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

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
  - DECREASES %rsp by 8
  - THEN 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