William Hupp
Reverse Engineering I
CIS 5930

Reverse Engineering I Write-Up

Install the tools!

10 Points

Problem:

Provide a screen shot that you have the latest versions of the following tools installed. Install instructions are located in the slides.

- Radare2
- Ipython

If you wish to you wish to use different tool/s please provide a brief description of the tool/s and which tool/s from above are being replaced. NOTE: You are unsupported if you have technical problems with a tool not listed above. Also do not use non-free, non-open-source software.

flag{R3_w33k_0n3}

Solution:

flag{R3_w33k_0n3}

Journal

10 Points

Problem:

Keep a journal of all x86 instructions you did not already know. You must have a minimum of 10 instructions in your journal. If needed find 10 instructions online that you don't know and add them to your journal.

Submit your journal as part of homework submission.

flag{3asy_p0int5}

Solution:

imul –The imul instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above).

The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e. bx) operand must be a register. – **imul bx, 12h**

The three operand form multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.

mov - The mov instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address. - **mov eax, ebx (copy the value in ebx into eax)**

lea - Load effective address - The lea instruction places the *address* specified by its second operand ([val]) into the register specified by its first operand. Note, the *contents* of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region. - **lea eax, [val] (the address of val is placed in eax)**

inc - Increment - The inc instruction increments the contents of its operand (eax) by one. - **inc eax**

dec – Decrement - The dec instruction decrements the contents of its operand (eax) by one – **dec eax**

not - Bitwise logical not - Logically negates the operand contents (that is, flips all bit values in the operand eax) – **not eax**

neg - Negate - Performs the two's complement negation of the operand contents (eax) – **neg eax**

cmp - Compare - Compare the values of the two specified operands (eax, ebx), setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand. - **cmp eax, ebx**

push – Push onto the stack - The push instruction places its operand (eax) onto the top of the hardware supported stack in memory. Specifically, push first decrements ESP by 4, then places its operand into

the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses. **push eax**

pop – Pop the stack - The pop instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register (eax) or memory location). It first moves the 4 bytes located at memory location [SP] into the specified register or memory location, and then increments SP by 4. - **pop eax**

flag{3asy_p0int5}

.bss

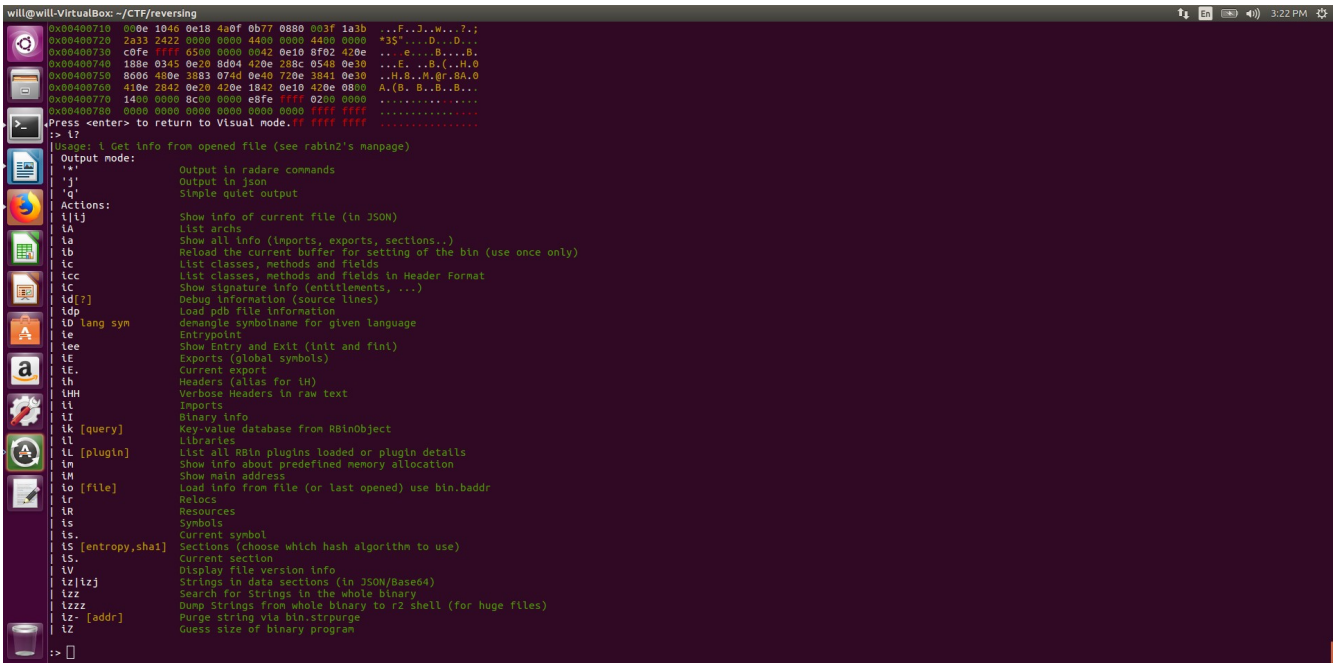
10 Points

Problem:

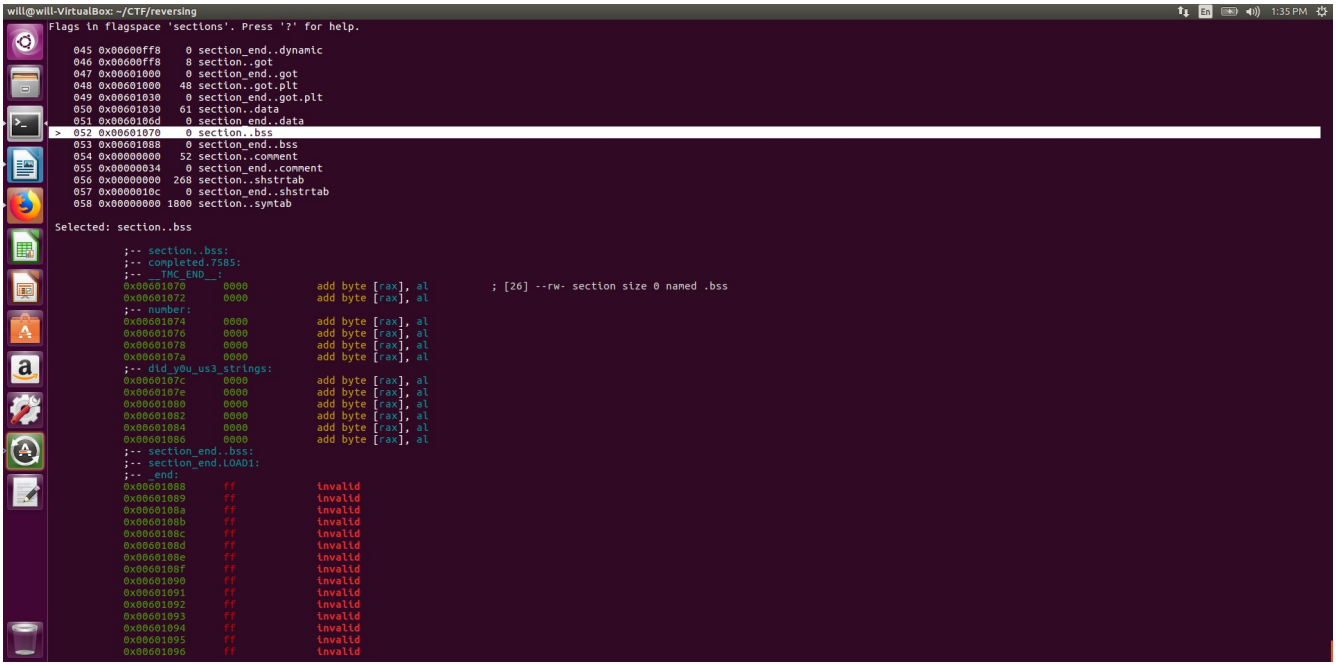
Analyze the challenge binary and find the flag in the .bss section. Add flag{} to your found flag. NOTE: 1pt deduction per flag submitted that is from a different section.

Solution:

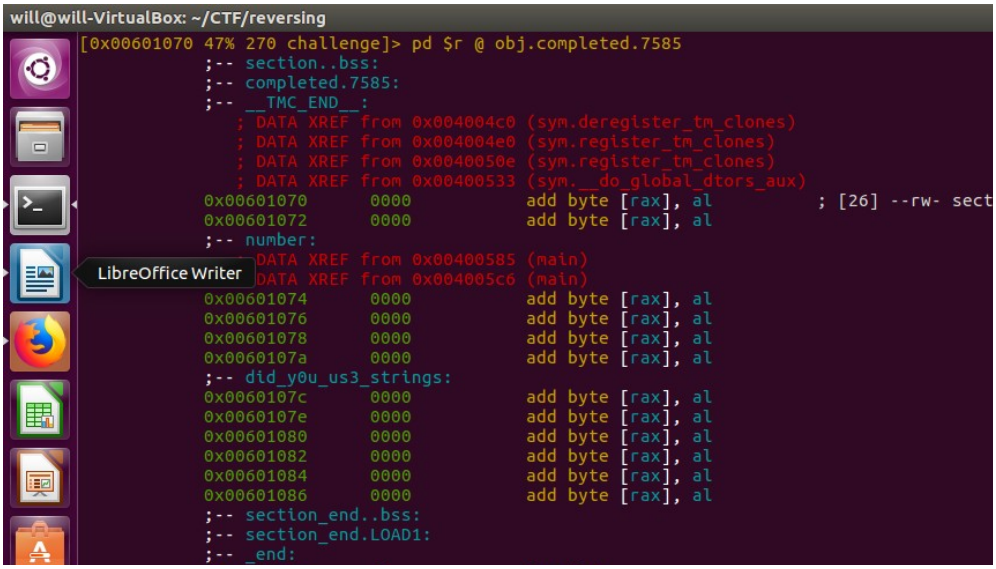
This is a reversing challenge on a file called ‘challenge’. Since it is a reversing problem I must use Radare. I started Radare with ‘r2 challenge’. I immediately typed ‘aaaa’ to analyze the binary. I then hit ‘V’ to enter visual mode. I then used the ‘:i?’ command.



I tried ‘iS’ under Brandon’s suggestion.



I find the bss section, which is where the flag is. The second column contains the address of that section in Radare. The address for bss is 0x00601070. I navigate there with ‘s 0x00601070’. After hitting ‘V’ again I page through to the instructions with ‘p’ and see the flag after scrolling down.



flag{did_y0u_us3_strings}

.data

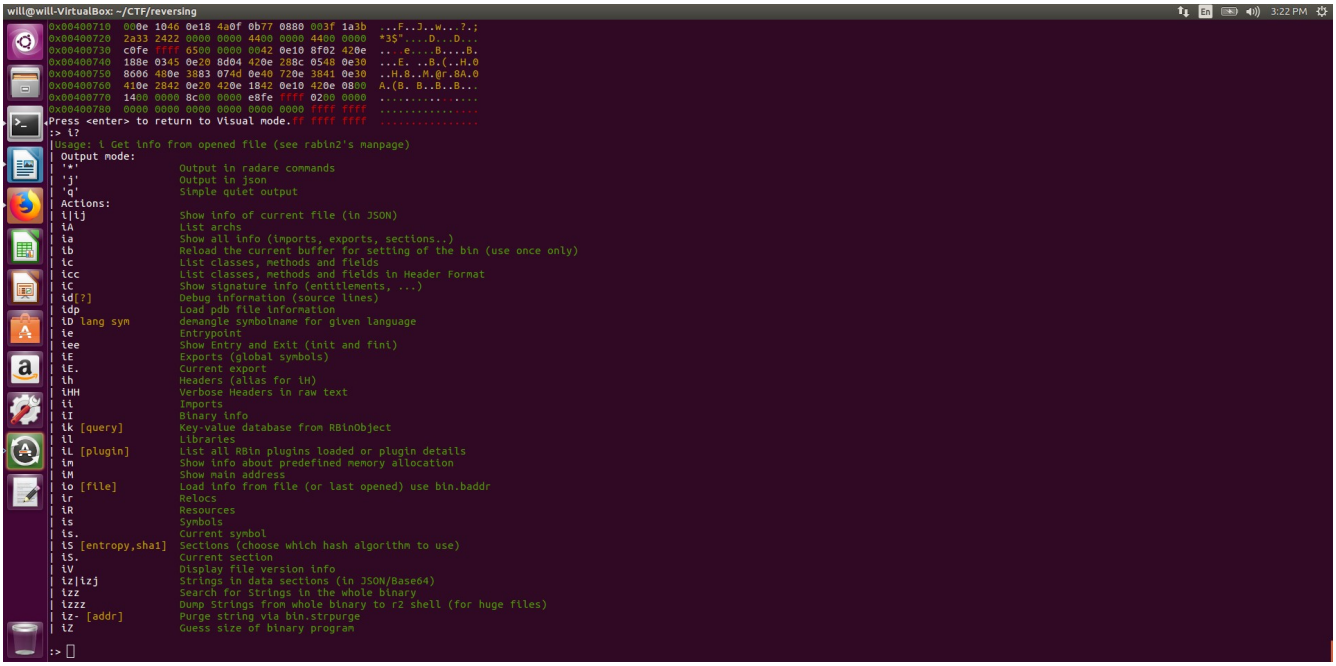
10 Points

Problem:

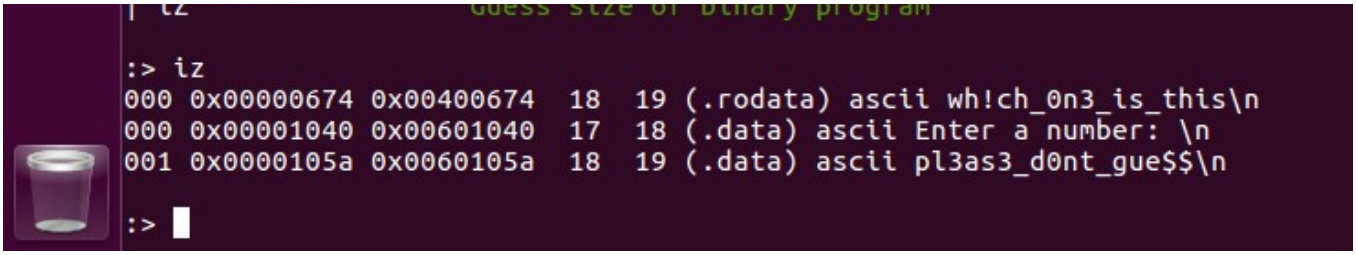
Continue to analyze the challenge binary and find the flag in the .data section. Add flag{} to your found flag. NOTE: 1pt deduction per flag submitted that is from a different section.

Solution:

This is a reversing challenge on a file called ‘challenge’. Since it is a reversing problem I must use Radare. I started Radare with ‘r2 challenge’. I immediately typed ‘aaaa’ to analyze the binary. I then hit ‘V’ to enter visual mode. I then used the ‘:i?’ command.



I notice the ‘iz’ command gives strings contained in the data section.



I used that command and see there are 2 strings in the .data section. One of them looks like a flag and sure enough it is.

flag{pl3as3_d0nt_gue$$}

.rodata

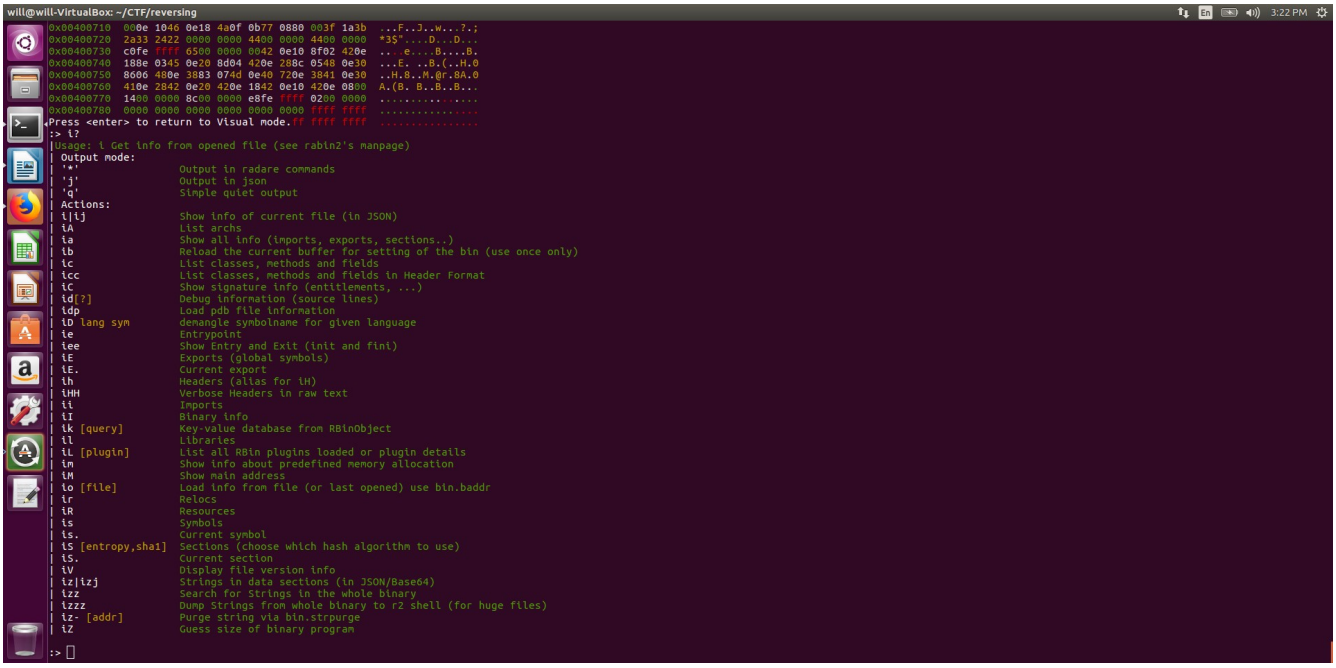
10 Points

Problem:

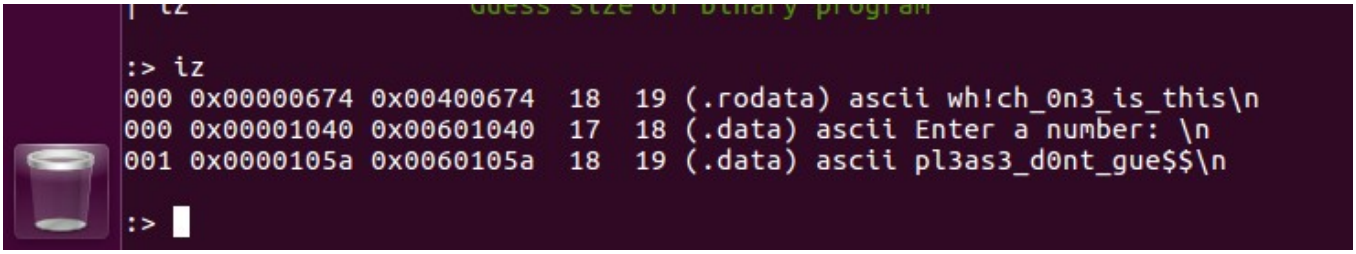
Continue to analyze the challenge binary and find the flag in the .rodata section. Add flag{} to your found flag. NOTE: 1pt deduction per flag submitted that is from a different section.

Solution:

This is a reversing challenge on a file called ‘challenge’. Since it is a reversing problem I must use Radare. I started Radare with ‘r2 challenge’. I immediately typed ‘aaaa’ to analyze the binary. I then hit ‘V’ to enter visual mode. I then used the ‘:i?’ command.



I notice the ‘iz’ command gives strings contained in the data section.



I used that command and see there is 1 string in the .rodata section. It looks like a flag and sure enough it is.

flag{wh!ch_0n3_is_this}

Warm up

10 Points

Problem:

Continue to analyze the challenge binary and provide the following information in your write up:

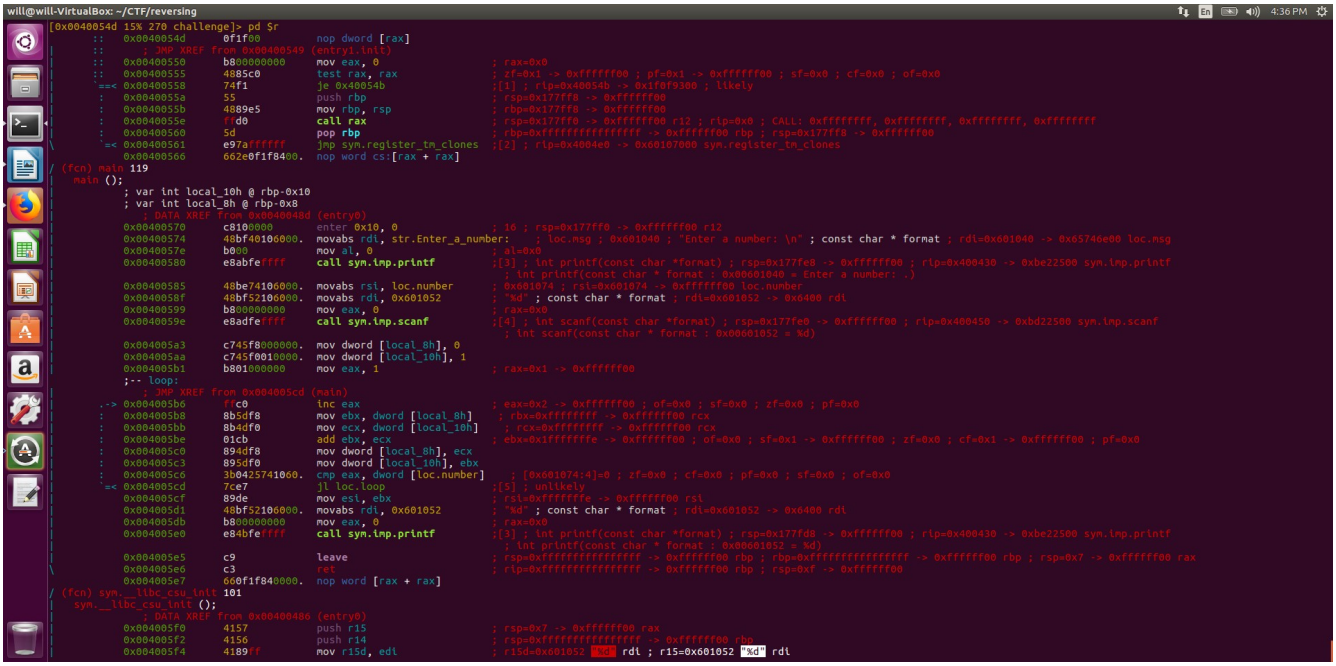
- What constructs are used in this binary? Describe how you identified each construct.
- What common programming challenge is this binary implementing (calculating) ? The answer is one word. Flag format: flag{md5(to_lower(answer))}

Solution:

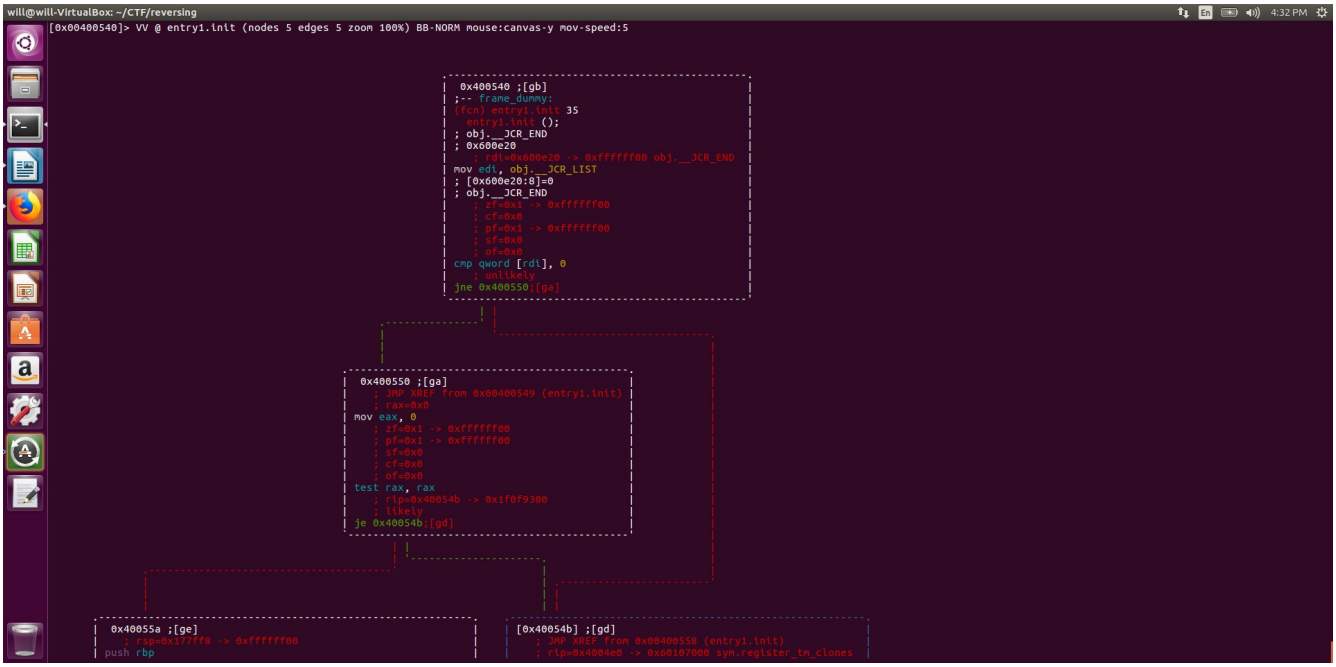
If else statement and while loop

As per usual, I opened Radare and typed ‘aaaa’ to analyze the binary

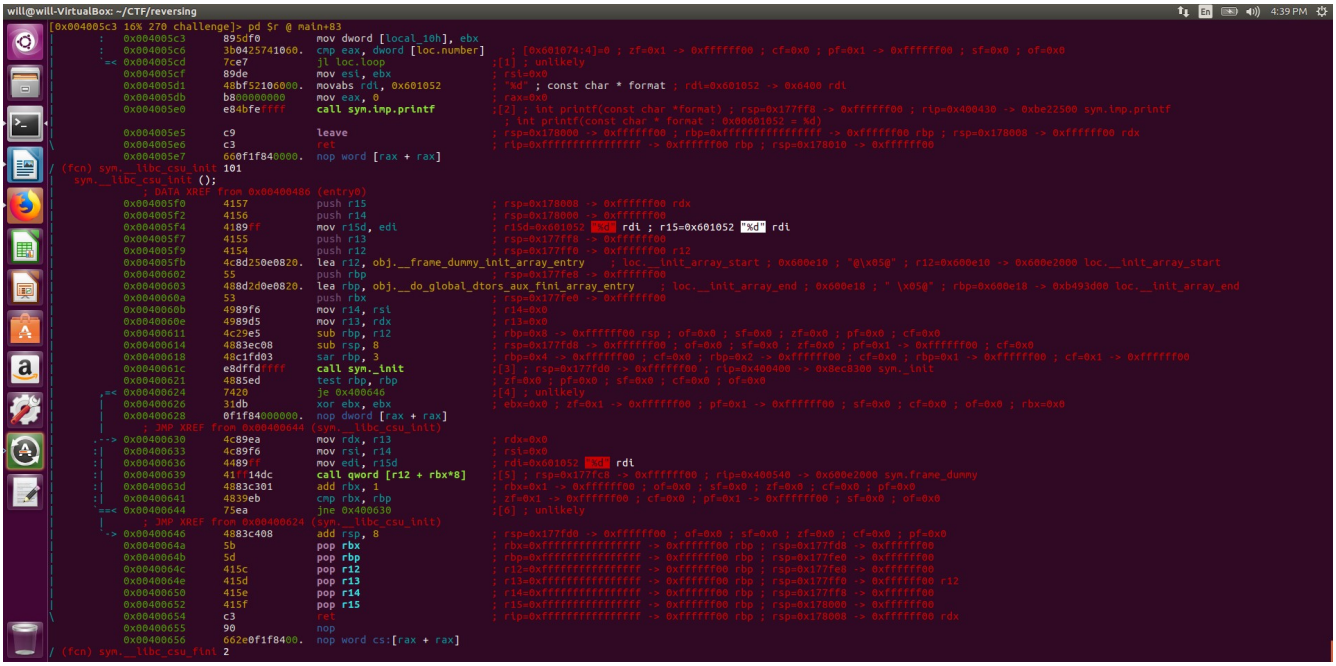
If statement: I started by analyzing main by typing ‘V’ and then ‘p’ twice to get to the instructions and scrolled to main().



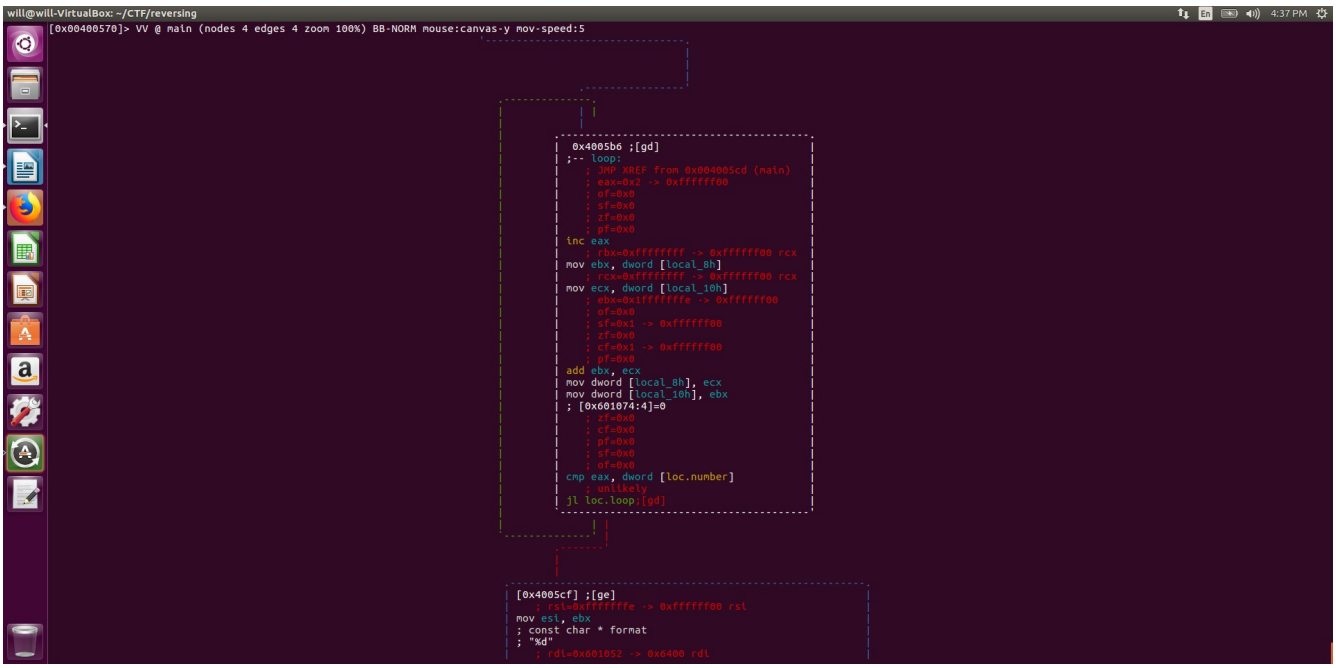
I hit ‘V’ again to open graph view and noticed two if statements. This is an if statement because there is a compare block up top. It is a compare block because it has a comparison then a conditional jump that has two paths that the program can take during execution as indicated by the red and green arrows. At the bottom of this jump there is a then block (green line) and an else block (red line). According to the notes, this is the structure of an if statement.



While loop: Scrolling down when viewing the instructions of the program, I notice an ‘sym_libc_csu_init’ function.

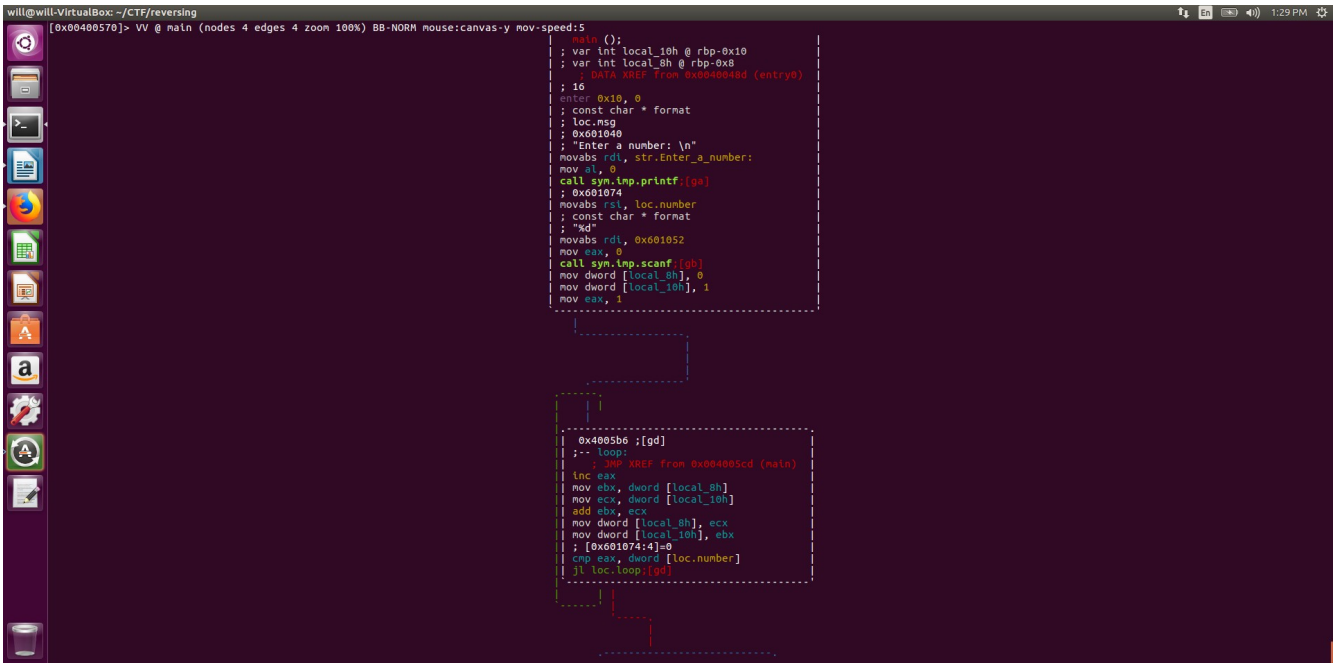


I opened that up in graph mode with ‘V’ and see the following:

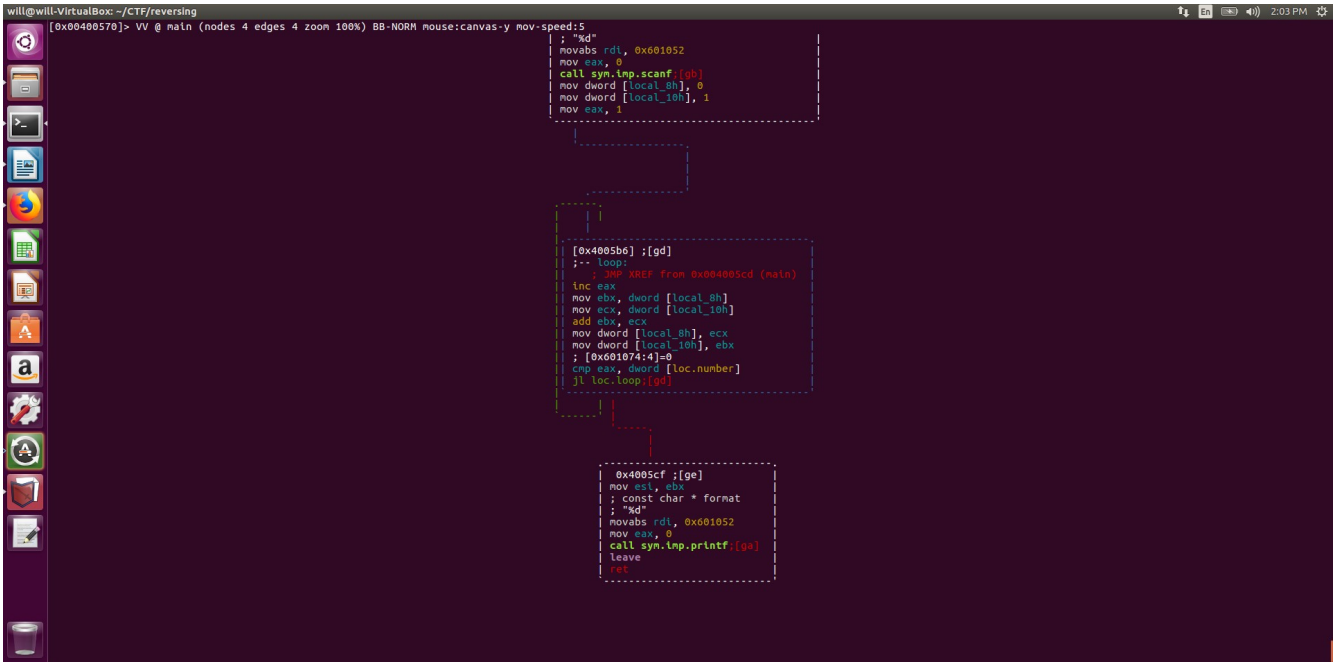


This is a while loop because at the end of a basic block there is a conditional jump that loops back to the start of a basic block. As indicated by the notes, this is the structure of a loop, since a block returns to itself. It is not a for loop because there is no incrementing block. Therefore, it must be a while loop. Also, Radare explicitly says that it is a loop.

The programming challenge this code is implementing is the Fibonacci sequence. I found out this information by analyzing main() and opening graph view.



In this portion of code, after the user enters a number, local_g8 and local_10h are initialized to 0 and 1. These are the base cases for the Fibonacci sequence. Eax is intialized to 1 and is used as the counter for the loop. An addition is performed on 0 and 1, initially, and is stored in ebx. The result of the addition (ebx) and the second operand (ecx) are used as the operands for the next loop iteration. This is accomplished with the instructions ‘mov dword[LOCAL_8h], ecx’ and ‘mov dword[LOCAL_10h], ebx’ Every iteration those local variables are used as the addition operands. The cmp eax (the counter), dword[loc.number] does the comparison for the loop to make sure the loop doesn't run more than user_enterered_number - 1 times. The loop runs as many times as the user entered number -1 as indicated by ‘jl loc.loop;’ After the loop the value of the final addition (ebx) is printed and then the program returns. This is all done in main.



I used the command ‘chmod 777 challenge’ followed by ‘./challenge’ to run the program and confirmed my results that the program is running a Fibonacci sequence. The Fibonacci value of 3 is 2 and the Fibonacci value of 10 is 55.

After running Fibonacci through an md5 encoder found at https://www.md5hashgenerator.com/ I get the flag.

flag{ef15d8edd00a6960c9c16937cbf14212}

Number

20 Points

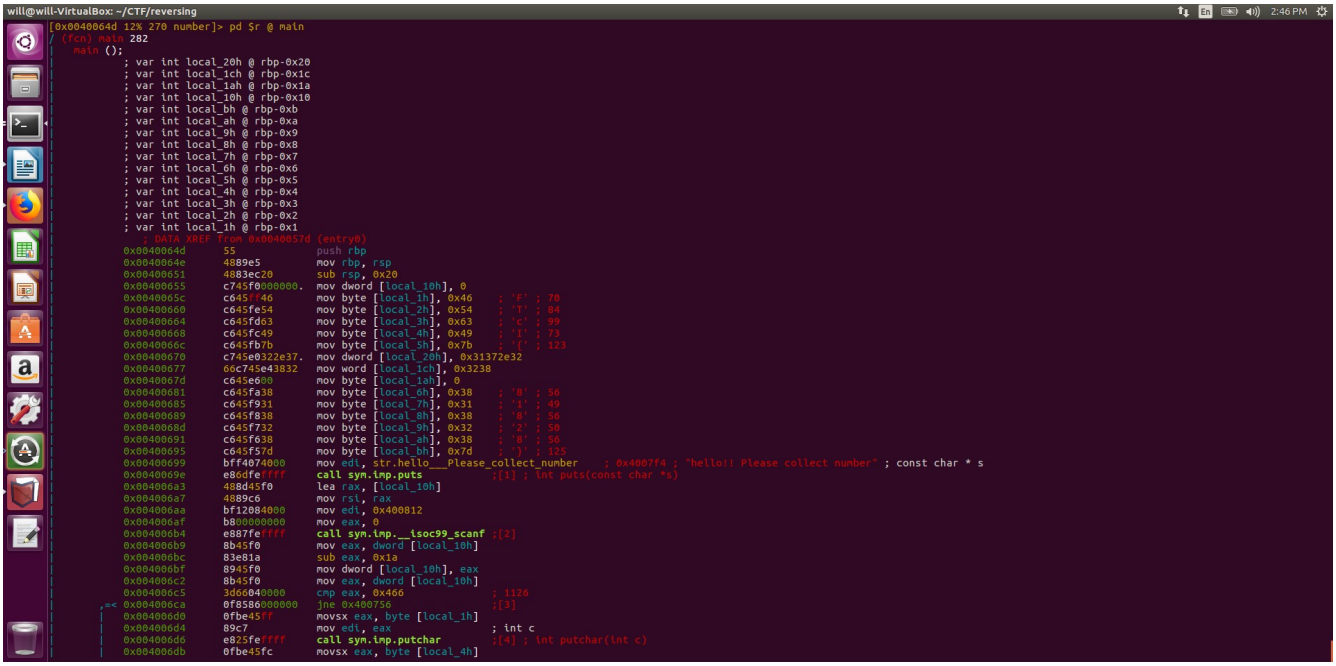
Problem:

Find the flag! Include all your steps in your write up.

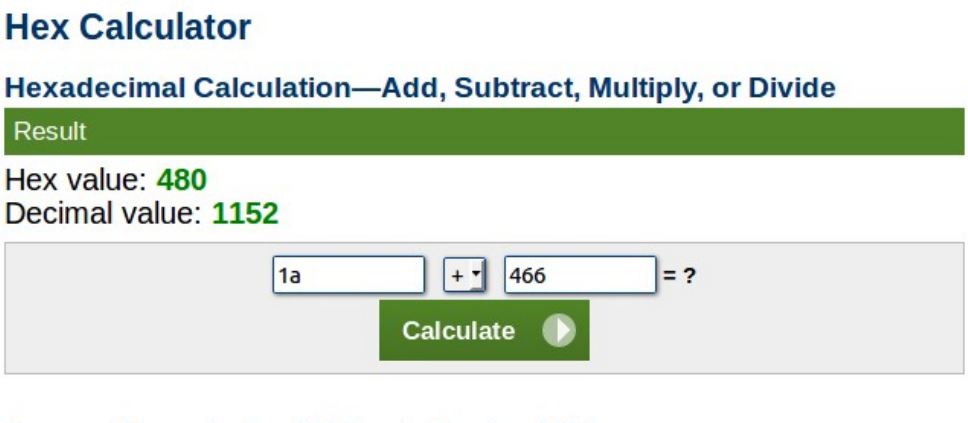
Flag format: FIT{*}

Solution:

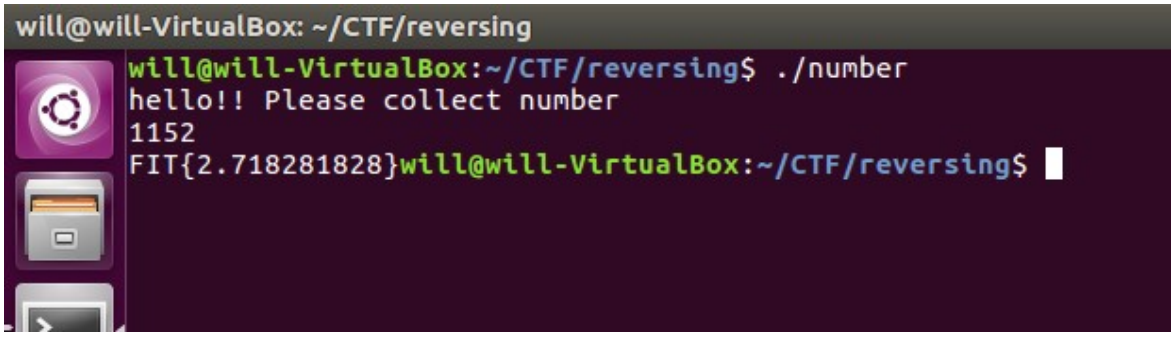
I started by downloading the file ‘number’ and changing the permissions so that I could run the program. For this problem I loaded the file ‘number’ into Radare2. I then proceeded to type ‘aaaa’ to analyze the binary and then ‘V’ to see the instructions. I paged through to get to the instructions list with ‘p’. The first thing I did was search for main. I scrolled down and found this in main:



The user is asked to collect a number. That value is stored in ‘local_10h’ and then eax. Then, 0x1a is subtracted from the user entered number. This is done at ‘sub eax, 0x1a’. Then a comparison is done on 0x466. If the result of the comparison isn’t 0x400756 the program jumps and prints “No” (I found this out by typing in a wrong answer into the program). The comparison is checked with ‘jne 0x400756’. When the jump isn’t taken several character manipulations take place which I’m assuming is the flag. To get the input that allows the jump not to be taken I added 0x1a to 0x466. This is done to reverse the subtraction done earlier in the program. I used the calculator found at http://www.calculator.net/hex-calculator.html?number1=1a&c2op= and got the following results.



I entered the decimal value 1152, which is the result of the hex addition, into the program and got the flag.



Flag = FIT{2.718281828}

Got Time?

20 Points

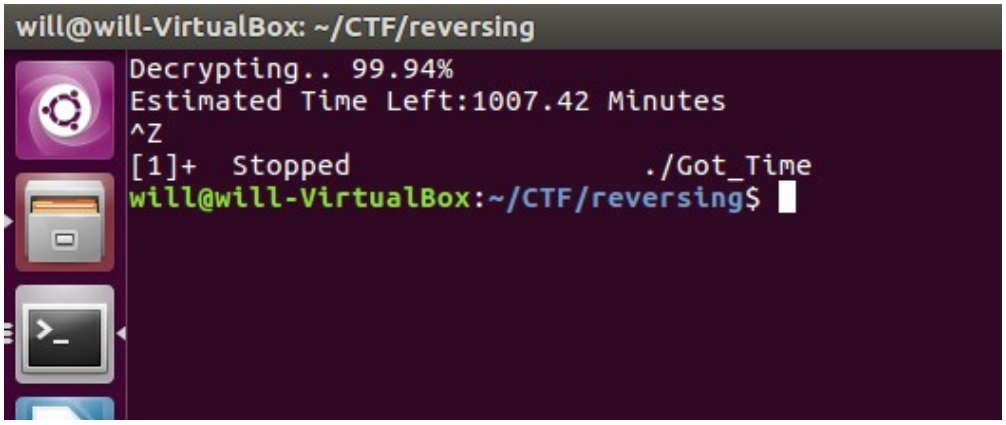
Problem:

Find the flag! Include all your steps in your write up.

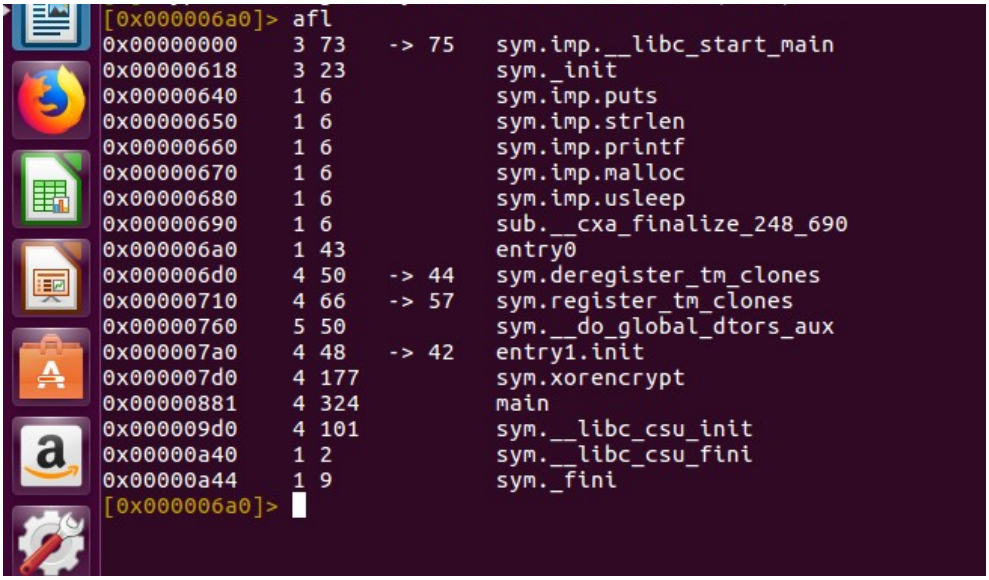
Flag format: TAMPA{*}

Solution:

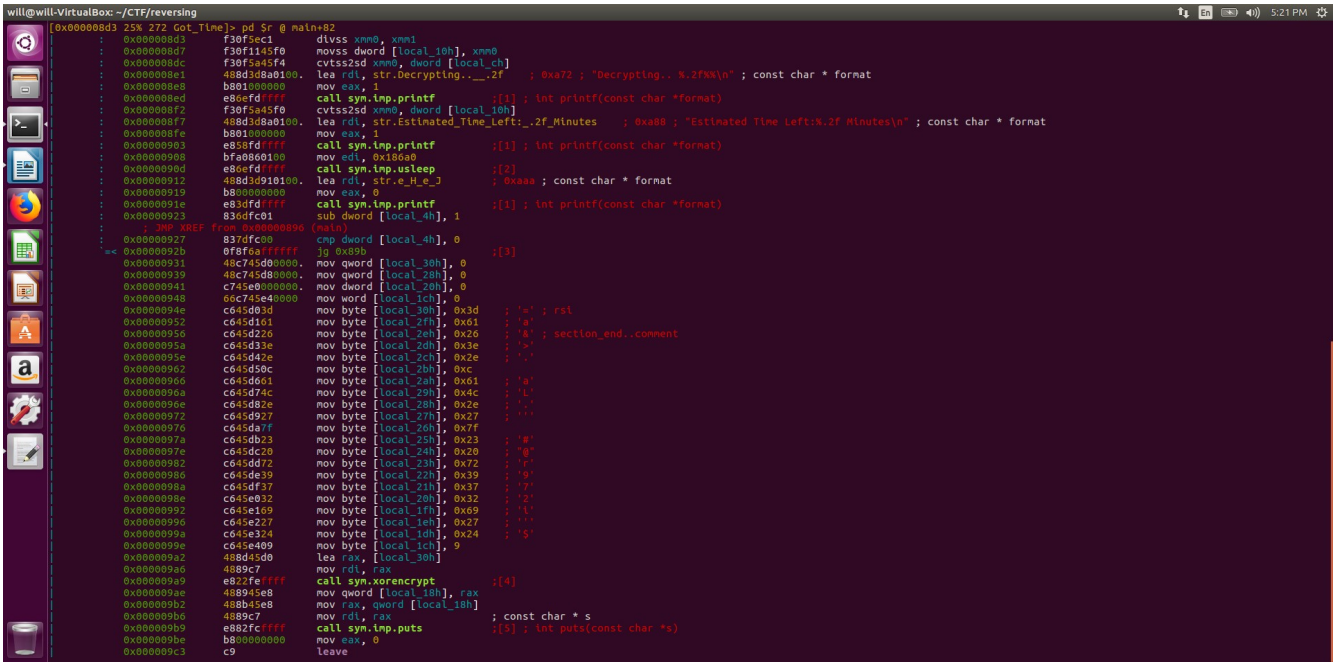
The first thing I did was download the file ‘Got_Time’. I ran ‘./Got_Time’ after changing the permissions to allow execution.



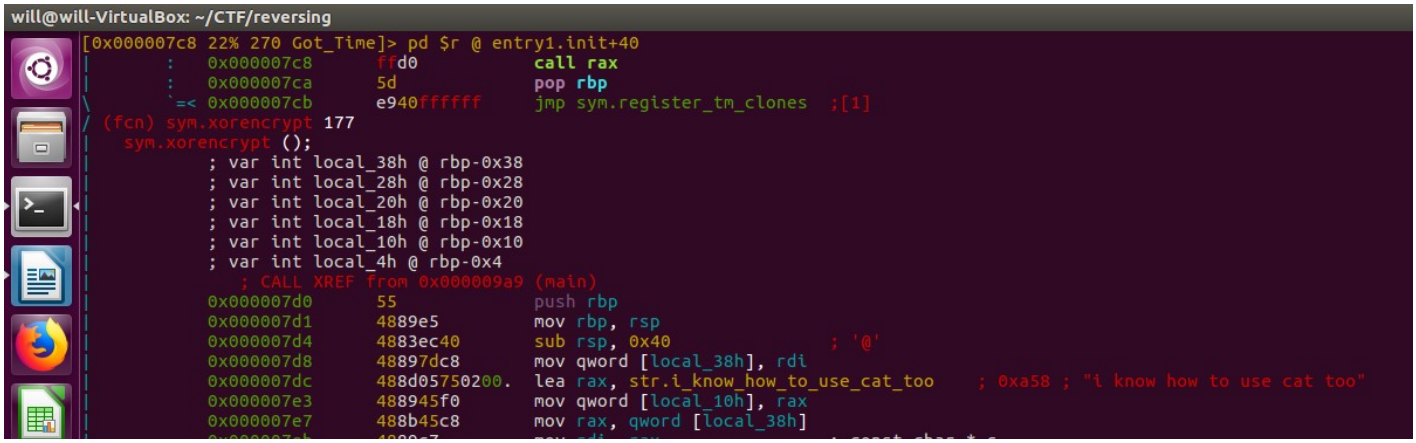
The program is decrypting the flag, but it takes 1000 minutes. This isn’t practical, so I will have to analyze it with Radare. After opening Radare, I typed ‘aaaa’ to analyze the binary. I then used the ‘afl’ command to view functions.



I noticed a function called ‘sym.xorencrypt’ and ‘main’. This lead me to believe the flag is being xor encoded. I seeked to main with ‘s 0x881’ which is the address of main. I then typed ‘V’ and then hit ‘p’ a couple times to view the instructions. After scrolling down main I notice a ton of local variables which are characters that are initialized right before the call to ‘sys.xorencrypt’.



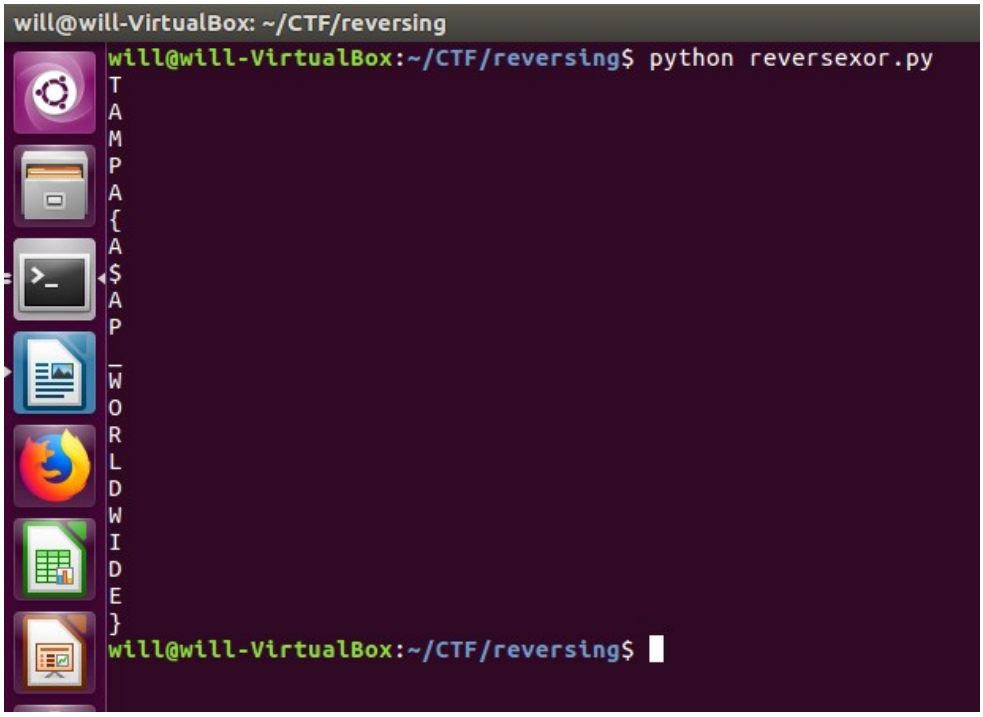
These have to be important. I then scroll up to the XOR function and notice the string “i know how to use cat too”



The rest of the function uses many local variables and looks really intimidating. It uses all sorts of different hex values, moves, and adds. I tried the value ‘0x40’ as the key because it is the only hard coded byte in the function. This didn’t work. Since I can’t seem to find a solid key for the XOR encryption in the function, I will choose to ignore the rest of the code. “i know how to use cat too” appears to be the important part of the function because it is the only string. Going out on a limb, I wrote a python script which XORs the first character of the string “i know how to use cat too” with the first local variable in main’s hex value, XORs the second character of the string with the second local variables in main’s hex value, and continued this process for every local variable in main. The code is attached as ‘reversexor.py’. I chose to ignore the very first local variable’s value because it was 0, and I assumed it was generated on accident by Radare. Below is a brief part of the code.

```
z=0x3d^ord('i')
print chr(z)
z=0x61^ord(' ')
print chr(z)
z=0x26^ord('k')
print chr(z)
z=0x3e^ord('n')
print chr(z)
z=0x2e^ord('o')
print chr(z)
```

The script used every character of the string ‘i know how to use cat too’ except for ‘ too’. After running the python script I found the flag.



Flag = TAMPA{A$AP_WORLDWIDE}