William Hupp
CIS 5930
Reverse Engineering II

Reverse Engineering II Write-Up

More tools!

10 points

Problem:

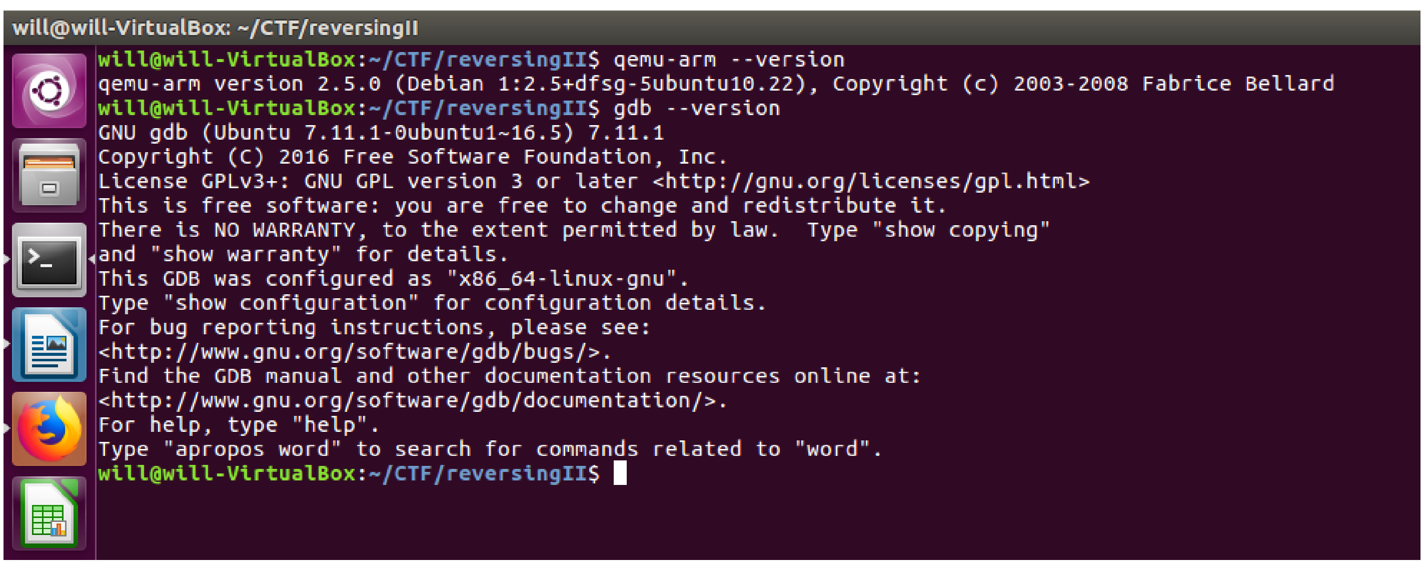
Provide a screen-shot showing that you have the latest versions of the following tools installed. Install instructions are located in the slides.

- pwndbg
- QEMU

If 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_tW0}

Solution:



flag{R3_w33k_tW0

Journal II
10 points

Problem:

Continue your journal of x86 instructions. You must have a minimum of 10 new instructions, 20 total instructs, in your journal. If needed find 10 new instructions online that you don't know and add them

to your journal.

Submit your journal as part of homework submission.

flag{r0und_tw0}

Solution:

OUT – output to port - Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.
The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. **-OUT DX, EAX (Output doubleword in EAX to I/O port address in DX)**

PMAXSW – maximum of packed signed integers - Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand. **- PMAXSW mm1, mm2/m64 (Compare signed word integers in *mm2/m64* and *mm1* and return maximum values.)**

PMINSW – minimum of packed signed integers - Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand. **- MINSW *mm1, mm2/m64 (Compare signed word integers in mm2/m64 and mm1 and return minimum values.)***

PMULHRSW - packed multiply high with round and scale -PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand. - **PMULHRSW** ***mm1, mm2/m64 (Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to mm1.)***

POR – *bitewise logical OR - Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0. -* ***POR mm, mm/m64 (Bitwise OR of mm/m64 and mm.)***

PSHUFB – *packed shuffle bytes - PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.-* ***SHUFB mm1, mm2/m64 (Shuffle bytes in mm1 according to contents of mm2/m64.)***

PSLLW – shift packed data left logical - Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand. - **PSLLW** ***mm, mm/m64 (Shift words in mm left mm/m64 while shifting in 0s.)***

PSUBB – subtract packed integers - Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation.

Overflow is handled with wraparound, as described in the following paragraphs. - **PSUBB *mm, mm/m64 (Subtract packed byte integers in mm/m64 from packed byte integers in mm.)***

RCL – *rotate - Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1). -* ***RCL r/m8, 1(Rotate 9 bits (CF, r/m8) left once.)***

SUBPS – *subtract packed single precision floating point values - Performs a SIMD subtract of the packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand. -* ***SUBPS xmm1, xmm2/m128 (Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.)***

flag{r0und_tw0}

Run me!

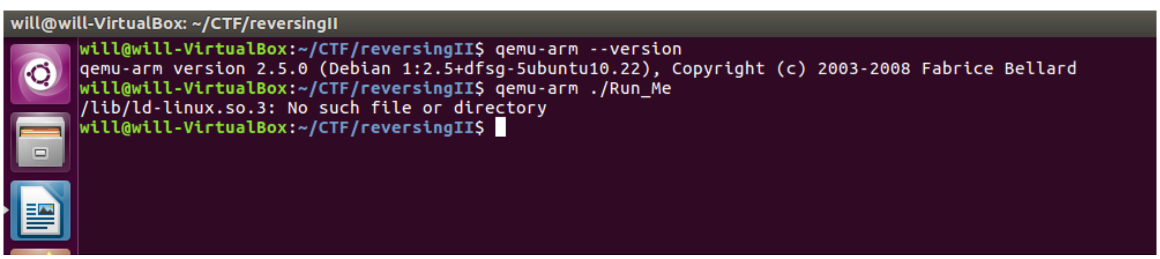
20 points

Problem:

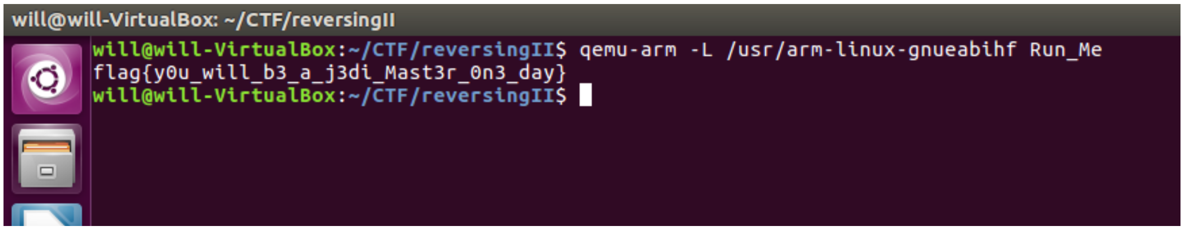
Run the file. Get the flag. Include all steps in your write up.

Solution:

First, I installed the two files and changed the permission on Run_Me to make it executable. I ran file on Run_Me and got the following ‘ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.3, for GNU/Linux 3.2.0, BuildID[sha1]=7106456e068decc7cd4cb5b261bfadd3252e573b, not stripped’ Since it is an ARM file I will need to use qemu. After I installed qemu with the instructions on the slides I, for some reason, still have version 2.5.0. When I ran the command qemu-arm ./Run_Me I get the following error.



After googling the error, I found a github that told me to use the following commands to fix the problem. First, run ‘sudo apt-get install gcc-arm-linux-gnueabihf libc6-dev-armhf-cross qemu’. After I did that I had to use the following command to run qemu, ‘qemu-arm -L /usr/arm-linux-gnueabihf Run_Me’. After running that command I get the flag.



Flag = flag{y0u_will_b3_a_j3di_Mast3r_0n3_day}

Got_Time? II

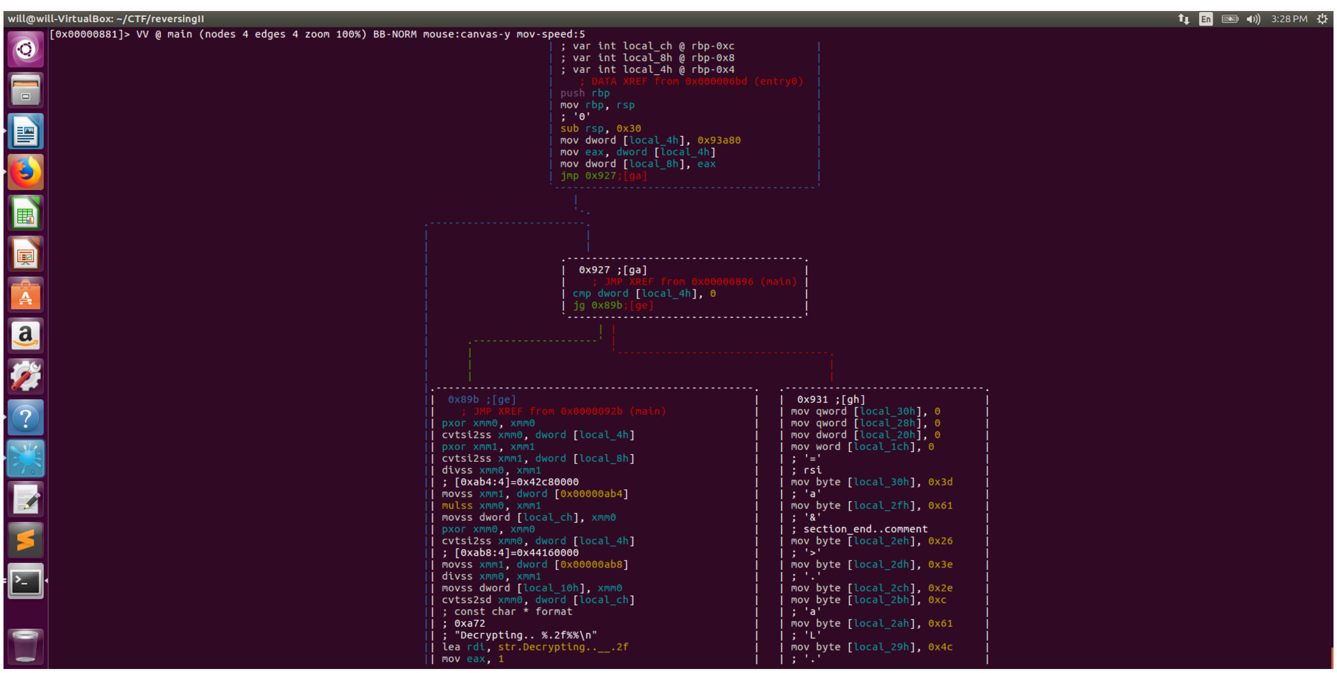
20 points

Problem:

Solve the Got_Time challenge again. This time you must use dynamic analysis to get the flag. (AKA: Have the program, Got_Time, print the flag.)

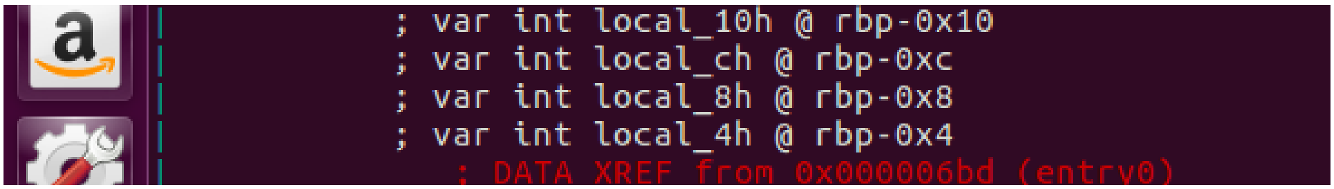
Solution:

Analyzing the binary with Radare, I hit ‘aaaa’, then ‘V’ and ‘p’ twice to view instructions. I used ‘s main’ to get to the main of the program. I hit ‘V’ again to enter graph mode. Analyzing the graph I get the following.



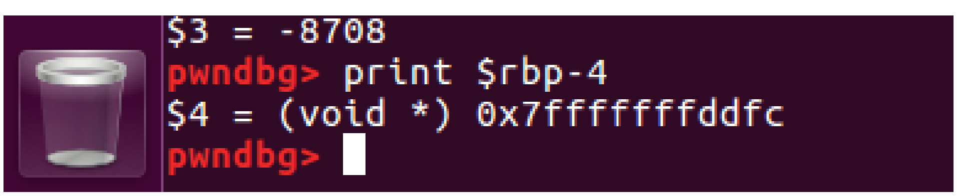
The left basic block is where the sleeping takes place. Here we notice the command ‘cmp dword [local_4h], 0’ followed by ‘jg 0x89b’ before the if statement. We want the value of the compare to be less than 0x89b so that the red branch is taken. If the red branch is taken, the entire sleeping portion of the code is skipped so the flag will display right away. Since cmp a, 0 is the same as cmp a-0, we have to set the value of dword[local_4h] to be less than 0x89b since it is branching on greater than 0x89b as indicated by ‘jg’. Only then can we skip the portion of the code that sleeps. We can do this with pwndbg.

Note: According to thee variable decelerations in main, local_4h is stored at ebp-4. This is important cause we can’t access the memory address of local variables.

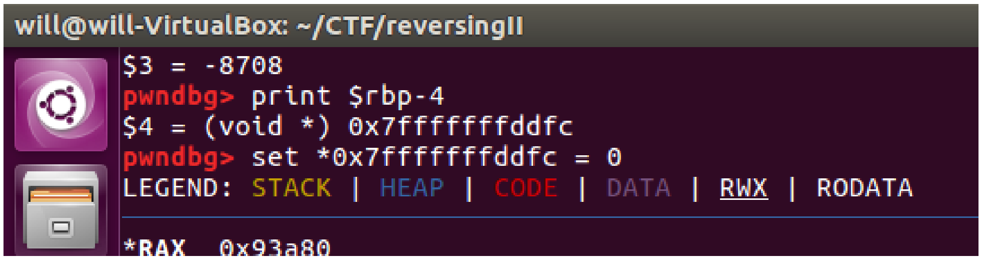


I started pwdbg with ‘dbg Got_Time’. I then typed ‘start’ to start the program. This yielded the following results.

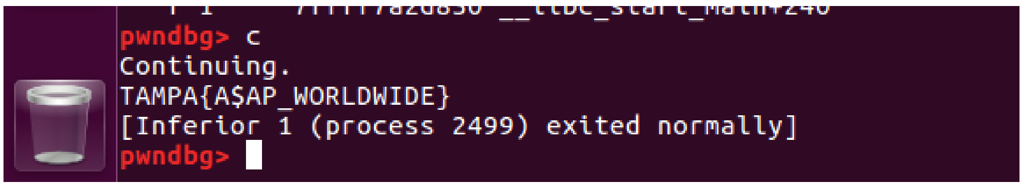
We see that the comparison we were looking at is 5 instructions from the start of main. We move to this instruction with ‘s 5’. Now we have to set dword[local_4h] to be less than 0x89b. We can do this by setting the memory address of local_4h to 0. Since local_4h is stored at ebx-4, we find that address with ‘print $rbp-4’. I found that command with a little bit of googling.



We then set that memory address to 0 with ‘set *0x7fffffffdfc = 0’. I used * instead of $ because it was an address and not a register.



Then continuing the program with ‘c’ we get the flag.



Flag = TAMPA{A$AP_WORLDWIDE}

Arrrrggggggggggs

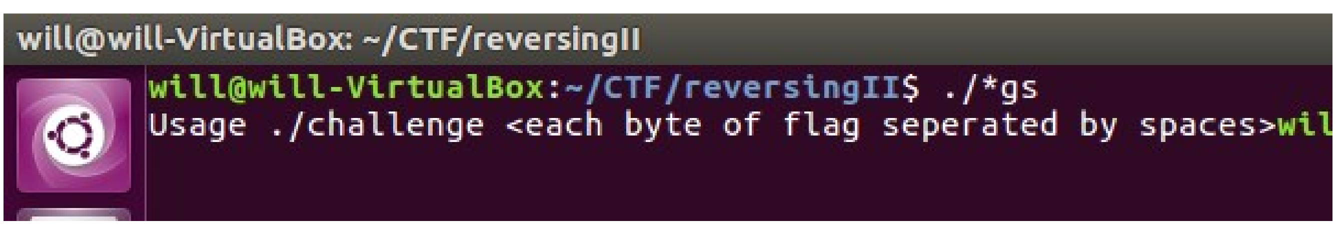
20 points

Problem:

Use static and dynamic analysis to find the flag! Include all your steps in the write up.

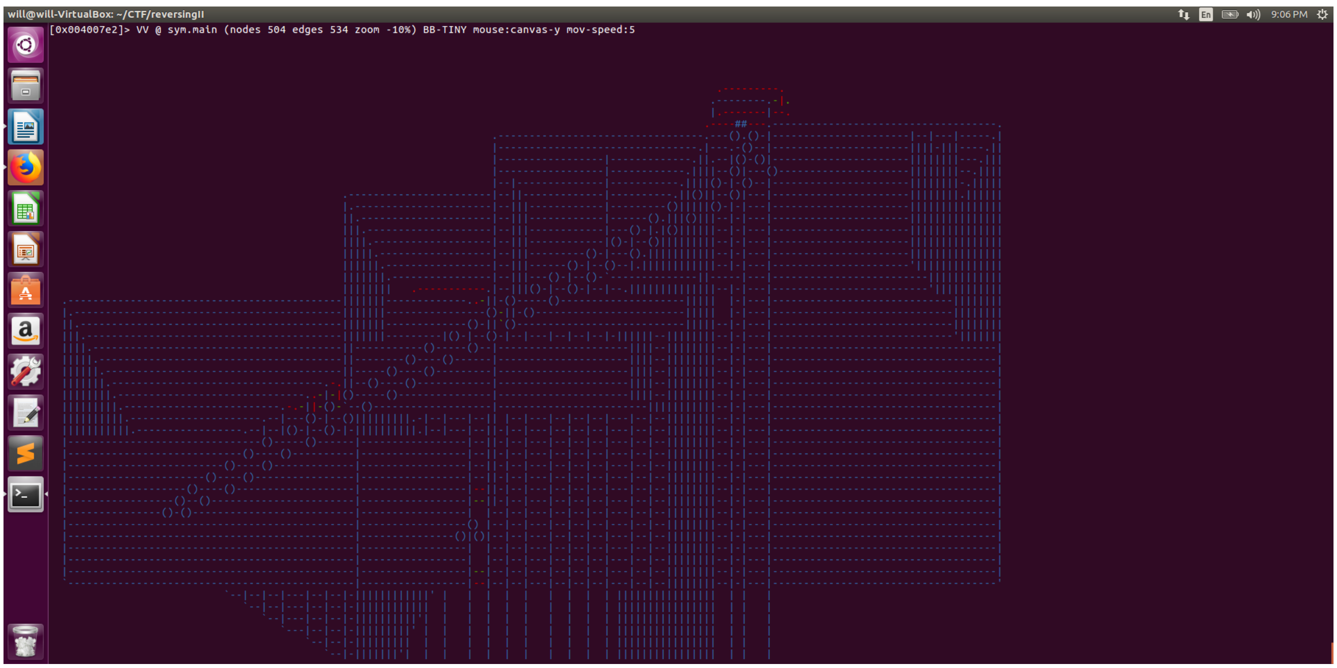
Solution:

Running the program, I get the following results.



So the input is in the form ./argggggggggs flag { … }. I proceed to open the file in Radare to analyze it using the command ‘r2 *gs’. I type ‘aaaa’ then ‘s main’ to go to main. I then open graph view with ‘V’ and switch type ‘p’ twice to view the instructions. It’s complicated so I open graph view with ‘V’ again.

The first thing I notice is there is a ‘cmp dword[local_44h] 0x1f on the initial branch. 0x1f is 31. Following that is a jump on equal. Scrolling through the graph, I notice there are 30 compare calls in different blocks in the graph. This leads me to believe the flag is 30 characters. Based off the usage of running the program, each compare block checks each of the arguments to the program.



Zooming out we say the layout of the map. This represents 30 if else statements. After stepping through all 30 if statements to get to the final block which calls a “success” function that prints nothing but hacked, I see that making it through the program doesn’t print a flag. This makes me believe I need to do something with the bytes being compared because there is a big outside while loop that appears to loop through the arguments. Analyzing the first if statement we see the following:

The variables appear to be flag bytes in an array since each variable is separated by just one byte.
In the right hand side we the if statement is performing arr[0]+arr[1]-arr[2]. The result of that comparison is being compared to 0x51. I believe the array decreases from 30h for increasing indexes because the stack pops the highest value first. For the next comparison if statement we have the following:

This is performing the following, arr[0]-arr[1]+arr[2] and that result is being compared to 0x35. We know that these indexes hold because the variables are the same as the previous if statement. Lets analyze the next if statement.

Here we have arr[1]-arr[0]+arr[2] that that result is being compared to 0x37. This pattern continues 10 more times for all 30 if statements, repeating the same structure for comparing against different hex values. However, the next three if statement would use arr[3], arr[4], and arr[5] as arguments as indicated by the changing variables in relation to local_30h. This pattern continues every three if statements. If these conditions do not hold the program exits. Since looping through the program correctly only prints 'hacked', I will have to use python to get the flag. I stored the comparison bytes in a list and iterated over the list, checking all three conditions and adding to the flag if the conditions are met. The code is attached as 'arg.py'.

```
import string
list = [0x51, 0x35, 0x57, 0x5a, 0x9c, 0x42, 0x62, 0x8c, 0x5c, 0x26, 0xaa, 0x3c, 0x1d, 0xa1, 0x45,
0xa3, 0x1b, 0x45, 0x93, 0x2b, 0x3b, 0x92, 0x56, 0x2c, 0x43, 0x59, 0x4b, 0x75, 0x7d, 0x7d]
flag=""
x =  string.printable
for i in range(0,len(list),3):
        for a in x:
                for b in x:
                        for c in x:
                                if ord(a)+ord(b)-ord(c) == list[i] and ord(a)-ord(b)+ord(c) == list[i+1] and ord(b)-
                                 ord(a)+ord(c) == list[i+2]:
                                        flag += a+b+c
                                        print flag
```

Flag = CTF{Now_th1s_1s_t0_g3t_ANGRyy}

Focus

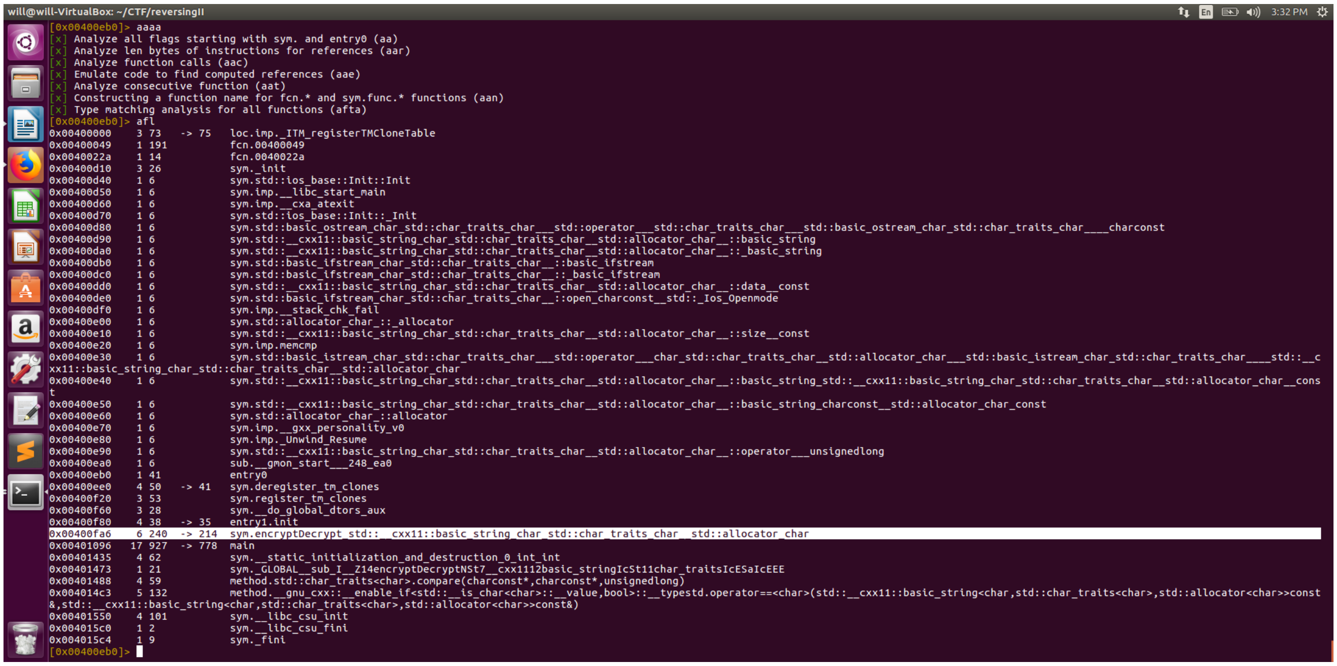
20 points

Problem:

Use static and dynamic analysis to find the flag! Include all your steps in the write up.

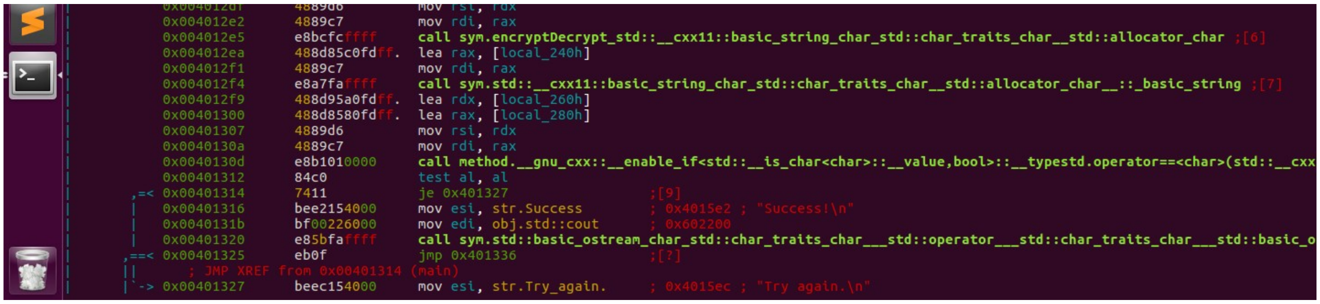
Solution:

The challenge gives us two files ‘Focus’ and ‘file.txt.enc’. Inside ‘file.txt.enc’ is a string that appears to be the encoded flag. Using ‘file’ on Focus I see that it is a Linux executable. After changing the permissions to run it with ‘chmod 777’, I run the program and it prompts for user input. I’m assuming you have to input the flag. Lets analyze the ‘Focus’ file with Radare using ‘r2 Focus’. I type ‘aaaa’ to analyze the binary, then ‘afl’ to view the program’s functions.

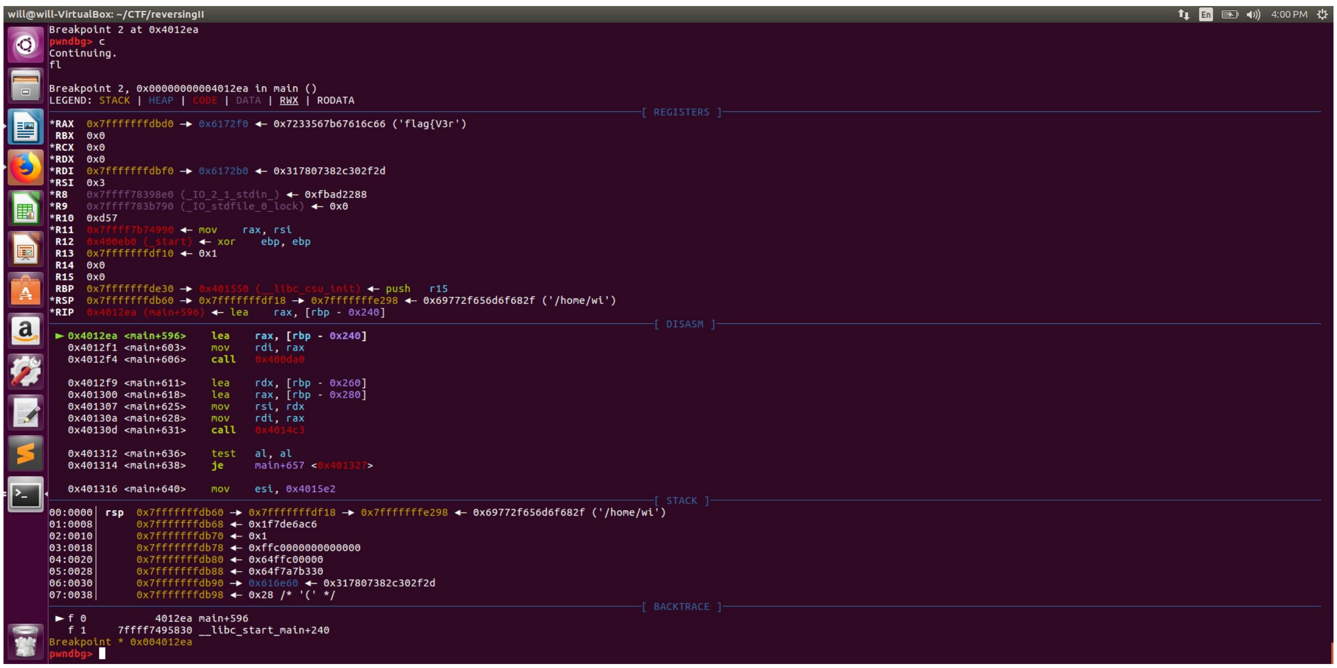


Here we notice a function called ‘sym.encryptDecrypt_std’. I’m going to try to find where this function is called in main. I type ‘s main’ to seek to main and then ‘VV’ to enter graph mode. After scrolling through main, I notice the function call.

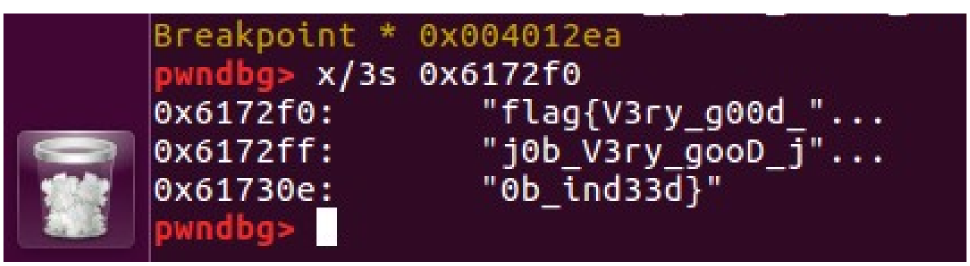
Right after the function call, local_240h is moved into eax. I’m assuming this is the return from the decrypt function. We will have to dynamically see what is in that register. First, I note the address of where the ‘lea rax, [local_240h]’ instruction is located. I note that address by exiting graph mode with ‘q’



The address is 0x004012ea. I start pwndbg to analyze the rax register at that location. I start pwndbg with ‘gdb Focus’ then type ‘start’ to begin execution. I then set a breakpoint at the memory address of the instruction lea rax, local_240h] with b* 0x004012ea and typed ‘c’ to execute to that breakpoint. I type ‘flag’ as sample input then analyze the rax register.



Here we can see rax is set to ‘flag{v3r’ Since there is no loop over the decrypt function, the whole flag should we be printing. We can see the entire flag with ‘x/3s 0x6172f0’. 0x6172f0 is the address of the rax register. X/s is a gdb command for examining strings. I needed to specify 3 consecutive memory locations because the entire flag can’t be stored in one memory location. This is done with the 3 in ‘x/3s’.



flag{V3ry_g00d_j0b_V3ry_gooD_j0b_ind33d}