Nick Petro (nap67@pitt.edu)
November 1, 2016
COE 0449
Project 2 Write-up
Wednesday 10am Lab Section

nap67_1

1.1 – Procedure

The first program was fairly straightforward and I did not really get tripped up in trying to find the password. I began by running the file through the **mystrings** program as recommended by the description document. I then scrounged around the resulting string output until about midway through I found a promising set of strings that looked like this "**MvfeqpudzNkTBMVLpIIUQxMdixJSorry! Not correct!Congratulations!**" which I figured looked quite promising. I took the string "**MvfeqpudzNkTBMVLpIIUQxMdixJ**" which came right before the sorry or congratulations and tried that as the password. It worked! I ran the mystrings program on it a second time to ensure the string did not change and then ran it one more time with the same password and it worked. This led me to conclude that it is indeed the password.

1.2 – Solution

The output of my mystrings program allowed me to find the correct password: "**MvfeqpudzNkTBMVLpIIUQxMdixJ**".

1.3 – Other Notes

- I attempted to use the mystrings program rather than jumping straight into the debugger because I felt it was the simplest approach and was also recommended by the lab description. This turned out to work for this program only and neither of the others, although I did try running the mystrings program on the other two just to be sure.
- Because this was the largest program (with a much larger file size), in addition to containing explicit names of functions, I think the executable was likely **statically linked** and left with an **intact symbol table**.

nap67_2

2.1 – Procedure

I began attempting to find the password for this program the same way I did for the first one, by running it through the **mystrings** program. I had no luck with that on this one, and quickly knew that using **gdb** was the only option to move forward. After entering the debugging environment, I first ran '**disas main**' and was met with a relatively short list of assembly instructions and no clearly named functions. I then ran the disas command on each of the functions and began **analyzing the assembly code**.

From there I noticed a couple things while trying to determine where to set my breakpoints. This first was at **0x0804853d <+75>** (main <+75>) which read "**cmp $0xe,%eax**". Immediately following this was a **jump-if-less-than instruction** which I took to mean that the password would need at least 0xe (15) characters. From there I set a number of breakpoints and went on the check a number of values at each of them running both '**print**' and '**x/s (char*)**' or '**x/f (char*)**' for **$eax, $ebx, $ebp, $edx, $esp, $al**. As it turned out, $al contained the ascii value for 'a' and was being compared to my string input. I took this as a possible solution and so I attempted a 15 character long string of a's like so "**aaaaaaaaaaaaaaa**". This worked!

After running through the program again, I noticed my string characters being compared against other asci values for different vowels. I tried different passwords with different vowels and they all worked so long as they were at least 15 characters long and only contained vowels.

2.2 – Solution

My initial solution was: "**aaaaaaaaaaaaaaa**", however after delving a little deeper into the assembly code (and more attempts), I realized that the password could be **any string longer than 15 characters so long as it contained only vowels**.

2.3 – Other Notes

- Due to the files very small size, it is clear that it is not statically linked. In addition, I feel as though there had to be a minimal symbol table based on the lack of function names and calls or jumps to addresses such as **0x8048348 (fget)** instead.
- Running mystrings on this program provided no real help. I found a few strings around the "Sorry/Congratulations" outputs that I tried, but with no luck, so I knew that debugging with gdb was necessary.
- For this program I used the '**ni**' command in gdb a lot to step through the instructions and check key values at each of the steps, which was extremely helpful.

nap67_3

3.1 – Procedure

I began this program just as the other two by running it through the mystrings program. As expected, this produced no fruitful results, and in this case, not even a viable string to try in my opinion. So I then attempted to 'disas main' just as in the last one, and to my surprise I was met with "**No symbol table is loaded. Use the "file" command.**". This caused me to scour the '**help**' pages to find a viable command to help me out. I found that I could run '**info file**' and be met with an address for the "**Entry Point:**". I used this address as my breakpoint, and went to work running 'disas' on the '**address,+50**' which showed me the next 50 addresses and gave me something to work off of. I quickly realized that there were a series of calls to different functions that I had to work through to get to the section of assembly that was doing the real work for the password. I followed the same pattern of adding a breakpoint and running the 'disas' command followed by the address,+50 until I got to a section of code with a <**printf**> and <**put**> calls which looked to me to be promising.

From there I found the approximately 20 lines that were looped through a number of times, 10 to be exact, which I determined by using the '**ni**' command. This lead me to believe that the **password had to have at least 10 characters**. From there I noticed a few lines that formed the "meat" of the functionality and consisted of a **subtraction of 0x31** from each character value of my input, followed by a **comparison to 0x4**, which resulted in a **jump if it was above 0x4**. This jump meant that it would skip an important addition later on, for a final comparison that determined if "Congratulations!" was printed in some capacity. The final comparison **compared 0x8 to a value** that was only added to if the value of my input (after subtraction) was less than 0x4. **Only if the value was equal to 0x8 would the "Congratulations!" string be printed**. So I set off by inputting a string of eight 4's followed by 2 of any key input, "**44444444gg**". It worked!

I then looked at the code a little closer and noticed **another subtraction of 0x1** before some of these other instructions and so I tried, "**55555555gg**". Again it worked. I upped it again to, "**66666666gg**", and had no luck which meant that I had figured it out. After ensuring that I didn't miss any other conditions, and running through a number of other tests, I determined that I had in fact discovered the formula for the password. It could be longer than 10 characters, but **only the first 10 mattered**. They could consist of **8 numbers between 0-5 and 2 of any other value (including newline) in any order** and it would unlock the program.

3.2 – Solution

My initial solution was: "**44444444gg**", but after more deliberation and a bunch of 'ni' commands, I determined the formula was **any string whose first 10 characters consisted of exactly 8 numbers between 0-5 and 2 of any other character (including a newline)**.

3.3 – Other Notes

- This file was the smallest of all because there was **no symbol table to load**. It may still have been statically linked because there were explicit calls to '**getchar**', '**tolower**', and '**printf**' while I was performing the disassembly.
- I realized a little while after finishing the project that I could have used **objdump** and redirected the output to **a.out** to look at all of the assembly code, but it was kind of a moot point because I had finished, although I will keep that in mind for the future.
- There were quite a few "**jmp**" commands and this program was by far the most difficult to figure out. I would say it just goes to show how **a lot of jumps can produce very difficult to read code!**
- I found the assembly code was easiest to follow when performed in **reverse**, so for this program I worked backwards to find which jump (or lack thereof) lead me to the final printf that I wanted. From there I worked through the loop in reverse and it was truly **reverse engineering** that allowed me to solve this program.
- I had to really understand the order of the **logical and arithmetic instructions** in this program to be able to find the solution. I actually converted some of the assembly to **pseudo coded c** so that I could understand what the program was doing, and in the end that is how I was able to recreate and understand the formula.