**CS449 Project2 Write-up**

Jingxia Zhu(jiz119)

**Executable1:**

**Procedure:**

* **Attempt1:** All programs are written in C so there must be a main function. In gdb, I then placed a breakpoint at main and did a disassembly. In disassembly, I found the following:

repz cmps BYTE PTR ds:[esi],BYTE PTR es:[edi]

This instruction is doing repetitive comparisons for every byte in the two registers, $esi and $edi. I assume it is doing the comparison of the correct password and the user-input string. The register $esi may have the correct password and the register $edi may have the user input. But the string in $esi is “WyisPlsLJaauEOyBEfNWLgB” and this is not the correct password after I tried. I then randomly typed in a password string “abd” and found the string in $edi became “bd” while $esi still had “WyisPlsLJaauEOyBEfNWLgB”. So, I guess it might be the chomp function that is chopping off the first character. I think that there should be another character appended before “WyisPlsLJaauEOyBEfNWLgB” but I could not know which one that was.

* **Attempt2:** I changed my mind and tried to think in another direction. Since a simple password may just be a single fixed string, I guess it might just be hidden in the strings output. So, the first thing I did was to output strings of the first executable file using the strings function. I input the command, “strings -a -n 4 jiz119\_1”, and in the strings output, I found the following:

pWyisPlsLJaauEOyBEfNWLgB

Sorry! Not correct!

Congratulations!

It seemed to me that the string “pWyisPlsLJaauEOyBEfNWLgB” before “Sorry! Not correct!” and “Congratulations!” was most likely to be my password for this executable since this string just has one more character, ‘p’, in the beginning compared to the string I got in attempt1. I was pretty sure it is the correct password; I tried it, and it is correct.

**Solution:** pWyisPlsLJaauEOyBEfNWLgB

=====================================================================

**Executable2:**

**Procedure:**

* **Attempt1:** With the experience of solving the password of the first executable, I tried the strings program but there was no useful information.
* **Attempt2:** Then I tried to solve the password in gdb and used command “dissas /m”, it gave me the disassembled code. From there, I found the following:

0x080486c7 <+172>: call 0x8048428 <strncmp@plt>

I assume this strcmp() function is to compare two strings, one is the correct password and another one is the user input. To get a closer look, I set break point at main and went down instruction by instruction until the address 0x080486c7, I found the following:

=> 0x080486c7 <main+172>: e8 5c fd ff ff call 0x8048428 <strncmp@plt>

So, I now I am sure the password should be in the $ebx register. I got the following:

0xffffd383: "10.215.33.250 54332 22"

It looks like that "10.215.33.250 54332 22" is highly possible to be my password for the second executable file. I tried it, and it worked.

* **Attempt3:** I tested on the other day, however, found the old password did not work anymore. Then I tried to figure out why. The first half of the old passcode looks really like an IP address, and this information might be acquired from the function getenv(). At that time, I tried the combination of my current IP address: 130.49.19.236 and the second half of my old passcode: 54332 22. (So, this is the password I input: “130.49.19.236 54332 22”); and it worked. I also tried to change the second half of the passcode randomly, using different characters combinations in different lengths (It even worked without the second half). But they all worked well. I also tried to append other combinations at the beginning of my current IP address and to take apart the IP address by other characters; however, these all failed. In conclusion, as long as it started with the IP address, the passcode can successfully unlock the second executable file.

**Solution:**

* **General solution:** A string starts with your current IP address
* **Passwords worked for me:** 10.215.33.250 54332 22, 130.49.19.236 54332 22

=====================================================================

**Executable3:**

**Procedure:**

* **Attempt1:** I tried the strings program but found nothing really useful again.
* **Attempt2:** Then I tried to use gdb but I could not even find the main. “b main” gave me error message saying that “Function main not defined”.
* **Attempt3:** I used objdump to see if I could find more useful information.

I found that the .text section and I think this is the start of the assembly code as well. It makes sense because when writing MIPS, we write the assembly code starting in the text section. I can see there are two counter variables from the following:

804842b: c7 45 f0 00 00 00 00 mov DWORD PTR [ebp-0x10],0x0 //counter2

8048432: c7 45 f4 00 00 00 00 mov DWORD PTR [ebp-0xc],0x0 //counter1

We can see their address and they are set to 0 at the beginning. I assume there might be two loops and the loop time depends on the comparison result of the two counters. Let’s see the following(not necessarily in consecutive order):

8048439: eb 10 jmp 804844b <puts@plt+0xf7>

804844b: 83 7d f4 0f cmp DWORD PTR [ebp-0xc],0xf

80484a2: 83 7d f0 04 cmp DWORD PTR [ebp-0x10],0x4

80484a6: 75 16 jne 80484be <puts@plt+0x16a>

80484be: c7 04 24 f2 85 04 08 mov DWORD PTR [esp],0x80485f2

Right after set the two counters to 0, jump to address 8048439, where counter1 needs to compare with 0xf (16 in decimal). So, I think maybe the total input length that the program checked would be 16 bytes. And counter2 needs to be compared with 0x4(4 in decimal). So, there might be a smaller portion of the string with the length of 4 and will be checked with more conditions. The inside loop times constrained by counter2, and the outside loop times constrained by counter1.To find more information for the conditions, I checked all “cmp” instructions and found 5 possible conditions. Please take a look at the following important instructions (not necessarily in consecutive order): (to save space I did not list the corresponding jump instructions but discussed below.)

8048470: 83 f8 5b cmp eax,0x5b //91--> [

804847a: 83 e8 28 sub eax,0x28 // subtract 40

804847d: 83 f8 01 cmp eax,0x1 //1

8048484: 83 f8 7b cmp eax,0x7b //123 --> {

8048489: 83 f8 7d cmp eax,0x7d //125 --> }

804848e: 83 f8 5d cmp eax,0x5d //93 --> ]

The first instruction compared with 91, corresponding to ‘[’ in ASCII table. If greater than 91, it then jumps to compare with 123, corresponding to ‘{’, then 125, corresponding to ‘}’, then 93, corresponding to ‘]’. If not equal to 93, then it jumps out of the inner loop and starts again from the beginning of the outer loop(counter1++). If less than or equal to 91, then subtract 40 and compare with 1; if greater than 1, jumps out of the inner loop and starts again from the beginning of the outer loop(counter1++). That is, the ASCII number should be less than or equal to 41. So, I assume the password will begin with any combination of 4 characters from ‘[’, ‘{’, ‘}’, ‘]’, and characters with ASCII value less than or equal to 41, which will include ‘(’, and ‘)’, followed by any other lengths of character combinations except characters that are just mentioned. But the checkable length will still be just16 bytes. Then I start to try possible password combinations. “%+[]asdfghjkljhgfdsadfgh” failed but “()[]asdfghjkljhgfdsadfgh” succeed. I reviewed the text section but could not find strong evidence to prove that why characters like ‘%’ and ‘+’ that have smaller ASCII value than 41 failed to form the correct password. I guess then maybe other functions made this possible. There might be other functions that constrained the value to be greater than or equal to 40 before getting into the loops. In conclusion, the password will work as long as it starts with any combination of 4 characters from ‘[’, ‘{’, ‘}’, ‘]’, ‘(’, and ‘)’, followed by any lengths of character combinations except characters that are just mentioned.

**Solution:**

* **General solution:** A string with length of at least 16 that starts with any combination of 4 characters chosen from ‘[’, ‘{’, ‘}’, ‘]’, ‘(’, and ‘)’. The next 5 to 16 characters will be combinations of any characters except ones that are just mentioned. The part that is longer than 16 can be any characters, including ‘[’, ‘{’, ‘}’, ‘]’, ‘(’, and ‘)’, since this part is not actually checked. (You can just have the first 16 characters and that 16-char string is actually counted as the real password that the program prompt shows. - see note2)
* **Passwords worked for me:** ()[]aaaaaaaaaa, (){}abbbbbbbbbbbbb
* **Note1:** some of my successful password attempts shows only 15 characters but do not forget that I have to hit “enter” at the end; so that’s actually 16 characters)
* **Note2:** For example, I input 20 characters (including the “enter”) as: ()[]asssssddddddwww. It gave me the prompt saying “Congratulations! Unlocked with passphrase ()[]asssssdddddd”. The program only checks the first 16 characters. So even though I input a longer string the prompt saying the file was unlocked with passphrase “()[]asssssdddddd”. Although I typed in characters longer than 16, but as long as the input worked well to unlock the file, I count that as a valid solution for unlock the file.