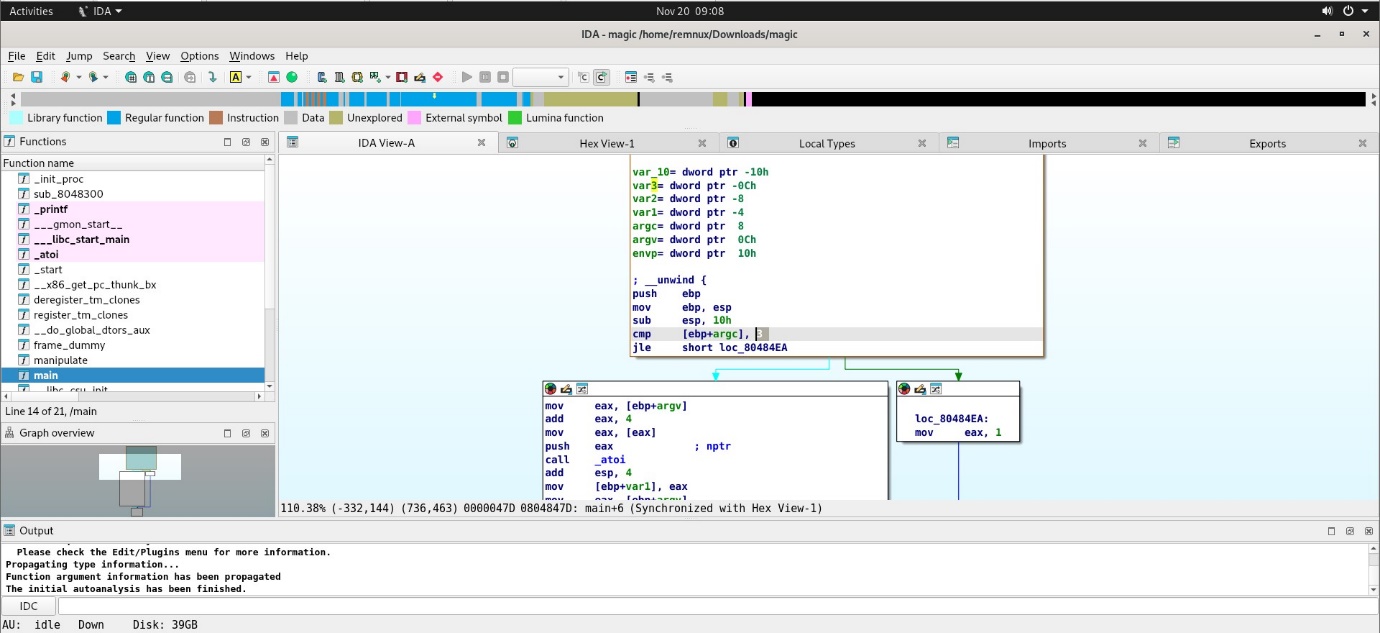
Name: Rithik Sarvesh Bharathiraja

UID: 120395246

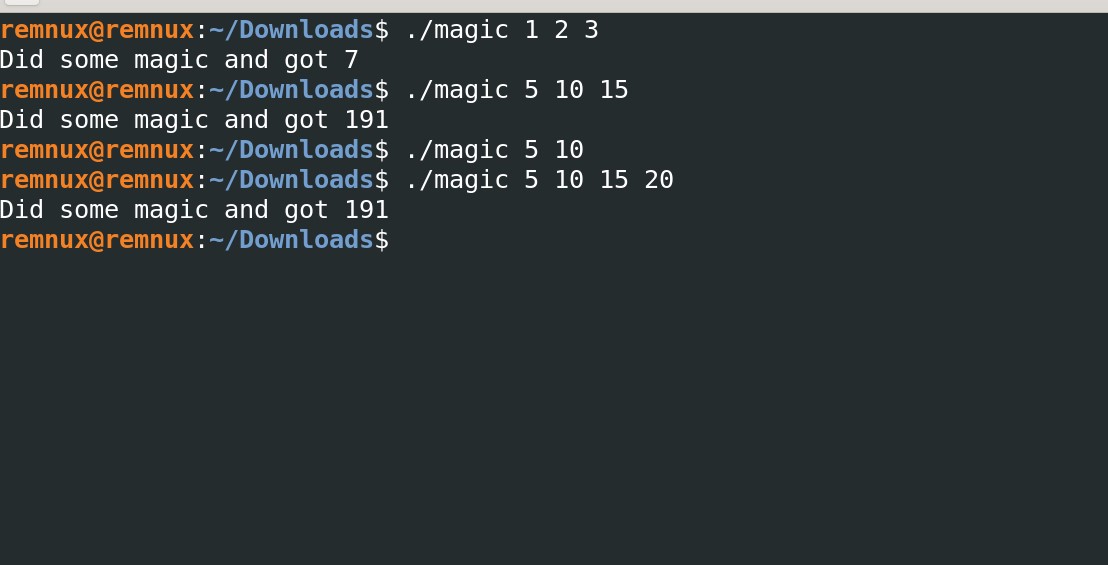
Task: Assignment 4

Course: ENPM696

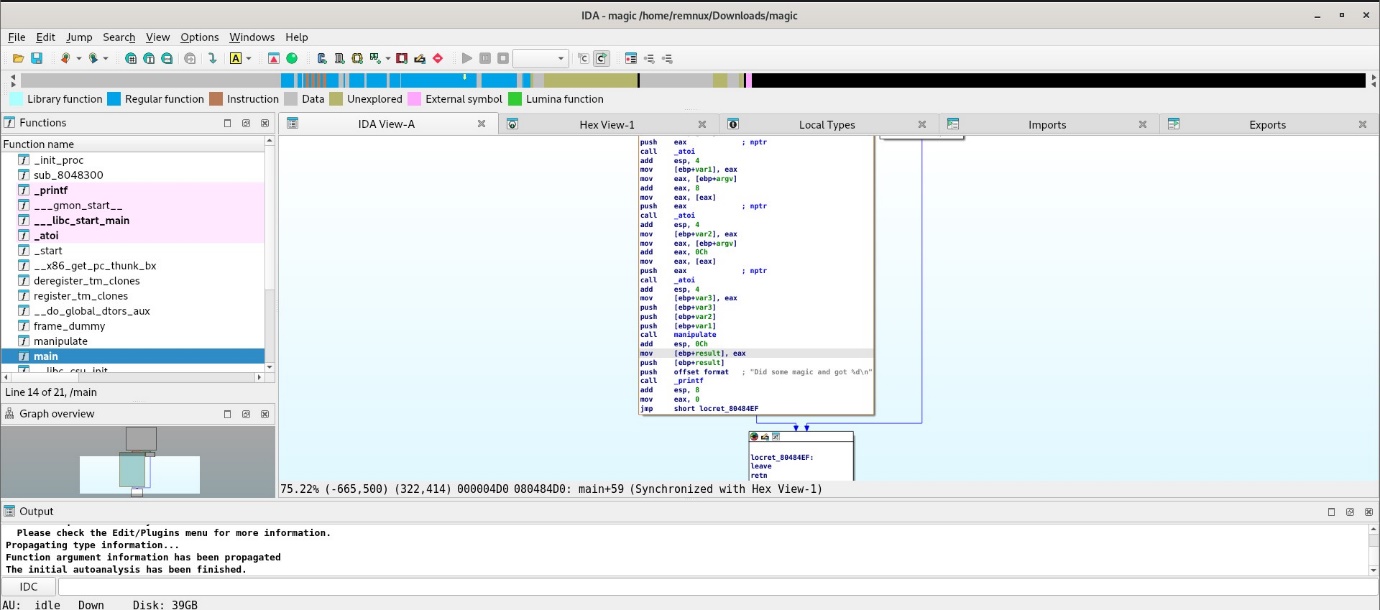
1. a) argc is compared with 3. Following that “jle” instruction is used. By that, we can infer that this binary requires more than 4 arguments (including the binary name) to execute its functionality.



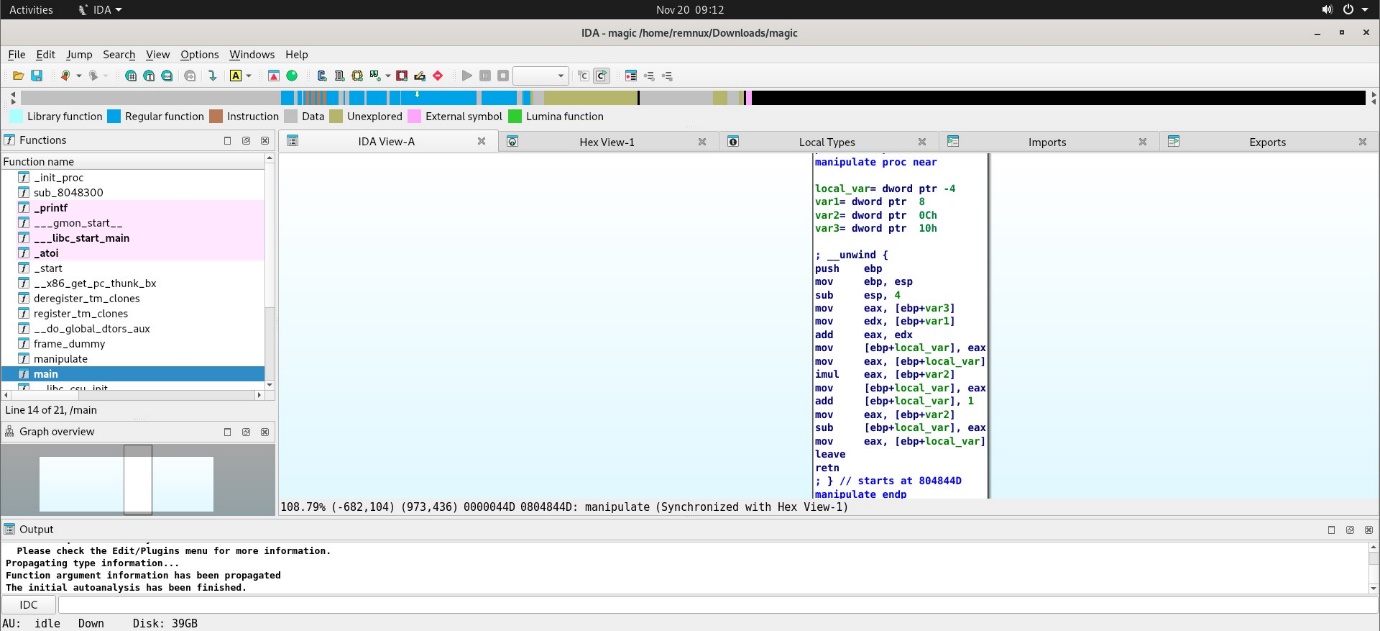
When given the 3 arguments with the binary name, it prints the following line, “Did some magic and got {int}”. If 4 arguments were given, it takes only the first three arguments.



b) In the main function, all the three arguments were fed into the atoi() function to convert them into an integer. After all the three were converted, all of those arguments were fed to the next function manipulate()



In the manipulate() function, first and third arguments were added together, later the summation is multiplied by the second argument. And then, the result is incremented by 1 and subtracted by second argument again. Finally, the result is returned to the main function where it is printed along with the print statement.



Let’s say, there are three variables, var1,var2, var3

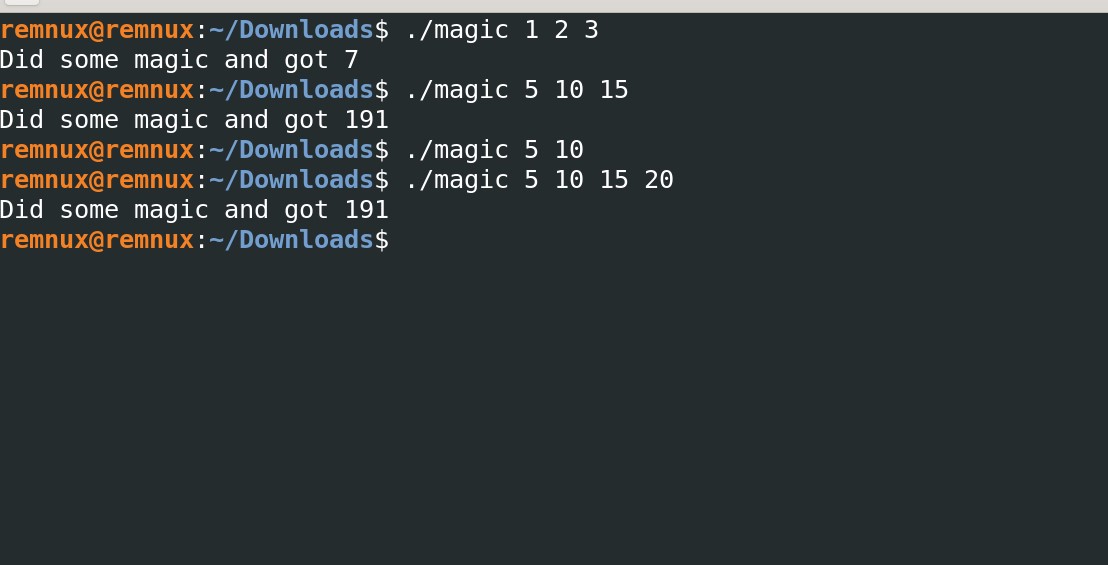
Result = [(var1+var3)\*var2]+1-var2

For input 5, 10, 15

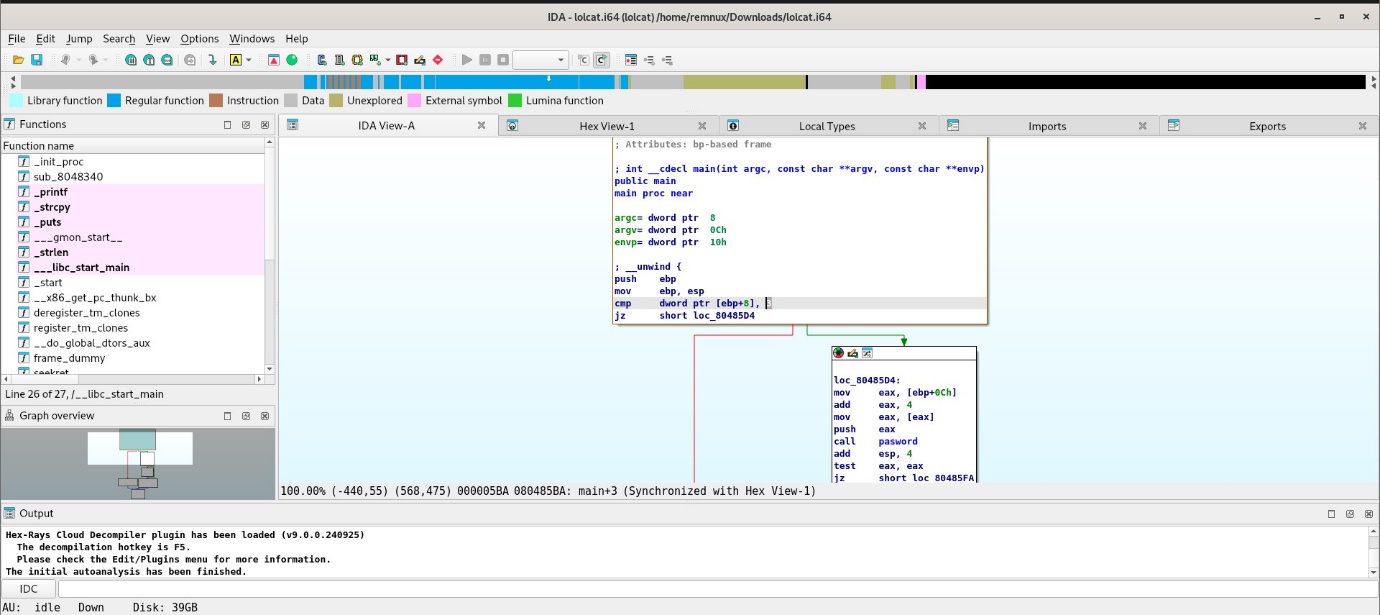
Result = [(5+15)\*10]+1-10 = 191

For input 1, 2 and 3

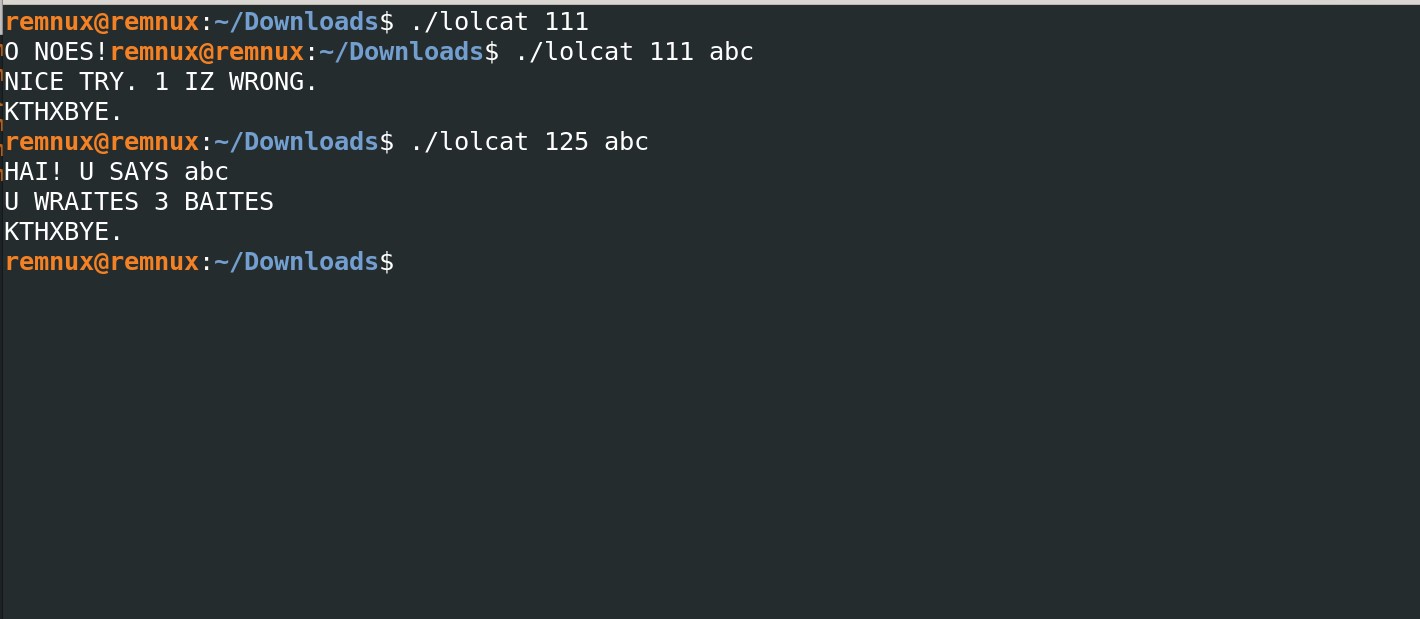
Result = [(1+3)\*2]+1-2 = 7



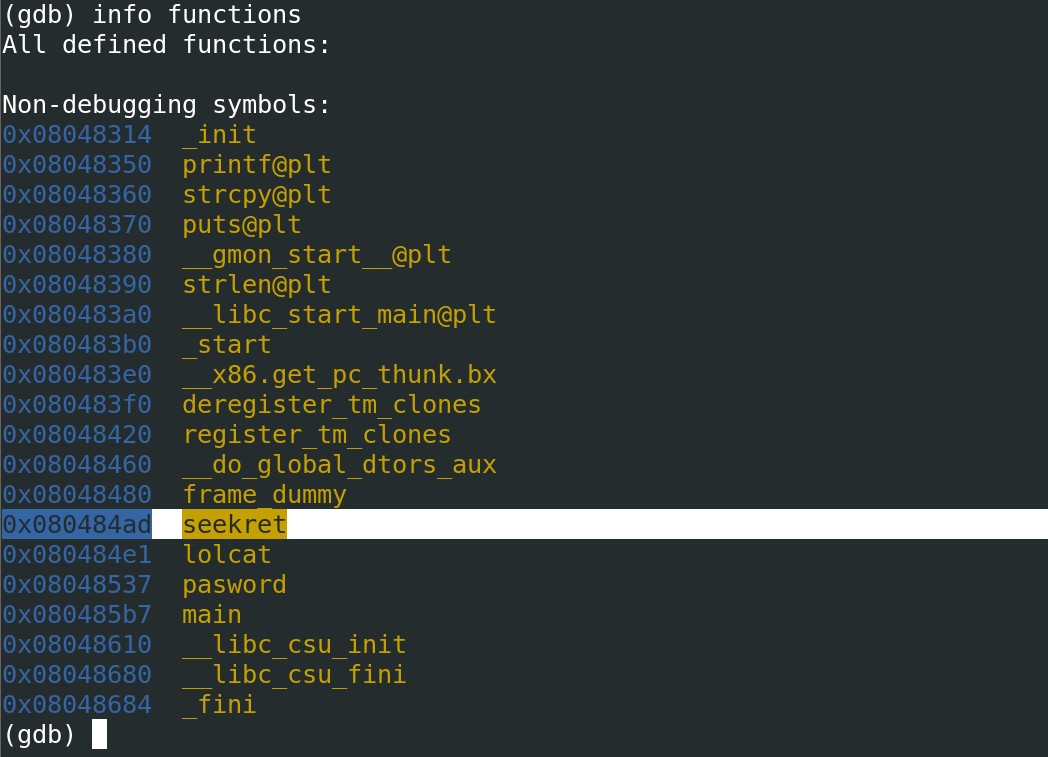
(ebp+8) was compared with 3 which means it takes 3 arguments as input (including the binary name) or else the program exits.



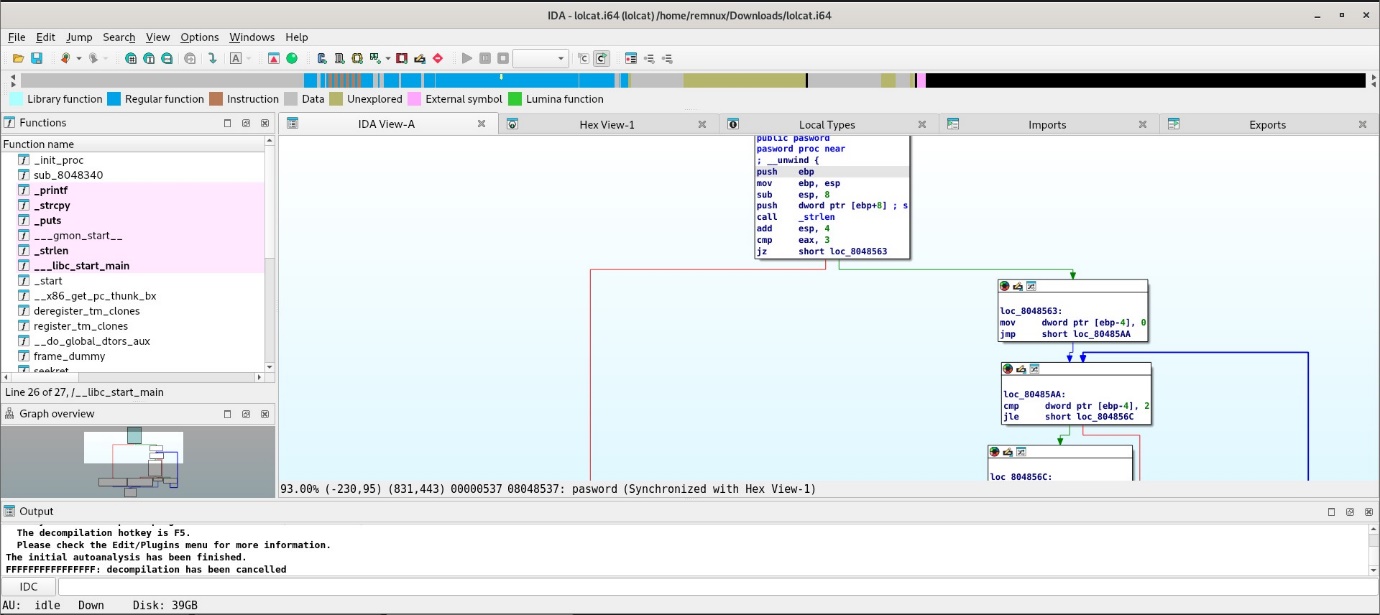
When only one input is given, the program exits with the print statement “O NOES!”. When given 2 arguments, it performs different functionality.



There are three functions other than the main() in this binary: seekret(), lolcat() and pasword(). First the pasword() is being called. If entered the right password, lolcat() will be called after that. If not, the program will be exited. seekret() was defined in the binary but it was never called.



The pasword() function requires of input size 3. Otherwise, it exits. Once it finds that the first given input is of size 3. It runs a loop.

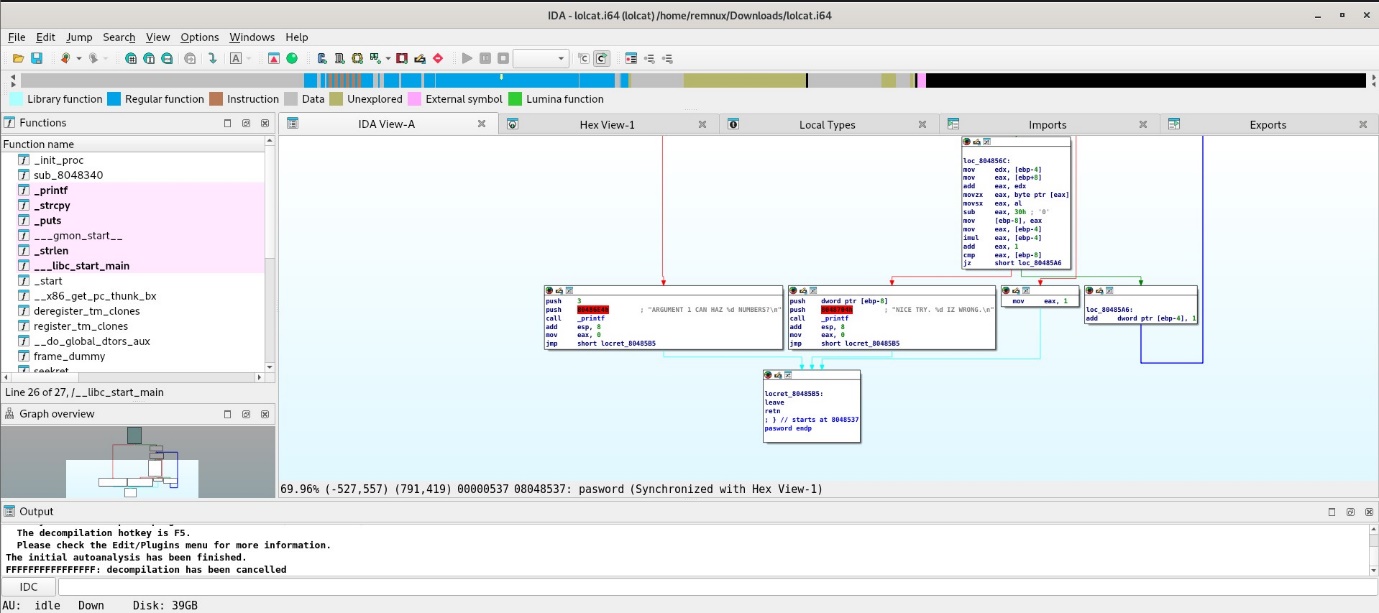


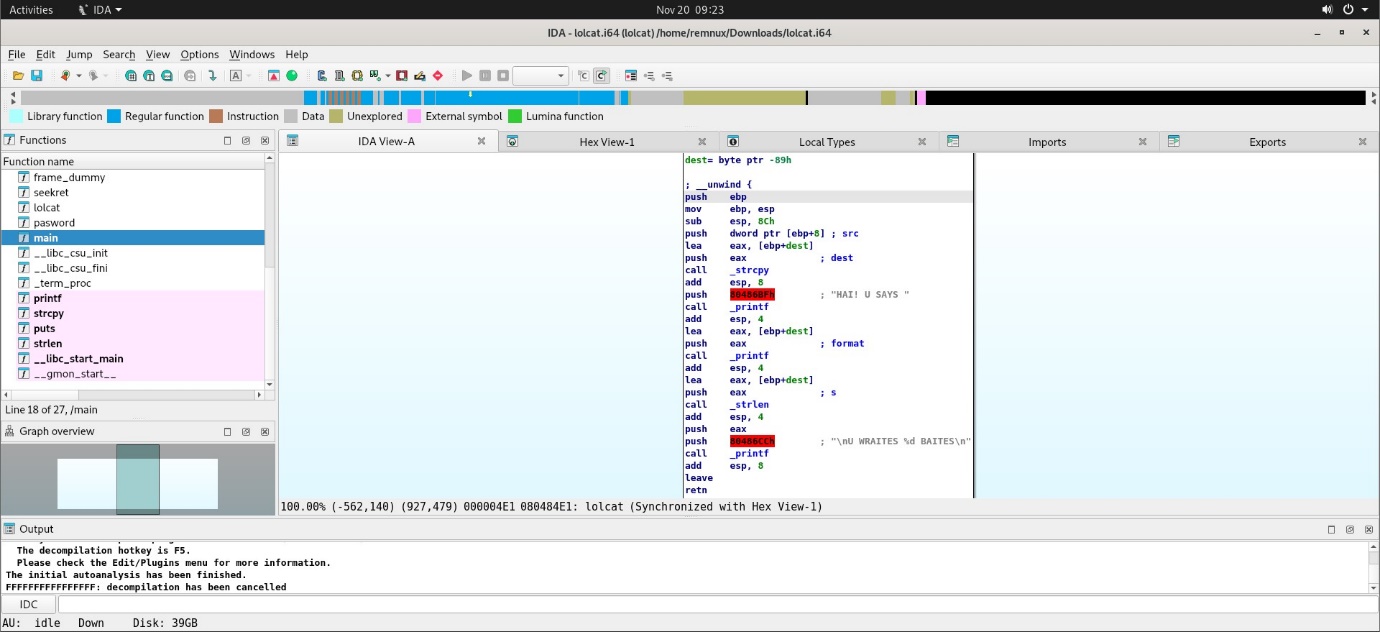
Here we can infer that some operation is being performed on the i value of the loop and ith value of the first argument.

The function takes the hex value of the argument. And then, it subtracts 0x30 from it. Later, the result is being compared with i2 + 1.

We know that the first argument is of size 3. Thus, possible i values are 0,1,2.

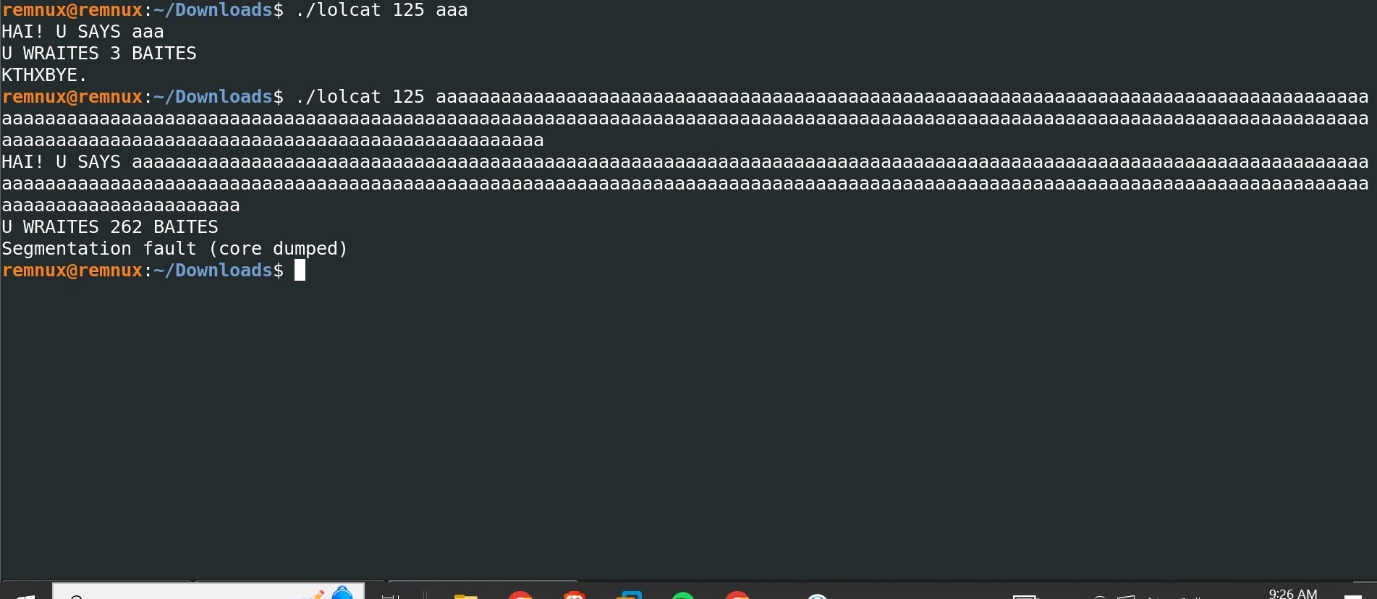
For that, i2 + 1 will be 1,2,5. ith value of the arguments are subtracted by 0x30. Thus, they should 0x31,0x32, 0x35. So that, they can match the other side of the equation. With the ASCII table, the text representation of those hex numbers are 1,2 and 5. Thus, the password is “125”. With that, lolcat() can be executed.



Inside the lolcat(), we can see that the user-defined input is being stored in a 0x89 byte buffer using strcpy(). And then, it prints a print statement and followed by the user-defined input (or the second argument) and its size in another pre-defined print statement. 

If it’s not a protected and writable stack, we can use this vulnerability to perform buffer overflow to reach the EIP register to make it run the instruction address we desire.

To know that, let’s construct a payload with password ‘125’ at first and then the second argument. When we gave 262 bytes(that’s a random number) of second argument, we can see that, code is terminated due to segmentation fault.(Interesting ?!). Let’s try to crafty payload to verify this.



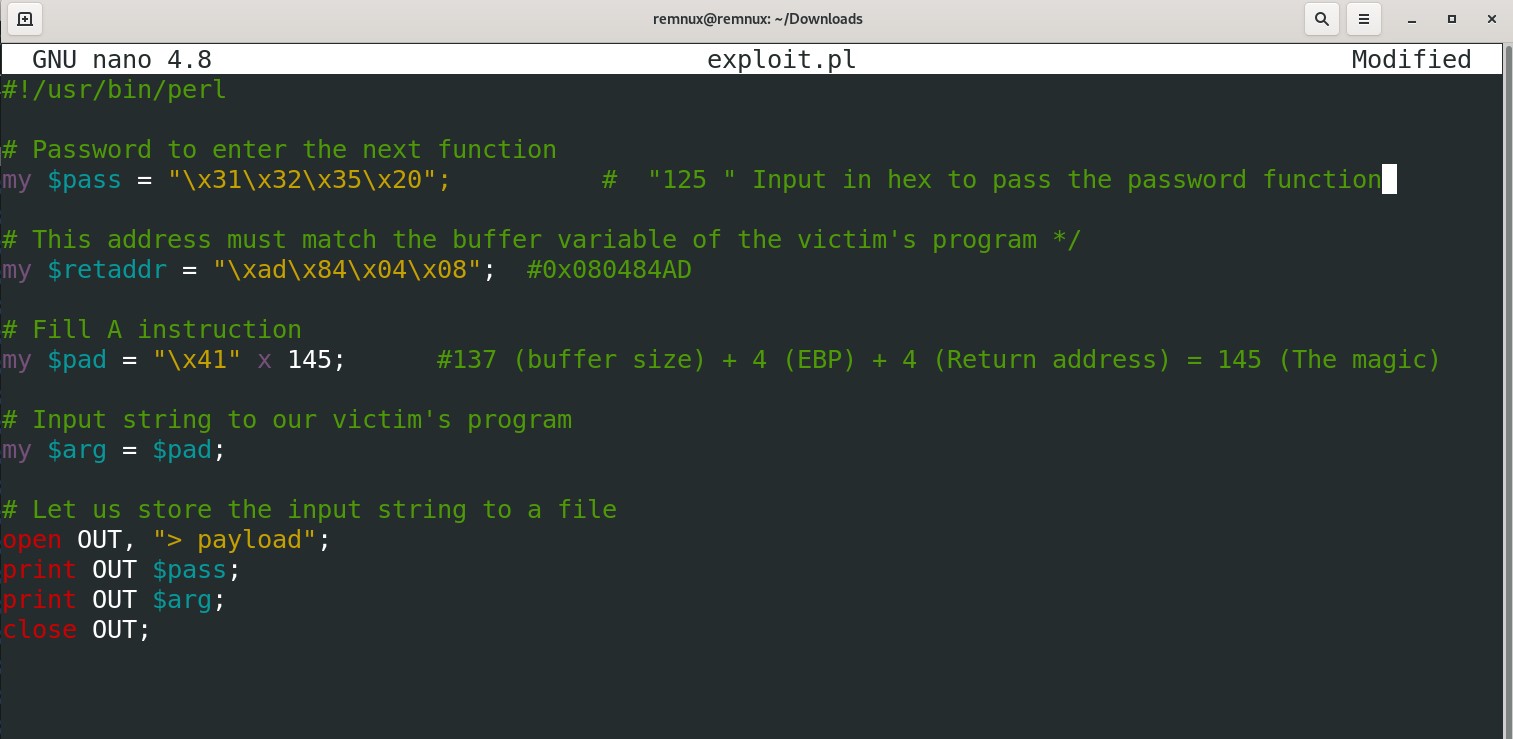
This perl code was part of the ENPM691(It was put in pre-requisite for a reason). Here we generate 145 ‘A’ to test the vulnerability. (0x41 is the hexadecimal value for “A”).

First, let’s pass the hexadecimal value of the first argument followed by the space character to separate it from the second argument.   
Reason for that magical number – 145  
Size of the buffer -0x89 = (137 in decimal)

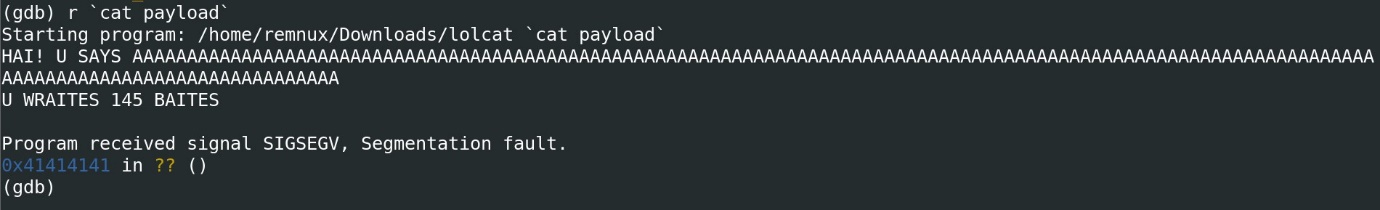
EBP size – 4 bytes

Return address size – 4 bytes

Thus, in total it’s 145 bytes. If we were right, the program should try to execute the address 41414141.



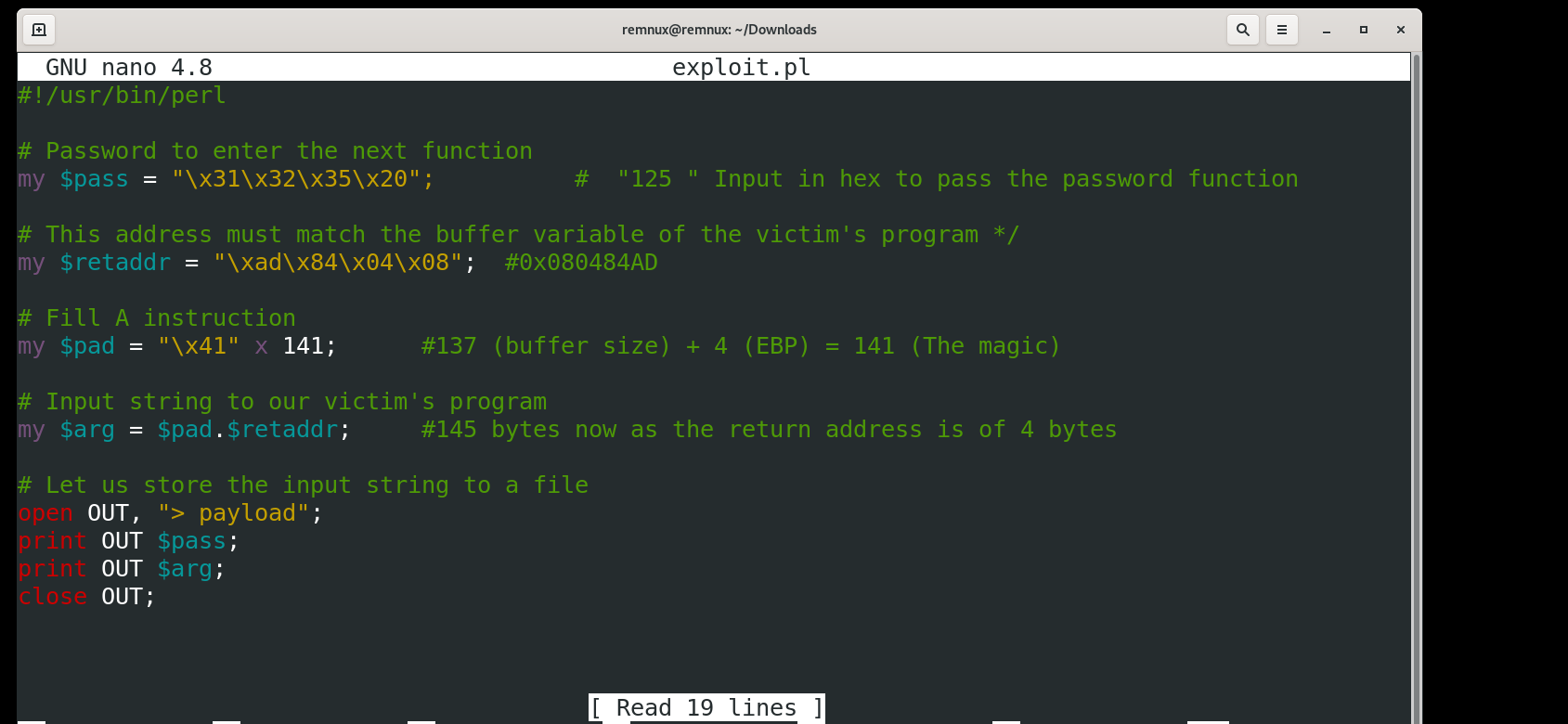
With the given input, it’s verified that it’s vulnerable to buffer overflow and it has a writable stack too.



In the exploit.pl file, this time let’s change the parameter.

First, first argument is present and followed by 141 ‘A’ to reach the return address space.

In the return address space, let’s pass the address of the seekret() function which is 0x080484ad (It’s given in reverse order as it’s Little Endian)



When the payload is passed on to the program. It prints the following statement. “U DISCOVERED MAH SEEKRET  
IT 42”

. Let’s try to find the logic behind that number 42.



Integer 7 is moved to the (ebp-4). And then, Subtracted by 1. The result (6) is multiplied with (ebp-4) which has 7 in it. Finally, the result (42) is printed along the statement “IT %d”.

