# Architektury systemów komputerowych

Lista zadań nr 6

Na zajęcia 15 kwietnia 2021

Należy być przygotowanym do wyjaśnienia semantyki każdej instrukcji, która pojawia się w treści zadania. W tym celu posłuż się dokumentacją: x86 and amd64 instruction reference[1]. W szczególności trzeba wiedzieć jak dana instrukcja korzysta z rejestru flag «EFLAGS» tam, gdzie obliczenia zależą od jego wartości.

W trakcie tłumaczeniu kodu z asemblera x86-64 do języka C należy trzymać się następujących wytycznych:

- Używaj złożonych wyrażeń minimalizując liczbę zmiennych tymczasowych.
- Nazwy wprowadzonych zmiennych muszą opisywać ich zastosowanie, np. `result` zamiast `rax`.
- Instrukcja `goto` jest zabroniona. Należy używać instrukcji sterowania `if`, `for`, `while` i `switch`.
- Pętle «`while`» należy przetłumaczyć do pętli «`for`», jeśli poprawia to czytelność kodu.

**Uwaga!** Przedstawienie rozwiązania niestosującego się do powyższych zasad może skutkować negatywnymi konsekwencjami.

**Zadanie 1.** Poniższy wydruk otrzymano w wyniku deasemblacji rekurencyjnej procedury zadeklarowanej następująco: «`long puzzle(long n, long *p)`». Zapisz w języku C kod odpowiadający tej procedurze. Następnie opisz zawartość jej **rekordu aktywacji** (ang. *stack frame*). Wskaż **rejestry zapisane przez funkcję wołaną** (ang. *callee-saved registers*), zmienne lokalne i adres powrotu.

```
1  puzzle:
2          push  %rbp
3          xorl  %eax, %eax
4          mov   %rsi, %rbp
5          push  %rbx
6          mov   %rdi, %rbx
7          sub   $24, %rsp
8          test  %rdi, %rdi
9          jle   .L1
10         lea   8(%rsp), %rsi
11         lea   (%rdi,%rdi), %rdi
12         call  puzzle
13         add   8(%rsp), %rax
14         add   %rax, %rbx
15 .L1:    mov   %rbx, (%rbp)
16         add   $24, %rsp
17         pop   %rbx
18         pop   %rbp
19         ret
```

**Uwaga!** Wskaźnik wierzchołka stosu w momencie wołania procedury musi być wyrównany do adresu podzielnego przez 16.

**Zadanie 2.** Poniżej zamieszczono kod procedury o sygnaturze «`struct T puzzle8(long *a, long n)`». Na jego podstawie podaj definicję typu «`struct T`». Przetłumacz tę procedurę na język C, po czym jednym zdaniem powiedz co ona robi. Gdyby sygnatura procedury nie była wcześniej znana to jaką należałoby wywnioskować z poniższego kodu?

```
1  puzzle8:
2        movq  %rdx, %r11
3        xorl  %r10d, %r10d
4        xorl  %eax, %eax
5        movq  $LONG_MIN, %r8
6        movq  $LONG_MAX, %r9
7  .L2:  cmpq  %r11, %r10
8        jge .L5
9        movq  (%rsi,%r10,8), %rcx
10       cmpq  %rcx, %r9
11       cmovg %rcx, %r9
12       cmpq  %rcx, %r8
13       cmovl %rcx, %r8
14       addq  %rcx, %rax
15       incq  %r10
16       jmp .L2
17 .L5:  cqto
18       movq  %r9, (%rdi)
19       idivq %r11
20       movq  %r8, 8(%rdi)
21       movq  %rax, 16(%rdi)
22       movq  %rdi, %rax
23       ret
```

**Wskazówka:** Zauważ, że wynik procedury nie mieści się w rejestrach %rax i %rdx, zatem zostanie umieszczony w pamięci.

**Zadanie 3.** Zakładamy, że producent procesora nie dostarczył instrukcji **skoku pośredniego**. Rozważmy procedurę «`switch_prob`» z poprzedniej listy. Podaj metodę zastąpienia «`jmpq *0x4006f8(,%rsi,8)`» ciągiem innych instrukcji. Nie można używać **kodu samomodyfikującego się** (ang. *self-modyfing code*), ale można użyć dodatkowego rejestru. Zastanów się czy potrafisz również zastąpić instrukcję **pośredniego wywołania procedury**, np. «`call *(%rdi,%rsi,8)`», gdyby jej brakowało.

[1] `http://www.felixcloutier.com/x86/`

**Zadanie 4.** Poniżej widnieje kod wzajemnie rekurencyjnych procedur «M» i «F» typu «long (*)(long)». Programista, który je napisał, nie pamiętał wszystkich zasad **konwecji wołania procedur**. Wskaż co najmniej dwa różne problemy w poniższym kodzie i napraw je!

```
1  M:   pushq   %rdi
2       testq   %rdi, %rdi
3       je      .L2
4       leaq    -1(%rdi), %rdi
5       call    M
6       movq    %rax, %rdi
7       call    F
8       movq    (%rsp), %rdi
9       subq    %rax, %rdi
10 .L2: movq    %rdi, %rax
11      ret
12 F:   testq   %rdi, %rdi
13      je      .L3
14      movq    %rdi, %r12
15      leaq    -1(%rdi), %rdi
16      call    F
17      movq    %rax, %rdi
18      call    M
19      subq    %rax, %r12
20      movq    %r12, %rax
21      ret
22 L3:  movl    $1, %eax
23      ret
```

**Zadanie 5.** Skompiluj poniższy kod źródłowy kompilatorem gcc z opcjami «-Og -fomit-frame-pointer -fno-stack-protector» i wykonaj deasemblację **jednostki translacji** przy użyciu programu «objdump». Wytłumacz co robi procedura alloca(3), a następnie wskaż w kodzie maszynowym instrukcje realizujące przydział i zwalnianie pamięci. Wyjaśnij co robi robi instrukcja «leave».

```
1  #include <alloca.h>
2
3  long aframe(long n, long idx, long *q) {
4    long i;
5    long **p = alloca(n * sizeof(long *));
6    p[n-1] = &i;
7    for (i = 0; i < n; i++)
8      p[i] = q;
9    return *p[idx];
10 }
```

**Zadanie 6.** Poniżej widnieje kod procedury o sygnaturze «long puzzle5(void)». Podaj rozmiar i składowe rekordu aktywacji procedury «puzzle5». Procedura «readlong», która wczytuje ze standardowego wejścia liczbę całkowitą, została zdefiniowana w innej jednostce translacji. Jaka jest jej sygnatura? Przetłumacz procedurę «puzzle5» na język C i wytłumacz jednym zdaniem co ona robi.

```
1  puzzle5:
2       subq    $24, %rsp
3       movq    %rsp, %rdi
4       call    readlong
5       leaq    8(%rsp), %rdi
6       call    readlong
7       movq    (%rsp), %rax
8       cqto
9       idivq   8(%rsp)
10      xorl    %eax, %eax
11      testq   %rdx, %rdx
12      sete    %al
13      addq    $24, %rsp
14      ret
```

**Zadanie 7.** Procedurę ze zmienną liczbą parametrów używającą pliku nagłówkowego stdarg.h[2] skompilowano z opcjami «-Og -mno-sse». Po jej deasemblacji otrzymano następujący wydruk. Co robi ta procedura i jaka jest jej sygnatura? Jakie dane są przechowywane w rekordzie aktywacji tej procedury? Prezentację zacznij od przedstawienia definicji struktury «va_list».

```
1  puzzle7:
2       movq  %rsi, -40(%rsp)
3       movq  %rdx, -32(%rsp)
4       movq  %rcx, -24(%rsp)
5       movq  %r8, -16(%rsp)
6       movq  %r9, -8(%rsp)
7       movl  $8, -72(%rsp)
8       leaq  8(%rsp), %rax
9       movq  %rax, -64(%rsp)
10      leaq  -48(%rsp), %rax
11      movq  %rax, -56(%rsp)
12      movl  $0, %eax
13      jmp .L2
14 .L3:  movq  -64(%rsp), %rdx
15      leaq  8(%rdx), %rcx
16      movq  %rcx, -64(%rsp)
17 .L4:  addq  (%rdx), %rax
18 .L2:  subq  $1, %rdi
19      js  .L6
20      cmpl  $47, -72(%rsp)
21      ja  .L3
22      movl  -72(%rsp), %edx
23      addq  -56(%rsp), %rdx
24      addl  $8, -72(%rsp)
25      jmp .L4
26 .L6:  ret
```

**Wskazówka:** Przeczytaj rozdział §3.5.7 dokumentu opisującego ABI dostępnego na stronie przedmiotu.

[2] https://en.wikipedia.org/wiki/Stdarg.h

**Zadanie 8 (bonus).** Kompilator GCC umożliwia tworzenie funkcji zagnieżdżonych w języku C. Skompiluj poniższy kod z opcją «-Os -fno-stack-protector», po czym zdeasembluj go poleceniem «objdump». Zauważ, że procedura «accumulate» korzysta ze zmiennej «res» należącej do rekordu aktywacji procedury «sum». Zatem w miejscu wywołania «accumulate» (wiersz 3) musimy dysponować wskaźnikiem na tą procedurę i jej środowisko. Wyjaśnij w jaki sposób kompilator przygotował procedurę «accumulate» do wołania w «for_range_do» oraz w jaki sposób «accumulate» otrzymuje dostęp do zmiennych ze środowiska «sum». Przy pomocy debuggera GDB i nakładki gdb-dashboard[3] zaprezentuj, instrukcja po instrukcji (polecenie «si»), co się dzieje w trakcie wywołania funkcji w wierszu 3.

```
1  static void for_range_do(long *cur, long *end, void (*fn)(long x)) {
2    while(cur < end)
3      fn(*cur++);
4  }
5
6  long sum(long *a, long n) {
7    long res = 0;
8
9    void accumulate(long x) {
10     res += x;
11   }
12
13   for_range_do(a, a + n, accumulate);
14   return res;
15 }
```

**Uwaga:** Użycie tej konstrukcji wymaga, by stos był wykonywalny, co może ułatwić hakerom przejęcie kontroli nad programem.

[3] https://github.com/cyrus-and/gdb-dashboard