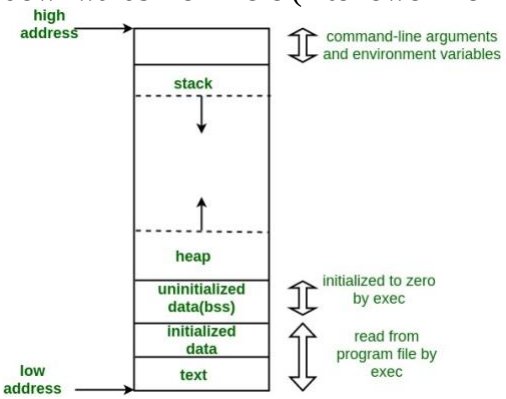
zpanzarino3 903305160

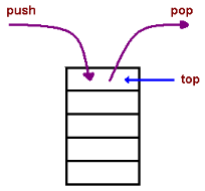
Zachary Panzarino

Stack Buffer Overflow

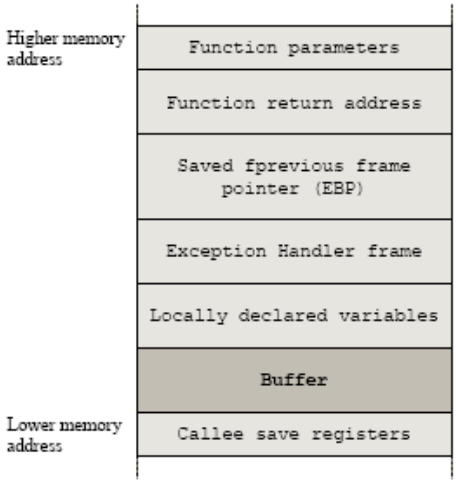
The stack is generally stored higher parts of memory. It is placed at the opposite end of the address space as the heap, giving them both room to grow into the free space. Therefore, the stack grows downwards since it is located at the top of the address space. Specifically, this address might start at 0xFFFF at the top of the stack. The rest of the stack will be written downwards from here (into lower memory). The image below shows where the stack is located.



The stack functions as a last in first out data structure. The most recent data pushed onto the stack will be the first data values retrieved from the stack. This can be seen below.



The stack contains a lot of information about a program, such as the program's function call chain, local variables, parameters, and return values. These variables are things that are essential to executing a program, so the system is consistently reading and writing to the stack while executing a program. A typical layout of the stack can be seen below. This diagram shows where the register values, function arguments, and local variables would be placed on the stack.



All of this information that is stored on the stack is crucial to the control flow of the program. However, the buffer can become so large that it begins to overwrite other aspects of the program,

which is where a security vulnerability is. If there is a buffer of size ‘non-binary’ is allocated by the function, a stack overflow will occur, and essential program information will be overwritten. For example, if a processing word size is 4 bytes, then a data type of 4 bytes can be read from one word. However, if the data type of 4 bytes is not stored at the start of a word, then it will span across multiple words, requiring extra memory read functions. When this is the case, the frame pointer and return address can be changed, returning to an incorrect memory location.

A program that contains a stack overflow vulnerability might look like the following.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char buffer[16];
    strcpy(c, argv[1]);
}
```

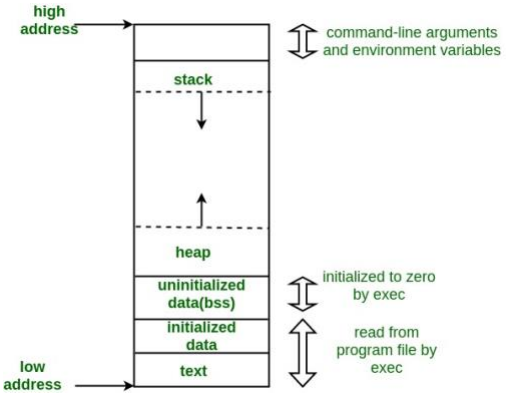
In this program, if the input is more than 16 bytes, an overflow will occur. The buffer and frame pointer must both be overwritten before reaching the return address. Therefore, an input of 24 bytes should result in an overflow. In this case, the buffer would look as follows.

argc	4 bytes
argv	4 bytes
Return Address	4 bytes
Frame Pointer	4 bytes
Buffer	16 bytes
Remainder of Stack	

For an overflow, the stack would be written up over the frame pointer, return address, and arguments. Any input that reaches 24 bytes has the potential to overwrite the return address, potentially launching a different return function.

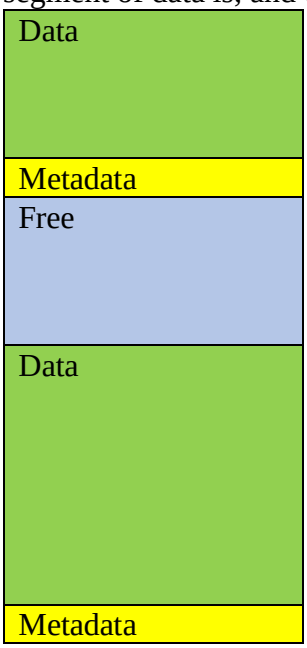
Heap Buffer Overflow

The heap is located opposite the stack in an address space. If the stack is in higher parts of memory, then the heap will be located in lower parts of memory. However, there is still some program information that is stored below the stack. A diagram of where the stack is located is shown below.



The heap is used for managing dynamically allocated memory. This memory can be accessed in a program through malloc, realloc, and free (along with some others). While the stack grows downwards, the heap grows upwards, so that they are both able to grow into the free space. Over time, the heap may become fragmented as certain parts are freed and reused in different parts of

a program. This means that heap memory is not contiguous. In addition, the heap is shared between multiple programs. The heap is structured with metadata that helps explain what each segment of data is, and typically looks like the following diagram.



A program that contains a heap overflow vulnerability might look like the following.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char* chunk1 = malloc(16);
    char* chunk2 = malloc(16);
    strcpy(chunk1, argv[1]);
    free(chunk1);
    free(chunk2);
    return 1;
}
```

In this program, two chunks of size 16 bytes are allocated. If the argument is longer than 16 bytes, it can overwrite some of the data on the heap. Essentially, it will overwrite some of the metadata and data of the second chunk. When doing this, an exploit can make the system believe that chunk 2 has already been freed, causing issues when chunk 1 is freed (because the system merges free chunks together into a larger chunk). Using a negative value for the size of the chunk 2 (overwritten in the metadata) will cause the system to think that it has already freed this chunk. Therefore, the chunks are unlinked from their place in the heap and marked as free. Utilizing this exploit can lead to an unauthorized user running shell code. You can see what this heap might have looked like in the diagram below.

Previous size	4 bytes
Chunk size	4 bytes
Allocated Chunk	16 bytes
Previous size	4 bytes
Chunk size	4 bytes
Allocated chunk	16 bytes
Rest of heap	

Exploiting Buffer Overflow

In order to exploit the buffer overflow, I utilized gdb to find the addresses of the system functions that we needed. To find the address for system, I ran "print system", which revealed that system was located at 0xb7e57190. I used the same method for finding the address of exit, running "print _exit", which revealed that exit was located at 0xb7eccbc4. Finding the address of the shell was a bit harder and required more commands. I ran "info files" to get some information about loaded sections. This gave me a list of files, containing a range of memory addresses and a name to go along with each range. Next, I started searching through these ranges in order to find the shell. To do this, I used the "find" command. I decided to start at 0xb7fde114 – 0xb7fde138, since this was the first of the ranges that had a name containing "/lib", and most executable system files can be found in that directory. I continued searching through the list until I found where the shell was located. The command that eventually revealed this to me was 'find 0xb7ff67c0, 0xb7ffa7a0, "sh"', which revealed to me that the shell program was located at 0xb7ff7e5c. These were the only three addresses that I needed to run the exploit, so the next step was to modify "data.txt" in a way that would execute the intended exploit. First, I started with 21 lines containing "aaaaaaaa" to fill up the buffer and overwrite the frame pointer. It is important that these values are less (in hex) than the address values so that they are placed beforehand when being sorted. Next, I used the address of the system function to replace the return address, having this function called rather than the correct one. The function argument for this call would be two lines down, so I put the address of "sh" two lines below that, causing it to be passed as the argument to the function, launching a shell. Finally, I put the address of exit between those two in order for it to properly exit from the original program without causing a segmentation fault. The output of my exploit can be seen below.

zpanzarino3 903305160

```
ubuntu@ubuntu-VirtualBox:~/Desktop$ gcc sort.c -o sort -fno-stack-protector
ubuntu@ubuntu-VirtualBox:~/Desktop$ echo $$
2362
ubuntu@ubuntu-VirtualBox:~/Desktop$ echo $0
bash
ubuntu@ubuntu-VirtualBox:~/Desktop$ ./sort data.txt
Current local time and date: Tue Sep 17 19:40:01 2019


Source list:
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xb7e57190
0xb7eccbc4
0xb7ff7e5c

Sorted list in ascending order:
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
b7e57190
b7eccbc4
b7ff7e5c
$ echo $$
2512
$ echo $0
sh
$ exit
ubuntu@ubuntu-VirtualBox:~/Desktop$
```

Mac, OSX Mojave, 64 bit

zpanzarino3 903305160

Open Question

Both return-oriented programming and jump-oriented programming are code-reuse attacks that rely on using code that is already loaded into memory in order to exploit a vulnerability. After taking control of the call stack, an attacker can utilize “ret” statements in programs that are already in memory, essentially executing code in a specific sequence. These sequences of code are referred to as gadgets. Without injecting any code, an attacker can execute Turing-complete behavior through these sequences. However, jump-oriented programming does not require usage of the stack or “ret”. It uses a dispatcher gadget to dispatch functional gadgets, except these gadgets rely on an indirect branch at the end rather than “ret”. Since these attacks utilize instructions already in memory, they can be used to bypass security defenses such as code-signing and executable space protection. An attacker might use jump-oriented programming because, unlike return-oriented programming, it does not rely on control of the stack and modifying the return address. Stack vulnerabilities are highly monitored and secured due to many known potential vulnerabilities, so this attack may be easier to execute than return-based programming.

Works Cited

Arpaci, and Dusseau. “The Abstraction: Address Spaces.” *Operating Systems*, 2019, pp. 1–9, pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf.

Bletsch, Tyler, et al. “Jump-Oriented Programming: A New Class of Code-Reuse Attack.” www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf.

“BUFFER OVERFLOW 6: The Function Stack.” *Tenouk's C Tutorial*, www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html.

Ferguson, Justin N. “Understanding the Heap by Breaking It.” *IOActive*, www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf.

“Memory Layout of C Programs.” *GeeksforGeeks*, 30 Jan. 2019, www.geeksforgeeks.org/memory-layout-of-c-program/.