AERE361 Lab 6

Jordan Reese

February, 23 2018

1 Password

This code is meant to test our knowledge of how memory is stored in C. I took a look at the code, and I saw a few things I didn't like, but nothing was inherently wrong with the code. So I decided to compile and test it. I entered in the password cyc1nenat10n and the program gave me root access. This makes sense because the that is the correct password for the program. I then entered in hawk3y35 and this gave me an incorrect password error. This again makes sense because that password is in fact incorrect. The trick is when you enter in the longer password. At first glance we should notice a problem with this password. Buff is only 15 characters long and so we exceed the length. This we are potentially ruining memory assignments. I entered the lengthy password and it printed wrong password, but gave me root access anyway. This is because the password was long enough to change the memory address value for match. To put this in easier to explain terms, match as a memory address assignment to it. So if you manage to stumble upon that memory address you could change the value for match. This happens when a password entry exceeds 28 characters.

There are a couple things I could think of to fix this problem. The first idea was a simple fix. All I would do is check the string length of the input and if it is longer than the actual password, then bug out of the code and print an error message. This still doesn't fix the problem of overwriting the memory, it just prevents root access for an incorrect password. The other thing that could strengthen the program is give match an exact match rather than just checking if (match). This would create only one condition that could give root access. The second much more complicated way I thought of would be very hard for me to program because I don't have the knowledge to do so. I would check each input as the user put the character in the command line. If that number was correct then keep checking new numbers, if not then don't check anymore numbers until the string is fixed. We would have to factor in backspaces, which would be tricky but this is the most secure way I could think of to fix the problem of static memory assignments.

2 $\mathbf{e}^x Code$

This code was relatively easy to think through. I had a bit of a hard time determining what the error precision was. But luckily for me, I had just done a piece of code in AERE 362 that was roughly the same idea. We need to find converging numbers using a precision number. Once the number stop changing by amounts larger than the preset precision, then the series isn't really creating anything new. So you can stop the series when the precision is met, because anything after than would be almost pointless to calculate. Once I thought through what I needed to set up the math was just a simple fraction. The tricky part of the fraction was the factorial. I decided a function would be pretty useful here. I didn't want to clutter the fraction by nesting loops to create a factorial. The function is a simple for loop that just multiplies and overrides values, until the factorial is found. It even works for 0!. I felt there wasn't any tricking coding or algorithm problems for this particular code. It was straight forward.