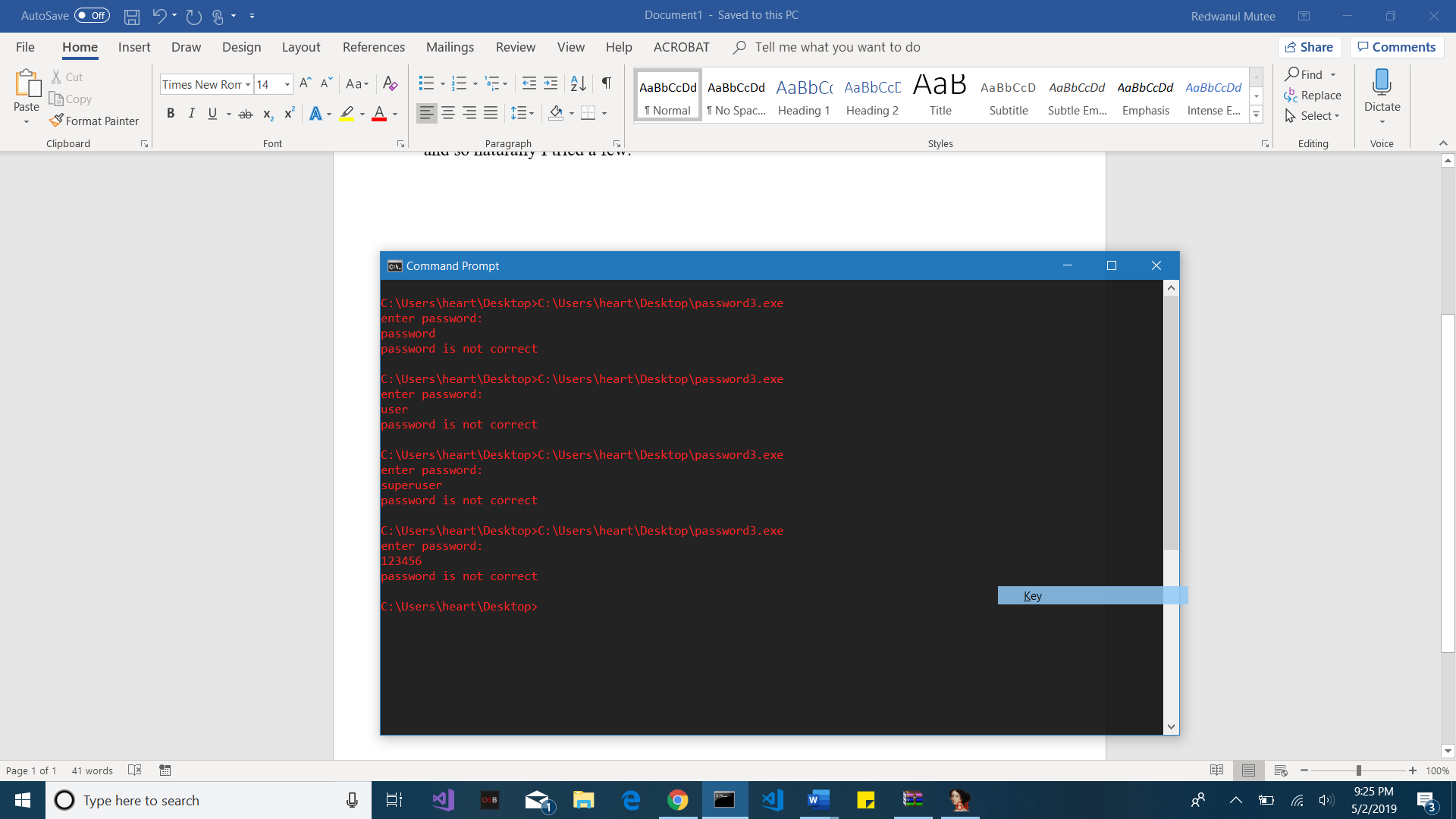
Redwanul Mutee Rm4243

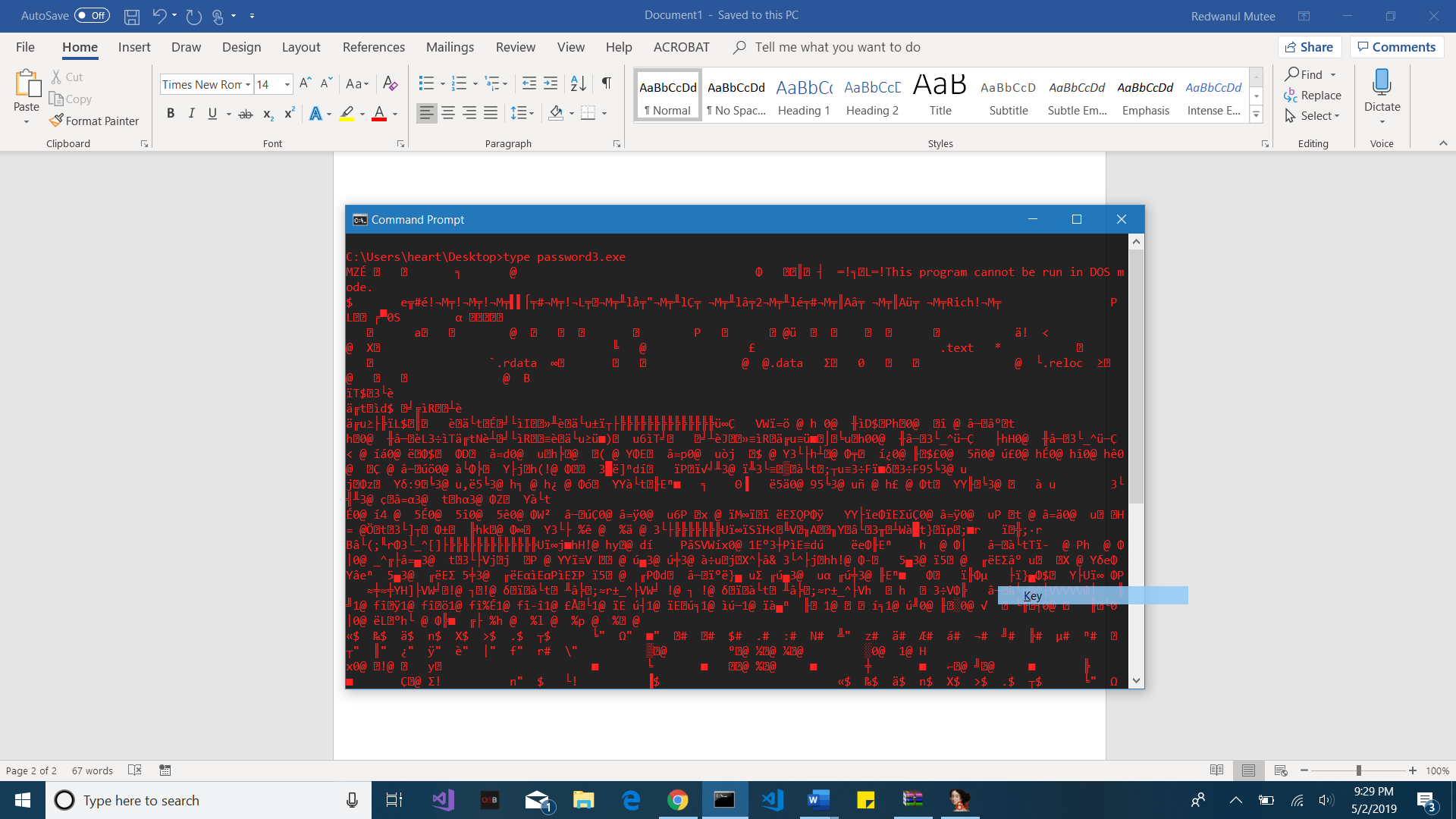
CSGY-9163 Application Security

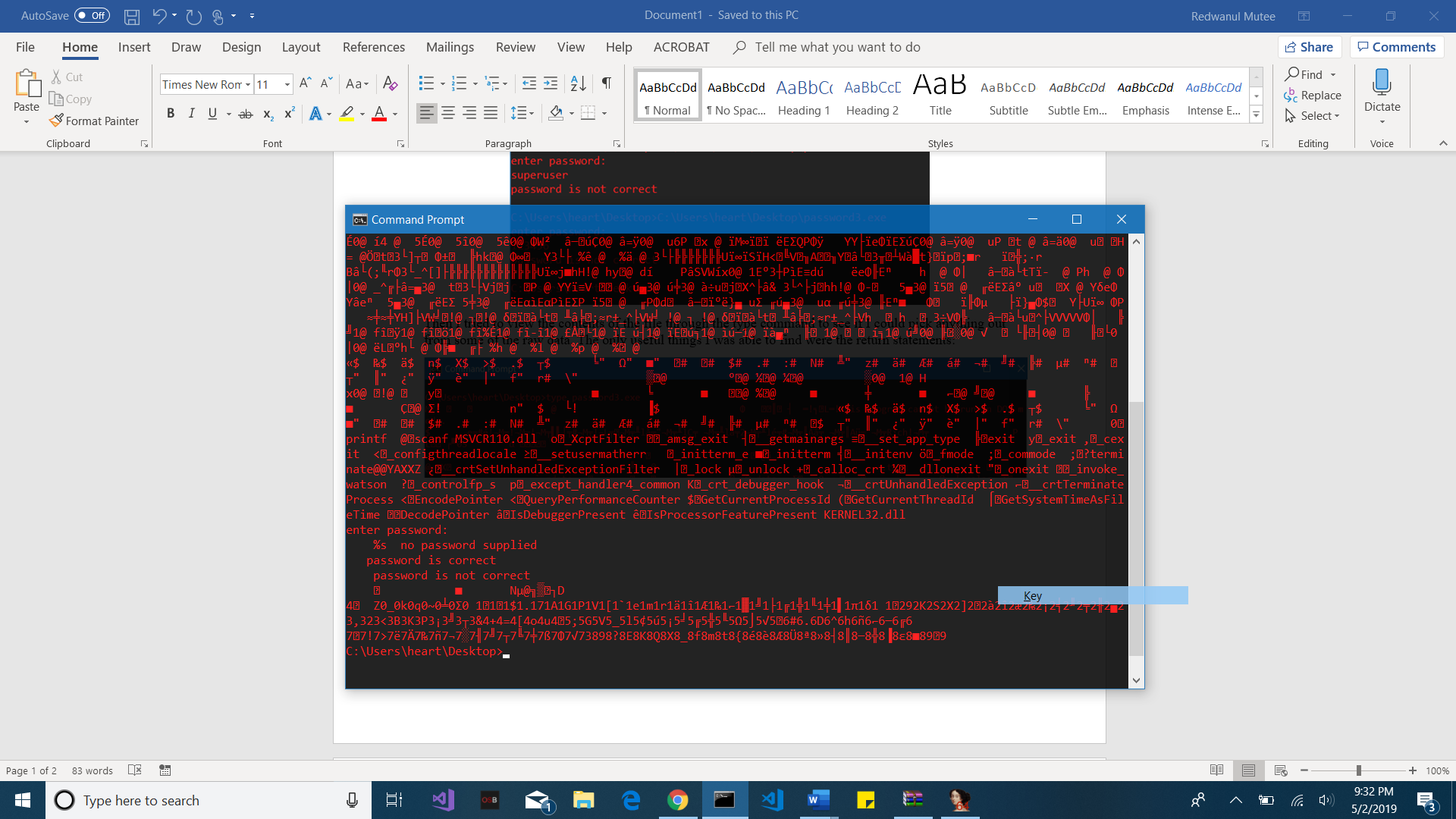
**Lab 3: Reverse Engineering**

From the <https://challenges.re/40/> website, I downloaded the windows version of the challenge 40 executable (password3.exe) . When I ran the program in cmd, it wanted a password and so naturally I tried a few.



Then I tried to view the contents of the file through the type command to see if I could pick anything out from some of the raw data. The only useful things I was able to find were the return statements:





I switched to my Linux VM that had Binary Ninja and downloaded the Linux version of the challenge. I opened the password3 file using Binary Ninja and analyzed the high-level disassembly and obtained the following code blocks:



The program uses puts to display the string “enter password” and uses scanf to request the user for input. If no input is provided, then control moves to the following block which uses puts displays the string “no password supplied”



When a password is entered it undergoes two checks; the input is processed through two consecutive functions. If it passes the first function check then it proceeds to the second function check. The first function labeled sub\_10000dd1 and the second function labeled sub\_10000e04 are shown below. The first function seems to perform an operation on the initial input and checks if the output is equal to 553. The second function seems perform another operation on the initial input and checks if the output is equal to 0xD404F501 which is equal to 16,441,996,54510 .

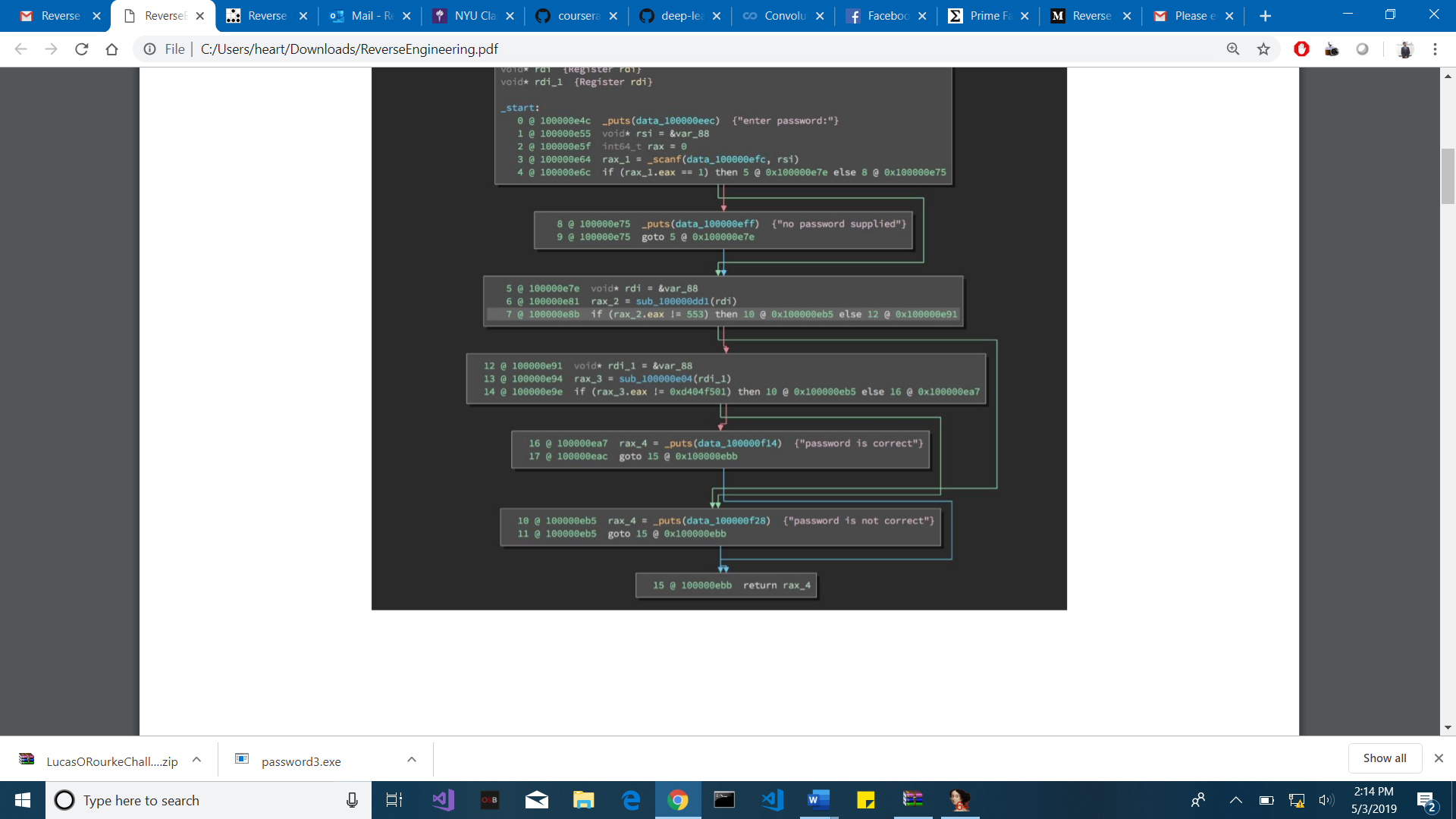


-

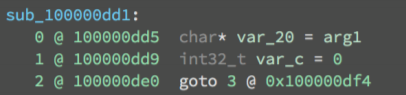
If the input does not pass through either of these checks then control flows to a code block that displays “password is incorrect” and exits. If it passes through the two checks, then the control flows to a code block which displays “password is correct” and exits. Therefore, if we inspect what operations each of these functions are carrying out before doing the checks, then we can generate passwords that meet such requirements.

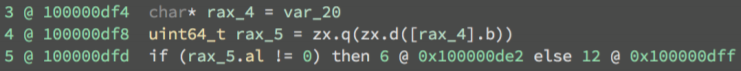


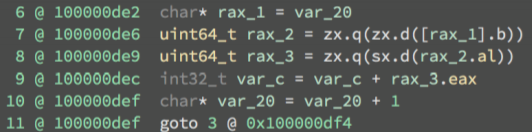


🡪

The first function sub\_10000dd1 takes the initial input and calculates the total sum found when the integer value of all the characters in the input are added together. It does this through a for loop on the input length. A variable called var\_c is also initialized to 0. In each iteration, the integer value of the current character is added to var\_c. This value is then ultimately returned. Earlier we saw that the first function was performing an operation on the initial user input and checking if the output is equal to 553. This means that the integer sum of all the characters in the input has to be 553. The disassembly of this function is shown below:

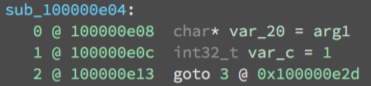


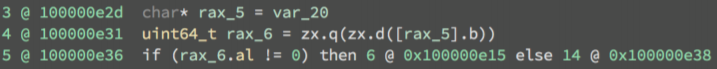


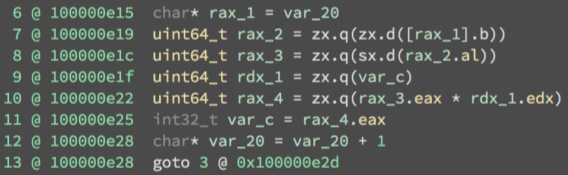




The second function sub\_10000e04 takes the initial input and calculates the total product found when the integer value of all the characters in the input are multiplied together. It does this through a for loop on the input length. A variable called var\_c is also initialized to 1. In each iteration, the integer value of the current character is multiplied by var\_c. This value is then ultimately returned. Earlier we saw that the second function was performing an operation on the initial user input and checking if the output is equal to 0xD404F501 or 16,441,996,54510. This means that the integer product of all the characters in the input has to be 16,441,996,54510 . The disassembly of this function is shown below:





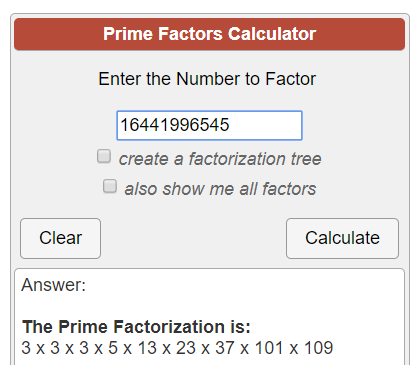




Now we know the following facts:

1. The input are ASCII chracters in the range 0-255 (perhaps more likely to be letters in 0-128)
2. The integer sum of all the characters in the input has to be 553
3. The integer product of all the characters in the input has to be 16,441,996,54510

If we wanted numbers that add upto about 500 and multiply to about 10,000,000,000 then we could use the number 100 about 5 times since 100\*5 = 500 and 100^5 = 10,000,000,000. Therefore I suspect that the input length is 5 meaning we need 5 numbers that add upto 553 and multiply to 16,441,996,545. To further this thought, I decided to check the prime factorization of 16,441,996,545.



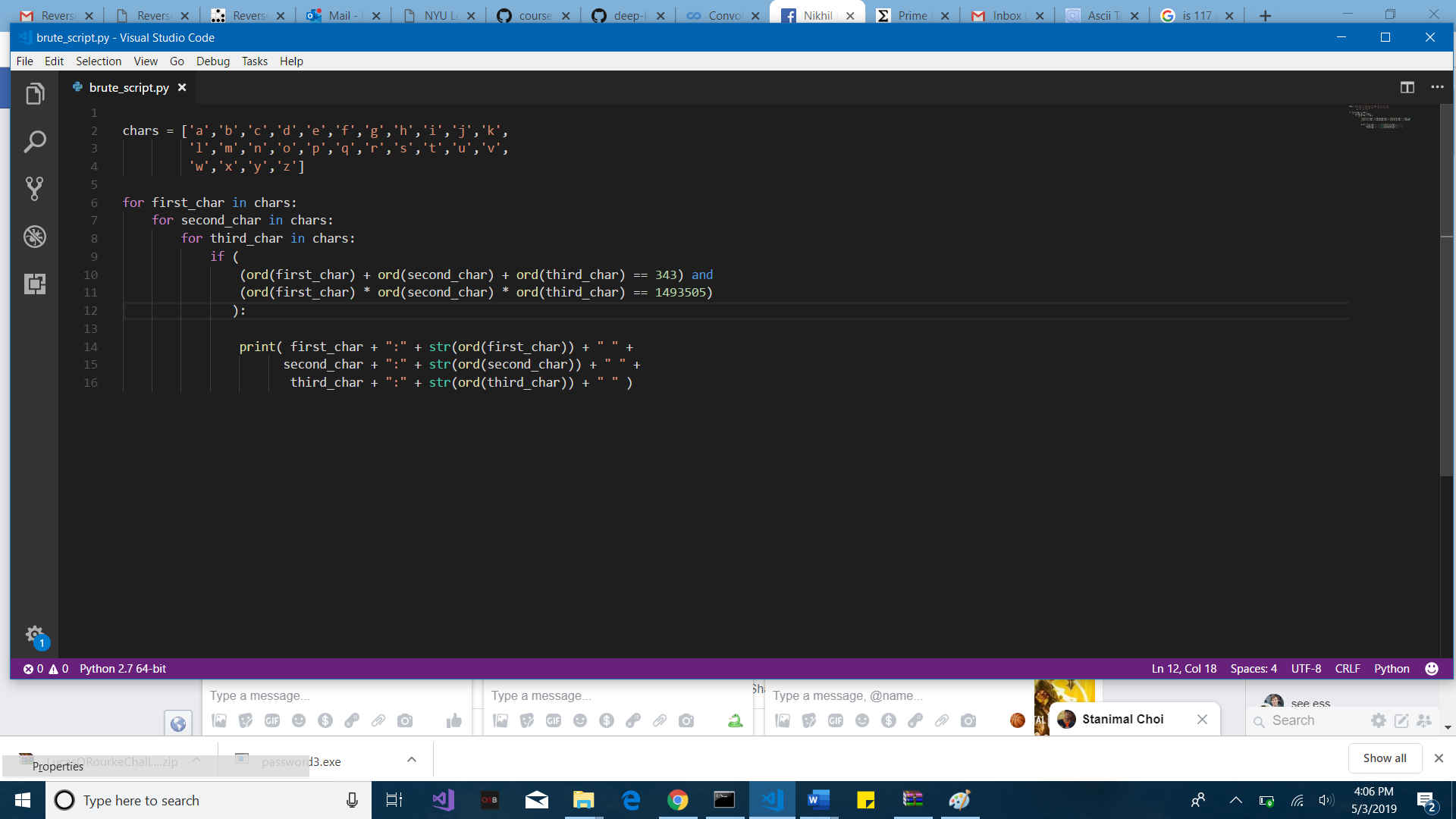
In the prime factorization above, we could use different combinations to come up with many different factors. However, the numbers 101 and 109 stood out because they represent the characters ‘e’ and ‘m’ respectively. If we treat these as two possible characters in a password that we suspect to be of length 5, then we need to find 3 more characters that satisfy the following requirements (obtained from previous requirements):

553 – (101 + 109) = 343   
16,441,996,545 / (101\*109) = 1,493,505

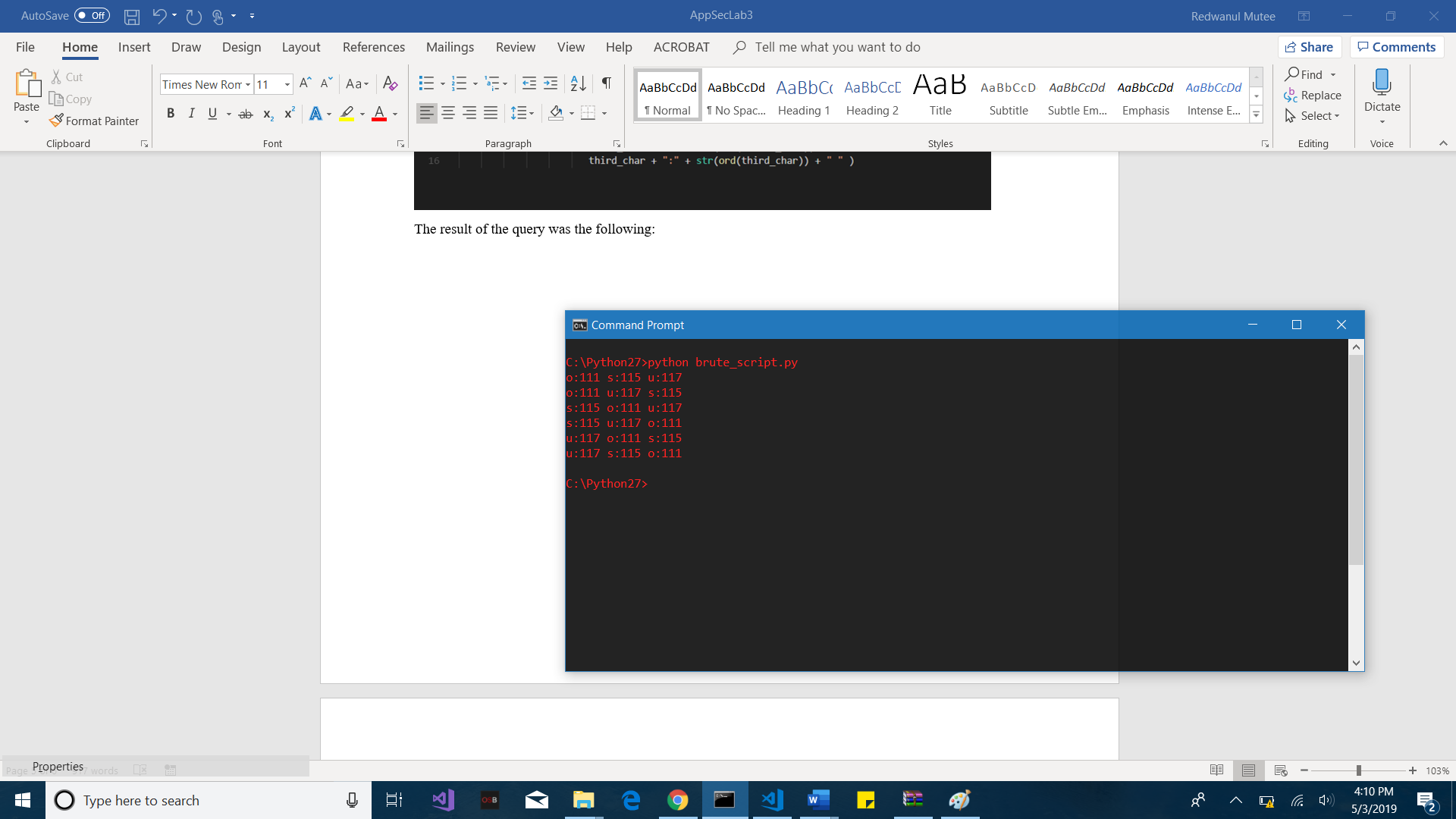
1. The integer sum of the three remaining characters in the password has to be 553
2. The integer product of the three remaining characters in the password has to be 1,493,505

Now we just have to use a simple python script that loops through the lowercase chracters and finds which 3 chracters adds upto 343 and multiplies to 1,493,505.

I wrote the script very fast in python and since it is performing a simple operation, it’s not going to look anything amazing. It has three nested loops and checks for the later requirements mentioned earlier.



The result of the script was the following:



It gives us the numbers 111, 115, and 117. As it turns out, along with the previous numbers of 101 and 109 the following are true:

1. 111 \* 115 \* 117 \* 109 \* 101 = 16,441,996,545
2. 111 + 115 + 117 + 109 + 101 = 553

**101 = ‘e’   
109 = ‘m’   
111 = ‘o’   
115 = ‘s’   
117 = ‘u’  
  
MOMENT OF TRUTH:** Any of the 5! = 120 permutations of the above letters should work as passwords for the challenge.

