Project 2 Write-Up

Zach Sadler
zps6@pitt.edu
Thursday 3pm Lab Section

zps6_1

1.1 Procedure

For the first program, I tried the same procedure as the puzzle lab. I tried **'disas main',** which then printed out a listing of the assembly code. From there I thought it would be helpful to locate and examine the '**call'** to '**strcmp**.' So, I set a **breakpoint** at the location of the call to strcmp, and then did **x/s (char *)** for both of the arguments of stcmp (based on values being pushed to small offsets of the **stack**), with **x/s (char *)0x8090b08** providing the correct answer.

1.2 Solution

My call of x/s (char *)0x8090b08 gave the correct answer:
qALOkANiONTVajBUqPMgnZsX.

1.3 Post-Mortem/Notes

- I was fairly confident this method would work, since I figured there would be at least one program which had a simple strcmp of a string literal, just as in the lab.

- Based on the huge file size (over 60 times larger than the next program), and the explicit names of functions (like strcmp), I think the executable was very likely **statically linked** and left with an intact **symbol table**.

- After solving through the x/s method above, I realized that I could have also used the **mystrings**.c program, which reveals the following string somewhere in the middle of the program:
  u!@9?SuW?@D=?XZS?WVS?[^_?WVSQ?tAP?[^_?WVS?[^_?[^_?**qALOkANiONTVajBUqPMgnZsX**Sorry! Not correct!Congratulations!
  Unlocked with passphrase %s
  Had I ran the mystrings program on this file, I would likely have noticed the strange string before the correct/incorrect results.

zps6_2

2.1 Procedure

For the second program, I initially tried the same procedure as before, but had no luck. Instead when I did 'disas main' I was met with a very short list of assembly instructions. The first few did arithmetic operations, and while at first that looked promising I instead decided to follow the call to '**d'**. I did **'disas d'**, and found calls to '**c'** and '**e'**, as well as other, unnamed functions. Instead of **'e'** I found a call to **'s'.**

Armed with these strangely-named functions, I set a number of breakpoints and began the same method of using x/s (char *), on **$eax, $ebx, $edx, $edi, $esp,** and **$esi** as I continued through my breakpoints, trying to read the arguments passed to different functions or other interesting calls**.** This allowed me to see potential strings in those registers, and while some looked interesting, I eventually found a very promising string, which turned out to be the solution.

2.2 Solution

Eventually I did **x/s (char *)$ebx** during **breakpoint 0x080484b4** which gave the correct answer: **3.141593**.

2.3 Post-Mortem/Notes

- This file was significantly smaller than the last one, and it seems clear that it is not statically linked, because of the lower file size. In addition, I would say there is a minimal symbol table, based on the lack of clear names for function calls in some instances (instead just calls to 0x0480409b or likewise).

- Use of mystrings on this executable provided no help at all, just a few lines of seeming garbage. While I did try a few choice strings (since there were so few to choose from), I figured a more advanced technique would be needed.

- After the fact, I learned about the gdb command **'layout next'**, which can be used successively to enable the user to have a three part screen with registers at the top, then assembly code, then gdb command line. I think that using this would have been much easier than spamming x/s (char *)$eax, $ebx, … but then again I do not think it would automatically convert the register values to strings so it's hard to tell.

zps6_3

3.4 Procedure

For the third program, I laughed pretty hard at the beginning. My initial call to 'disas main' was met with "No symbol table is loaded. Use the "file" command." After searching through the '**help files'** page and realizing that it probably wanted me to load a symbol table which I just didn't have, I tried a different approach. Instead, after digging through the '**help data'** command I found the '**print'** and '**inspect'** commands.

I attempted to do '**print inspect'**, but accidentally entered that as my password… which I was surprised to see was correct!! So I laughed a long while, and then started examining more closely. It turned out that only the **first ten characters matter**. Then I discovered that the phrase *int012345 worked when * was one of **'q', 'w', 'e', 'r', 't', 'y'.** Through more trial and error, I determined that it would work as long as the **first two characters** were taken from 'qwerty', with no repeats. Then I discovered my final solution:

3.5 Solution

Any string whose first ten characters contain exactly 2 non repeating characters from: 'q', 'w', 'e', 'r', 't', 'y'

3.6 Post-Mortem/Notes

- This file was smallest of all, because (as noted), no symbol table was loaded. I think there is a chance this executable was statically linked, since in the **objdump** there is explicit mention of '**getchar'**, '**tolower'**, and '**printf'.**

- As I mentioned, I did an **objdump** which I redirected using the **shell command >>** to a file a.txt. Using this I was able to read through the assembly code even though I could not disassemble the executable.

- Just as in the last one, use of mystrings on this executable provided no help at all, just a few lines of seeming garbage.

- Although I initially found my answer through guessing and some luck, I did go through later using 'layout next' and '**objdump'** commands to read over the assembly code. I figured that the solution would not be a strcmp with a string literal since so many possible solutions existed. Instead, I spent time looking **for logical and arithmetic operations**, and special properties of the ASCII codes for 'q','w','e','r','t','y'.

- Translating to binary:
  q 0111 0001
  w 0111 0111

e 0110 0101
r 0111 0010
t 0111 0100
y 0111 1001
did not reveal much, nor did examining some of the 'and' and 'add' literals:
add 0xd08b6
1101 0000 1000 1011 0110

and 0x8000
1000 0000 0000 0000.
jne 0xb48794
1011 0100 1000 0111 1001 0100

- I was able to figure out much of what the assembly code did, but it was just too convoluted and had too many pure '**jmp**' commands to follow (a good example of why **goto can produce difficult to read code**!). If I had to guess, I would say there is liberal use of goto in this program.

- I was not able to find the solution through direct interpretation of the assembly code, but I was able to determine the pattern of the solution through deductive reasoning and careful thinking.

Thanks for reading.