1) **An explanation of what happens in the stack when a function is called, including a description of purposes of the rbp, rsp, and rip registers as used in the Intel x86 architecture.**

Whenever a function is called, the stack evaluates and pushes parameters onto the stack, it then pushes return addresses onto the stack, and then it branches into the destination function. Rbp is the base pointer for x86. it keeps local variables and function parameters from the rsp register so that whenever rsp is changed, the offset rbp still contains those values. Rbp is assigned to rsp's current address, this sets the new stack frame. Rsp also points to the last item on the stack. The RIP or the register instruction pointer, points to the next instruction to be executed.

2) **A short description of what (in general) happens during a buffer overflow attack.**

An buffer overflow attack is when you can use the lack of buffer checking, overload the buffer and jump to whatever instructions you want directly.

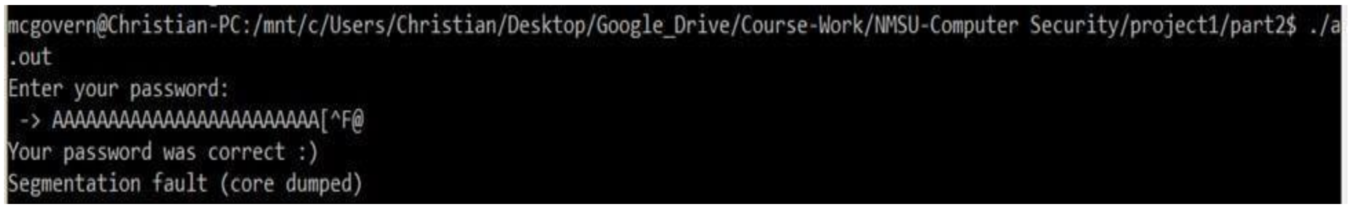
3) **An explanation of why the code used to take input in bof.c is dangerous, and what procedure should be used instead.**

The code in bof.c uses gets(); which is a dangerous function and is now defunct, I would use fgets(); instead. I also would have some if else logic on how much was inputted to people couldn't even try to overflow the buffer.

4) **A list of commands you used in gdb in order to examine the stack, as well as their output at each step.**

I did not need gdb to complete this assignment. All I needed was the a.dump to tell me where the success function was in memory, then I could overload the buffer with 24 characters to overload the buffer(8 + 16) then the ascii value of the hex value location of the Success(); function. 40065b turned into [^F@ and the program successfully jumped to Success.

5) **A copy of the input you provided to the program in order to perform the exploit.**



```
mcgovern@Christian-PC:/mnt/c/Users/Christian/Desktop/Google_Drive/Course-Work/NMSU-Computer Security/project1/part2$ ./a
.out
Enter your password:
 -> AAAAAAAAAAAAAAAAAAAAAAAA[^F@
Your password was correct :)
Segmentation fault (core dumped)
```