Lesson 6: Buffer Overflow Intro

Objectives:

(a) Describe the buffer overflow attack, determine what features of C make it possible, and identify who is responsible for memory management in C.

(b) Demonstrate the ability to craft simple buffer overflow attacks.

(c) Explain how specific buffer overflow attacks work by describing stack operations.

1. The Buffer Overflow Attack Introduction

The first major attack on DoD computer networks took place in February of 1998 and lasted for over a week. The hackers gained administrative (i.e., “root”) access on UNIX machines at 7 Air Force sites and 4 Navy sites, gaining access to logistical, administrative and accounting records. The method used in this early attack—a buffer overflow—has been used countless times ever since. Many famous attacks—the Morris Worm, the Code Red Worm, the SQL Slammer Worm, the Twilight Hack, Blaster, Conficker—used the buffer overflow as a primary attack vector. The recent Stuxnet worm used the buffer overflow as one of many attack vectors.

You may have learned in SY110 that the Department of Homeland Security worked together with the SANS Institute, Apple and Oracle back in 2011 to develop a list of the top 25 software vulnerabilities, and the “classic buffer overflow” came in third, behind SQL injection and OS command injection.

In February 2013, the security firm Sourcefire surveyed Common Vulnerability Scoring System (CVSS) data from 1988 to 2012, and found that buffer overflows were the most-often reported vulnerability. Of the vulnerabilities assigned a category of “high severity”, buffer overflows comprised over a third of the total. Security analyst Paul Roberts notes that “the stubborn staying power of buffer overflows for more than two decades – despite gallons of industry ink spilled on the problem – is dispiriting and has to get us thinking about what it is we’re doing wrong as an industry.”

Bottom Line: The buffer overflow attack is still an exceedingly common attack vector, hence why we learn about it in this course.

2. In a Nutshell

The basis for the buffer overflow attack can be appreciated by examining the following section of C code:

```c
int k = 1000;
char my_stuff[512];
my_stuff[k] = 'A';
```

What happens if this code is executed? This array is only allotted 512 bytes; i.e., this array holds character variables `my_stuff[0]` through `my_stuff[511]`. The programmer who wrote the third line of code seems unaware that the last element of the array is `my_stuff[511]`, since this third line of code assigns a value to the non-existent variable named `my_stuff[1000]`. When this code is executed, a byte of memory 488 bytes beyond the end of the array will be overwritten with the character ‘A’.

This error will not be caught at compile-time. In a nutshell, the problem is that C compilers do not check for reading or writing beyond the bounds of an array. In fact, since arrays are a consecutive group of memory locations, our compiler is fully capable of generating that machine language instruction that stores `my_stuff[1000] = ‘A’` in memory.

This is a big concern because almost all major operating systems are written in C. Additionally, many popular applications are also written in C.

You might be wondering: What exactly happens when the code above is run? The unfortunate answer is: Who knows? Perhaps nothing noticeable will occur. Perhaps disaster will occur.
Practice Problem 8.1

What feature of the C language makes a buffer overflow attack possible?

Solution:

3. Back to the Stack

Recall that when a program is to be executed, the operating system reserves a block of main memory for the program’s instructions. The text segment holds the machine language instructions of the program, which we typically view as assembly-language instructions.

3.1 Stack mechanics

The stack is the portion of memory that the program has available to store information during execution. For example, the program’s variables are stored on the stack. Let’s look at the following program and relying on our understanding of the stack mechanics from the previous chapter, let’s diagram the stack frame.

```
void user_fxn(int a, int b)
{
    char alphanode[?]: ← function variable
    printf("Enter your alpha code!");
    scanf("%s", alphanode);
}
```

The program begins at the `main` function, and the main variables are placed on the stack building from the bottom up.

We keep track of the stack using the base pointer (`ebp`) which points to the bottom of the stack (specifically the memory location immediately following the bottom of the stack) and the stack pointer (`esp`), which points to the top of the stack. Each function gets to place its variables on the stack. The part of the stack that belongs to a function is called that function’s stack frame. In the above program, `main( )` has a stack frame as does `user_fxn( )`.

When we reach the point in executing the program that we arrive at the function call, `user_fxn(x, y);` in main, the function arguments are placed on the stack in preparation for the function call. The stack prior to executing the function call now looks like this:

```
Function Arg
Memory for fxn arg x
Memory for fxn arg y
```

```
Main Variables
Memory for y
Memory for x
```

We will then prepare to execute the function call and reset `esp` and `ebp` to hold the values of the top and bottom of the stack frame for `user_fxn`. Similarly, `eip` will reset to hold the address of the first instruction in the text segment for `user_fxn`.

After all machine language instructions in the text segment for `user_fxn` is executed, we will need to jump back into the `main` function where we left off. For this reason, we need to keep the stack frame for `main` undisturbed. In order for this to happen, we place the return address for the next instruction in `main` that will run after the function has completed on the stack, and the old value of the base pointer (for main’s stack frame) on the stack.
After placing the necessary addresses to pick up where we left off when we return to main, we begin executing the instructions in the text segment for `user_fxn` and the ebp and esp registers now hold the address of the top and bottom of the stack frame for `user_fxn`. On the stack frame for `user_fxn` we allocate space for the function variable `alphacode`, so the stack now looks like the following:

*Recall from the last chapter, the organization of the stack and note that our example conforms to this as it must, building each block from the bottom up.*

### 3.2 Overflow!

Consider that the function `user_fxn`, as part of its code, prompts the midshipman to enter his or her alpha code. It uses the character array named `alphacode` to hold the value that the midshipman enters. We have seven bytes reserved on the stack frame for `user_fxn` for `alphacode`, which is enough space to safely store 6 characters and a null terminator. As long as the midshipmen follow the instructions and enters their alpha code when prompted, our program will work as designed.

What if our midshipman enters their alpha code beginning with the letter “m” when prompted? Then the null would be placed outside the bounds of the array. Or perhaps our midshipman was sleepy, and when he was prompted to enter his alpha code (which happens to be `151234`) he dozed off for a micro-nap and accidentally entered:

```
1512344444444444444444444 <enter>
```

In this case, he entered a total of 25 characters. Think about this… what happens? When the 25 characters (plus the NULL which is automatically appended) are written into the array `alphacode`, the characters entered beyond the sixth will result in overwriting memory outside the bounds of the array!

And what is just outside the bounds of this array? Two very important values for returning to main after the function instructions are complete: the return address and the prior ebp address. Vulnerable as they may be, we know that our C compiler will not prevent us from accessing outside the bounds of the array!
Suppose this occurs. What will happen when function `user_fxn` is finished executing? If the return address was indeed overwritten, then the return address will now consist of some of the characters that were in the midst of the alpha string that was entered. The CPU will “pick up” the “new” return address and attempt to continue the “fetch, decode, execute” cycle from that location in main memory. However, it is unlikely when the return address is overwritten, that the resulting address is an address that is still within the memory allocated to the program. If our program tries to access memory outside that memory allocated to our program, the OS will crash our program with a **segmentation fault**. A segmentation fault occurs when a program attempts to access memory outside the region of main memory that it has been allotted.

When this is done intentionally, this sequence of events is called a **buffer overflow attack**!

**Practice Problem 8.2**

Describe the mechanism by which a segmentation fault occurs.

Solution:

The following two practice problems take a closer look at how buffer overflows can occur.

**Practice Problem 8.3**

For the `pawn` function below, is it possible to overwrite the value of the variable `cost` by entering a string beyond the bounds of the array `item` on the stack during the `scanf` function call? Explain.

```c
void pawn()
{
    char item[12];
    int cost = 100;
    printf("What have you come to sell? ");
    scanf("%s", item);
}
int main()
{
    pawn();
}
```

Solution:
Practice Problem 8.4

When the `echo_string` function is called in `main` from the following code sample, the stack pictured below is created. Note: `main` has no variables declared, and the function takes no arguments.

```c
#include<stdio.h>
void echo_string()
{
    int count;
    char entered_string[10];
    printf("Enter a string: ");
    scanf("%s", entered_string);
    for(count=0; count < 10; count=count+1)
    {
        printf("%s\n",entered_string);
    }
}
int main()
{
    echo_string();
}
```

Assuming there is no padding (extra spaces) when the frame is created. How many characters can be entered before the return address is changed? Completely overwritten?

Solution:

4. A Possible Solution: Don’t Use C!

If this problem exists simply because C compilers do not check for reading or writing beyond the bounds of an array, an easy way to solve this problem would be to avoid using the C language altogether. In fact, more modern programming languages such as Java and C# will not allow a programmer to access beyond the bounds of an array. Why not simply abandon C and announce to the world: Problem Solved?

We cannot simply abandon C since too many C programs are in circulation. Moreover, programmers would not want to abandon C even if a magic wand could suddenly convert all C legacy code into Java programs! Recall from an earlier lecture that even today, most programmers are programming in C and prefer to program in C.

The C programming language is very popular because it executes quickly and it provides the programmer with a high level of control over the program. But with this power comes responsibility: data integrity in C is the **programmer’s responsibility**. If the responsibility for data integrity were taken away from the programmer and given to the compiler instead, the compiler would consistently and constantly check that we never run beyond the bounds of an array (which is good), but program execution would be much slower (which is bad). Generally, users want their programs (whether they be operating systems, office software, application programs or games) to execute quickly. C executes quickly since the compiler does not verify data integrity. Yet, with the responsibility for data integrity resting on the programmer's shoulders, buffer overflow errors can occur if the programmer is not careful.

A good analogy is provided by USNA instructor Nick Rosasco:

*C is like a workbench with saws and power tools and high-voltage drops and spinning lathes all out in the open, without safeguards and protections. For a master craftsman who knows his job very well, this environment would be ideal for productive work, with the understanding that the craftsman has to be responsible for his safety. For the novice, this environment would be very dangerous.*

Conversely, a workbench that required the user to constantly interact with multi-level interlocked protection mechanisms and cumbersome safety features would be much safer for the novice, but would drive the skilled craftsman insane. As with workbenches, so with programming languages: The intentional lack of safety in C translates into greater flexibility and improved performance... and risk.
CH 6. Problems

Name: _______________________

1. What feature of the C language make a buffer overflow attack possible?

2. Answer the following questions concerning how a program is stored in memory during its execution.
   (a) Which segment of memory has contents that remain unchanged during program execution?
   (b) Does the programmer have complete control over how the stack is organized?
   (c) What is the relationship between the order in which variables appear in a function and the order in which these same variables are stored in a function's stack frame?
   (d) What CPU registers are used to define the boundaries of a stack frame?
   (e) Suppose main calls a function named fun. After all the commands of fun have executed, how does the program know where to continue at the exact location in main where it left off?
   (f) Is a source code file permitted to have more than one function?
   (g) If your answer to (f) was "no", explain why that is the case. If your answer to (f) was "yes", explain how the operating system knows where to begin executing your program if the source code file contains multiple functions.

3. Given the following code snippet:
   ```c
   char first_name[6] = "Alice";
   strcpy(first_name, "Alexander");
   ```
   (a) Will the C compiler state that there is an error?
   (b) What potentially dangerous situation occurs because of the snippet above?
   (c) What is the minimum size necessary for the array first_name to prevent this error?
   (d) There are at least two ways to change the above code to prevent the above error from happening. Describe one.
4. When the greetings function is called in main from the following code sample, the stack pictured in the figure that follows is created.

```c
#include<stdio.h>
void greetings()
{
    int name_len = 15;
    char name[name_len];
    int year = 2014;

    printf("Enter your name: ");
    scanf("%s", name);
    printf("Hello: %s! The current year is %d.\n", name, year);
}

int main()
{
    greetings();
}
```

(a) Assuming there is no padding (extra spaces) when the stack frame is created, how many characters must the user enter to overwrite only the first byte of the return address?

(b) Is it possible to change the value of year by performing a buffer overflow attack? Why or why not?