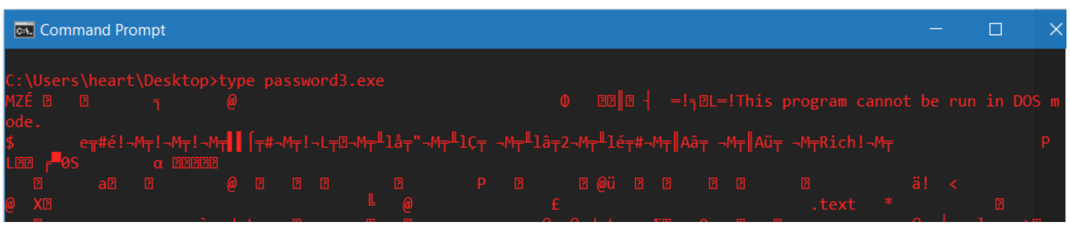
Redwanul Mutee Rm4243
CSGY-9163 Application Security

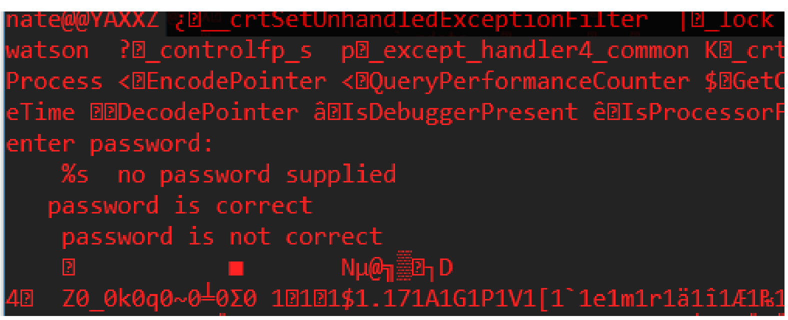
Lab 3: Reverse Engineering

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.

```
Command Prompt

C:\Users\heart\Desktop>C:\Users\heart\Desktop\password3.exe
enter password:
password
password is not correct

C:\Users\heart\Desktop>C:\Users\heart\Desktop\password3.exe
enter password:
user
password is not correct

C:\Users\heart\Desktop>C:\Users\heart\Desktop\password3.exe
enter password:
superuser
password is not correct

C:\Users\heart\Desktop>C:\Users\heart\Desktop\password3.exe
enter password:
123456
password is not correct

C:\Users\heart\Desktop>
```

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:

```
void var_88  {Frame offset -88}
int64_t __saved_rbp  {Frame offset -8}
void* const __return_addr  {Frame offset 0}
int64_t rax  {Register rax}
int64_t rax_1  {Register rax}
uint64_t rax_2  {Register rax}
uint64_t rax_3  {Register rax}
int64_t rax_4  {Register rax}
void* rsi  {Register rsi}
void* rdi  {Register rdi}
void* rdi_1  {Register rdi}

_start:
   0 @ 100000e4c  _puts(data_100000eec)  {"enter password:"}
   1 @ 100000e55  void* rsi = &var_88
   2 @ 100000e5f  int64_t rax = 0
   3 @ 100000e64  rax_1 = _scanf(data_100000efc, rsi)
   4 @ 100000e6c  if (rax_1.eax == 1) then 5 @ 0x100000e7e else 8 @ 0x100000e75
```

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"

```
8 @ 100000e75  _puts(data_100000eff)  {"no password supplied"}
9 @ 100000e75  goto 5 @ 0x100000e7e
```

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 $16{,}441{,}996{,}545_{10}$.

```
5 @ 100000e7e  void* rdi = &var_88
6 @ 100000e81  rax_2 = sub_100000dd1(rdi)
7 @ 100000e8b  if (rax_2.eax != 553) then 10 @ 0x100000eb5 else 12 @ 0x100000e91
```

```
12 @ 100000e91  void* rdi_1 = &var_88
13 @ 100000e94  rax_3 = sub_100000e04(rdi_1)
14 @ 100000e9e  if (rax_3.eax != 0xd404f501) then 10 @ 0x100000eb5 else 16 @ 0x100000ea7
```

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.

```
16 @ 100000ea7  rax_4 = _puts(data_100000f14)  {"password is correct"}
17 @ 100000eac  goto 15 @ 0x100000ebb
```

```
10 @ 100000eb5  rax_4 = _puts(data_100000f28)  {"password is not correct"}
11 @ 100000eb5  goto 15 @ 0x100000ebb
```

```
0x100000ebb → return rax_4
```

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:

```
sub_100000dd1:
   0 @ 100000dd5  char* var_20 = arg1
   1 @ 100000dd9  int32_t var_c = 0
   2 @ 100000de0  goto 3 @ 0x100000df4
```

```
3 @ 100000df4  char* rax_4 = var_20
4 @ 100000df8  uint64_t rax_5 = zx.q(zx.d([rax_4].b))
5 @ 100000dfd  if (rax_5.al != 0) then 6 @ 0x100000de2 else 12 @ 0x100000dff
```

```
 6 @ 100000de2  char* rax_1 = var_20
 7 @ 100000de6  uint64_t rax_2 = zx.q(zx.d([rax_1].b))
 8 @ 100000de9  uint64_t rax_3 = zx.q(sx.d(rax_2.al))
 9 @ 100000dec  int32_t var_c = var_c + rax_3.eax
10 @ 100000def  char* var_20 = var_20 + 1
11 @ 100000def  goto 3 @ 0x100000df4
```

```
12 @ 100000dff  uint64_t rax_6 = zx.q(var_c)
13 @ 100000e03  return rax_6
```

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 $16{,}441{,}996{,}545_{10}$. This means that the integer product of all the characters in the input has to be $16{,}441{,}996{,}545_{10}$. The disassembly of this function is shown below:

```
sub_100000e04:
   0 @ 100000e08  char* var_20 = arg1
   1 @ 100000e0c  int32_t var_c = 1
   2 @ 100000e13  goto 3 @ 0x100000e2d
```

```
3 @ 100000e2d  char* rax_5 = var_20
4 @ 100000e31  uint64_t rax_6 = zx.q(zx.d([rax_5].b))
5 @ 100000e36  if (rax_6.al != 0) then 6 @ 0x100000e15 else 14 @ 0x100000e38
```

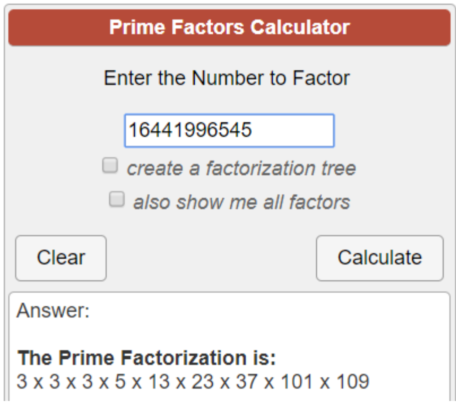
```
 6 @ 100000e15  char* rax_1 = var_20
 7 @ 100000e19  uint64_t rax_2 = zx.q(zx.d([rax_1].b))
 8 @ 100000e1c  uint64_t rax_3 = zx.q(sx.d(rax_2.al))
 9 @ 100000e1f  uint64_t rdx_1 = zx.q(var_c)
10 @ 100000e22  uint64_t rax_4 = zx.q(rax_3.eax * rdx_1.edx)
11 @ 100000e25  int32_t var_c = rax_4.eax
12 @ 100000e28  char* var_20 = var_20 + 1
13 @ 100000e28  goto 3 @ 0x100000e2d
```

```
14 @ 100000e38  uint64_t rax_7 = zx.q(var_c)
15 @ 100000e3c  return rax_7
```

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 $16,441,996,545_{10}$

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

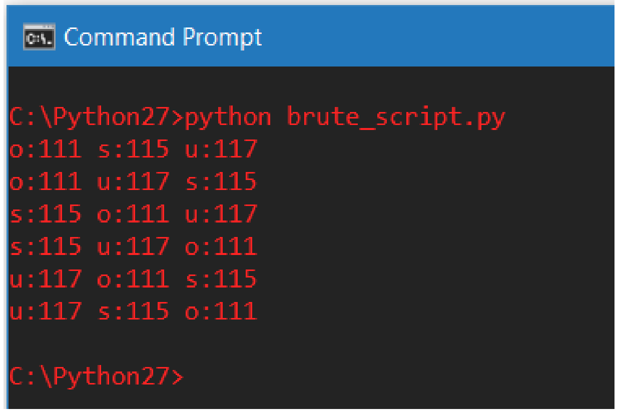
Now we just have to use a simple python script that loops through the lowercase chracters and finds which 3 chracters adds upto 343 and multiplies to 1,493,505.

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.

brute_script.py

```python

chars = ['a','b','c','d','e','f','g','h','i','j','k',
         'l','m','n','o','p','q','r','s','t','u','v',
         'w','x','y','z']

for first_char in chars:
    for second_char in chars:
        for third_char in chars:
            if (
                (ord(first_char) + ord(second_char) + ord(third_char) == 343) and
                (ord(first_char) * ord(second_char) * ord(third_char) == 1493505)
                ):

                print( first_char + ":" + str(ord(first_char)) + " " +
                       second_char + ":" + str(ord(second_char)) + " " +
                        third_char + ":" + str(ord(third_char)) + " " )
```

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.