CS 449 Project 3:

**Password 1**: SGNayMCVXcaQRcIDleTwMHY

My approach to this first executable was to try and disassemble the program in gdb.

When I performed disas main, I noticed there were a bunch of hexadecimal addresses. I thought that if I tried inspecting the address of the strings via x/s.

First, I tried inspecting the contents of eax, which yielded: 0x80d691c <stdin>: ""

Next, I tried inspecting the contents of esi, which yielded:

0x80b388c <\_\_dso\_handle+4>: "SGNayMCVXcaQRcIDleTwMHY"

I thought, huh, “SGNayMCVXcaQRcIDleTwMHY” seems kinda unusual, maybe it’s the password!! So, I tested the string, and it worked!!

**Password 2**: 3.141593

My approach to the second executable was similar to the first executable.

First, I disassembled main using gdb.

What I found were some pushes, a call to some function, followed by some pops.

So, I figured, since there’s a hexadecimal address with the function call, if I did disas 0x8048545, + 100, I would be able to see more of the assembly code.

I notice there’s a test followed by a jump… So, I figure that if I put a breakpoint on the jump address 0x80485b6, and inspected some variables, the password would be stored in some variable…

Immediately, I think, “hey $esi held the password last time, so maybe it’ll hold it this time too!!”

So, I did x/s $esi and it returned the string: 0xffffcf58: "3.141593"

I enter "3.141593" and it’s correct!!

**Password 3:** Any 16-character string, where exactly 9 of the characters are “9,0,4,c,s,” where “9,4,0,c,s” can be repeated.

Note: Any string exceeding 16 characters will be spliced into a substring containing ONLY 16 characters.

A more in-depth analysis of the x86 assembly code can be found here:

<https://pastebin.com/WGG9mzb1>

**Attempt 1:**

My first approach was similar to the approaches to get passwords 1 and 2. When I ran gdb, followed by disas main, I got a message saying: No symbol table is loaded.

So, I’m thinking, “huh, there’s no main… so maybe mystrings will help”

I go and run ./mystrings gol6\_3, and I get a list of strings, what catches my attention immediately are getchar and \_\_libc\_start\_main.

This provides an inkling about the password, it seems to be involving some search for certain characters in the password…

I decide to disas \_\_libc\_start\_main and notice the assembly code is rather short, so I decide to disas 0x8048304, +100, and when I perform x/s on all of the hexadecimal addresses present, I’m stuck in a corner, so I decide to scrap this idea, since it seems to be getting nowhere.

**Attempt 2:**

I’m thinking “hey, since gdb didn’t work, maybe I should try inspecting all the contents in the file!!” Straight away, I’m thinking that objdump will clue me in on some hints about the passcode.

I spend a couple hours analyzing and figuring out which flags I want to put with objdump, and I deduce that objdump -M intel -d gol6\_3 is the best bet.

I get a huge chunk of assembly code, which overwhelmed me at first, but upon closer analysis, I realize, “hey this kinda looks like MIPS!!” I eye in on the .text section of assembly code and start making some connections to MIPS.

In MIPS, the .text section is where all of the local variables are allocated, so this must be where I can find some indication of what the passcode could be!!

Looking more in-depth into the .text, I notice there is a function that seems to be looking at the input of the user.

Within this function, I notice that ebp and ebx are pushed onto the stack, and with some googling, I figure out that ebp is analogous to the frame pointer in MIPS.

As I delve more into the code, I notice that there are two local variables that are allocated by moving the ebp down. These two local variables are initialized to 0, which lead me to believe they are counters.

I notice that one of the local variables, 0xc is incremented and compared to the 0xf (15), which leads me to the conclusion that 0xc is a loop counter, and terminates when the loop counter is 15. In other words, this loop terminates after 16 iterations.

This fact alone leads me a major clue: The password must be 16 characters long.

Diving past this loop, I notice the counter is reset and is used to iterate through the user-inputted string to search for specific characters.

I also notice inside of the loop, there seems to be another nested loop that there are a bunch of compare statements, supplemented with a bunch of jump statements.

I notice cmp eax,0x39 followed by a jump statement that increments the other local variable, 0x10. This is the same for cmp eax,0x30, cmp eax,0x34, cmp eax,0x63, and cmp eax,0x73.

As I search in the ASCII table for the values, I find that 0x39 = 9, 0x30 = 0, 0x34 = 4, 0x63 = c, and 0x73 = s.

Based on this information, I begin to observe that 0x10 is serving as a count to keep track of these how often certain characters appear.

What really tipped me off was an equivalent cmp eax,0x39, followed by a different jump statement, which printed the failure string. This led me to the conclusion that if the user-inputted passcode was missing any of the characters “9,0,4,c,s” it would not be accepted.

Looking past these series of jump and compare statements, cmp DWORD PTR [ebp-0x10], 0x9 stood out. This comparison was supplemented by a jump, jne 80484bb <puts@plt+0x167> which printed the failure string. On the other hand, if the counter was equal to 9, the success string would be printed.

All of these findings bring me to the following conclusion:

Any password that is 16 characters long, where 9 of the characters are “9, 0, 4, c, s”, where “9, 4, 0, c, s” can be repeated will be accepted as a password.

However, the password must contain “9, 0, 4, c, s” or it will not be accepted…

The following are examples of valid/invalid passwords:

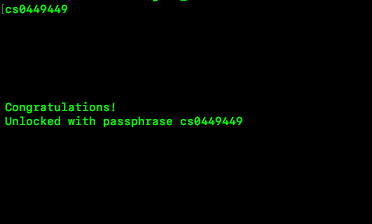
**VALID:** “99004ccssabdefgh”, “cs9\_world9904cs\_”, “449cs0449LOVEIT!”

**INVALID:** “I\_HATE\_CS0449”, “JAVA\_IS\_BETTER\_THAN\_C”, “cs0449\_is\_better\_w/o\_jarrett”

**There’s also some weird behavior:**

1) If the user has 9 of the specific characters, but is less than 16 characters, the user will be prompted to press the [RETURN] key until the string is 15 characters, long, however this will be mirrored to the number of characters it takes to get to 16-characters after the congrats string is printed.

For Example: With a 9-character password…



Since it the password is 16-characters long, the user is prompted to enter [RETURN] 6 times (15-9) to get to 15-characters, and the effect is mirrored so there are 7 new lines (16-9) to get to 16-characters.

2) If the user enters a string greater than 16-characters long, it will be truncated such that the string is 16-characters long.

For Example: With a 20-character password…



449cs0449LOVEIT!rrrr is truncated to 449cs0449LOVEIT!.

­­