

Reverse Engineering II Write-Up

More tools!

10 points

Problem:

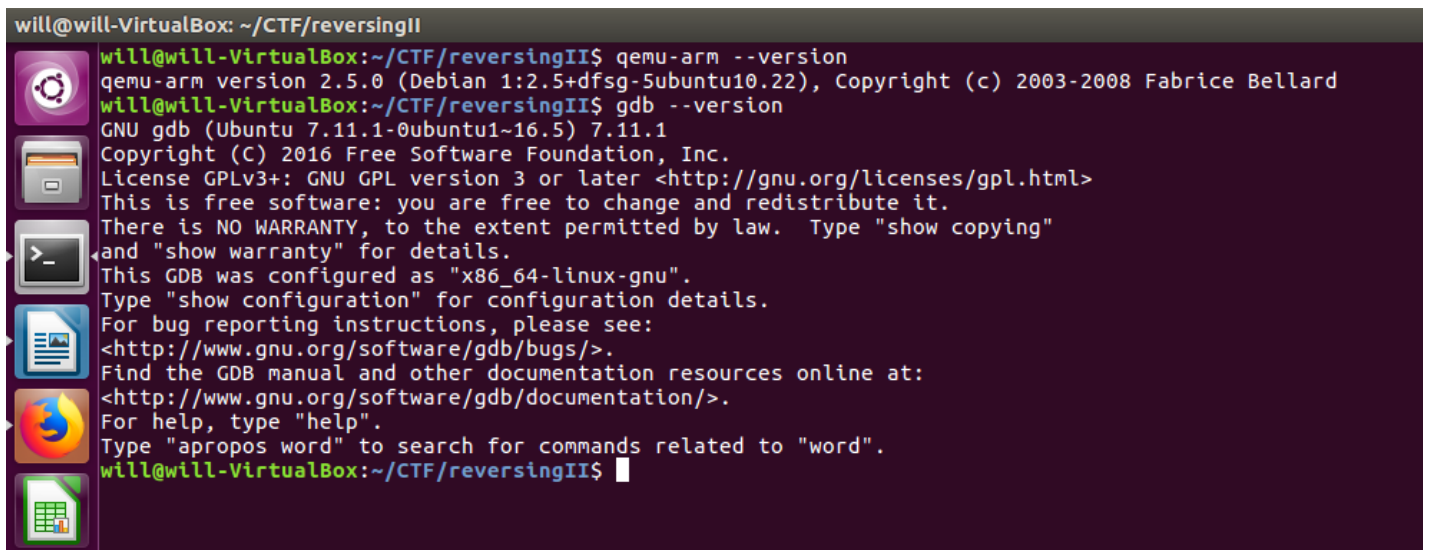
Provide a screen-shot showing that you have the latest versions of the following tools installed. Install instructions are located in the slides.

- 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:



The screenshot shows a terminal window titled "will@will-VirtualBox: ~/CTF/reversingII". The user has run the command `qemu-arm --version`, which outputs: `qemu-arm version 2.5.0 (Debian 1:2.5+dfsg-5ubuntu10.22), Copyright (c) 2003-2008 Fabrice Bellard`. Then, the user has run `gdb --version`, which outputs: `GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1`, followed by copyright and license information. The terminal also shows instructions for using GDB, such as `show warranty`, `show configuration`, and `help`.

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 instructions, 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 – bitwise 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!

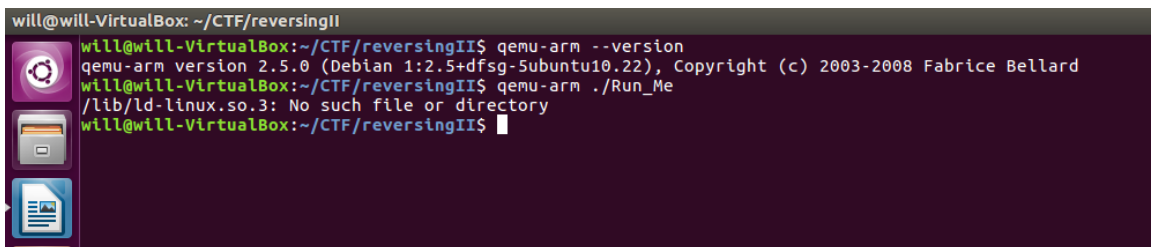
20 points

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.



```
will@will-VirtualBox: ~/CTF/reversingII
will@will-VirtualBox:~/CTF/reversingII$ qemu-arm --version
qemu-arm version 2.5.0 (Debian 1:2.5+dfsg-5ubuntu10.22), Copyright (c) 2003-2008 Fabrice Bellard
will@will-VirtualBox:~/CTF/reversingII$ qemu-arm ./Run_Me
/lib/ld-linux.so.3: No such file or directory
will@will-VirtualBox:~/CTF/reversingII$
```

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-gnueabi lib64-dev-armhf-cross qemu'. After I did that I had to use the following command to run qemu, 'qemu-arm -L /usr/arm-linux-gnueabi Run_Me'. After running that command I get the flag.

```
will@will-VirtualBox: ~/CTF/reversingII
will@will-VirtualBox:~/CTF/reversingII$ qemu-arm -L /usr/arm-linux-gnueabihf Run_Me
flag{y0u_will_b3_a_j3di_Mast3r_0n3_day}
will@will-VirtualBox:~/CTF/reversingII$
```

Flag = flag{y0u_will_b3_a_j3di_Mast3r_0n3_day}

Got_Time? II

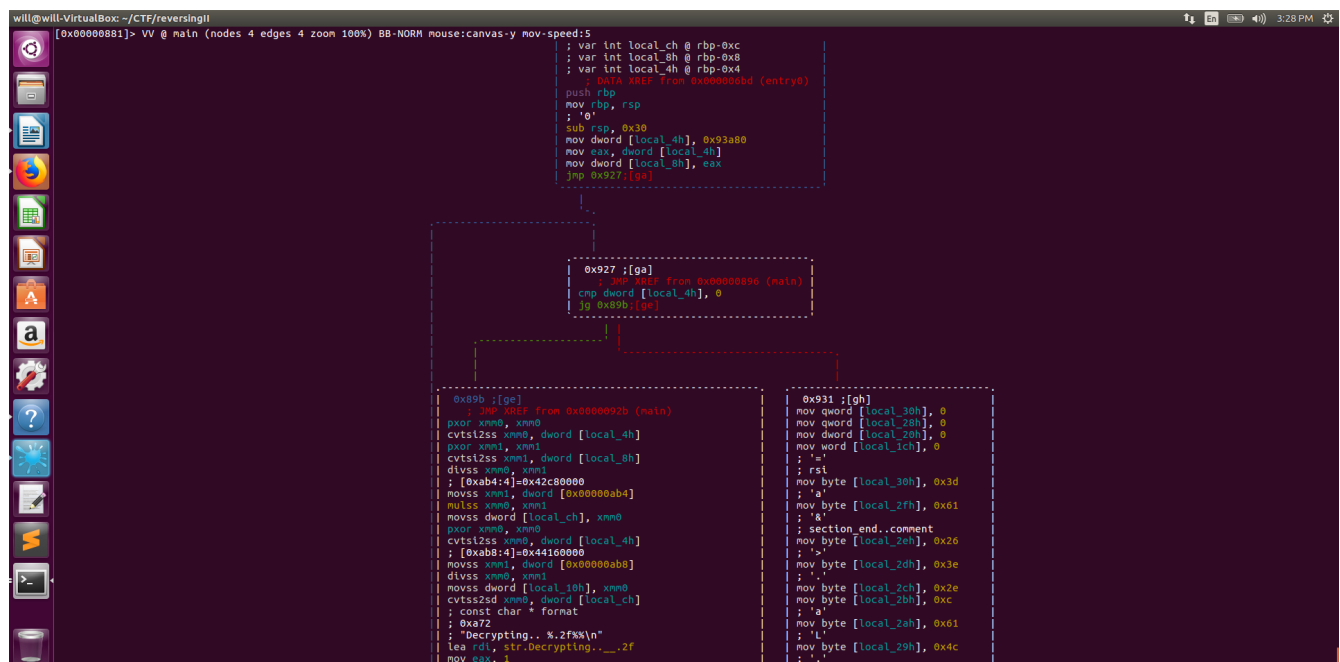
20 points

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

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

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.



```
$3 = -8708
pwndbg> print $rbp-4
$4 = (void *) 0x7fffffffddfc
pwndbg>
```

We then set that memory address to 0 with 'set *0x7fffffffddfc = 0'. I used * instead of \$ because it was an address and not a register.

```
will@will-VirtualBox: ~/CTF/reversingII
$3 = -8708
pwndbg> print $rbp-4
$4 = (void *) 0x7fffffffddfc
pwndbg> set *0x7fffffffddfc = 0
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
*RAX 0x93a80
```

Then continuing the program with 'c' we get the flag.

```
pwndbg> c
Continuing.
TAMPA{A$AP_WORLDWIDE}
[Inferior 1 (process 2499) exited normally]
pwndbg>
```

Flag = TAMPA{A\$AP_WORLDWIDE}

Arrrrrggggggggggs

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.

```
will@will-VirtualBox: ~/CTF/reversingII
will@will-VirtualBox:~/CTF/reversingII$ ./*gs
Usage ./challenge <each byte of flag seperated by spaces>wil
```

So the input is in the form ./arggggggggggs 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 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 $\text{arr}[0] + \text{arr}[1] - \text{arr}[2]$. 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, $\text{arr}[0] - \text{arr}[1] + \text{arr}[2]$ 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 $\text{arr}[1] - \text{arr}[0] + \text{arr}[2]$ 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 $\text{arr}[3]$, $\text{arr}[4]$, and $\text{arr}[5]$ 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
```

```
TypeError: unsupported operand type(s) for -: 'str' and 'str'
will@will-VirtualBox:~/CTF/reversingII$ nano arg.py
will@will-VirtualBox:~/CTF/reversingII$ python arg.py
CTF
CTF{No
CTF{Now_t
CTF{Now_th1s
CTF{Now_th1s_1s
CTF{Now_th1s_1s_t0
CTF{Now_th1s_1s_t0_g3
CTF{Now_th1s_1s_t0_g3t_A
CTF{Now_th1s_1s_t0_g3t_ANGR
CTF{Now_th1s_1s_t0_g3t_ANGRyy}
will@will-VirtualBox:~/CTF/reversingII$
```

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.

```

will@will-VirtualBox: ~/CTF/reversingll
[0x00400000]-> aaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze ten bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[x] Emulate code to find computed references (aae)
[x] Analyze consecutive function (aat)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (afta)
[0x00400000]-> afd
0x00400000 3 73 -> 75 loc_imp_1TM_registerTMCcloneTable
0x00400049 1 191 fcn.00400049
0x0040022a 1 14 fcn.0040022a
0x00400d10 3 26 sym_init
0x00400d40 1 6 sym.std::ios_base::Init::Init
0x00400d50 1 6 sym.imp._lib_start_main
0x00400d60 1 6 sym.imp._cxa_atexit
0x00400d70 1 6 sym.std::ios_base::Init::Init
0x00400d80 1 6 sym.std::basic_ostream_char_std::char_traits_char_std::operator_std::char_traits_char_std::basic_ostream_char_std::char_traits_char_std::charconst
0x00400d90 1 6 sym.std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::basic_string
0x00400da0 1 6 sym.std::basic_ifstream_char_std::char_traits_char_std::basic_ifstream
0x00400db0 1 6 sym.std::basic_ifstream_char_std::char_traits_char_std::basic_ifstream
0x00400dc0 1 6 sym.std::basic_ifstream_char_std::char_traits_char_std::basic_ifstream
0x00400dd0 1 6 sym.std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::data_const
0x00400de0 1 6 sym.std::basic_ifstream_char_std::char_traits_char_std::open_charconst_std::ios_openmode
0x00400df0 1 6 sym.imp._stack_chk_fail
0x00400e00 1 6 sym.std::allocator_char_std::allocator
0x00400e10 1 6 sym.std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::size_const
0x00400e20 1 6 sym.imp.memcmp
0x00400e30 1 6 sym.std::basic_istream_char_std::operator_char_std::char_traits_char_std::allocator_char_std::basic_istream_char_std::char_traits_char_std::c
0x00400e40 1 6 sym.std::basic_string_char_std::allocator_char_std::allocator_char_std::basic_string_std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::cons
0x00400e50 1 6 sym.std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::basic_string_charconst_std::allocator_charconst
0x00400e60 1 6 sym.std::allocator_char_std::allocator
0x00400e70 1 6 sym.imp._gxx_personality_v0
0x00400e80 1 6 sym.imp.Unwind_Resume
0x00400e90 1 6 sym.std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::operator_unsignedlong
0x00400ea0 1 6 sub.gmon_start_248_ea0
0x00400eb0 1 41 entry0
0x00400ee0 4 50 -> 41 sym.deregister_tm_clones
0x00400f20 3 53 sym.register_tm_clones
0x00400f60 3 28 sym._do_global_dtors_aux
0x00400f80 4 38 -> 35 entry1_init
0x00400fa6 6 240 -> 214 sym.encryptDecrypt_std::_cxx11::basic_string_char_std::allocator_char
0x00401096 17 927 -> 778 main
0x00401435 4 62 sym.static_initialization_and_destruction_0_int.int
0x00401473 1 21 sym.GLOBAL_sub_I_214encryptDecryptNS7_cxx11basic_stringlcStlchar_traitslctSalCEE
0x00401488 4 59 method_std::char_traits_char_std::compare(charconst*,charconst*,unsignedlong)
0x004014c3 5 132 method__gnu_cxx::_enable_if_std::_is_char_char::_value,bool::_typestd.operator==<char>(std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::const
&_std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::const)
0x00401550 4 101 sym._libc_csu_init
0x004015c0 1 2 sym._libc_csu_fini
0x004015c4 1 9 sym._fini
[0x00400000]->

```

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.

```

0x401191:[g]
; 0x63:4=-1
; 'c'
; 99
cmp dword [local_24ch], 0x63
jg 0x4012b8:[g]

0x4012b8:[g]
; 0x60:0x6f from 0x00401198 (main)
lea rdx, [local_240h]
lea rax, [local_240h]
mov rsi, rdx
mov rdi, rax
call sym.std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::basic_string_std::_cxx11::basic_string_char_std::char_traits_char_std::
lea rdx, [local_240h]
mov rsi, rdx
mov rdi, rax
call sym.encryptDecrypt_std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::[gt]
lea rax, [local_240h]
mov rdi, rax
call sym.std::_cxx11::basic_string_char_std::char_traits_char_std::allocator_char_std::basic_string:[gu]
lea rdx, [local_240h]
lea rax, [local_240h]
mov rsi, rdx
mov rdi, rax
call method__gnu_cxx::_enable_if_std::_is_char_char::_value,bool::_typestd.operator==<char>(std::_cxx11::basic_string_char_std::char_traits_char_std::
test al, al
je 0x401327:[g]

```

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'

```

0x004012e1 4889d0 mov rsi, rax
0x004012e2 4889c7 mov rdi, rax
0x004012e5 e8bcfcffff call sym.encryptDecrypt_std::__cxx11::basic_string_char_std::char_traits_char_std::allocator_char::[6]
0x004012e8 488d85c0fdff lea rax, [local_240h]
0x004012f1 4889c7 mov rdi, rax
0x004012f4 e8a7faffff call sym.std::__cxx11::basic_string_char_std::char_traits_char_std::allocator_char::__basic_string::[7]
0x004012f9 488d95a0fdff lea rdx, [local_260h]
0x00401300 488d8580fdff lea rax, [local_280h]
0x00401307 4889d6 mov rsi, rdx
0x0040130a 4889c7 mov rdi, rax
0x0040130d e8b1010000 call method.__gnu_cxx::__enable_if_std::__is_char_char::__value,bool::__typestd.operator==<char>(std::__cxx
0x00401312 84c0 test al, al
0x00401314 7411 je 0x401327 ;[9]
0x00401316 bee2154000 mov esi, str.Success ; 0x4015e2 ; "Success!\n"
0x0040131b bf9b226000 mov edi, obj.std::cout ; 0x002200
0x00401320 e85bfaffff call sym.std::basic_ostream_char_std::char_traits_char_std::operator__std::char_traits_char_std::basic_o
0x00401325 eb0f jmp 0x401336 ;[?]
; JMP XREF from 0x00401314 (main)
-> 0x00401327 beec154000 mov esi, str.Try_again. ; 0x4015ec ; "Try again.\n"

```

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.

```

will@will-VirtualBox: ~/CTF/reversingll
Breakpoint 2 at 0x4012ea
pwndbg> c
Continuing.
fl
Breakpoint 2, 0x00000000004012ea in main ()
LEGEND: STACK | HEAP | CODE | DATA | RMW | RODATA

[ REGISTERS ]
RAX 0x7fffffffdbd0 -> 0x6172f0 -> 0x7233567b67616c66 ('flag{V3r}')
RBX 0x0
RCX 0x0
RDX 0x0
RDI 0x7fffffffdbf0 -> 0x6172b0 -> 0x317807382c302f2d
RSI 0x3
R8 0x7ffff7b398e0 (_IO_2_1_stdin_) -> 0xfbad2288
R9 0x7ffff7b3b790 (_IO_stdfile_0_lock) -> 0x0
R10 0xd57
R11 0x7ffff7b74990 -> mov rax, rsi
R12 0x400eb0 (.start) -> xor ebp, ebp
R13 0x7fffffffdf10 -> 0x1
R14 0x0
R15 0x0
RBP 0x7fffffffde30 -> 0x401550 ([libc_csu_init] -> push r15
RSP 0x7fffffffdbd0 -> 0x7fffffffdf10 -> 0x69772f65d6df682f ('/home/wl')
RIP 0x4012ea (main+596) -> lea rax, [rbp - 0x240]

[ DISASM ]
0x4012ea <main+596> lea rax, [rbp - 0x240]
0x4012f1 <main+603> mov rdi, rax
0x4012f4 <main+606> call 0x4014c0
0x4012f9 <main+611> lea rdx, [rbp - 0x260]
0x401300 <main+618> lea rax, [rbp - 0x280]
0x401307 <main+625> mov rsi, rdx
0x40130a <main+628> mov rdi, rax
0x40130d <main+631> call 0x4014c3
0x401312 <main+636> test al, al
0x401314 <main+638> je main+657 <0x401327>
0x401316 <main+640> mov esi, 0x4015e2

[ STACK ]
00:0000 rsp 0x7fffffffdbd0 -> 0x7fffffffdf10 -> 0x69772f65d6df682f ('/home/wl')
01:0000 0x7fffffffdbd0 -> 0x177de6ac6
02:0010 0x7fffffffdb70 -> 0x1
03:0010 0x7fffffffdb70 -> 0xffc0000000000000
04:0020 0x7fffffffdb80 -> 0x64ffc00000
05:0020 0x7fffffffdb80 -> 0x64f7a7b330
06:0030 0x7fffffffdb90 -> 0x317807382c302f2d
07:0030 0x7fffffffdb90 -> 0x28 /* '(' */

[ BACKTRACE ]
> f 0 4012ea main+596
f 1 7ffff7495830 _libc_start_main+240
Breakpoint * 0x004012ea
pwndbg>

```

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'.

```

Breakpoint * 0x004012ea
pwndbg> x/3s 0x6172f0
0x6172f0: "flag{V3ry_g00d_"...
0x6172ff: "j0b_V3ry_gooD_j"...
0x61730e: "0b_ind33d}"
pwndbg>

```

flag{V3ry_g00d_j0b_V3ry_gooD_j0b_ind33d}

