Ben Sottile

Buffer Overflow Attack

4.1. How are the addresses decided for the following variables a and i, i.e., during the runtime, how does the program know the address of these two variables?

```
void foo(int a) // Stack Frame
{ int x;
      //Stack
      Frame
}
```

4.2. In which memory segments are the variables in the following code located?

```
int i = 0; // Data Segment
 void func(char *str)
{
char *ptr =
      malloc(sizeof(int)); // in
      stack
 char buf[1024]; //updates
      buffer size
int j; //in Stack
static int y;  //In BSS
}
```

4.3. Please draw the function stack frame for the following C function.

```
int bof(char *str)
{
char buffer[24];
strcpy(buffer,str); return 1;
}
```

Return address of function

Str = x ←----------

4.4. A student proposes to change how the stack grows. Instead of growing from high address to low address, the student proposes to let the stack grow from low address to high address. This way, the buffer will be allocated above the return address, so overflowing the buffer will not be able to affect the return address. Please comment on this proposal.

That address will have to be stored somewhere and and instead of having to guess people who want to find the return address will just read where ever you storing the address and overwrite directly.

4.5. In the buffer overflow example shown in Listing 4.1, the buffer overflow occurs inside the strcpy() function, so the jumping to the malicious code occurs when strcpy() returns, not when foo() returns. Is this true or false? Please explain.

It occurs when the foo() returns. Because the return address being overwritten is foo()'s

4.6. The buffer overflow example was fixed as below. Is this safe ?

```
int bof(char *str, int size)
{ char *buffer = (char *) malloc(size);

    /* The following statement has a buffer overflow problem */ strcpy(buffer, str);

    return 1;
}
```

Depends, if the user input is limited sure. If the user exploits the fact that the buffer will try to match anything andputs in a super huge string hoping to overwrite something then no.

4.7. Several students had issue with the buffer overflow attack. Their badfile was constructed properly where shell code is at the end of badfile, but when they try different return addresses, they get the following observations. Can you explain why some addresses work and some do not?

buffer address : 0xbffff180 case 1 : long retAddr = 0xbffff250 -> Able to get shell access case 2 : long retAddr = 0xbffff280 -> Able to get shell access case 3 : long retAddr = 0xbffff300 -> Cannot get shell access case 4 : long retAddr = 0xbffff310 -> Able to get shell access case 5: long retAddr = 0xbffff400 -> Cannot get shell access

I guess its possible they are returning outside of the stack frame so they are no longer a privileged program

4.8. The following function is called in a privileged program. The argument str points to a string that is entirely provided by users (the size of the string is up to 300 bytes). When this function is invoked, the address of the buffer array is 0xAABB0010, while the return address is stored in 0xAABB0050. Please write down the string that you would feed into the program, so when this string is copied to buffer and when the bof() function returns, the privileged program will run your code. In your answer, you don't need to write down the injected code, but the offsets of the key elements in your string need to be correct. Note: there is a trap in this problem; some people may be lucky and step over it, but some people may fall into it. Be careful.

```
int bof(char *str)
{ char buffer[24]; strcpy(buffer,str); return 1;
}
```

(Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code)(Return address of my code)(NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP)(My Code)

4.9. HHH

In this problem, we will figure out how overflowing a buffer on the heap can lead to the execution of malicious code. The following is a snippet of a code execution sequence (not all code in this sequence is shown here). During the execution of this sequence, the memory locations of buffer and the node p, which are allocated on the heap, are depicted in Figure 1. You can provide your input (up to 300 bytes) in userinput, which will be copied to buffer. Your job is to overflow the buffer, so when the target program gets to Line Ì, it will jump to the code that you have injected into the heap memory (assuming that the heap memory is executable). The return address is stored at location 0xBBFFAACC.

```
struct Node { struct Node *next; struct
Node *pre;
};
```

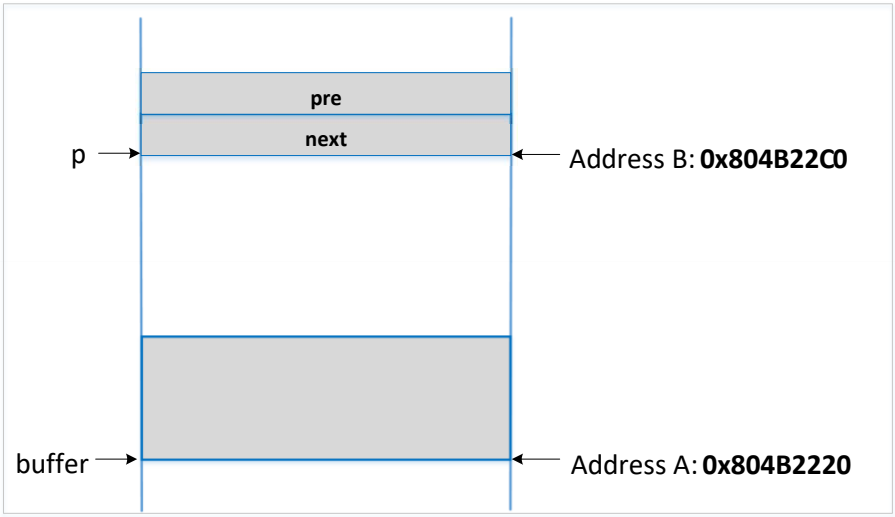

Figure 1: Figure for Problem 4.9.

```
// The following is a snippet of a code execution sequence.

  struct Node *p = malloc(sizeof(struct Node)); struct Node *q;
  char *buffer = malloc(100);

  /* Code omitted: Add Node p to a linked list */

  // There is a potential buffer overflow in the following strcpy(buffer, user_input);

  // remove Node p from the linked list
  q = p->pre;        Ê q->next = p->next;
        Ë

   return;                              Ì
```

Hint: You still want to place the starting address of your malicious code into the return address field located at 0xBBFFAACC. Unlike stack-back buffer overflows, where you can naturally reach the return address field via overflowing, now the buffer is on the heap, but the return address is on the stack; you cannot reach the stack by overflowing something on the heap. You should take advantage of the operations on the linked list (Lines Ê and Ë) to modify the return address field.

This is a simplified version of how a buffer overflow on the heap can be exploited. The linked list is not part of the vulnerable program; it is actually part of the operating system, which uses it to manage the memory on the heap for the current process. Unfortunately, the linked list is also stored on the heap, so by overflowing an application's buffer, attackers can change the values on this linked list. When the OS operates on the corrupted linked 4

list, it may change the return address of function, and trigger the execution of the injected code.

(Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code)(Return address of my code)(NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP)(My Code)

4.10. This problem is built on top of Problem 4.9.. Assume that the structure for Node becomes the following (a new integer field is addeded to the beginning). Please redo Problem 4.9. It should be noted that in Figure 1, the variable p will now point to the area 4 bytes below the next field.

```
struct Node { int value; struct Node *next;
struct Node *pre;
};
```

(Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code)(Return address of my code)(NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP) (NOP)(My Code)

4.11. Why does ASLR make buffer-overflow attack more difficult?

It randomizes memory address so its hard to simply overwrite the return address and or guess the address of your own code.

4.12. H

To write a shellcode, we need to know the address of the string "/bin/sh". If we have to hardcode the address in the code, it will become difficult if ASLR is turned on. Shellcode solved that problem without hardcoding the address of the string in the code. Please explain how the shellcode in exploit.c (Listing 4.2) achieved that.

It utilizes the pointer pushes the /bin and /sh at the very end of the code and since they know the value of the pointer when they push they know the value of the the string they will put in execuv to invoke.

4.13. When you construct the attack string in Problem 4.8., you have many choices when deciding what value to put in the return address field. What is the smallest value that you can use?

The smallest value you use is the address after the return address field any value lower than would mess with the code.

4.14. H

The following function is called in a remote server program. The argument str points to a string that is entirely provided by users (the size of the string is up to 300 bytes). The size of the buffer is X, which is unknown to us (we cannot debug the remote server program). However, somehow we know that the address of the buffer array is 0xAABBCC10, and the distance between the end of the buffer and the memory holding the function's return address is 8. Although we do not know the exact value of X, we do know that its range is between 20 and 100.

Please write down the string that you would feed into the program, so when this string is copied to buffer and when the bof() function returns, the server program will run your code. You only have one chance, so you need to construct the string in a way such that you can succeed without knowing the exactly value of X. In your answer, you don't need to write down the injected code, but the offsets of the key elements in your string need to be correct.

```
int bof(char *str)
{ char buffer[X]; strcpy(buffer,str); return 1;
}
```

(Whatever)(Whatever)(Whatever) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code)(Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code) (Return address of my code)(Return address of my code)(NOP) (My Code)

I figured 4 bytes an address so you can do whatever until you reach 3 bytes then you fire your return address of and some NOPs to ensure wherever your address leads it is likely you code.