Lesson 8: Buffer Overflow Attack

Objectives:

(a) Describe how a buffer overflow attack can be used to gain `root` access to a computer.

(b) Describe two techniques that a hacker can use to make it simpler to craft a buffer overflow.

(c) Given parameters of an attack and the layout of the stack for a program, determine if the attack could be successful.

(d) Describe technical solutions that have been proposed to prevent a program from being exploited by a buffer overflow.

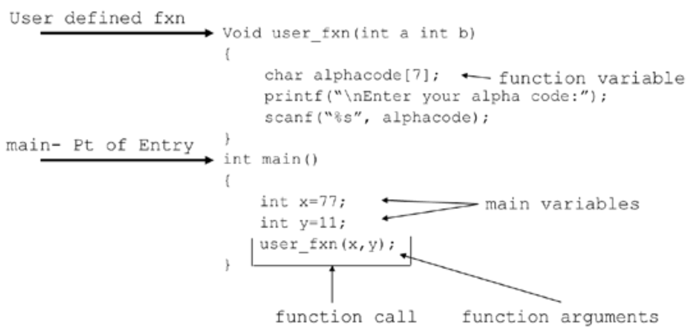
Back in Chapter 6, we noted that the very first major attack on DoD computer networks took place in February of 1998 and lasted for over a week. The hackers gained administrative (i.e., "`root`") access on UNIX machines at 7 Air Force sites and 4 Navy sites, gaining access to logistical, administrative and accounting records. The method used in this early attack—*a buffer overflow*—has been used countless times ever since.

But how does this attack occur and why is it successful? How do we distinguish a buffer overflow attack from an accident where you write outside the bounds of that array??

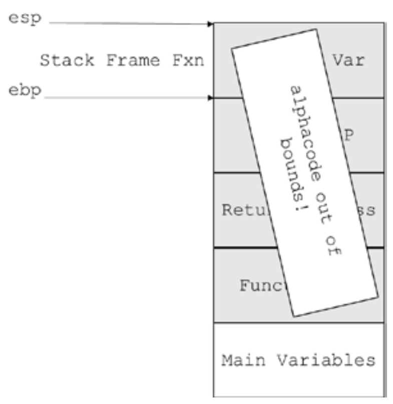
1. Anatomy of a Buffer Overflow Attack

Using the same example program from Chapter 6, we recall that when prompted to enter your alpha code during the function call, you may safely enter 6 characters (with room for the NULL terminator). Additional characters will begin to overwrite memory beyond the bounds of the array potentially imperiling the values of the `prior ebp` and the `return address` stored on the stack.

In this example, writing outside the bounds of the array was caused by mistake and was a mere annoyance. If the return address is overwritten, then the return address will consist of some of the characters that were in the `alphacode` that was entered. This may cause a segmentation fault and crash our program if the memory allocated on the stack for the return address does not point to a valid address. We recover from it by running our program again and ensuring that we don't slip up and enter too many characters. But a buffer overflow attack is capable of being so much more.



```
Void user_fxn(int a int b)
{
    char alphacode[7];
    printf("\nEnter your alpha code:");
    scanf("%s", alphacode);
}
int main()
{
    int x=77;
    int y=11;
    user_fxn(x,y);
}
```

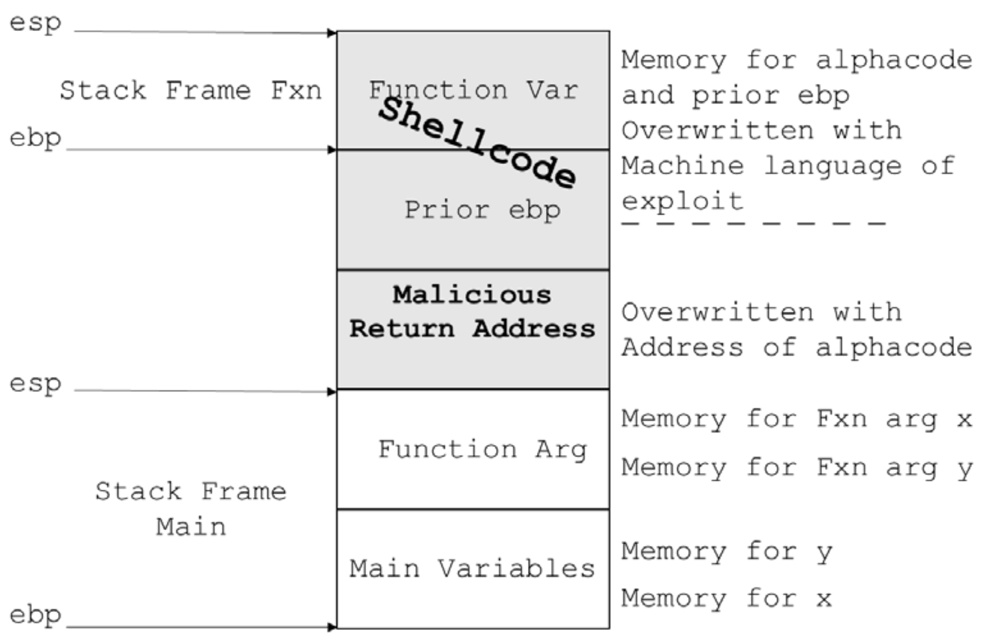


1.1 What makes an attack malicious?

The idea behind the ***buffer overflow attack*** is that the adversary is placing its own executable program, or **exploit,** in memory and making it execute. Instead of entering their alpha code when prompted the user is placing an exploit in the stack memory and writing beyond the bounds of the character array of `alphacode.`

The key to making the exploit execute is that the return address is also overwritten and changed to the address of `alphacode`. If we overwrite the program's return address with the value of the address of `alphacode`, then the return address will bring us to the start of the executable program (i.e. the exploit), as the exploit was entered onto the stack at the address of variable `alphacode`! That is to say that the instruction pointer will "pick up" the "new" return address when the function is done executing, and the executable code that the adversary placed on the stack will then start executing.

Again, the exploit involves the adversary placing its own program in memory and making it execute. The exploit is also referred to as "shellcode" because it normally will contain instructions to open a "shell", which is a window like you've been using in the VM to run your programs. The attack will result in the stack being configured as in the following diagram.



1.2 <u>Required Elements</u>

1.2.1 <u>Exploit code or shellcode</u> Shellcode is a list of carefully crafted instructions that can be executed once the code is injected into a running application – like a virus inside a cell – but it isn't really a standalone executable program. Shellcode is considered exploit code or exploit payload. Buffer overflows are the most popular way of injecting a shellcode[1] because after the instructions of the called function are done executing, the executable code that the adversary placed on the stack will begin executing. This takes advantage of the stack mechanics for function calls. After all of the called function instructions are finished executing, the value of the return address on the stack is loaded into `eip` as the address of the next instruction to be executed. We remember from Chapter 5 that the return address is placed on the stack to allow the CPU to "pick up" where it left off in `main` prior to the function call.

1.2.2 <u>Malicious Return Address</u> The return address must be overwritten with a deliberate value, ideally the beginning of the memory block designated for the input string used by the function. In order to find these locations you would first attempt to enter a ridiculously long value (but one that you know) when prompted to enter something, and you would check to see if that causes the program to behave erratically or crash with a segmentation fault. This is an example of a technique known as "*fuzz testing*" or "*fuzzing*". In general, fuzzing is the attempt to find soft spots in a program. When a segmentation fault occurs, the OS will output a "core" file that contains all the information needed by the debugger to reconstruct the state of execution when the invalid operation caused a segmentation fault. If fuzzing is successful, you can then analyze the hex dump of the core file to see where your input string resides in memory. You then use this info, plus the source code (usually available for UNIX) to attempt to find where the return address is stored. Crafting the ideal malicious return address is time consuming though not altogether challenging.

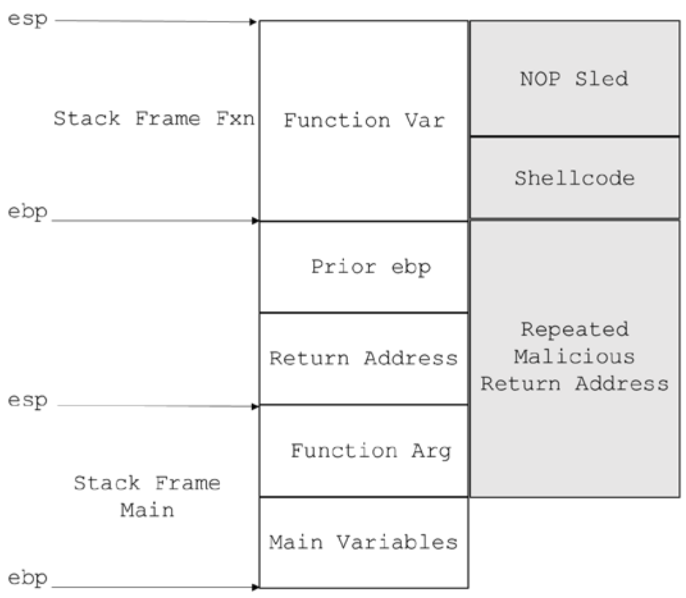
1.3 <u>Optional Elements</u> Crafting a buffer overflow attack is not easy. Hackers use two clever techniques to make the process a little more manageable.

1.3.1 <u>NOP Sled</u> A series of NOP instructions (assembly language "no operation" instructions, which are actual instructions to do nothing) are added to the exploit before the shellcode. This series of NOP instructions is called the "NOP sled". The NOP sled lets the hacker be a little bit off with their malicious return address. The return address just has to point anywhere within the NOP sled. Otherwise, the return address would need to be the precise address of the start of the shellcode.

1.3.2 <u>Repeated Malicious Return Address</u> The malicious return address is repeated many times at the end of the shellcode. The repetition gives the hacker a number of chances to get the malicious return address correctly positioned to overwrite the actual return address.

1.4 <u>Typical Layout</u> Utilizing the NOP sled and Repeated Return Address to make our buffer overflow a success, the typical layout of the exploit in a buffer overflow attack looks as follows when laid over the stack of a function:

[1] What is the relationship between shellcode and exploit code. http://searchsecurity.techtarget.com/answer/What-is-the-relationship-between-shellcode-and-exploit-code. Accessed 29SEP2016



1.5 <u>Attack Sequencing</u> The buffer overflow attack would happen in the following sequence:

1. Vulnerable program is run from the command line.
2. During a function call, exploit is injected causing a buffer overflow and overwriting the return address value of the stack which is intended to return CPU to the next instruction in `main` following the function call.
3. Function call instructions finish executing. Malicious return address is loaded into `eip`
4. CPU begins “fetch, decode, execute” cycle from the malicious return address. Malicious return address points to somewhere within the NOP sled.
5. CPU continues to move through the instructions arriving at the shellcode and continuing through effectively “launching” the exploit.

Practice Problem 10.1

A program that contains a user-defined function and function call. The program takes a user input on the command line that is copied onto the function's stack frame at address `ebp`−120. Using the debugger and pausing execution of the program *within* the function produces the following information:

```
(gdb) i r eip ebp esp
eip        0x0804834a  0x0804834a
esp        0xbffff700  0xbffff700
ebp        0xbffff7a0  0xbffff7a0
```

(a) How many bytes are on the function's stack frame?

(b) Suppose a hacker is attempting to perform a buffer overflow attack on this program. If the buffer overflow attack consists of running the program with a command line argument comprised of 80 NOPs followed by a 64-byte exploit followed by an appropriate malicious return address repeated 20 times, would you expect this attack to work? Why or why not?

(c) If the buffer overflow attack consists of running the program with a command line argument comprised of 20 NOPs followed by a 64-byte exploit followed by repeating `0x0804834b` twenty times, would you expect this attack to work? Why or why not?

(d) If the buffer overflow attack consists of running the program with a command line argument comprised of 10 NOPs followed by a 64-byte exploit followed by repeating `0xbffff700` twenty times, would you expect this attack to work? Why or why not?

(e) If the buffer overflow attack consists of running the program with a command line argument comprised of 40 NOPs followed by a 64-byte exploit followed by repeating `0xbffff740` twenty times, would you expect this attack to work? Why or why not?

2. Consequences of a Buffer Overflow

Running the shellcode which had been injected during the buffer overflow certainly sounds like a nefarious proposition but what does it mean? The consequences of the buffer overflow center around the content of the shellcode. What operations is our machine performing that are not a part of the original program and do we have the permissions to perform them? A quick google search for "examples of shellcode" will turn up numerous results ranging from the script to `setuid`, to the script to open a new command prompt.

In Chapter 7 we explored the `setuid` permission. When an executable program has the `setuid` flag set, then whenever the program is executed, it will behave as though it were being executed by the owner.

Consider the case where `root` is the owner of a program with the `setuid` flag set. If a user were to perform a buffer overflow on that program and execute shellcode that opens the command prompt, then it would appear as if `root` were giving the command and open the command prompt for `root`.

THIS IS A HUGE DEAL! Why? Because `root` is the user who has privileges to read, write, execute and even delete any user files on the entire computer! You're not supposed to be `root`!

The buffer overflow itself pales in comparison to the privilege escalation that results from the buffer overflow. Much like a thief picking a lock, once the overflow is executed, the attackers work has only just begun.

3. Defenses against the Buffer Overflow Attack

Now that we understand the problem, how can we prevent a program from being exploited by a buffer overflow?

3.1 <u>Compiler bounds check</u> One technique that would surely work: we could add code to the compiler to check the bounds on each and every array reference (i.e. we make the compiler responsible for ensuring data integrity). But this would significantly slow down all programs, and so is not a solution at all—it would be akin to preventing injuries in automobile accidents by imposing a national 5 MPH speed limit.

3.2 <u>Careful coding</u> We can minimize buffer overflow exploits by careful coding. The programmer is responsible for data integrity, and must be vigilant in testing and retesting all code for potential problems. Several C library functions are notorious for inviting buffer overflow problems. In our EC310 class, the `strcpy` function is a well-known culprit. The designers of C have, in fact, provided an improved version of this particular command. The improved version, named **`strncpy`**, introduces some protection against writing beyond the end of an array. The format for the `strncpy` command is:

```
strncpy(destination_string , source_string ,number_of_characters_to_copy)
```

The `strncpy` command's third argument is the number of characters to copy. The programmer can ensure, through the use of this third argument that we do not write beyond the bounds of the string called `destination_string`.

The function `scanf` was also revised to permit the user to have some control of the total number of characters read in from the keyboard.

The battle between hackers and programmers never ends. When hackers first started to take advantage of the fact that `strcpy` allows us to enter strings of any size into buffers of fixed size, programmers responded by writing the `strncpy` function. Hackers quickly learned that if the source string is longer than the specified number of bytes to be copied, the `strncpy` function does not automatically append a terminating NULL to the string that is copied. Thus, if the programmer is not careful, a new set of hacks can be developed, based on the existence of strings sitting in memory without a NULL terminator.

3.3 <u>System Hardening</u> The role of the system administrator is also very important in detecting anomalous behavior and preventing the escalation of privileges. All executables on a system ought to be periodically reviewed to ensure least privilege principles are followed. This means that users have the bare minimum permissions necessary to accomplish their functions on the system. We introduced the `sudo` command in Chapter 7 but acknowledged while it exists, it would not be realistic for every user to have this permission (as you do currently). The fewer opportunities for privilege escalation that exist will greatly reduce the impact of buffer overflow attacks.

3.4 <u>Technical Solutions</u> Beyond awareness and careful coding, several technical solutions have been proposed.

3.4.1 <u>The non-executable stack</u> This approach forbids the operating system from executing instructions that are on the stack. With this approach, the `eip` register would never be permitted to hold an address that is in the stack's address range (between `ebp` and `esp`). Machine language instructions would not ordinarily be found on the stack (the machine language instructions would be in the text segment of memory), so there is no reason for the `eip` to ever point to an address on the stack frame.

It must be noted that this solution still poses some problems. First, it does not prevent a buffer overflow; rather, it prevents a buffer overflow from running the injected exploit. This approach does not protect against an adversary crashing our machine on a segmentation fault. Also, some highly specialized applications actually depend on having executable code on the stack.

3.4.2 <u>The canary</u> This approach entails placing a specific known value on the stack just prior to the return address. This known value is termed a "canary" (since a canary was used in coal mines to provide an advance indication of a building of dangerous gasses). An attempt to overwrite the return address will necessarily overwrite the canary. Before a function returns to `main`, the canary is checked to see if it's value has been altered. If the canary has been altered, the program is halted. Again, this does not prevent a buffer overflow attack, but if properly executed it will prevent running of the injected exploit.

Hackers have found ways to defeat the use of a canary. First, if known canaries are used (for example, if a canary with value `-1` is always used, the hacker can perform a buffer overflow, overwriting the return address while making sure that the canary is overwritten with the correct value (`-1`). If the programmer uses a pseudo-random canary, the hacker can attempt to read the canary value as part of the exploit, taking care to overwrite it with the prior value.

3.4.3 Address Space Layout Randomization (ASLR) In this technique, the stack is placed in random memory locations, preventing the hacker from easily predicting the location of the return address. We can still inject the shellcode and corrupt the memory but cannot determine a suitable value for the malicious return address preventing the execution of the exploit. Of course the locations of the stack are not completely random, but are usually arranged according to a fixed number of possible options. Starting with their operating system called Vista, Microsoft used ASLR with 256 different possible options for the stack layout and then include a pseudo-random offset within the layout. A common counter-hack (not covered in this class) involves using format string vulnerabilities to determine the return address location. Of course hackers are also studying the various layout options and, eventually but certainly, hacks will be developed for each of the layout options.

Practice Problem 10.2

Briefly describe two technical solutions that have been proposed to prevent a program from being exploited by a buffer overflow.

Solution:

CH. 8 Problems

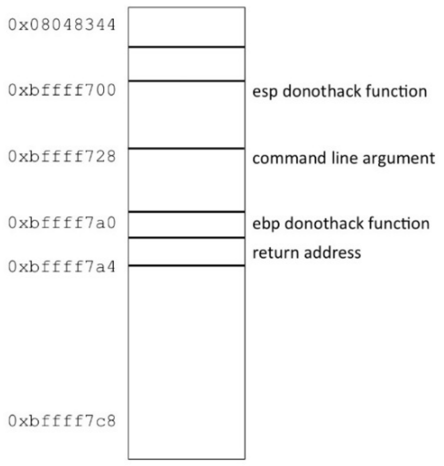
Name:____________________

1. Order these three main components of a buffer overflow exploit as they will appear on the stack:

   shellcode
   malicious return address
   nop sled

2. At some point in a C program called *hackthis.c*, a function named *donothack* is called. A hacker is attempting to take advantage of a vulnerability in the code to initiate a buffer overflow attack when the function *donothack* is called and a command line argument is copied into memory in the function's stack frame.

Analyzing the code and using the debugger on *hackthis.exe* a sketch of the memory layout emerges as depicted to the right (*note*: sketch is not to scale). For each of the following values explain why they should or should not be used as the *malicious repeated return address* in a buffer overflow attack?



(a) `0x0804834b`

(b) `0xbffff700`

(c) `0xbffff740`

(d) `0xbffff7b4`

3. Aside from careful programming and the modification of several specific C commands, list and briefly describe two technical solutions that have been proposed to prevent a program from being exploited by a buffer overflow.

4. Explain why the buffer overflow described in this chapter is much more insidious than the buffer overflows described in Chapter 6.