Exam 2 topics

Lectures

- Programming with Memory
- x86 Basics
- x86 Control Flow
- x86 Procedures, Call Stack
- Representing Data Structures
- Buffer Overflows
- Processes Model
- Shells
- (Basics of) Memory Alloc, Caching

Labs

- Pointers in C
- x86 Assembly
- x86 Stack
- Data structures in memory
- Buffer overflows
- Processes

Topics

- C programming: pointers, dereferencing, arrays, structs, cursor-style programming, using malloc
- x86: instruction set architecture, machine code, assembly language, reading/writing x86, basic program translation
- Procedures and the call stack, data layout, security implications
- Processes, shell, fork, wait
- Basics of: malloc implementation, caching

Assignments

- Pointers
- x86
- Buffer
- Concurrency
- Malloc checkpoint

Exam 2: ISA
December 10 (during last lab)

Practice problems

For Exam 2: ISA

x86 short answer practice problems

ex

1. Which x86 instructions implicitly change the stack pointer? How do they change it?

2. What are some things defined by the *word size*? What is the word size we have been using for x86 in class?

3. Describe the general idea of a buffer overflow exploit in C code compiled to x86.

4. Describe how a child process's memory is related to the memory of the parent process.

x86 short answer practice problems

ex

1. Which x86 instructions implicitly change the stack pointer? How do they change it?

pushq	popq	call	ret
`%rsp -= 8`	`%rsp += 8`	`%rsp -= 8`	`%rsp += 8`

2. What are some things defined by the *word size*? What is the word size we have been using for x86 in class?

   Register size, address size, pointer size
   NOT instruction size (variable-width instruction size)

3. Describe the general idea of a buffer overflow exploit in C code compiled to x86.

   Buffer overflow occurs when code lacks bounds checking in writing untrusted input to a destination region of memory that is too small. Buffer overflow attacks can overwrite the return addresses on the stack to point to further exploit code.

4. Describe how a child process's memory is related to the memory of the parent process.

   The child process starts with a copy of the state of the parent's memory. It is a private copy: the child and the parent do not share memory once the child is created.

2-D array practice problem

ex

```
long a[2][3];
```

1. Draw a picture of how this array is laid out in memory, labeling the indices and byte offset of each element (starting with `a[0][0]` at offset `+0`);

Recall: `index = C*r + c`

scale by element size

```
long get_elem_1_2(long a[2][3]){
  return a[1][2];
}
```

2. Write x86 assembly code to implement this function.

2-D array practice problem: solution

ex

```
long a[2][3];
```

1. Draw a picture of how this array is laid out in memory, labeling the indices and byte offset of each element (starting with `a[0][0]` at offset `+0`);

Recall: `index = C*r + c`

scale by element size

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
+0	+8	+16	+24	+32	+40

```
long get_elem_1_2(long a[2][3]){
  return a[1][2];
}
```

2. Write x86 assembly code to implement this function.

Since we know the size, we can calculate
C*r+c = 3*1+2 = 5, 5*sizeof(long) = 5*8 = 40

```
movq 40(%rdi),%rax
retq
```

x86 arithmetic practice problem

ex

```
long funmath0(long x, long y) {
    return x + 4*y + 21;
}
```

```
long funmath1(long x, long y) {
    return 2*x + 4*y + 21;
}
```

```
long funmath2(long x, long y) {
    return 6*x + 5*y + 21;
}
```

Implement the above functions in x86 *without* `addq` or `mulq`.
You can use `leaq` and any other x86 instruction.

Recall: addressing modes can only multiply by 1, 2, 4, or 8.

x86 arithmetic practice problem

ex

```
long funmath0(long x, long y) {
    return x + 4*y + 21;
}
```

```
long funmath1(long x, long y) {
    return 2*x + 4*y + 21;
}
```

```
long funmath2(long x, long y) {
    return 6*x + 5*y + 21;
}
```

3 possible answers:

```
funmath0:
  leaq    21(%rdi,%rsi,4), %rax
  ret

funmath1:
  leaq    (%rdi,%rsi,2), %rax
  leaq    21(%rax,%rax), %rax
  ret

funmath2:
  leaq    (%rdi,%rdi,2), %rdx
  leaq    (%rsi,%rsi,4), %rax
  leaq    21(%rax,%rdx,2), %rax
  ret
```

Implement the above functions in x86 *without* `addq` or `mulq`.

You can use `leaq` and any other x86 instruction.

Recall: addressing modes can only multiply by 1, 2, 4, or 8.