Jack Napolitano

Cracking the Passwords

Password 1 – jon18_1

This was by far the easiest to find.

I initially ran jon18_1 through mystrings. Upon looking through the massive output of text I could not find anything that appeared to be a password.

Then using gdb, I started by using same the procedure that we used in the lab. I quickly came about the string I was entering, which was stored in %edi at 0x08048302. I noticed that before the compares between %edi and %esi, there was a line:

=> 0x8048302 <main+51>: mov $0x80b316c,%esi

Then I set a breakpoint right after that line and called **x/s $esi** which provided me with the string "DLjcqIBXEKwzqUybNqOVWGkBd." I re-ran the program and this was the correct password.

What annoyed me, however, is that using ctrl+f and "DLjcqIBXEKwzqUybNqOVWGkBd " in my text file that I got from mystings, I found that the password was there the whole time; I just missed it.

Password 2 – jon18_2

This one took me a while. I couldn't find anything useful in the text dump from mystrings so I began the ever so fun process of examining the assembly instructions. I set a breakpoint in main and tried disas, then setting more breakpoints and checking registers with **x/s (reg)**. I got frustrated with this because I couldn't really find anything. I decided to pursue a more surefire way of finding the code. The slow and painful **stepi** and analyzing what is happening as the program runs. After a whole lot of **stepi**'s I found "G" in %edi. I think this is a start… OOooooookkkaayy. Bad choice and a waste of 30 minutes…

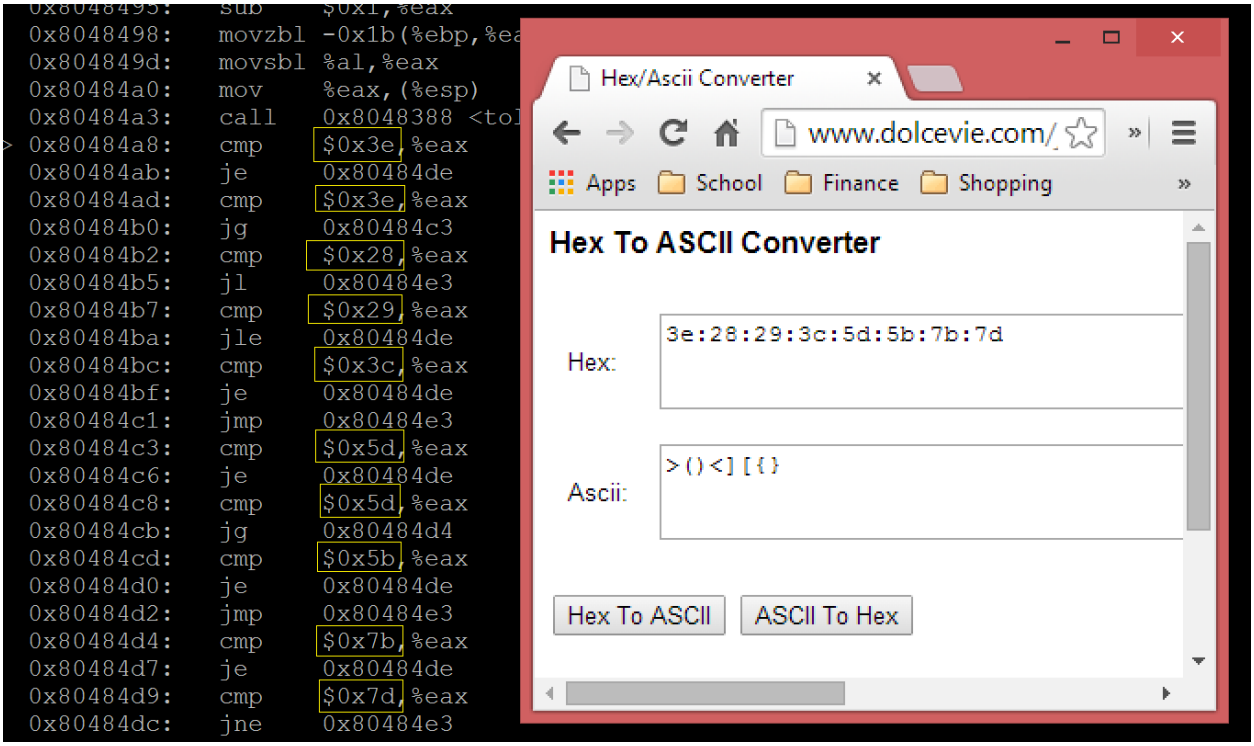
Back to square one.

Before I ran my program this time, I entered **info functions** and found the functions "s", "c', "e", and "d". I set break points at each of these and began examining them. When I was in "s' I ran **x/s $esi** and was given the value "2.718282". I then re-ran the program and it worked!

Password 3 – jon18_3

This one was tricky. My first step was to run it through mystrings and see what I got. The output didn't really seem too helpful so I started playing around in gdb with the program. I tried to set a breakpoint at main, but there was no main function. Then I tried to set a breakpoint in "printf" because I figured I needed to set a breakpoint somewhere in the program so I could get some starting point. The entire program ran without hitting my breakpoint so I was getting confused and decided to go and look at the output file again. I noticed in my output text file that there was "puts", "tolower", "printf', and "getchar" as different functions. I decided it wouldn't hurt to try and set breakpoints at these functions so I set a breakpoint at "tolower" [**(gdb) b tolower**] . Once I ran the program with this new breakpoint I finally got to the breakpoint so I tried **(gdb) disas**. This displayed me with the instructions that seemed to be possibly modifying my input string, but that really wasn't what I was concerned with. My goal was to

find something that led me to the password I could input to unlock this program. Because I was not interested in what was in the assembler code for the *tolower* function, I used **(gdb) nexti** until I was out of the tolower function. After a bunch of **nexti**'s, I got a message "***0x080484a8 in ?? ()***". This was interesting. I tried to use **disas** but that wouldn't work here so I asked gdb to display the 100 instructions after the program counter with **(gdb) display /100i $pc**. I noticed that the program was comparing a hex value to the %eax register. I pulled up a hex to ASCII converter online and entered the values on the list (seen below).



Now I have a set of ASCII values **< > [ ] ( ) { }** that seem to have some significance. I tried entering all of the characters as the password but that wasn't working. Now I tried entering a bunch of "**{**" as the password because I knew that the hex code was **0x7b**. I went back to the top of the "??" function and displayed the next ten instructions with **display /10i $pc** . Then I stepped through the code one line at a time once again with **nexti**. I noticed when I hit the line **0x080484d4: cmp $0x7b, %eax** then the following line **0x080484d7 je 0x8048de** that the **je**, or "jump if equal" call was satisfied when I entered one more **nexti.** It then jumped down to some new assembly code. As I continued to step I noticed some values in memory were being incremented. As I parsed over these lines of code it was more apparent what was happening. I repeatedly passed over the code:

0x80484e7: cmpl $0xa,-0xc(%ebp)

0x80484eb: jle 0x8048492

0x80484ed: cmpl $0x2,-0x10(%ebp)

0x80484f1: jne 0x8048509

It took me a while to realize what was happening here. The first two lines say, if the number we are storing in memory is less than 10, jump to **0x8048492.** This is happening because the bottom two lines

are a comparison we only want to do after we have compared every char of our password to the set of ASCII values we found earlier. So we continue to loop through the hex comparisons, then jump and increment our value in **-0xc(%ebp)** until we have reached 10. Why 10? Turns out that is the length of the password. (I explain another way I determined that below. [1]) So now that we hit 10, we do that comparison we see on the bottom 2 lines. We are checking to see if **$0x2** is equal to a value we have stored in memory. Turns out, we incremented this value, **-0x10($ebp)**, every time one of our *special* characters, **< > { } [ ] ( ),** were found. So why are we comparing it to 2? I finally realized for us to have a working password, it must only contain EXACTLY 2 of the characters from our set of special characters. To make sure I was right. I re-ran my program with a breakpoint at the beginning of "??". I entered "<<aaaaaaaa" as my password and stepped through the instructions. Furiously fighting though an onslaught of carpal tunnel, I stepped though 10 iterations until it finally got to the step where it compared 2 to the number of special chars I encountered. Lo and behold, my passphrase worked. I tested the program with passwords like "<>aaaaaaaa", "<aaaaaaaa>", "(hhhhhhhh]" and they all worked.

Thus, the password requires exactly 2 characters from **< > { } [ ] ( )** and 8 other characters.

[1] *Another thing I noted when attempting random passwords was that when I entered a password that was less than 9 characters, it went to a new line and waited for me to enter another character. But with 9 characters, it would go directly to the "Sorry! Not Correct!" statement. Also, I saw that if I entered the letter "a" and hit enter, it would go to a new line and I could do this 5 times with a single character. This led me to believe that the password that is expected by the program should be of a certain length. The reason that I could enter 9 chars and would go to the "Sorry! Not Correct!" statement is because it contained 9 characters and the new line escape value, so it actually did have 10 chars. Thus the password needed to be 10 chars long.*