Joshua Reno
CS 4235: Intro to Information Security, Fall 2018
Project 1: Buffer Overflow

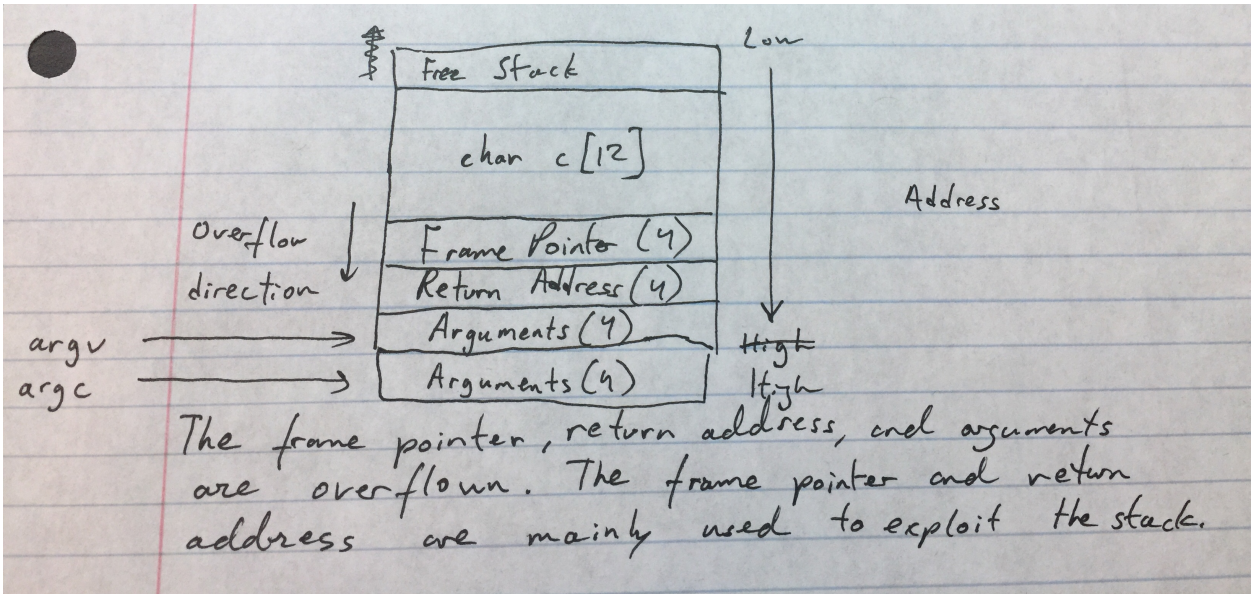
1. Stack Buffer Overflow

a. Memory Architecture

The stack is located in a reserved part of memory within a thread. Within memory, the stack grows upward where the memory address decreases. From bottom to top, the stack contains the return address, the frame pointer, and parameters. When functions call each other, register values that need to be saved when returned to are saved as are parameters into the called function. These variables are stored in the stack frame. Local variables are also stored in a functions stack frame, a structure within the stack. Regarding control flow, when one function calls another, the parameters of the called function are first saved. Then the return address is pushed onto the stack. This is all done by the callee function. The called function push the frame pointer, sets the frame points, allocates space for local variables, runs the function, resets the frame pointer, pops the old frame pointer, and runs the callee function. When a buffer does not align with the stack, the frame pointer and return address can be overrun, returning to an illegal memory location. When the stack grows, it takes up space in free memory.

b.

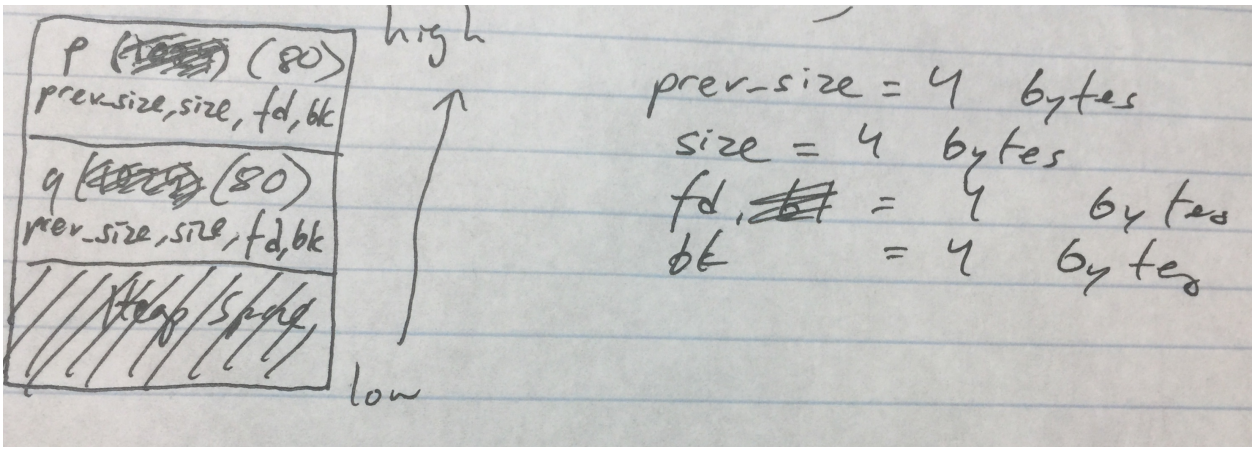
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char c[12];
    strcpy(c, argv[1]);
}
```



2.

a. Both the stack and the heap are stored in a computer's RAM. The heap deals with dynamic memory allocation (malloc, calloc, etc). While the stack is located in higher memory address space, the heap is located in lower memory address space. The stack grows down while the heap grows up into empty memory space. Unlike a stack, there is no order in the heap. Both the stack and the heap are allocated by the operating system.

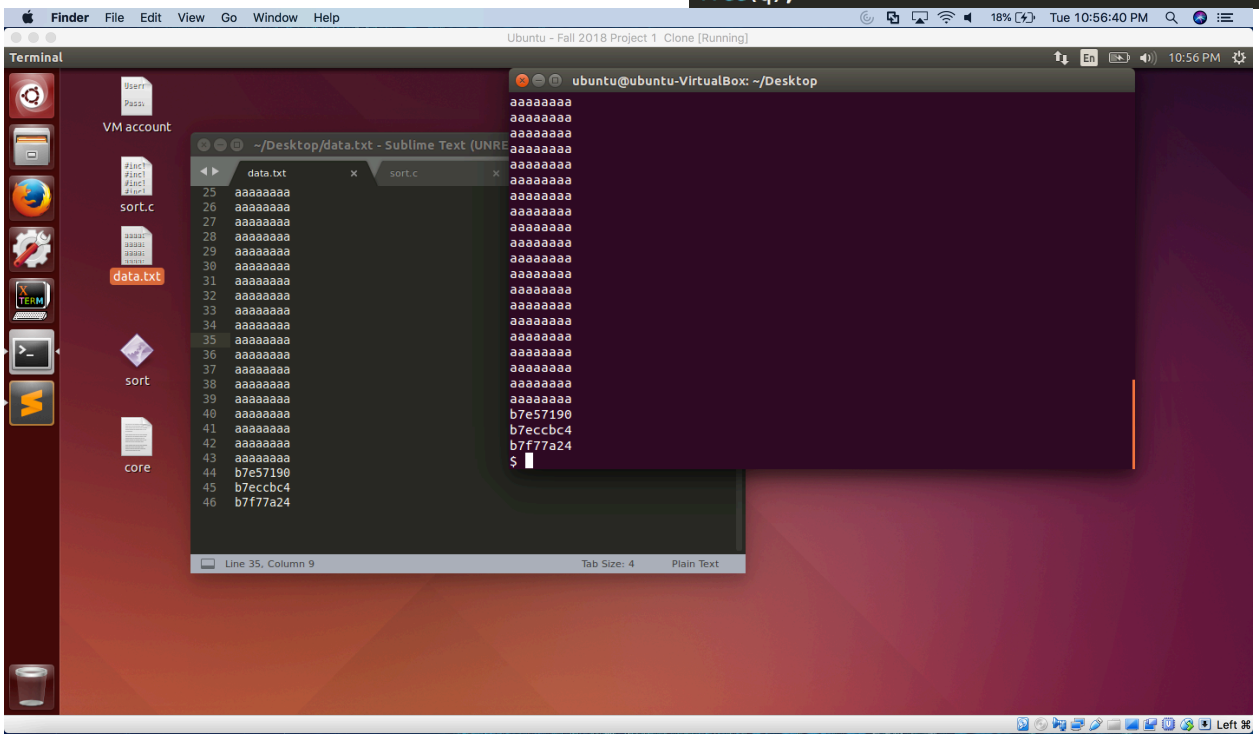
b.



p and q are adjacent in memory. A successful exploit will fill p's data with a "fake chunk". This will adjust the fd and bk pointers. The exploit will overflow data into q's header that sets prev_size and size. When q is freed, the fake chunk will also be freed and unlinked. Now that the pointers have changed for p, the attack can change p's data.

```c
struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[10];
};

unsigned long long *p, *q;
struct chunk_structure *fake_chunk, *chunk2_hdr;
char data[20];

p = malloc(0x80);
q = malloc(0x80);

fake_chunk = (struct chunk_structure *)p;
fake_chunk->fd = (struct chunk_structure *)(&p - 3);
fake_chunk->bk = (struct chunk_structure *)(&p - 2);

chunk2_hdr = (struct chunk_structure *)(q - 2);
chunk2_hdr->prev_size = 0x80;
free(q);
```

3. MacOS 10.13.6 64 bit



4. Code reuse attacks involves using code often in the standard library and executing it. Code reuse attacks use buffer overflow (as an example) to take control of the control flow of a program. Return oriented programming (ROP) uses code sequences terminating in “ret” instructions are executed using control of the stack. Return oriented programming can b easily detected due to reliance on the stack and consecutive execution. Jump oriented programming (JOP) uses neither the stack nor “ret” instructions. Just like ROP, JOP uses chains of primitive operations although it ends in an indirect branch rather than a “ret”. JOP relies on “dispatcher gadgets” or jump-oriented gadgets in the GNU libc library. These indirect jump instructions are crucial because almost every (known) ROP detection method relies on “ret” and the use of the stack. A “jmp” gadget, which makes it hard to control program flow, is replaced with the dispatcher gadget discussed above, which natively determines the next functional gadget to use. This makes JOP possible since the “jmp” ending in a functional gadget reallocates control of the program flow to the dispatcher gadget.

Works Cited

Bletsch, Tyler, et al. “Jump-Oriented Programming: A New Class of Code-Reuse Attack.” *NUS Computing*, National University of Singapore, 2011, www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf.

Kapil, Dhaval. “Heap Exploitation.” *Heap Exploitation by Dhaval Kapil*, Dhavalkapil.com, heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html.

Ying, Kailiang. “Heap Based Buffer Overflow --- Introduction.” *Heap Overflow*, Http://Yingkailiang.blogspot.com, 8 Jan. 2013, yingkailiang.blogspot.com/2014/01/heap-based-buffer-overflow-introduction.html.