Jacob Kilver

Introduction to Information Security

Assignment 1

21 Sept 2016

# Question 1

#### Stack Buffer Overflow

The stack is where data for recently executed code is stored. For instance, short term variables are stored on the stack. A stack buffer overflow occurs when a data is written to a variable that exceeds the amount allotted to in in the stack. Most often this is in the case of user inputs that are not validated before being used.

To understand the stack buffer overflow issue, we have to understand how the stack grows. Let’s assume a stack as in lecture which grows from higher to lower addresses. As variables are created, they are stored on the stack. If data is written to a variable on the stack that exceeds the space allotted to it, the data in the adjacent spots above it are overwritten. An attacker can use this vulnerability to attack just the program (by altering local variables) or the entire system by corrupting the return address to point to malicious code.

As an example, take the code in Listing 1. This program has a vulnerability because it uses the gets() function, which does not validate the size of user input. The program creates a couple of local variables and then waits for user input. However, since the input array is only one character in length, any input greater than one character will overwrite other variables on the stack.

Listing 1: Excerpt of vulnerable program

|  |
| --- |
| int main(int argc, char \*argv[])  {  int var1 = 10;  int var2 = 20;  char userInput[1];  gets(userInput);  …  } |

A diagram of the stack from the example above can be seen in Figure 1. We see that the input arguments for the main() method are pushed to the stack, followed by the return address, so that the OS knows where to return the program execution to after main() completes. Following this the local variables are pushed to the stack. When the user inputs data that exceeds one character, the var1 and var2 stack locations are overwritten. If enough data is read in, the return address will be overwritten. When the program execution completes and the return address is accessed, execution will not return to the calling function. If the return address is not carefully crafted, this will result in moving to an illegal memory location and a segmentation fault will result. If the return address is carefully crafted, program execution could return to a location where malicious code is present that could compromise the entire system.

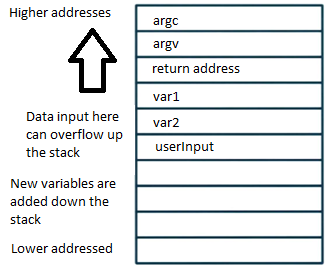


Figure : Diagram of a typical stack

#### Heap Buffer Overflow

The heap is used for longer term storage than the stack. Generally, these attacks are more difficult than stack buffer overflows because there are no return addresses to overwrite. Instead, the heap contains function pointers.

Listing : Heap buffer overflow vulnerable code example [Computer Security: Principles and Practices 3rd ed., 369]

|  |
| --- |
| /\* Record type to allocate on heap \*/  typedef struct chunk  {  char inp[64]; // vulnerable input buffer  void (\*process)(char \*); // pointer to function to process inp  } chunk\_t;  void showlen(char \*buf)  {  int len;  len = strlen(buf);  printf(“buffer5 read %d chars\n”, len);  }  int main(int argc, char \*argv[])  {  chunk\_t \*next;    setbuf(stdin, NULL);  next = malloc(sizeof(chunk\_t));  next->process = showlen;  printf(“Enter value: “);  gets(next->inp);  next->process(next->inp);  printf(“buffer5 done\n”);  } |

# Question 2

# Question 3