Erik Roeckel

Efr11

Project 3- Write up

1. **Executable 1: efr11\_1**

For the first password I just ran the executable (efr11\_1) with mystrings.c. After examining all of the strings I noticed there was a random string: lDZLLWATZmyBLFZjrgGKihDkccydeb followed by a “Congratulations” followed by “Unlock with passphrase %s” **printf** message. After noticing this I assumed that the random string presented was mostly likely the password, so I entered it after running the executable and it was correct. Although I found this password in about five minutes, please let me walk you through the process of how I debugged this first executable, thinking I was working on the second executable, for about an hour and a half. First, I ran GDB and put a break point at main. Then when in main I noticed the line **repz cmps BYTE PTR ds:[esi],BYTE PTR es:[edi]** so I googled it and realized that it was comparing two strings. So to inspect the registers there I put a break point at this line and continued through the program. Once at this break point I inspected the registers using **i r**, then I examined the value at the **esi** register and its value was **“lDZLLWATZmyBLFZjrgGKihDkccydeb”.** It was at this point that I realized I was debugging the first executable rather than the second.

**Correct Password:** lDZLLWATZmyBLFZjrgGKihDkccydeb

1. **Executable 2: efr11\_2**

First of all, you might find this hard to believe but this one gave me a harder time than the third executable did. I started this in the same way as the last process, by putting a break in the main() function. After inserting a break in the main() function I ran the program. Once stopping at this breakpoint I ran the **disas** command to see the contents and calls of the main() function. From here I assumed that **fgets@plt** got input from the user so I skipped that for the time being. The next function I examined was <c>, since I wasn’t exactly sure what the function did. After putting a break at the c() function, I stepped through it using si command and realized that it most likely had nothing to do with the password I was looking for, was mostly just chomping or taking the last letter from the string inputted from the user. For this reason, I decided to examine the next function call in main(). The next call was to p() function, so I put a break point here and used the c command to continue the running of program. Once I hit this break point I used **si** to step through it, and the disas command to display its contents. During this I found a significant loop in the p() function from lines **<+31> - <+72>.** When stepping through this loop I realized that the **eax** register held the length of my input string, but no register specifically carried the password string that I was looking for. I looked back at assembly of main and noticed a loop from lines that compares the size of my input string **eax** to the hexadecimal value **0xa**, which I calculated to be 10. I then realized that this means the password must be greater than 10 characters. After finding out this information I re-ran the program and stepped back through the p() function. By stepping through, and consistently checking the registers with the command **i r,** as well as examining the values in those registers with **x/s** I noticed that the loop in p() was somehow comparing the first letter of the input to the last letter of the input. Realizing this I decided to put a break at the line where this comparison was occurring in p() and run through the program entering an input longer than 10 characters where the first and last letter are the same. I preceded to enter **adsgeherhergbreba**, and step through the program (at this point I am stuck in the loop in p()). My first step through passed, but then something weird happened, my second step through failed. So, I went back to the p() function to examine what could possibly be giving my this result and then I found the pattern. I realized that this loop wasn’t just comparing the first and last letters of the input, but rather every character entered in the first half of the string must be repeated in reverse order. The fancy definition of a word(string) that follows a pattern like this is a palindrome.

**Correct Passwords:**

abcdeffedcba

abcdefghhgfedcba

aaaaaaaaaaaa

bbbbbbbbbbbb

cccccccccccc

Etc…(any palindrome, greater than 10 letters, palindrome being a string that reads the same backwards as it does from the front)

1. **Executable 3: efr11\_3**

To start off, I first tried to put a break at main, but I wasn’t allowed so I assumed there was no main function. I used the command **objdump –x efr11\_3** to see info from the executable in order to get the starting address. Using this command, I saw that the start address was **0x08048370** and set a break at this point. I then ran the program and after getting to this break I ran the command **disas $pc,+40 and noticed a function call in there named <\_\_libc\_star\_main@plt>.** From here I put a break at the address of this function and continued in order to reach this function call. Once at this function call I used **si** to step into the function itself. Once I was in the function I used **disas $pc, 120**, to examine the contents of the **<\_\_libc\_star\_main@plt>.** When examining the contents of this function I saw there was once again a **<puts@plt>** function as well as a function **<getchar@plt>**. My attention was immediately drawn to the getchar() function because its very familiar to me. After running the program, and stepping through this getchar() function I realized when examining the registers that this functions takes in one character from input each loop through (which is what I expected). Knowing this I removed that breakpoint and ran the program from scratch eventually taking me back into the **<\_\_libc\_star\_main@plt>** function. Once here again, I decided to increase the amount of instructions visible to me, so I did **disas $pc,+400** and this was very overwhelming. Being that there were so many lines in this I decided to quit gdb and run **objdump –d –Mintel efr11\_3** and this is where I found the meat of the program. While scrolling through this assembly code, I realized a really interesting and important looking section of code in the **.text** segment. This segment of code ranged from address **0x08048470- 0x0804848e,** and contained multiple calls to **<puts@plt>** as well as comparing the input string to hexadecimal values (this is particulary what drew my attention). I first calculated the hexadecimal values in the context of their jumps as follows: **0x39 = 9, (jg)0x39 = >9, 0x30 = 0, 0x34 = 4, (unconditional jump in between these), 0x63 = c, 0x73 = s**. After finding these hexadecimal values I tried quitting gdb and using just these values to make a password, but this didn’t work. So I went back to my previous spot looking at the assembly in **.text**, and this is where I realized that all those comparisons to hexadecimal values happen within a loop, this loop compares this input string to the hexadecimal value **0x10** = 16. I then realized that this loop makes it so that the input string can not be less than 16 characters, this made a lot of sense because any time I would try to enter a password less than 16 characters it would make me press enter a bunch of times. Then looking deeper into the assembly code I noticed another line of code, a couple lines after this one, that seemed to be comparing characters in our input string to those of the hexadecimal values stated above. Being that this line of code**: DWORD PTR [ebp-0x10],0x8** had an 0x8 I noticed that this was checking to make sure the input string had exactly 8 of the specified hexadecimal characters contained within the first 16 letters. This also meant that the other 8 or more characters couldn’t be one of those values or the password would be incorrect.

**Correct Passwords:**

904csrrrsssrrrrrr

ssssssssqqqqqqqqqqqq

904crrrrqqqssssrrrrrrrr

Etc…The string can be anything more than 16 characters which must include exactly 8 of the special characters being **(9, >9(kinda irrelevant), 0, 4, c, s)** within the boundaries of those first 16 characters and the other 8+ characters must not be any of these special characters.