Final_Assignment_Report

Final Report Evan Haaland

Executive Summary

When I first downloaded the program I uploaded it to my ubuntu VM to run a few commands to get more information about the file. I could see right away that it is a windows executable file. This meant that I would have to perform any debugging on a windows VM/machine. After running more commands I could see some of the text contained in the program. The text helped me realize that I am trying to get the password for the software.

I decided to continue my static analysis using ghidra in the ubuntu VM. The decompiler was very useful and sped up the process a good amount. I unfortunately was not able to find the main function right away. I had to go through some of the functions to see the flow of the program. To help with this I would try and find functions of interest such as the one that prints out any messages and try and find any other references to the functions that are called in the function I am analyzing. I started to rename any of the functions that I saw to help understand what the program is doing.

I was able to find the function that runs specifically if we get the password correct. The only issue is that we don't get the password from this. This made me realize that I am close to figuring out where the password is generated. I started to go through the function calls before the function that calls the message box. I saw that one function takes in my command line argument and performs and operation on it with another string. I saw that there is a function that generates this string. I saw that the function implements random numbers. At first I thought this meant there would be multiple passwords until I saw that the random number was seeded which meant the values would never change.

I originally tried to use an online c compiler to run the code and see if I could get the seeded numbers statically but I did not see if I was getting the same numbers that I would get if I were to run the program. This made me realize that I should run the program with a debugger so that I can see the exact seeded values that the program receives. When I ran in it the debugger I set break points and the function and I could then retrieve the string in full.

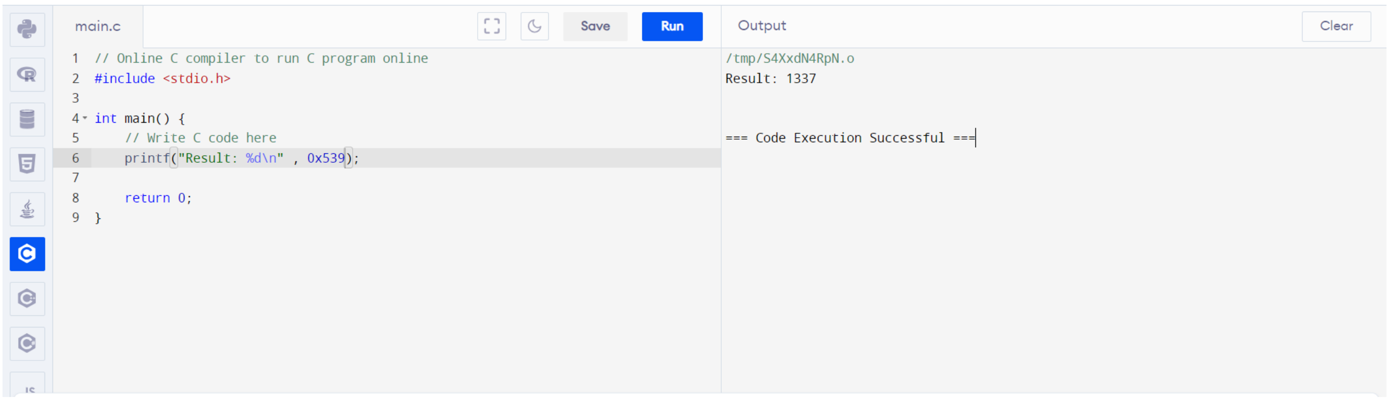
I broke the first round of the encryption. I wanted to try and do it all statically but doing it dynamically made it much faster. When I looked at the 2nd method of encryption I realized that it was using a memcpy() call that checks if the memory from buf1 matches the memory from buf2. I realized that buf2 is predetermined and buf1 is the value after encryption is performed with the string I typed in and the generated string from the program. I realized that I could use the information in buf2 to reverse the xor encryption to get the value I needed.

I wanted to automate this with python but was not able to do it quickly. I got out a notepad .txt file and used python to get the binary values for the data in buf2 and the encrypted string. I manually did the math for the xor and than turned the binary into decimal and then into ascii. Once I did this I got the password. I ran the password and got access granted.

Technical Summary

When I first ran `file softwarechallenge.exe` I could see that I was dealing with windows executable file. This meant that if I had to do any dynamic analysis I would have to move my debugging to a windows VM/Machine. When I ran `strings softwarechallenge.exe` I was able to see strings with *access denied* and *access granted* messages meaning that this program is most likely checking for a password. I would also see other windows function calls. These some what gave me an understanding of what I should be looking at in the software.

When I decided to move to using ghidra I was not able to find the main function. I got sent to the entry which did not help me to much. I decided to analyze the other functions to see if I could find obvious clues of what to look at. I luckily was able to find the function that displays the message boxes and I was able to see where it asks for the user input and where it sends the access granted and denied messages. From there I saw that there is a function that specifically runs after the access granted window box. This function contained the cod `Result" %d` So I know that it is printing something that could be of interest. - I used a online c compiler so that I can see that the result is 1337 when printed out. I tried using this as a password but it did not work. This meant I had more work to do.

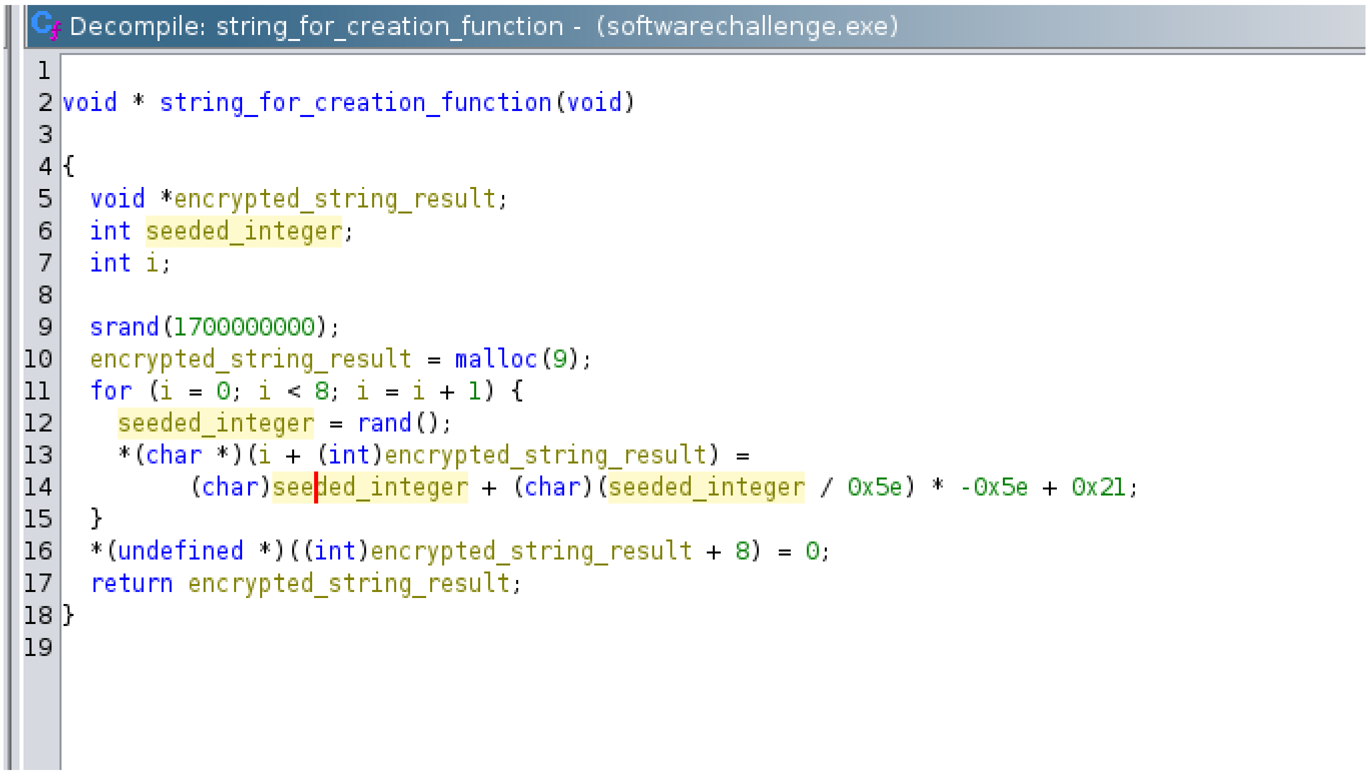


```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {
    // Write C code here
    printf("Result: %d\n" , 0x539);

    return 0;
}
```

```
/tmp/S4XxdN4RpN.o
Result: 1337


=== Code Execution Successful ===
```

I decided to analyze the function more and see what is being called before the access granted and access denied checks. I noticed that there was a function in which a xor encryption was called. I realized that this function took in the parameter which was the user input. I used the ghidra renaming features to rename the function so that I would be able to notice it a lot better in other lines of code. When looking at the xor encryption I was able to change some of the variable names. This made it helpful so that I could get more readability to what the function as actually doing. I saw that that there was another string that was being used in the xor encryption. I saw that the string was created by calling another function at the beginning of the current function. I decided to look into this function.

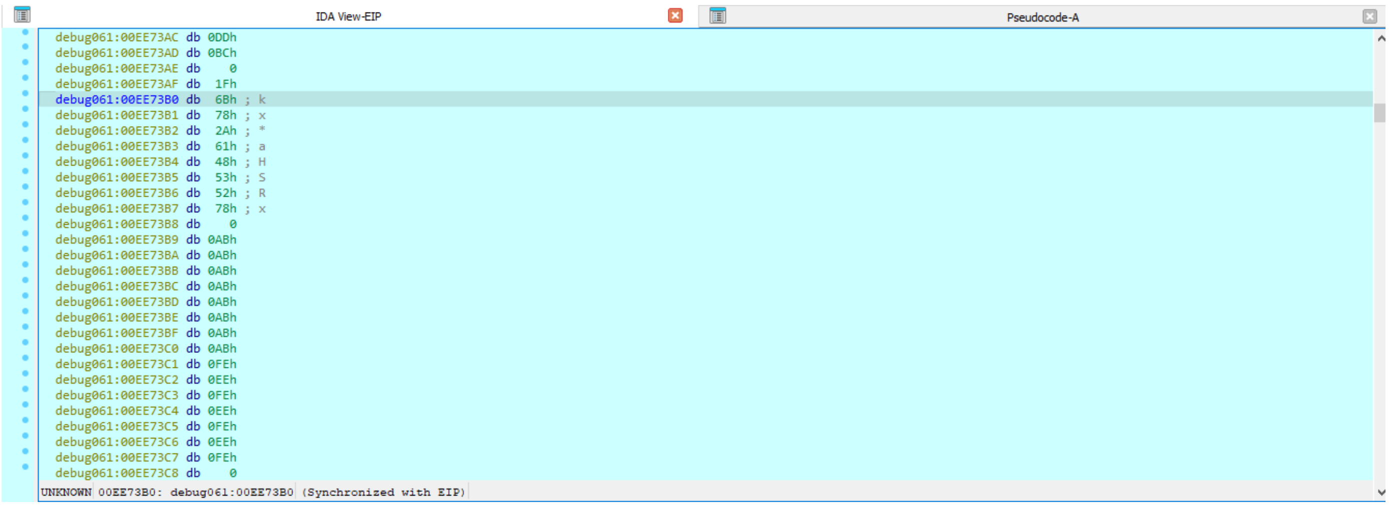
When opening the new function I decided to rename it something along the lines of **generate_encrypted_string** just so I can distinguish this encryption function from the one before it. Using the decompiler I could see that it was using `malloc(9);` to create a string with 9 characters. I saw that the functions `srand()` and `rand()` were used. I realized that a random integer was created to help generated some of the ascii characters with the xor encryption. At first I thought this was a way to make sure that there is a random password every time. When I read up on what `srand()` does I realized that it generates a seeded value so that the `rand()` would generate the same results every time.



Decompile: string_for_creation_function - (softwarechallenge.exe)

```

void * string_for_creation_function(void)

{
  void *encrypted_string_result;
  int seeded_integer;
  int i;

  srand(1700000000);
  encrypted_string_result = malloc(9);
  for (i = 0; i < 8; i = i + 1) {
    seeded_integer = rand();
    *(char *)(i + (int)encrypted_string_result) =
         (char)seeded_integer + (char)(seeded_integer / 0x5e) * -0x5e + 0x21;
  }
  *(undefined *)((int)encrypted_string_result + 8) = 0;
  return encrypted_string_result;
}

```

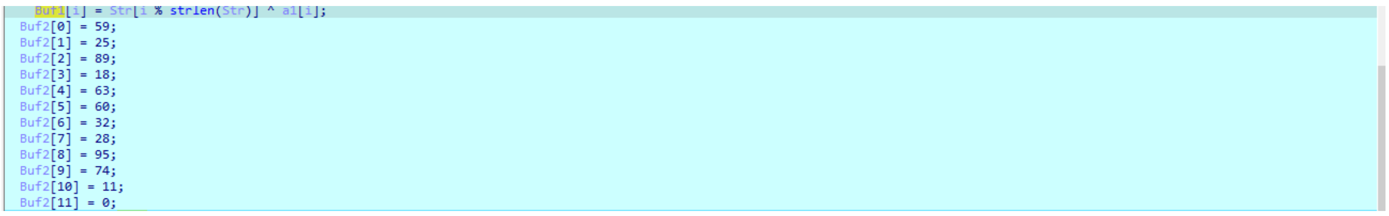
I originally tried to use the same online c compiler to rewrite the function to generate the seeded values but I was not getting expected results. I wanted to quickly do it statically but was not able to. I decided it was time to use the ida freeware debugger on my windows machine. When I switched do ida freeware it took me a second to find the functions I needed to look at. Luckily, the address were the same so I was able to find them. I attached a break point at the end of the **generate_encrypted_string** function and I was able to see that the string generated was **kx*aHSRx** . Using the debugger made this a lot faster. I was able to just run the function and get the whole string at once versus having to statically go through character by character.



```
debug061:00EE73AC db 0DDh
debug061:00EE73AD db 0BCh
debug061:00EE73AE db    0
debug061:00EE73AF db  1Fh
debug061:00EE73B0 db  6Bh ; k
debug061:00EE73B1 db  78h ; x
debug061:00EE73B2 db  2Ah ; *
debug061:00EE73B3 db  61h ; a
debug061:00EE73B4 db  48h ; H
debug061:00EE73B5 db  53h ; S
debug061:00EE73B6 db  52h ; R
debug061:00EE73B7 db  78h ; x
debug061:00EE73B8 db    0
debug061:00EE73B9 db 0ABh
debug061:00EE73BA db 0ABh
debug061:00EE73BB db 0ABh
debug061:00EE73BC db 0ABh
debug061:00EE73BD db 0ABh
debug061:00EE73BE db 0ABh
debug061:00EE73BF db 0ABh
debug061:00EE73C0 db 0ABh
debug061:00EE73C1 db 0FEh
debug061:00EE73C2 db 0EEh
debug061:00EE73C3 db 0FEh
debug061:00EE73C4 db 0EEh
debug061:00EE73C5 db 0FEh
debug061:00EE73C6 db 0EEh
debug061:00EE73C7 db 0FEh
debug061:00EE73C8 db    0
UNKNOWN 00EE73B0: debug061:00EE73B0 (Synchronized with EIP)
```

I now had to go back and see how the encryption worked for the other xor encryption. I kept trying to find a compare function to see where the string is checked. I could not find a cmp instruction but I did find that the access granted and access denied calls were determined whether a boolean value was true or not. I realized that the function that passes in the command line argument returns a boolean based on the value of memcpy(). I realized that I am using memcpy() to compare two strings and I did them to be identical to get the access granted message.

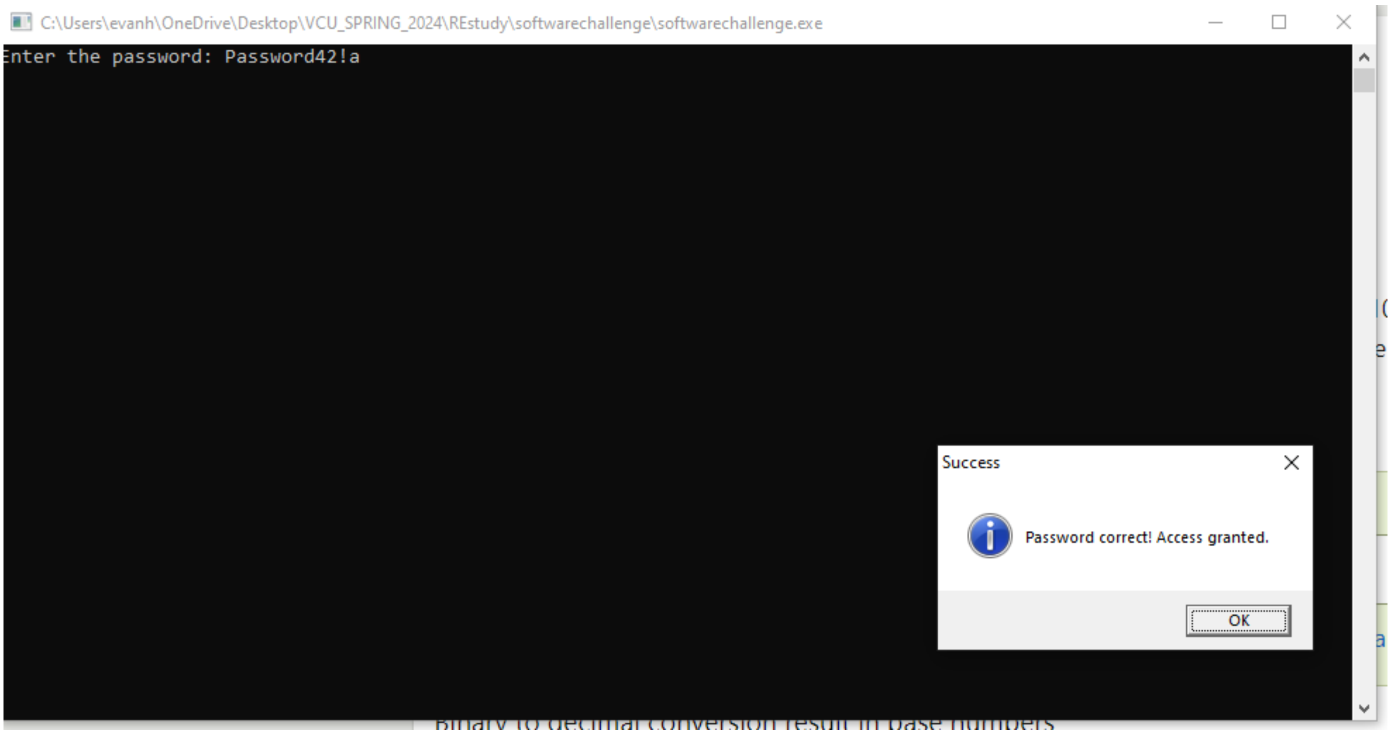
I decided to compare the two decompilers. I recommend doing this because the ida and ghidra decompilers both had differences that helped my understanding of the code. Ida showed me the values for buf2[] with a more obvious naming convention did ghidra. This made me realize that I actually knew the memory contents of buf2. I then saw that the encryption algorithm gave the results for buf1[] element by element using the string I typed in and the encrypted string from the previous function.


```
    Buf1[i] = Str[i % strlen(Str)] ^ a1[i];
  Buf2[0] = 59;
  Buf2[1] = 25;
  Buf2[2] = 89;
  Buf2[3] = 18;
  Buf2[4] = 63;
  Buf2[5] = 60;
  Buf2[6] = 32;
  Buf2[7] = 28;
  Buf2[8] = 95;
  Buf2[9] = 74;
  Buf2[10] = 11;
  Buf2[11] = 0;
```

I originally tried to get the whole process in python but I was not able to do it quickly. I was spending too much time tyring to create the script than trying to solve the challenge. I decided to use python to get all the ascii and hex value into binary. I did this so I can copy them over into another file and manually do the xor to get the values I needed. I then took the binary numbers and converted them to ascii.

```
00111011 00011001 01011001 00010010 00111111 00111100 00100000 00011100 01011111 01001010 00001011 00000000 BUF2 (binary)
01010000 01100001 01110011 01110011 01110111 01101111 01110010 01100100 00110100 00110010 00100001 01100001 ANSWERS (binary)
01101011 01111000 00101010 01100001 01001000 01010011 01010010 01111000 01101011 01111000 00101010 01100001 Static Encrypted String (binary)

80 97 115 115 119 111 114 100 52 50 33 97
Password42!a
```

I was then able to put all the pieces together to get the password: **Password42!a** I ran this in the program and got access granted and the print out after it.


C:\Users\evanh\OneDrive\Desktop\VCU_SPRING_2024\REstudy\softwarechallenge\softwarechallenge.exe

```
Enter the password: Password42!a
```

Success

Password correct! Access granted.

OK

Impact (ways to improve)

I think the seeded password would have to change. Although we added more ways of encryption, once I saw there was a set static seeded value used to determine the "random" values for one of the encryption strings, I realized that I could run the program to get the values and they would stay the same the whole time.

Being able to see the methods used for encryption was very helpful for me to crack the password. Being able to see the C and windows function calls gave me a lot of ideas of where to start investigating. I did get confused on some functions and trying to figure out there functionality. Luckily I have had some experience so I was able to determine which functions were most likely generated by the compiler.