15-213: Introduction to Computer Systems
Written Assignment #3

This written homework covers Machine Programming (Data, Advanced).

Directions

Complete the question(s) on the following pages with single paragraph answers. These questions are not meant to be particularly long! Once you are done, submit this assignment on Canvas.

Below is an example question and answer.

Q: Please describe benefits of two's-complement signed integers versus other approaches.

A: Other representations of signed integers (ones-complement and sign-and-magnitude) have two representations of zero (+0 and -0), which makes testing for a zero result more difficult. Also, addition and subtraction of two's complement signed numbers are done exactly the same as addition and subtraction of unsigned numbers (with wraparound on overflow), which means a CPU can use the same hardware and machine instructions for both.

Grading

Assignments are graded via *peer review:*

1. Three other students will each provide short, constructive feedback on your assignment, and a score on a scale of 1 to 15. You will receive the maximum of the three scores.
1. You, in turn, will provide feedback and a score for three other students (not the same ones as in part 1). We will provide a *rubric*, a document describing what good answers to each question look like, to assist you. You receive five additional points for completing all of your peer reviews.

Due Date

This assignment is due on Wednesday, June 15 by 11:59pm Pittsburgh time (currently UTC-4). Remember to convert this time to the timezone you currently reside in.

Peer reviews will be assigned roughly 12 hours later, and are due a week after that.

Question #1

Explain two strategies for thwarting buffer overflow attacks. Compare their vulnerabilities. Describe an attack that one strategy would prevent, but the other would not.

1. randomization of the stack position
2. making the stack non-executable

Both strategies cannot prevent buffer overflows. They only prevent or mitigate the effects of buffer overflows.
Also, both strategies cannot prevent overwriting the return address on the stack.

Strategy 1 only shifts the stack position of the entire program by some offset, the attacker can still calculate the position of the injected code based on the %rsp register and the relative position of the injected code.

Therefore, strategy 2 can prevent code injection attacks. Strategy 1 makes the attack more difficult but cannot prevent it.

Question #2

Think about following structure

```
struct a {
        char c[3];
        size_t y;
        int z;
};
```

Using proper alignment, how much space does this struct take up in memory? Describe a way to save space with this struct and state how much space is saved. How would accessing values differ with or without padding in data structures like structs and arrays? Why is the padding used in these cases important?

The alignment of a is 8 bytes.
The memory layout is:

```
| c[0] | c[1] | c[2] |    |    |    |    | y | y | y | y | y | y | y | y |
|  z   |   z  |   z  | z  |    |    |    |   |   |   |   |   |   |   |   |
```

So it takes up 24 bytes in memory.

To save space:

```
struct a {
  char c[3];
  int z;
  size_t y;
};
```

The new memory layout is:

```
| c[0] | c[1] | c[2] |    | z  | z  | z  | z  | y | y | y | y | y | y | y | y |
```

And it only takes up 16 bytes.

With padding, the address of basic data types is always the multiple of their alignment. As a result, these data types are always stored within 4-byte/8-byte chunks. So loading them from memory is faster compared to loading values that span the 8-byte boundaries (unaligned).

The padding here makes both the struct and the elements inside aligned. Therefore, accessing struct elements is faster. Meanwhile, it makes the address calculation of an array of this struct much easier. We can use `index * sizeof(a)` to get the memory offset of an element.

Question #3

Scalar values (integers, floating-point numbers, and pointers) that are larger than one byte are usually required to be *aligned*, meaning that the address of such a value must be an integer multiple of its size. We said in several different classes that this requirement makes it simpler to design the CPU hardware. As of the "memory hierarchy" class, you now have enough information to understand why this is. Give a concrete example of something that the CPU would need to be able to do if a 4-byte integer could begin at any address, and would not need to be able to do if a 4-byte integer can only begin at an address that's evenly divisible by four.

From memory hierarchy class, we learned that the CPU retrieves 64-bit word from RAM at a time. So starting from address 0, the CPU can get data within 64-bit size that is located at the multiple of 64-bit address using a memory access.

Therefore, if we are loading a 4-byte integer that is *aligned*, we first find out the 8-byte aligned address that contains this integer within an 8-byte span, then we load the entire 8 bytes, and extract the integer part.

If we are loading an integer that is *not* aligned, we first need to check if it is located at an 8-byte aligned address. If it is, we repeated the above process. If not, we need to use two memory access to load this integer. The first access loads the part of integer that is within the first 8-byte aligned bits, the second access loads the other part of the integer. Then we put the loaded bits together and extract the 4-byte integer.