**Nathaniel Todd – ntt6/3922932**
**Project 2**

## Program 1

### Method:

The first thing I did with ntt6_1 was run it through mystrings.c. I noticed that I was able to pick out all the function names with ease. By quickly looking over the functions I noticed there were several functions similar to the puzzle lab we did in recitation. Such as chomp and string comparison functions.

I then ran ntt6_1 with gdb and placed a breakpoint at main. I used disas to view the assembly code and placed a breakpoint right before it calls printf. I was surprised to find out I did not hit this breakpoint. I then placed a breakpoint at the cmp function. When running the x/s $esi to check to see what repz was operating on I received the string: VdDuSVpYuQllUFHWHeJxXqLbThu. This could possibly be the passcode but when passed to the program, it doesn't accept it. From here I started to look for my string to see if it was being modified.

Curios about the repz I put a break before and after it. I found the string: jVdDuSVpYuQllUFHWHeJxXqLbThu and my input were present before repz but after repz both had one less character. I tried the string from the second paragraph with various characters appended to the beginning but to no avail. When passed " jVdDuSVpYuQllUFHWHeJxXqLbThu" the program accepts the input. After researching repz and CMPSB it seems that the program is comparing bytes of the strings at a time and the characters are terminating afterwards.

### Solution:

jVdDuSVpYuQllUFHWHeJxXqLbThu

## Program 2:

### Method:

With this program I started by using the objdump -D to dump the assembly into a text file. This made it much easier to step through the program while reading the addresses and code from a text file. I then loaded the program with gdb to start exploring it.

I found the main and set a breakpoint at the beginning of the main, right before the call of c. I saw that my input was stored into ebx and eax. When I hit the breakpoint after c, I was able to see my string was in ebx and edx with the \n character truncated. This lead me to believe that c removes the new line character because inside c you can tell it is comparing the last byte of the string to 0xa, which is the new line. This state remained the same after passing the p call. The only difference is that edx no longer contains my string. Directly after this there is a comparison of eax to 1, this could mean that p is responsible for testing the string in some way.

From here I inspected the different functions being called. The first function I was able to make out was s which was a string length function. After verifying my suspicions about function c, I attempted to figure out p.

Through inspecting funciton <p> I found that it's putting my string in several registers but it stores the last character into edi. As I stepped thorugh <p> I found that there was a branch that could not be accessed unless the length of my string is more than one. This allowed me to return and access a new branch of the main. Unfortunately, it almost immediately compares the

string length to 0xd or 13. Therefore I inferred my string needed to be larger than 13 characters but I needed to find another way to exit p and access the same branch of the main.

I looked more closely at the section where of p where the last character is stored. I noticed it was actually incrementing through the string. So each time it is taking the last letter, second to last letter, and so on and comparing it to the first, second, etc character of the string. Therefore I inferred that the password pattern was a palindrome of 13 or more characters.

**Solution:**

Palindromes larger than 13 characters, any characters can be used including whitespace characters. Examples:

1111racecar1111 or 123456dad654321

**Program 3:**

**Method:**

I started off with gdb this time around but noticed I could not place a breakpoint at the main because it couldn't find the main. I then ran the program through mystring.c but it didn't yield anything terribly useful. that this program ran through the compiler differently and possibly didn't generate symbol tables. Therefore I used objdump -D to dump the assembly into a text file.

I started by researhcing a funciton I found called __libc_start_main(). The function itself was very small but it sounded promising. I found that the "`__libc_start_main()` function shall initialize the process, call the *`main`* function with appropriate arguments, and handle the return from `main()`". So it seemed like it calls another function to act as the main. Exploring that funcition didn't seem to get me anywhere.

I started to look through the .text section because of it's size. I placed a break point at the first address of the .text and ran the program in gdb. I hit the break almost immediately! After doing some research it seem like this is the entry point of the program. When I went to test for this break the program kept waiting when I provided "hello" for test input. I played around with this input and discovered it require at least 10 characters.

I noticed that the .text section calls the aforementioned main and tolower. Which made me think that the pass phrase is independent of case. Addionally I noticed that there were several counters comparing 0xa (10), 0x5, and another 0x5. The 10 was familiar from the input but the fives are new. They could possibly be counting attributes of the password. As I stepped through the main, I noticed that the addresses are not in the dumped assembly.

As I looked through the lower end of .text. I placed a break after the main call but I noticed that I only hit *after* the program tells me my password is wrong. This made me scroll through the lower lines of .text that's when I noticed the push structures are following the calling convention of "push %ebp, mov %esp,%ebp, push %ebx". I now believe that the main is calling these functions in the .text.

Through exploring these functions in the .text I noticed it was iterating through the first 10 characters of the string and comparing characters. After taking a closer look at the 0x5 counter comparisons I realized it was counting the numbers of characters and numbers.

I then entered the password abcde12345 and it worked! To get a better idea of the requirements I set breakpoints in these functions and stepped through with different number and character combinations. By checking memory as it stepped through I saw it only accepted

numbers 1-5 (inclusive) but the character comparison didn't exclude any characters.

**Solution:**

A password containing 5 characters that aren't numbers (including whitespace characters) in the first 10 digits and 5 numbers between 1 & 5 (inclusive) in the first 10 digits. Examples:

abcde12345 or  a1b2c3d4e57777