CSO-Recitation 10

CSCI-UA 0201-007

R10: Assessment 08 & Buffer overflow & Compiler optimization

Today's Topics

- Assessment 08
- Breakout exercises
- Buffer overflow
- Compiler optimization

Assessment 08

Q1 Set_five

Given the following C function from Lab 1,

```
void set_five(int *p)
{
  *p = 5;
}

void test()
{
  int p = 0;
  set_five(&p);
}
```

The assembly for `set_five` function is:

```
0x00000000000005fa <+0>: ???
0x0000000000000600 <+6>: retq
```

The assembly for `test` function is:

```
0x0000000000000601 <+0>:     sub    $0x10,%rsp
0x0000000000000605 <+4>:     movl   $0x0,0xc(%rsp)
0x000000000000060d <+12>:    lea    0xc(%rsp),%rdi
0x0000000000000612 <+17>:    callq  0x5fa <set_five>
0x0000000000000617 <+22>:    add    $0x10,%rsp
0x000000000000061b <+26>:    retq
```

Q1.1 %rsp

Under normal program execution, suppose the value of %rsp is `0x7fff856001d8` **just prior to** executing the first instruction of `test`. What is the value of %rsp just prior to executing the first instruction of `set_five`?

A. 0x7fff856001d8
B. 0x7fff856001c8
C. 0x7fff856001e8
D. 0x7fff856001c0
E. 0x7fff856001d0
F. 0x7fff856001c4
G. 0x7fff856001cc
H. None of the above

- val(%rsp)= `0x7fff856001d8`
- sub $0x10, %rsp
  - val(%rsp)= `0x7fff856001c8`
- callq 0x5fa <set_five>
- callq label
  - DECREASES %rsp by 8
  - THEN stores the return address at the memory location given by the new %rsp
  - THEN jumps to the operand
- callq 0x5fa <set_five>
  - val(%rsp)= `0x7fff856001c0`

Given the following C function from Lab 1,

```
void set_five(int *p)
{
  *p = 5;
}

void test()
{
  int p = 0;
  set_five(&p);
}
```

The assembly for `set_five` function is:

```
0x00000000000005fa <+0>: ???
0x0000000000000600 <+6>: retq
```

The assembly for `test` function is:

```
0x0000000000000601 <+0>:     sub    $0x10,%rsp
0x0000000000000605 <+4>:     movl   $0x0,0xc(%rsp)
0x000000000000060d <+12>:    lea    0xc(%rsp),%rdi
0x0000000000000612 <+17>:    callq  0x5fa <set_five>
0x0000000000000617 <+22>:    add    $0x10,%rsp
0x000000000000061b <+26>:    retq
```

Q1.2

Under normal program execution, what is the 8-byte value stored under the address specified by %rsp **just prior to** executing the first instruction of `set_five`?

A. 0x7fff856001d8
B. 0x7fff856001c0
C. 0x000000000000060d
D. 0x0000000000000612
E. 0x0000000000000617
F. It could be any arbitrary 8-byte value

- callq label
  - DECREASES %rsp by 8
  - THEN stores the return address at the memory location given by the new %rsp
  - THEN jumps to the operand
- callq 0x5fa <set_five>
  - val(%rsp)= `0x7fff856001c0`
  - val((%rsp)) =0x0000000000000617

Q1.3

After executing instruction 0x0000000000000600 <+6>: retq in set_five, what's new %rip value?

A. 0x000000000000060d

B. 0x0000000000000612

C. 0x0000000000000617

D. 0x0000000000000604

E. 0x0000000000000608

- retq
  - Jumps to the location given by the value in memory located at %rsp
    - %rip = mem[%rsp]
    - val((%rsp)) =0x0000000000000617
  - <u>THEN INCREASES</u> %rsp by 8
    - %rsp= %rsp+8

Q1.4 p's location

Where is the local variable p stored?

A. some register

B. memory (data segment)

C. memory (stack)

D. memory (heap)

Given the following C function from Lab 1,

```
void set_five(int *p)
{
  *p = 5;
}

void test()
{
  int p = 0;
  set_five(&p);
}
```

The assembly for `set_five` function is:

```
0x00000000000005fa <+0>: ???
0x0000000000000600 <+6>: retq
```

The assembly for `test` function is:

```
0x0000000000000601 <+0>:     sub    $0x10,%rsp
0x0000000000000605 <+4>:     movl   $0x0,0xc(%rsp)
0x000000000000060d <+12>:    lea    0xc(%rsp),%rdi
0x0000000000000612 <+17>:    callq  0x5fa <set_five>
0x0000000000000617 <+22>:    add    $0x10,%rsp
0x000000000000061b <+26>:    retq
```

- Local variables -> stack
- If C's primitive data type and pointer
  - registers
  - stack
- Here, 0xc(%rsp) -> some place in memory
- -> stack

Q1.5 p's location

If your answer of 1.4 is memory, where in memory (aka what address) is p stored (assuming the value of %rsp is 0x7fff856001d8 just prior to executing the first instruction of test)?

A. 0x7fff856001d8
B. 0x7fff856001c8
C. 0x7fff856001d4
D. 0x7fff856001d0
E. 0x7fff856001cc

- val(%rsp)= `0x7fff856001d8`
- sub $0x10, %rsp
  - val(%rsp)= `0x7fff856001c8`
- where in memory is p stored
  - 0xc(%rsp)
  - `0x7fff856001d4`

Given the following C function from Lab 1,

```
void set_five(int *p)
{
  *p = 5;
}

void test()
{
  int p = 0;
  set_five(&p);
}
```

The assembly for `set_five` function is:

```
0x00000000000005fa <+0>: ???
0x0000000000000600 <+6>: retq
```

The assembly for `test` function is:

```
0x0000000000000601 <+0>:     sub    $0x10,%rsp
0x0000000000000605 <+4>:     movl   $0x0,0xc(%rsp)
0x000000000000060d <+12>:    lea    0xc(%rsp),%rdi
0x0000000000000612 <+17>:    callq  0x5fa <set_five>
0x0000000000000617 <+22>:    add    $0x10,%rsp
0x000000000000061b <+26>:    retq
```

Q1.6 set_five

What's the missing first instruction of set_five (aka the instruction corresponding to ???)

A. `movl $0x5,(%rdi)`
B. `movq $0x5,(%rdi)`
C. `movl $0x5,(%rsi)`
D. `movq $0x5,(%rdi)`
E. `movl $0x5, %edi`
F. `movq $0x5, %rdi`
G. `movl $0x5, %esi`
H. `movq $0x5, %rsi`

- *p=5;
- p -> the first argument
  - store in %rdi
- mov $0x5, (%rdi)
- p is int * type
  - movl $0x5, (%rdi)

Given the following C function from Lab 1,

```
void set_five(int *p)
{
  *p = 5;
}

void test()
{
  int p = 0;
  set_five(&p);
}
```

The assembly for `set_five` function is:

```
0x00000000000005fa <+0>: ???
0x0000000000000600 <+6>: retq
```

The assembly for `test` function is:

```
0x0000000000000601 <+0>:    sub    $0x10,%rsp
0x0000000000000605 <+4>:    movl   $0x0,0xc(%rsp)
0x000000000000060d <+12>:   lea    0xc(%rsp),%rdi
0x0000000000000612 <+17>:   callq  0x5fa <set_five>
0x0000000000000617 <+22>:   add    $0x10,%rsp
0x000000000000061b <+26>:   retq
```

Q2 array → Q2.1

What is the value of *c[1] after executing line 11?

A. 1

B. 2

C. 3

D. 4

E. 5

F. None of the above

- c's type is int **
- c[i] == *(c+i)
    - c+i -> int ** (pointer arithmetic)
    - c[1]==*(c+1) -> int *
    - c[1] == b == &b[0]
- c[1]++;
    - pointer arithmetic
    - c[1] = c[1]+1 == &b[1]
- *c[1]==b[1]==4

Given the following C code

```
1: void foo(int **arr, long i)
2: {
3:    arr[i]++;
4: }
5:
6: void test()
7: {
8:  int a[2] = {1, 2};
9:  int b[2] = {3, 4};
10: int *c[2] = {a, b};
11: foo(c, 1);
12:}
```

Q2.2 arr[i]++

If Line 3 is realized using one instruction, what's that instruction?

A. \`addl  $0x8,(%rdi,%rsi,8)\`
B. \`addl  $0x4,(%rdi,%rsi,8)\`
C. \`addl  $0x8,(%rdi,%rsi,4)\`
D. \`addl  $0x4,(%rdi,%rsi,4)\`
E. \`addq  $0x8,(%rdi,%rsi,8)\`
F. \`addq  $0x4,(%rdi,%rsi,8)\`
G. \`addq  $0x8,(%rdi,%rsi,4)\`
H. \`addq  $0x4,(%rdi,%rsi,4)\`

- arr[i]++;
- arr[i]
  - (%rdi, %rsi, size)
  - arr -> int **, arr[i] -> int * -> 8-byte
  - (%rdi, %rsi, 8)
- arr[i] = arr[i]+1;
- add: dest = dest + src
  - dest == arr[i] -> int * -> 8-byte
  - addq
- src
  - arr[i]=arr[i]+1 is a pointer arithmetic
  - “+1” -> 1*size(element)
  - arr[i] -> int *, arr[i][j] -> int -> 4-byte
  - 1*4 == 0x4
- addq $0x4, (%rdi,%rsi,8)

Q3 → Q3.1 location of p

Where is the local variable t in test stored?

A. some register

B. memory (data segment)

C. memory (stack)

D. memory (heap)

Given the following C code

```c
typedef struct kv_pair {
  long key;
  char* val;
} kv_pair;

void init_pair(kv_pair *p)
{
  p->key = -1;
  p->val = NULL;
}

void test()
{
  kv_pair t;
  init_pair(&t);
}
```

Q3.2 p->val

If Line 9 is realized using one instruction, what is that instruction?

A. \`movl $0x0,0x4(%rdi)\`
B. \`movq $0x0,0x4(%rdi)\`
C. \`movl $0x0,0x8(%rdi)\`
D. \`movq $0x0,0x8(%rdi)\`
E. \`movl $0x0,0x4(%rsi)\`
F. \`movq $0x0,0x4(%rsi)\`
G. \`movl $0x0,0x8(%rsi)\`
H. \`movq $0x0,0x8(%rsi)\`

- p->val = NULL;
- p is kv_pair * type
  - stored in memory (stack)
  - (%rdi)
- the start address of p->val
  - 0x8(%rdi)
  - because first 8-byte stores “key”, next 8-byte stores “val”
- “val” has type char *
  - 8-byte
  - movq
- movq $0x0, 0x8(%rdi)

Exercises

Discuss the answer

Ex1

1. Consider the following scenario:

%rsp is 0x7fffffffe448

The following instructions execute:

```
pushq %rbp
pushq %rax
```

What is the new value for %rsp?

- pushq src
  - <u>DECREASES</u> %rsp by 8
  - <u>THEN</u> stores the operand at the memory location given by the new %rsp
- %rsp -> 0x7fffffffe448
  - 0x7fffffffe440
  - 0x7fffffffe438
- 0x7fffffffe438

Ex2

2. Consider the following scenario:

%rsp is 0x7fffffffe448

The following instructions execute:

```
callq my_cool_function
```

While `my_cool_function` is executing, what is the value for %rsp?

When `my_cool_function` executes `retq`,what will be the value for %rsp?

- callq label
  - DECREASES %rsp by 8
  - THEN stores the return address at the memory location given by the new %rsp
  - THEN jumps to the operand
- %rsp -> 0x7fffffffe448
  - 0x7fffffffe440
- retq
  - Jumps to the location given by the value in memory located at %rsp
  - THEN INCREASES %rsp by 8
- 0x7fffffffe448

Ex3

3. Consider the following scenario:

%rsi is 5

%rdi is 8

The following instructions execute:

```
leaq 40(%rdi, %rsi, 8), %rax
```

What is the value of %rax?

```
Memory looks like this:
0x00:  10
0x08:  24
0x10:  32
0x18:  59
0x20:  23
0x28:  1
0x30:  66
0x38:  10000000
0x40:  2607
0x48:  2019
0x50:  111
0x58:  17
0x60:  32
```

- “lea” is no memory access, only do arithmetic calculation
- leaq 40(%rdi, %rsi, 8), %rax
- val(%rax) = val(%rdi)+8*val(%rsi)+40
    - =8+8*5+40 = 88 = 0x58

Ex4

4. Consider the following scenario:

%rsi is 5

%rdi is 8

The following instructions execute:

```
movq  40(%rdi,  %rsi,  8),  %rax
```

What is the value of %rax?

```
Memory looks like this:
0x00:  10
0x08:  24
0x10:  32
0x18:  59
0x20:  23
0x28:  1
0x30:  66
0x38:  10000000
0x40:  2607
0x48:  2019
0x50:  111
0x58:  17
0x60:  32
```

- movq 40(%rdi, %rsi, 8), %rax
- val(%rax) = mem[val(%rdi)+8*val(%rsi)+40]
  - = mem[0x58]
  - = 17

Buffer Overflow

Not all buggy memory references access “illegal” memory

Buffer Overflow

- Have learnt about the memory layout
- If an instruction tries to access some invalid memory
  - Segmentation fault occurs
- But not all buggy memory references access “illegal” memory
  - Buffer overflow exploits

Buffer Overflow

- Buffer overflow on the stack
- Buffer overflow overwrites the return address
  - attackers may carefully chosen return address, executes malicious code
  - code injection buffer overflow attacks

Defenses

- Write correct code to avoid overflow vulnerability
  - Use safe APIs to limit buffer lengths
  - Use a memory-safe language
- Mitigate attacks despite buggy code
  - will be an always on-going project, attack and defense itself are alternately developed
    - Security research domain
  - One idea to prevent control flow hijiacking: catch over-written return address before invocation
    - place special value (“canary”) just beyond buffer
    - GCC implementation: -fstack-protector

Compiler optimization

Tries to minimize or maximize some attributes of an executable computer program

Optimizing compiler

- Goal: generate efficient, correct machine code
  - generally implemented using a sequence of *optimizing transformations*
  - algorithms which take a program and transform it to produce a semantically equivalent output program that uses fewer resources and/or executes faster
- The compiler performs optimization based on the knowledge it has of the program
  - Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them
- Common optimization
  - code motion
  - use simpler instructions
  - reuse common subexpressions

Optimization -- GCC

- Without any optimization option, the compiler’s goal is to reduce the cost of compilation and to make debugging produce the expected results
- Turning on optimization flags makes the compiler
  - attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program
- When debugging your code, it may help to disable optimizations
  - Add –O0 to the Makefile CFLAGS
- Depending on the target and how GCC was configured, a slightly different set of optimizations may be enabled at each -O level
  - You can invoke GCC with -Q --help=optimizers to find out the exact set of optimizations that are enabled at each level

Optimization -- GCC

- gcc's optimization levels: -O, -O2, -O3, -Og, -O0, -Os, -Ofast
- With -O, the compiler tries to reduce code size and execution time
  - without performing any optimizations that take a great deal of compilation time
- -O2 optimize even more
  - turns on all optimization flags specified by -O, and it also turns on some other optimization flags: e.g. -finline-small-functions…
- -Og optimize debugging experience
  - offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience
  - enables all -O1 optimization flags except for those that may interfere with debugging