**CSCE 548 Project 2**

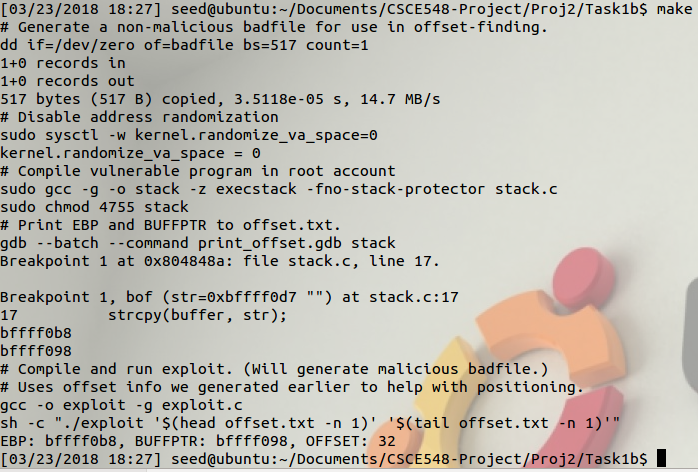
**3/23/2018**

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

**Task 1**

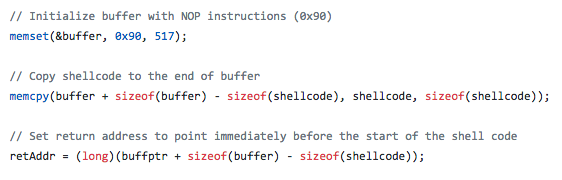
In this task, we have written an exploit.c program that conducts a buffer overflow attack against the vulnerable stack.c program using the given shellcode. We first use the gdb debugger on the stack.c to find the starting buffer address and the saved frame pointer EBP, which is later input into our exploit program. Afterwards, we compile and run our exploit. Finally, we run .*/stack* and verify that we have successfully obtained a root shell.

**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



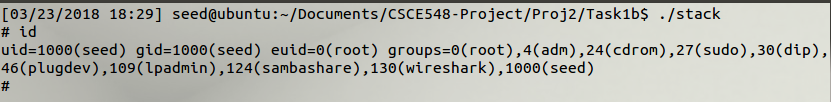
We initialize the buffer with all NOP instructions and place the shellcode at the end of the buffer. To implement the exploit correctly, we predict that the return address should be replaced with an address that is before the location of the actual shellcode but after the buffer’s location containing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

A description...

We calculate the buffer offset for the location of the 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 is the output after running *./stack*. We observe that we have root privilege because the effective user id is root.

**Task 2**

For this task we repeat the buffer overflow attack of task 1 with Ubuntu's address randomization enabled. Using the unmodified code of task 1 we observe that our attack fails with a segmentation fault. The reason for this failure is the address randomization; in this scheme the function stack occupies a random slot of memory each time the program executes. This means that the addresses returned by the gdb debugger for one execution of the program is not valid for the majority of subsequent executions, preventing us from running the program once to determine the address of the malicious code to use in a subsequent run.

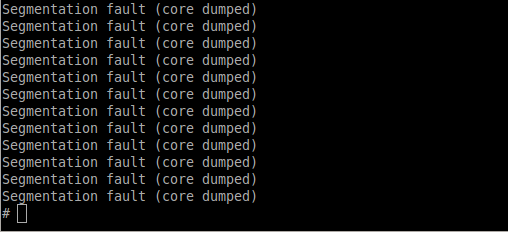
The solution is to run the Task1 attack in a loop; by chance one of the program executions will use a random address close to that returned previously by the gdb debugger. In the observations section we will discuss how we increased the probability of a successful run using this scheme.

**Observations:**

Outcome of simply running Task 1 attack:

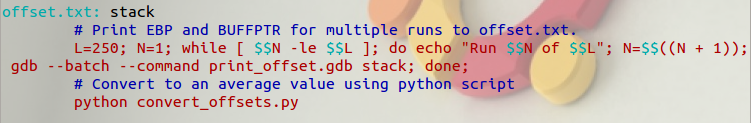
A description...

Outcome of running attack in a loop:

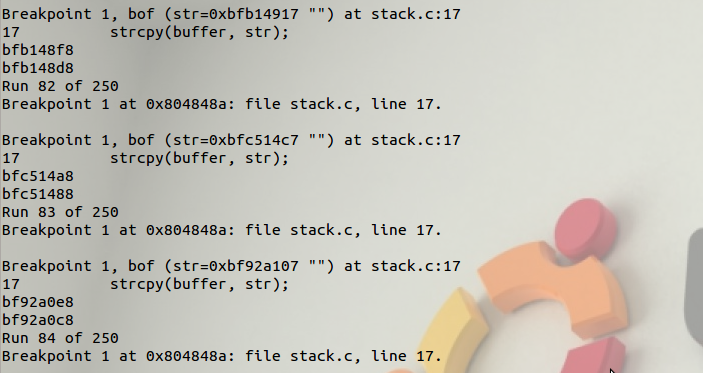


We can increase the probability of success with a modification to the attack. Primarily, instead of simply using one random address from gdb as our buffer pointer we can use the mean address found from many runs; this effectively places our address near the middle of the random range, thereby increasing the chance of being near the randomly generated address of a subsequent run.

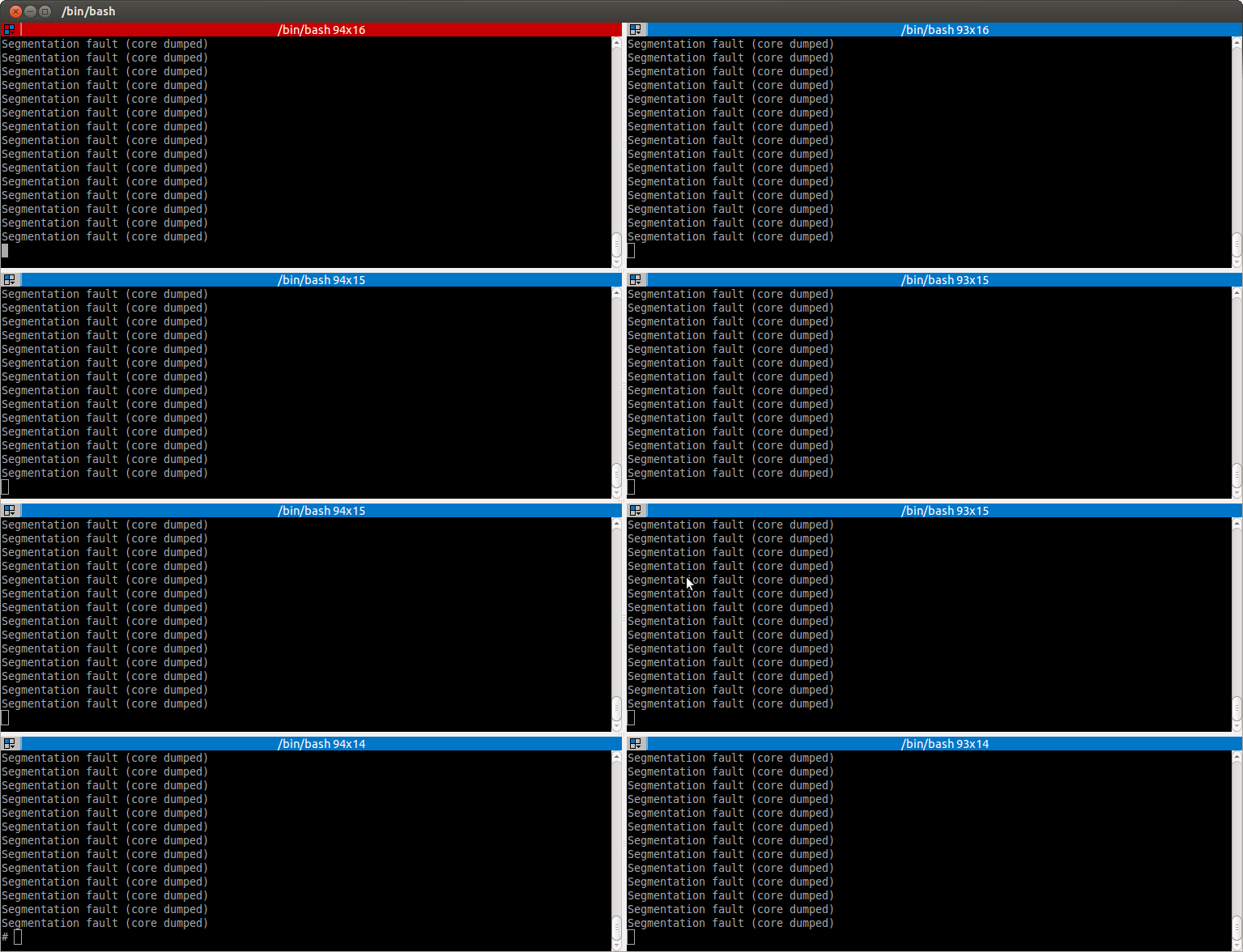
We use the following code in our Makefile to run the gdb debugger 250 times, printing the EBP and buffer pointer from each run to a file. A python script subsequently finds the mean address of these samples.



Snapshot of compiling our exploit code showing the gdb debugger running 250 times to sample the random address range.



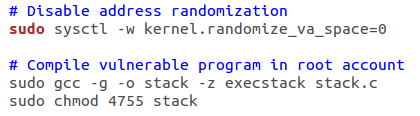
Finally, we can speed up the result by running multiple instances of the attack in parallel:



**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.

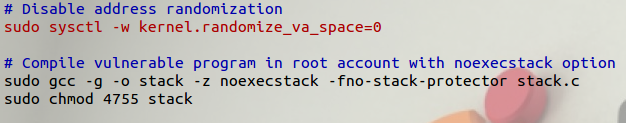
A description...

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.

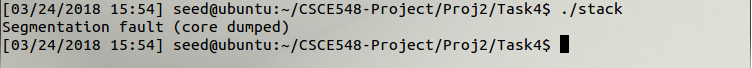
**Task 4**

In this task we recompile the vulnerable program with the noexecstack option which prevents any code on the stack from being executed. We observe that with this option enabled all of our attacks result in a segmentation fault.

**Observations:**



We compile the vulnerable program with the noexecstack option.



All of our attacks fail as soon as we attempt to execute code on the stack.