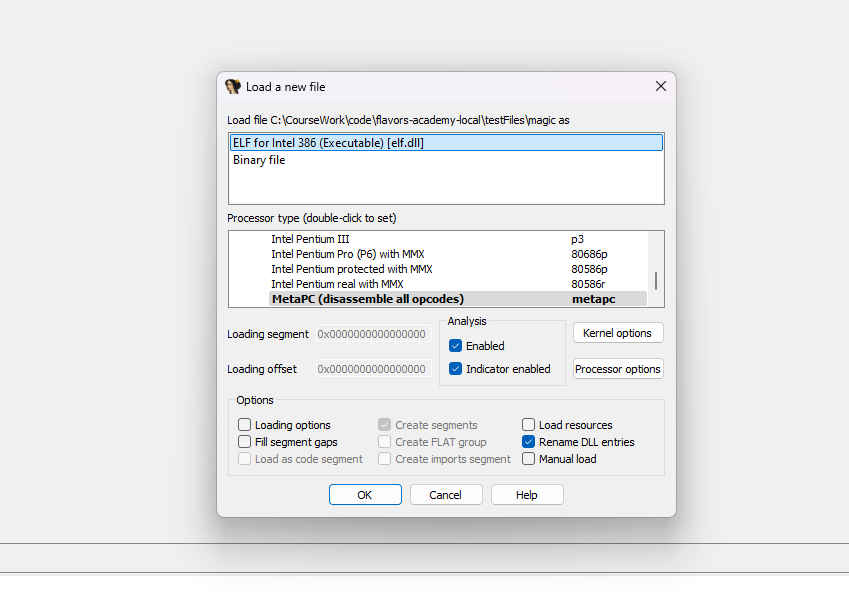
**Answer to the question no 1**

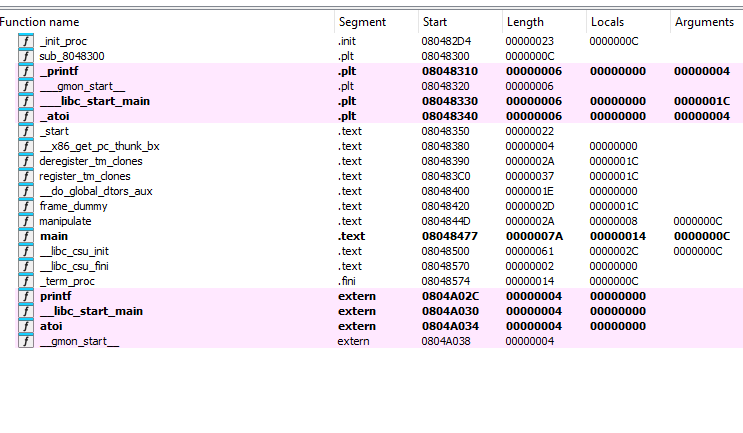
Tools used: Ida Pro and gdb in remnux virtual system.

Here are the steps and evidence for analyzing the binary:

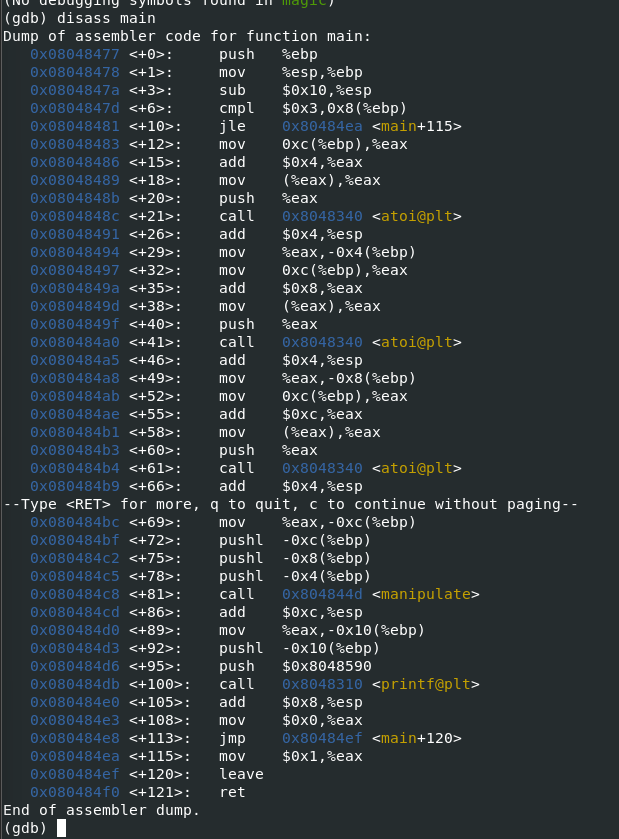
1. Opened magic binary in Ida pro and inside a terminal in remnux. I used chmod +x magic command to configure the permission of the binary. Then I opened that file in Ida pro.



1. I investigated the functions. It seems like it has main and manipulate functions.

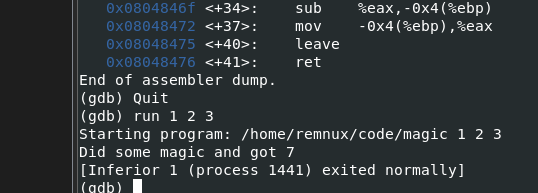


1. I went to gdb and disassembled main.

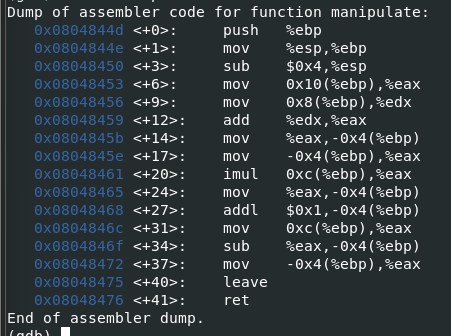


* In here it is using atoi to convert 3 arguments into integers:  
  0x0804848c <+21>: call 0x8048340 <atoi@plt>  
   0x080484a0 <+41>: call 0x8048340 <atoi@plt>  
   0x080484b4 <+61>: call 0x8048340 <atoi@plt>
* It is calling manipulate function here:  
  0x080484c8 <+81>: call 0x804844d <manipulate>
* It is pushing the result in here:  
  0x080484d6 <+95>: push $0x8048590.
* Finally, it is printing something here:  
  0x080484db <+100>: call 0x8048310 <printf@plt>

1. I ran the program in gdb with 3 argument 1, 2, 3 and it generated this result:



1. The "magic" of this program lies in the manipulate function, which performs operations on the three integer arguments and returns a result. Let’s look sat the disassembly of the function manipulate:

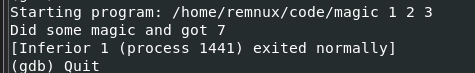


* The function takes 3 arguments:
  + Arg1 at 0x8(%ebp)
  + Arg2 at 0xc(%ebp)
  + Arg3 at 0x10(%ebp)
* Here it is loading the argument3 into %eax  
   0x08048453 <+6>: mov 0x10(%ebp),%eax
* Here it is loading argument 1 into %edx  
   0x08048456 <+9>: mov 0x8(%ebp),%edx
* Here it is conducting addition between argument 1 and argument 3:   
  0x08048459 <+12>: add %edx,%eax
* Here it is storing the result in a local variable.  
  0x0804845b <+14>: mov %eax,-0x4(%ebp)
* Here it is loading the addition result in %eax.   
   0x0804845e <+17>: mov -0x4(%ebp),%eax
* Multiplying argument 2 with the addition result.   
  0x08048461 <+20>: imul 0xc(%ebp),%eax
* Storing the multiplication result in a local variable.  
  0x08048465 <+24>: mov %eax,-0x4(%ebp)
* Incrementing multiplication result by 1.  
  0x08048468 <+27>: addl $0x1,-0x4(%ebp)
* Moving argument 2 into %eax  
  0x0804846c <+31>: mov 0xc(%ebp),%eax
* Subtracting the argument 2 from the latest addition result.   
  0x0804846f <+34>: sub %eax,-0x4(%ebp)
* Loading final result into %eax   
  0x08048472 <+37>: mov -0x4(%ebp),%eax
* Cleaning up stack frame   
  0x08048475 <+40>: leave
* Returning the result to main.  
  0x08048476 <+41>: ret

So, the magic binary’s manipulate function commute this:

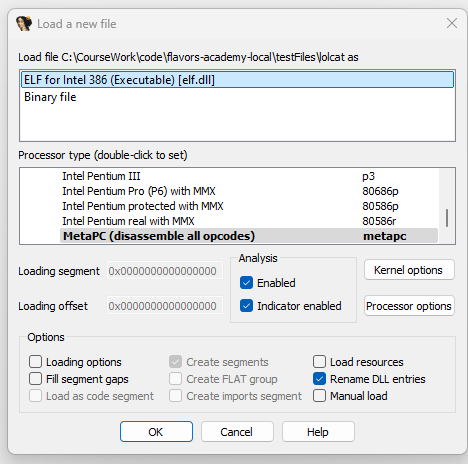
((argument 1 + argument 3) X argument2 + 1 ) – argument2)

Finally, it returns the result to the main function and the main function prints this out if the program is run with the arguments 1, 2, 3:

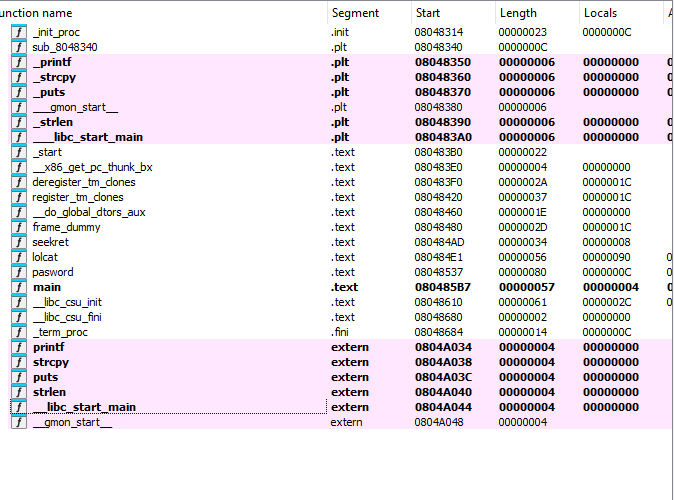


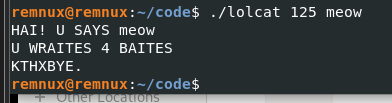
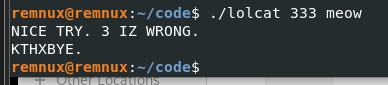
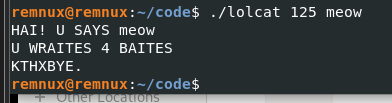
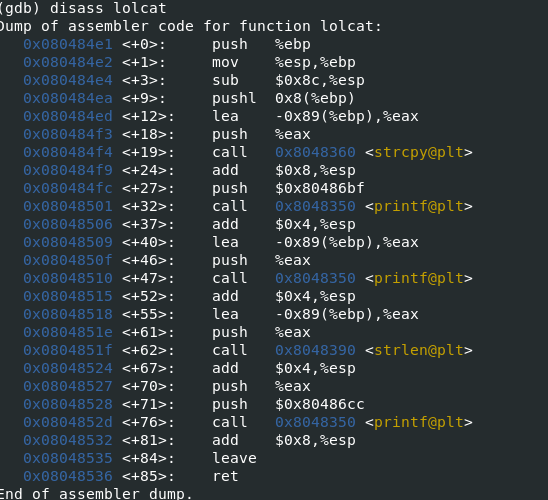
Answer to the question no 2:

1. Opened lolcat binary in Ida pro and inside a terminal in remnux. I used chmod +x magic command to configure the permission of the binary. Then I opened that file in Ida pro.

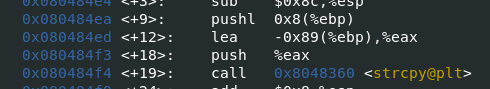


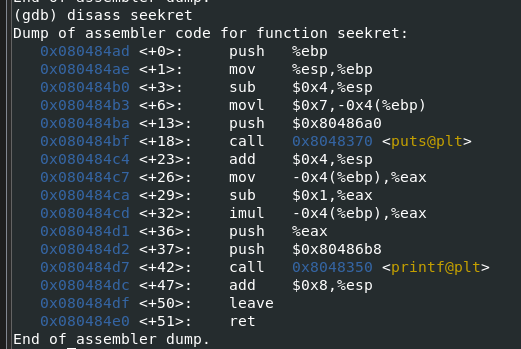
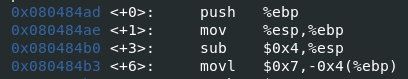
1. I investigated and saw that it has 4 functions, main, password, lolcat and seekret.



1. This program takes 2 arguments. The first argument validate as a password in the password function and the 2nd one is processed by lolcat, if the password is correct.
2. I ran the program using these 2 arguments i)125 ii) meow. This is the output:  
   
3. I ran the program with these 2 arguments i) 333 ii) meow and it generated this result:  
   
4. The lolcat function is called when the password validation in the password function succeeds. The lolcat function takes the second program argument as input, processes it, and prints:
   1. The input strings.
   2. The length of the string.
   3. To test the functionality of lolcat, I provided valid inputs to the program and analyzed its behavior using runtime testing and assembly code.
   4. Running ./lolcat 125 meow generates this output  
      
   5. lolcat successfully processes the second argument (meow), prints it, and reports the string length (4 bytes).
   6. Disassembly of the function lolcat  
      

**Evidence:**

* 1. The call to lolcat in main occurs after successful password validation  
     
  2. Copy the String to a Local Buffer:  
       
     argv[2] is copied into a local buffer at -0x89(%ebp) using strcpy
  3. Calculate the string length:  
     
  4. The length of the copied string (4 for "meow") is calculated using strlen.
  5. Print the length:  
       
     The program prints the length of the string (U WRAITES BAITES).

1. The **"secret"** in this program is the hardcoded value 42, which is computed and revealed in the seekret function.
   1. Disassembly of the secret function:  
      
   2. Setup and Initialization.  
      

The function initializes a local variable at -0x4(%ebp) to 7.

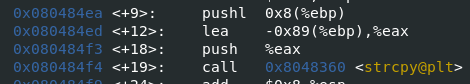
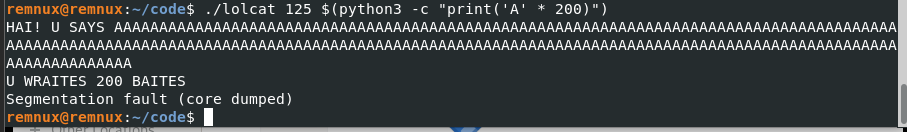
* 1. Compute the value:

  
This calculates = (7-1) X & = 42

* 1. Prints the result:  
       
     The value 42 is pushed to the stack and printed with a format string from memory.

Answer to the question no 4 & 5

Yes, the program has vulnerabilities. These vulnerabilities can be exploited under certain conditions. Below is a list of vulnerabilities, their evidence from the assembly code, and their implications.

* 1. Buffer overflow in lolcat:
     1. Buffer Allocation:  
        0x080484e4 <+3>: sub $0x8c,%esp  
        This program reserves 140 bytes for local buffer at -0x89(%ebp)
     2. Copy input without bounds checking:  
        
     3. strcpy copies the entire input string into the buffer without verifying its length.
     4. If argv[2] exceeds 140 characters, the stack is overwritten, potentially overwriting the return address.
     5. Exploitation  
        

This crashes the program.

* 1. Missing Input Validation in password:  
     The password function converts each character of argv[1] to an integer but does not validate whether the characters are numeric. If non-numeric characters are provided, the program may behave unpredictably.
     1. Evidence:  
        
     2. Character conversion in password:  
        
     3. The function processes each character of argv[1] as if it were numeric (e.g., '0' becomes 0, '1' becomes 1, etc.).
     4. For non-numeric characters (like 'a'), subtracting 0x30 (ASCII for '0') results in a nonsensical value:
     5. ASCII for 'a' is 97. 97−48=49
     6. For the input abc, the first character a converts to 49.
     7. The function compares this to formula char[n] = (index)^2 + 1
     8. The formula expects 0^2 + 1 = 1 for the first character
     9. 49 does not match 1 so the program fails.
     10. The program prints mismatch 49 value in this error message  
         