Insper

09 - Prática de Engenharia Reversa

Sistemas Hardware-Software - 2020/1

Igor Montagner

As aulas de hoje e quinta serão dedicadas a revisar conceitos básicos vistos nas últimas 5 aulas. A parte 1 repassa chamadas de funções e o uso da instrução `LEA`. A parte 2 contém exercícios intermediários que misturam dois conceitos vistos em aula (ex: loops e `LEA` ou funções + condicionais). Está indicado ao lado de cada exercício quais conceitos são exercitados e todos eles serão recebidos via repositório de atividades da disciplina.

Atenção: acesse a aba "Repositório de Atividades" no Teams para indicar seu usuário do blackboard e endereço do repositório privado no Github.

Todo exercício para entrega deverá ser colocado no repositório de atividades da disciplina.

Parte 1 - Revisão de funções e `LEA`

Todos os exercícios da revisão serão feitos com o arquivo `exemplo1` (compilado a partir de `exemplo1.c`).

Chamadas de funções

As chamadas de função são feitas usando a seguinte ordem para os argumentos inteiros:

1. `%rdi`
2. `%rsi`
3. `%rdx`
4. `%rcx`
5. `%r8`
6. `%r9`

Esta ordem nunca muda. Veja abaixo um exemplo de chamada de função.

```
0x065c <+4>:     mov    $0x6,%r9d
0x0662 <+10>:    mov    $0x5,%r8d
0x0668 <+16>:    mov    $0x4,%ecx
0x066d <+21>:    mov    $0x3,%edx
0x0672 <+26>:    mov    $0x2,%esi
0x0677 <+31>:    mov    $0x1,%edi
0x067c <+36>:    callq  0x64a <exemplo1>
0x0681 <+41>:    lea    0xa(%rax),%esi
```

Exercício: Traduza esta invocação de função para *C*:

Vamos agora analisar o código de `exemplo1`:

```
Dump of assembler code for function exemplo1:
   0x064a <+0>:     add    %esi,%edi
   0x064c <+2>:     add    %edi,%edx
   0x064e <+4>:     add    %edx,%ecx
   0x0650 <+6>:     add    %r8d,%ecx
   0x0653 <+9>:     lea    (%rcx,%r9,1),%eax
   0x0657 <+13>:    retq
```

Vemos na linha `exemplo1+9` que colocamos um valor no registrador `%eax` e depois finalizamos a função usando `retq`. Este é o segundo ponto que nunca muda: **o valor de retorno de toda função é colocado no registrador** `%rax` (ou uma de suas partes menores). A instrução usada é o `LEA` que veremos na seção a seguir.

Operações aritméticas usando `LEA`

Se usada de maneira literal, a instrução LEA (**L**oad **E**ffective **A**ddress) serve para calcular o endereço de uma variável local e é equivalente ao operador `&` em *C*. Porém, ela é frequentemente "abusada" para fazer aritmética. Um ponto importante quando usamos `LEA` é que todos os operandos são registradores de `64` bits.

Regra geral:

1. Se `LEA` for usada com o registrador `%rsp` então ela sempre representa o operador `&`
2. Se os registradores envolvidos foram usados como números inteiros em instruções anteriores, então ela representa uma conta com os valores dos registradores.

Vejamos o exemplo da função `exemplo1` acima:

```
   0x0653 <+9>:     lea    (%rcx,%r9,1),%eax
```

No exemplo acima `LEA` é usada para fazer aritmética. Sabemos disso pois, na chamada traduzida na parte anterior, elas recebem números inteiros (`%r9d = 6` e `%ecx = 4`). Seu primeiro argumento segue a seguinte lógica

`C(%R1, %R2, S)`

- `C` é uma constante
- `%R1` é um registrador
- `%R2` é um registrador (pode ser igual a `%R1`)
- `S` é `1, 2, 4` ou `8` (todos os tamanhos possíveis de registradores inteiros)

A operação acima calcula `C + %R1 + (%R2 * S)`. A operação `LEA` **nunca acessa a memória**, apenas move o resultado deste cálculo para o registrador destino. **Qualquer outra operação que use a sintaxa acima está fazendo um acesso a memória.** `LEA` **é a única exceção!**

Exercício: traduza a operação abaixo para *C*

```
   0x0653 <+9>:     lea    (%rcx,%r9,1),%eax
```

Exercício: Com estas informações em mãos, traduza `exemplo1` para *C*

```
Dump of assembler code for function exemplo1:
   0x064a <+0>:     add    %esi,%edi
   0x064c <+2>:     add    %edi,%edx
   0x064e <+4>:     add    %edx,%ecx
   0x0650 <+6>:     add    %r8d,%ecx
   0x0653 <+9>:     lea    (%rcx,%r9,1),%eax
   0x0657 <+13>:    retq
```

Retorno de funções

Vamos terminar nossa revisão analisando novamente a chamada de `exemplo1` no `main`:

```
   0x065c <+4>:     mov    $0x6,%r9d
   0x0662 <+10>:    mov    $0x5,%r8d
   0x0668 <+16>:    mov    $0x4,%ecx
   0x066d <+21>:    mov    $0x3,%edx
   0x0672 <+26>:    mov    $0x2,%esi
   0x0677 <+31>:    mov    $0x1,%edi
   0x067c <+36>:    callq  0x64a <exemplo1>
   0x0681 <+41>:    lea    0xa(%rax),%esi
```

Anteriormente já vimos que o `call` e os `mov` s acima fazem a chamada `exemplo1(1,2,3,4,5,6)` em *C*. A linha de baixo realiza uma operação aritmética com `%rax`.

Exercício: considerando que `%rax` armazena o valor de retorno de uma função, qual seria a tradução para *C* do bloco de código acima?

Parte 2 - exercícios intermediários

Os exercícios desta seção exercitam dois conceitos ao mesmo tempo. Cada um deles é disponibilizado via um arquivo `exI.o` na pasta `09-revisao` no repositório de atividades. A soluções devem ser colocadas no arquivo `solucao_exI.c` correspondente. Veja as instruções em cada arquivo para garantir que está implementando a função correta.

Importante: cada exercício estará disponível em uma página do handout de revisão juntamente com questões "padrão" para cada assunto. Essas questões são feitas para ajudar na compreensão dos programas. Faça-as com atenção e facilite sua vida.

Exercício: As função abaixo exercita os assuntos **Aritmética** e **Expressões booleanas**.

```
Dump of assembler code for function ex1:
   0x05fa <+0>:     lea    (%rdi,%rsi,1),%rax
   0x05fe <+4>:     lea    (%rax,%rdx,4),%rcx
   0x0602 <+8>:     imul   %rdi,%rdi
   0x0606 <+12>:    lea    (%rdi,%rsi,2),%rax
   0x060a <+16>:    add    %rax,%rdx
   0x060d <+19>:    cmp    %rdx,%rcx
   0x0610 <+22>:    setge  %al
   0x0613 <+25>:    movzbl %al,%eax
   0x0616 <+28>:    retq
```

1. Quantos argumentos a função acima recebe? Quais seus tipos? Declare a função abaixo.

2. As instruções `LEA` acima representam operações aritméticas ou a operação *endereço de* `&`? Como você fez esta identificação? .

3. Traduza as operações das linhas `ex1+0` até `ex1+15` para *C*

4. Nas linhas `ex1+18` e `ex1+21` é feita uma comparação. Qual e entre quais registradores? Onde é armazenado este resultado?

5. O quê faz a instrução `movzbl` em `ex1+24`? Juntando com a resposta da pergunta acima, traduza as instruções `ex1+18` até `ex1+27` para *C*.

Usando as perguntas acima preencha o arquivo de solução no repositório e execute os testes.

Exercício: O exercício abaixo exercita **Chamadas de funções** e **Condicionais**.

```
Dump of assembler code for function ex2:
   0x05ff <+0>:     push   %rbx
   0x0600 <+1>:     mov    %rdi,%rbx
   0x0603 <+4>:     mov    %rsi,%rdi
   0x0606 <+7>:     callq  0x5fa <vezes2>
   0x060b <+12>:    cmp    %rbx,%rax
   0x060e <+15>:    jle    0x613 <ex2+20>
   0x0610 <+17>:    add    %rbx,%rbx
   0x0613 <+20>:    add    %rbx,%rax
   0x0616 <+23>:    pop    %rbx
   0x0617 <+24>:    retq
```

1. Quantos argumentos a função acima recebe? Quais são seus tipos? Declare-a abaixo.

Vamos começar trabalhando na linha `ex2+7`, na instrução `call vezes2` . A chamada necessita usar o registrador `%rdi`, mas ele contém o primeiro argumento de `ex2`.

1. Em qual registrador é guardado o primeiro argumento de `ex2`? Isso é feito antes da chamada `call`.

2. Qual variável é passada como argumento para a função `vezes2` ?

3. Escreva abaixo a invocação de `vezes2`.

Você deve ter notado as instruções `push/pop %rbx` no começo/fim da função. Toda função pode usar os registradores de argumentos (vistos na parte 1) e o de valor de retorno como quiserem. Se precisarem mexer nos outros registradores a prática é salvá-los na pilha no começo da função e restaurá-los no fim. Assim não importa o que a função faça, para a função chamadora é como se não houvesse havido nenhuma modificação nos outros registradores.

Vamos agora olhar a condicional na linha `ex2+12`.

1. Após a chamada `call`, qual o conteúdo de `%rax`?

2. Juntando suas respostas nas questões de cima, qual é a comparação feita nas linhas `ex2+12, ex2+17` ?

3. Com essas informações em mãos, faça uma tradução do código acima para *C* usando somente `if+goto`.

Usando as perguntas acima preencha o arquivo de solução no repositório e execute os testes.

Exercício: O exercício abaixo exercita **Ponteiros** e **Expressões booleanas**.

```
Dump of assembler code for function ex3:
   0x05fa <+0>:     cmp    %rsi,%rdi
   0x05fd <+3>:     setl   %al
   0x0600 <+6>:     movzbl %al,%eax
   0x0603 <+9>:     mov    %eax,(%rdx)
   0x0605 <+11>:    cmp    %rsi,%rdi
   0x0608 <+14>:    sete   %al
   0x060b <+17>:    movzbl %al,%eax
   0x060e <+20>:    mov    %eax,(%rcx)
   0x0610 <+22>:    cmp    %rsi,%rdi
   0x0613 <+25>:    setg   %al
   0x0616 <+28>:    movzbl %al,%eax
   0x0619 <+31>:    mov    %eax,(%r8)
   0x061c <+34>:    retq
```

1. Quantos argumentos a função acima recebe? De quais tipos? Declare-a abaixo.

2. A função acima faz várias comparações. Liste quais e entre quais argumentos.

3. Onde é armazenado o resultado de cada comparação?

4. Com base em suas respostas acima, faça uma tradução linha a linha da função acima.

Usando as perguntas acima preencha o arquivo de solução no repositório e execute os testes.

Exercício: O exercício abaixo exercita **Chamadas de funções** e **Loops**.

```
Dump of assembler code for function ex4:
   0x05ff <+0>:     push   %rbx
   0x0600 <+1>:     mov    %rdi,%rbx
   0x0603 <+4>:     mov    $0x0,%eax
   0x0608 <+9>:     jmp    0x612 <ex4+19>
   0x060a <+11>:    mov    %rax,%rdi
   0x060d <+14>:    callq  0x5fa <mais_um>
   0x0612 <+19>:    cmp    %rbx,%rax
   0x0615 <+22>:    jb     0x60a <ex4+11>
   0x0617 <+24>:    pop    %rbx
   0x0618 <+25>:    retq
```

1. Quantos argumentos a função acima recebe? Quais seus tipos? E o valor de retorno? Declare a função abaixo.

2. A função acima tem um loop. Entre quais instruções? Use setas para identificá-lo.

3. É feita uma chamada para `mais_um`. Qual o argumento passado? Onde seu resultado é usado?

4. Faça uma tradução linha a linha da função acima usando somente `if+goto`

Usando as perguntas acima preencha o arquivo de solução no repositório e execute os testes.