**CSCE 548 Project 2**

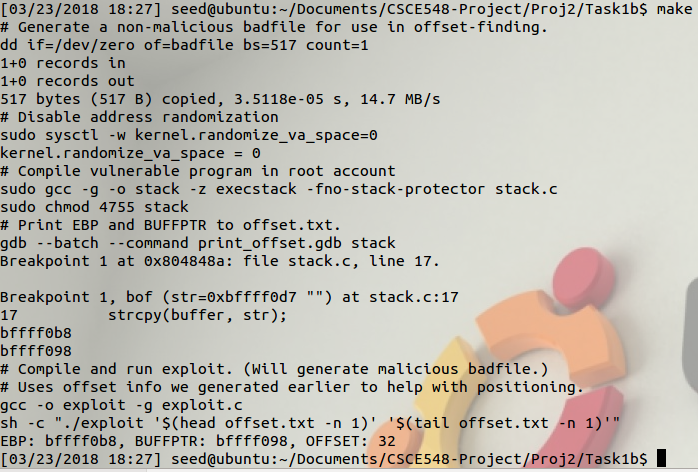
**3/23/2018**

**Group 1: Philip Conrad, Nathaniel Stone, Theodore Stone, Ming Wong**

**Task 1**

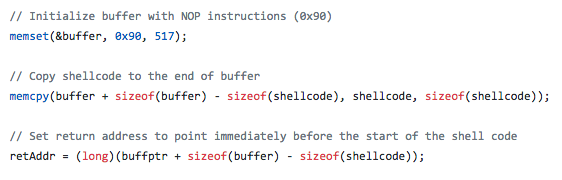
In this task, we have written an exploit.c program that conducts a buffer overflow attack against the vulnerable stack.c program using the given shellcode. We first use the gdb debugger on the stack.c to find the starting buffer address and the saved frame pointer EBP, which is later inputted into our exploit program. Afterwards, we compile and run our exploit. Finally, we run .*/stack* to check whether we have successfully obtained a root shell or not.

**Observations:**



The above screenshot presents all the commands utilized in this task before running the vulnerable program. We observe that the EBP is 0xbffff0b8 and buffer address is 0xbffff098, so the difference between the two pointers is 32 bytes.

Exploit.c screenshot



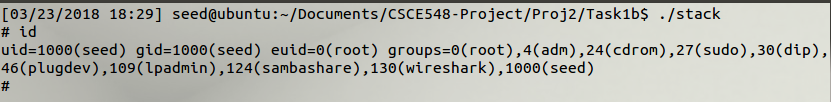
We initialize the buffer with all NOP instructions and place the shellcode at the end of the buffer. To implement the exploit correctly, we predict that the return address should be replaced with an address that is before the location of the actual shellcode but after the buffer’s location containing the return address.

Therefore, we override the return address to point immediately before the start of the shellcode. We observe that despite different machine environments, we are still able to launch the exploit successfully using the above offset from the buffer pointer.

Exploit.c screenshot

Macintosh HD:Users:ming:Desktop:Task1c.png

We calculate the buffer offset for the location of return address to be the number of bytes between the vulnerable buffer pointer and the local frame pointer of stack.c plus the size of the local frame pointer. Based on our success in launching the buffer overflow attack, we conclude that this is the correct buffer offset.



The above display the output after running *./stack*. We observe we have root privilege because the effective user id is root.

**Task 2**

TODO

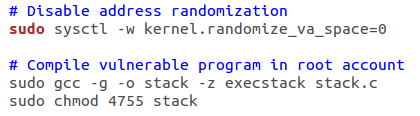
**Observations:**

TODO

**Task 3**

The goal of this task is to examine the Stack Guard protection scheme during a buffer flow attack. We re-activate the Stack Guard protection scheme when compiling the vulnerable stack.c program. Then we run *./stack* to check whether we have successfully obtained a root shell or not.

**Observations:**



We make sure that we turn off the address randomization from Task 2. Then we enable the Stack Guard protection by removing the –fno-stack-protector. We observe that we are still able to compile the stack properly because we only take out a compiler flag. We also notice that we can continue to compile and run our exploit program using the pointer addresses from gdb stack without any issues.

Macintosh HD:Users:ming:Desktop:Task3b1.png

When we run *./stack*, we obtain a program termination message. Moreover, the Stack Guard reports a stack smashing detection error message. The Stack Guard can detect the return address has been altered because it has placed a secret value or canary next to the return address on the stack. When the function returns, it first checks to ensure that the canary is not modified before jumping to the address pointed to by the return address word. If the canary is corrupted, then the program halts immediately and logs the intrusion attempt. Since we did overwrite the return address in this task, the above screenshot illustrates that the Stack Guard mechanism has effectively identified our stack-based buffer overflow and aborted our stack.c program.

**Task 4**

TODO

**Observations:**

TODO