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. Can you describe an attack that one strategy would prevent, but the other would not?

A non-exhaustive list of possible strategies, with attacks that (in principle) defeat each.

Stack randomization involves making the position of the stack vary so that different runs of the same program use different stack addresses. This makes it difficult for the attacker to determine where the exploit string is stored, which helps prevent code injection attacks since the exploit string needs to include the address of the injected code in order to direct execution of the program to it. However, attackers can use a brute force strategy and try many attacks, each using a different address for the injected code. Attackers can also prepend their exploit code with a nop sled -- a long sequence of nop instructions. These instructions have no effect except to increment the program counter, but make it easier for the attacker to overcome stack randomization. They no longer need to be able to correctly guess the address of the first instruction in their injected code, but any address within this long sequence of nops as the program will simply go through all the nops and then reach the exploit code.

Limiting the regions of memory that hold executable code can also protect against buffer overflow attacks. Even if an attacker can place exploit code on the stack and determine the address at which it is stored, if the stack is marked as non-executable, trying to execute the code will fail. This strategy is robust against the nop sled technique outlined above. However, as seen in attacklab, this can be overcome by return oriented programming attacks as they use only existing code.

Another strategy uses stack canaries to detect whether data stored on the stack has been corrupted. A secret value is placed on the stack, in between a buffer in memory that can be overflowed, and the nearest return address. Before returning from the function, the program checks whether the value has changed. Stack canaries reliably defeat any attack in which the attacker must overwrite all of the bytes in between the beginning of the vulnerable buffer and the return address. However, canaries are ineffective in situations where the attacker can choose *not* to overwrite some of the bytes in between the buffer and the return address (something like the function on slide 7 of 09-machine-advanced.pdf).

In principle, a program that *always* takes care not to write data outside the bounds of *any* memory allocation, no matter where it is or how it was allocated, is immune to buffer overflow attacks. The difficulty here is in being certain that every memory write is safe. Avoiding the use of inherently insecure C library functions (e.g. **`gets()`** and **`scanf()`**) is not enough. Coding in a higher-level language usually means the attacker has to attack the language interpreter rather than the individual program, but there is a long history of exploitable memory-corruption bugs (not exactly *buffer overflow* bugs) in the Java interpreter, for instance.

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?

With proper alignment it takes up 24 bytes of space.

c0	c1	c2	-	-	-	-	-	y	y	y	y	y	y	y	y
z	z	z	z	-	-	-	-								

To save space there are a few ways we can better organize our values. In all cases, we now use up 16 bytes of space, and we save 8 bytes of space.

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

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

Or

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

c[0]	c[1]	c[2]	-	z	z	z	z	y	y	y	y	y	y	y	

Or

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

z	z	z	z	c[0]	c[1]	c[2]	-	y	y	y	y	y	y	y	

The way values are accessed do differ with or without padding. With padding we know where array and struct elements are located based on the address alignment requirements and can be accessed based on multiples of the alignment requirement. Without padding, finding array or struct element addresses requires adding the sizes of the elements within the array or struct to discover its offset from the initial address of the array or struct.

Padding is important because it improves CPU performance. This is because it ends up being more costly/takes more time to calculate the address of an element offset without padding (as described above) than with padding.

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.

If a 4-byte integer can begin at any address, then part of it might be stored in one cache block, and the rest of it in the next higher cache block. A single CPU instruction would therefore need to be able to access two (consecutive) cache blocks. If a 4-byte integer can only begin at an address that’s evenly divisible by four, and the cache block size is also evenly divisible by four (which is always true, for the same reason), then any one CPU instruction will only ever need to access one cache block.

When we learn about virtual memory, we will learn that there’s a larger-scale structure called a “memory page” and that it would cause serious problems if a single CPU instruction might need to access more than one page. Alignment requirements also eliminate this possibility, because the size of a page is always a multiple of the cache block size.

Note that it is *possible* to design a CPU that can access more than one cache block, or even more than one page, in a single instruction; it just requires more circuitry. The x86 can in fact do this, but “unaligned” memory accesses are anywhere from 2× to 200× slower than aligned memory accesses (depending on the exact conditions).