0x8(%ebp)
7 80485cc: 89 7d fc         mov     %edi,-0x4(%ebp)
   Diagram da pilha neste ponto
8 80485cf: 8d 75 ec         lea     -0x14(%ebp),%esi
9 80485d2: 89 34 24         mov     %esi,(%esp)
10 80485d5: e8 a3 ff ff ff  call    804857d <gets>
   Modificar o diagrama da pilha neste ponto
```

- Preencha o diagrama que se segue da pilha (cada espaço representa 4 octetos) indicando a posição de `%ebp` com toda a informação disponível após a execução da instrução na linha 7 no código de montagem: valores hexadecimais (se conhecidos) dentro da caixa e identificação dos mesmo (por exemplo, "endereço de retorno") do lado direito.
- Modificar o diagrama para mostrar o efeito da chamada a `gets` (linha 10).
- Aquando da falha de segmentação para que endereço o programa tenta retornar?
- Que registo(s) têm o valor(s) corrompido) quando `getline` retorna?
- Além do potencial de "buffer overflow", que outras duas coisas estão erradas no código de `getline`?



Exercício 2.

No trecho de código montado, 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:
```

- 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.
- Crie uma tabela de uso de registos para esta função.
- Anote o código de montagem para explicar o seu funcionamento.
- Usando o servidor *sc.di.uminho.pt* compile com nível de otimização `O2` a mesma função. Compare o código produzido com o apresentado acima.

Exercício 3.

Nos excertos de código binário desmontado, algumas das informações foi substituído por Xs.

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

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

- Qual é o alvo da instrução `jb` abaixo?

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

- Qual é o endereço da instrução `mov`?

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

- Qual é o endereço alvo do salto?

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

- Explique a relação entre a anotação na direita e a codificação do octeto à esquerda.

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

Exercício 4.

Considere o código C, abaixo, onde `M` e `N` são constantes declarado 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];
}
```

- a) 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
```