[[1]](#footnote-1)

Buffer Overflow Vulnerability Lab

David Warren, CS428, dlwarren2@crimson.ua.edu

*Abstract*—This lab is about buffer-overflow vulnerabilities. Buffer overflow is when a program handles data beyond the boundaries of pre-allocated fixed length buffers. This allows a malicious user to change program behavior by either stopping a program from running, or even gaining control of the program. This lab consists of writing a buffer-overflow vulnerability and gaining root privilege of the system. Then, different protection schemes will be tested to evaluate the effectiveness of these different countermeasures.

# INTRODUCTION and Lab Definition

T

HIS lab is an exploration of the concepts of buffer-overflow vulnerabilities and some specific countermeasures. It allows the students to walk through how a buffer-overflow is exploited and allows for the students to witness the effectiveness of selected countermeasures. This is accomplished by first writing an exploit, and then using address space randomization, the StackGuard protection scheme, and finally the non-executable stack for Tasks 1-4 in that order [1]. The exploit is achieved from given shellcode in the lab definition with small modifications to the “badfile” file by inserting a new return address.

# Lab Setup

## Lab Environment

The first step in this lab was to create a suitable lab environment to conduct our exploration. To do this, an Ubuntu VM was created using VirtualBox and an Ubuntu image from SEED security labs’ SEEDUbuntu12.04. A setup document was referenced to create the VM from the pre-built VM image in the previous Lab 1 [2][3]. Then the necessary files from the lab were copied over from the description including call\_shellcode.c, exploit.c, and stack.c. Finally, call\_shellcode.c is compiled with the “execstack” flag and we compile stack with the execstack flag and without StackGuard protection. We are now ready to begin writing our exploit in Task 1.



Fig. 1. Compiling call\_shellcode.c with the execstack flag

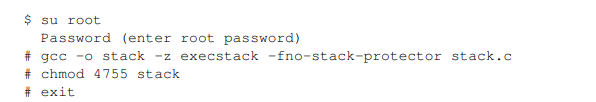


Fig. 2. Compiling stack.c with the execstack flag and no stack protection

# Task 1

## Summary of task

This task is completing the exploit.c code to construct the right conditions in the “badfile” input file. This can be done by overwriting the return address in the “bof” function of stack.c.

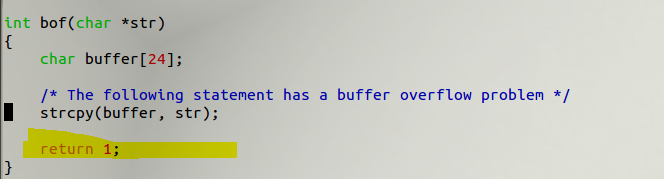


Fig. 3. Return pointer in stack.c we need to find address of.

## Process of task

To accomplish this, we will use gdb to disassemble the function and find the return address. To figure this out we will do as follows. First, we will disassemble the “bof” function seen above. Then, we set a breakpoint after the base pointer is set. We can see this instruction occur at the address 0x0804848a, so we will set a breakpoint there. Then, we will run through the function until we hit the breakpoint. After we hit the breakpoint, we will see what the address of the base pointer is in that function. We see at the end we have finally found the base pointer address.

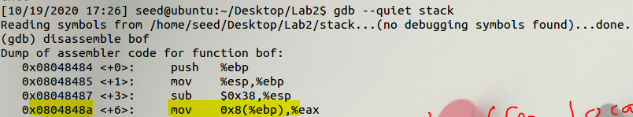


Fig. 4. Find where base pointer is set, and then add a breakpoint there.

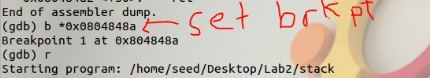


Fig. 4. Run program with breakpoint added.

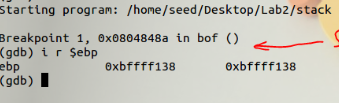


Fig. 5. Display the address of the base pointer.

Now that we have the address of the base pointer, we know the buffer local variable is assigned 32 bytes (0x20) below the base pointer from the instruction at address 0x08048491.



Fig. 6. Indication of where buffer is assigned, and therefore the return address.

Now all we have to do is some simple arithmetic to figure out where to place our new return address. We know the current return address is 4 bytes below the base pointer address, so start of buffer is 36 bytes below return address. This is what we will use to place our new return address in exploit.c, as you can see below.

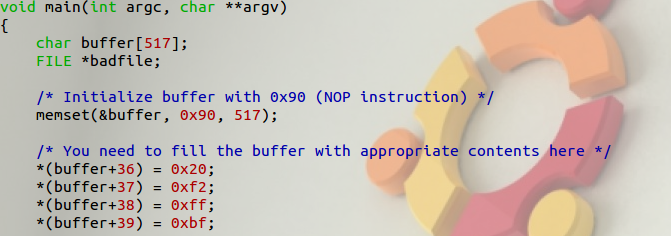


Fig. 7. New return address assigned.

Address is chosen to be in gap between 36 bytes in the buffer, and the shellcode. NOPs will execute until the program hits our new return address and then the shellcode will execute. NOPs are added with memset, and then shellcode is added below.

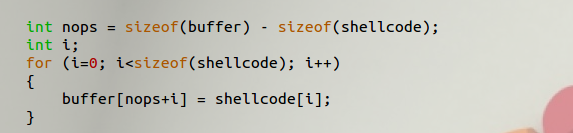


Fig. 8. Logic to determine where to put our shellcode.

After this has been completed, the exploit is ready to be compiled and run to generate the bad file, and then the buffer-overflow vulnerability can be exploited.

## Result of exploit

The exploit successfully works and we gain root access with shellcode. This can be seen after running the whoami command. This task was relatively simple once you understand how the stack is constructed, and can use the debugger to disassemble the stack program to find the base pointer.

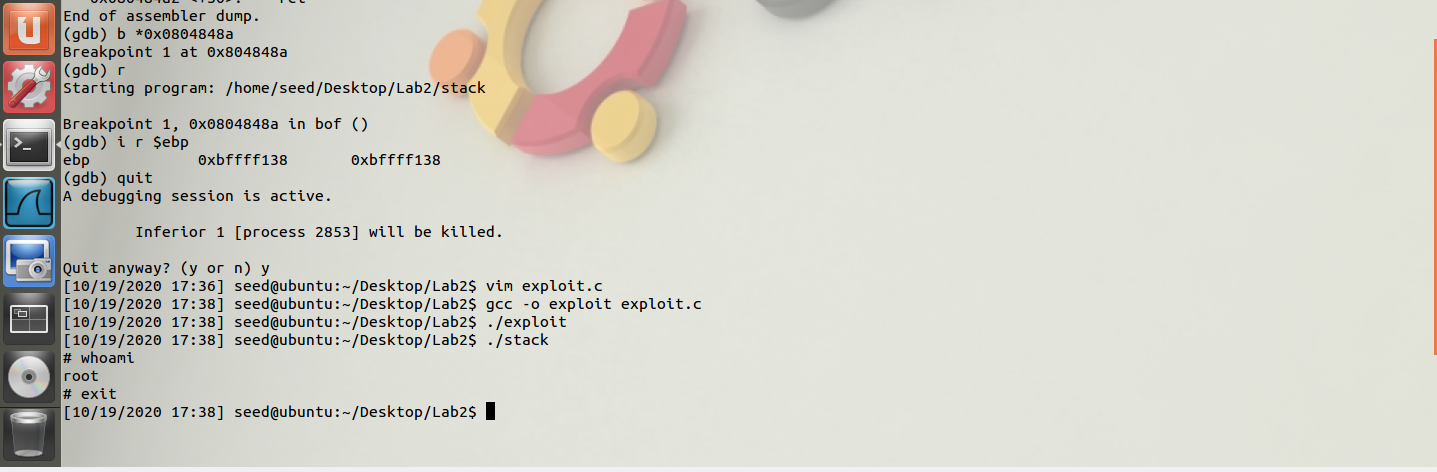


Fig. 9. Exploit success.

# Task 2

## Summary of task

This task is to attempt the exploit when address randomization is turned on with a space of “2.” Then the effectiveness of the countermeasure will be analyzed.

## Process of task

First, we must reenable address randomization with the command below.

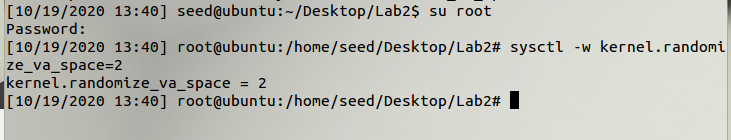


Fig. 10. Increase address randomization.

After this, the exploit is then run in a loop until we reach our shellcode. I achieved this twice. Once, I hit my shellcode in around 7 minutes which was incredibly lucky. I ran it again and it took almost 50 minutes. I suspect this is around the average time to reach shellcode with address randomization turned on with a value of “2.”

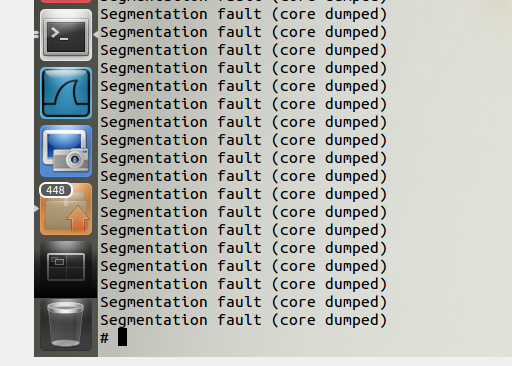


Fig. 11. Shellcode hit after brute force of around 50 minutes.

## Analysis of countermeasure

This countermeasure is quite poor. A brute force attempt success at around 50 minutes allows a malicious user to easily exploit the buffer-overflow vulnerability given a short amount of time. I was lucky on my first brute force attempt and reached shellcode after only about 7 minutes, which was incredibly fast. If the malicious user gets lucky like I did, they could easily use this exploit with little to no time for a security team to interrupt my attempts.

# Task 3

## Summary of task

This task is to attempt the exploit with only Stack Guard enabled. To do this, stack,c will be recompiled and then the exploit will be attempted again.

## Process of task

First, we must set address randomization value back to 0. Then we need to recompile stack.c with the execstack flag but without the -fno-stack-protector flag. The setup can be seen below.

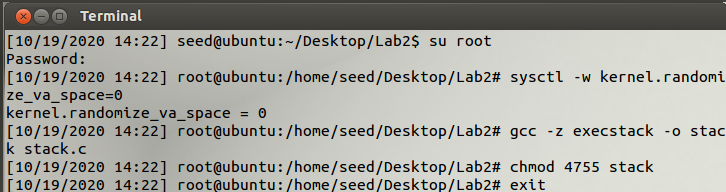


Fig. 12. Address randomization turned off, execstack on, and stack guard off.

Next, the exploit is recompiled and run again. The exploit failed this time, and stack guard caught stack smashing as the canary flag was overwritten by the memset instruction that attempted to insert NOPs over the canary.

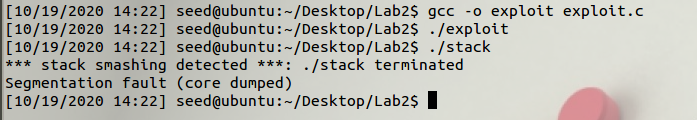


Fig. 13. Failed exploit due to stack guard.

## Analysis of countermeasure

This countermeasure was effective in preventing the exploit. Stack Guard caught when the memset instruction was writing NOPs and the canary was overwritten. Buffer overflow will not work in the presence of Stack Guard as the canary value would be almost impossible to guess and place in the “badfile.”

# Task 4

## Summary of task

The final task is to attempt the exploit with the non-executable stack enabled. To set up this task, stack.c must be recompiled with the -noexecstack flag enabled. The -fno-stack-protector must also be added so that the countermeasures do not interfere with one another, and we can attribute a successful block of the exploit to the non-executable stack.

## Process of task

First, we must recompile the stack program. This can be seen below. We also reinsure that address randomization is disabled for the same reasons as the stack guard.

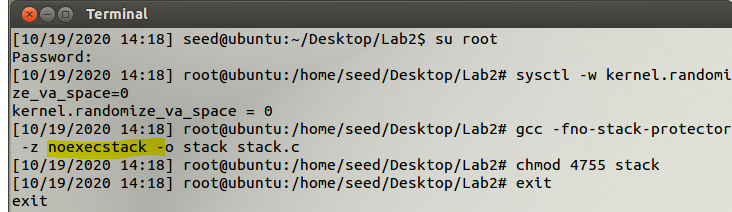


Fig. 14. Recompiling stack program with the -noexecstack flag.

Then, the exploit is attempted again. This time the exploit was successful. This was surprising to me, so I decided to look into why this was the case. We will discuss this in the following section.

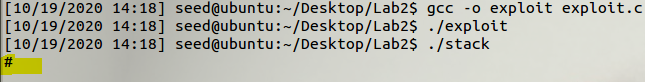


Fig. 15. Still getting root with -noexecstack?

## Analysis of countermeasure

Why did our exploit still work when non-executable stack was enabled? I looked at the lab documents on the SEED ubuntu website and found some documentation of the lab, along with some instructions on the non-executable stack [4]. First, the documentation wanted to ensure that the NX flag was enabled in VirtualBox. I checked and saw that it was as you can see below.

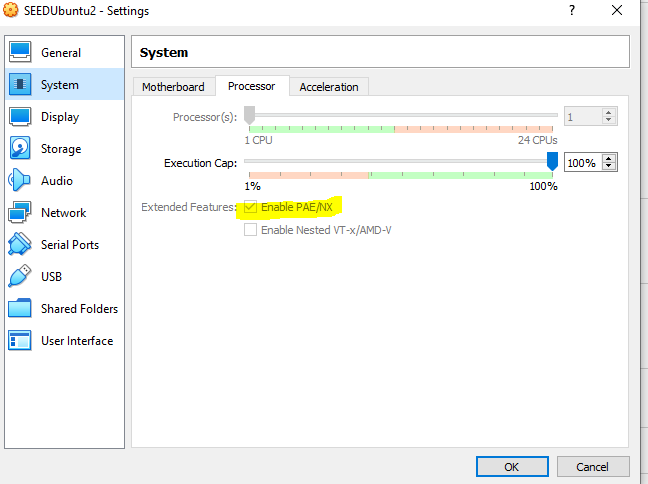


Fig. 16. NX flag enabled on processor.

This was not the problem. Then, I checked the documentation of the differences between Ubuntu11.04 and 12.04 concerning the NX bit [5]. What I found is that on Ubuntu 12.04, NX protection will not be supported at all. In the boot log of Ubuntu12.04 on an NX disabled host as can be seen below.

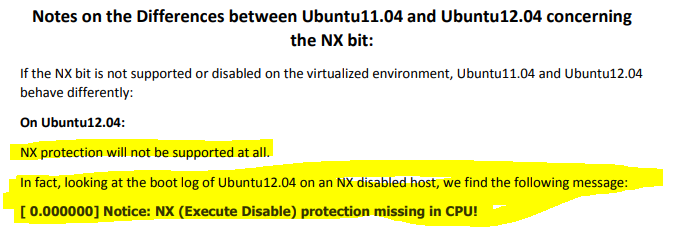


Fig. 17. Ubuntu12.04 NX bit protection status.

I wanted to check for myself so I looked in my own VirtualBox VM boot logs on the Ubuntu12.04 VM using the following command.



Fig. 18. Protection missing in CPU.

I confirmed what the documentation from SEED Ubuntu provided on my own machine and found that even with my NX bit enabled in VirtualBox, the NX protection was still missing in the CPU. This was consistent with what the SEED Ubuntu documentation stated concerning Ubuntu12.04. Based on what I know of Non-Executable Stack countermeasure, this is an effective way of preventing buffer-overflow exploits. However, this may not be available to all OS and could even not be enabled via a negligent programmer.

# Results and Conclusion

In conclusion, the buffer-overflow exploit was successfully used in 3 out of 4 tasks. In actuality, the exploit would have failed in Task 4 with another Ubuntu version that included stack protection when the NX bit was enabled. Completing the base exploit was simple after working with the debugger for a little while and reading about how the stack is constructed. The countermeasure of the Stack Guard was effective and the Non-Executable Stack would have been effective as well if it had been supported by the proper OS version. I would rate the Address Space Randomization as a poor countermeasure however, as I was able to reach my shellcode twice. Once, I reached it in only a few minutes through pure luck. This gives security teams a small window to intervene in an active security breach. Non-executable stack has another issue of not completely preventing buffer-overflow attacks. The return-to-libc attack is an example [1]. Overall, an undefended buffer-overflow vulnerability can be very dangerous and using multiple countermeasures is recommended to prevent attacks.

References

[1] Buffer Overflow Vulnerability Lab. Available:

http://www.cis.syr.edu/~wedu/seed/Labs\_12.04/Software/Buffer\_Overflow/Buffer\_Overflow.pdf.

[2] “How to use VirtualBox to Run Our Pre-built VM Image?” Available: http://www.cis.syr.edu/~wedu/seed/Documentation/Ubuntu12\_04\_VM/UseVirtualBox.pdf.

[3] Crypto Lab - Secret-Key Encryption. Available: http://www.cis.syr.edu/~wedu/seed/Labs\_12.04/Crypto/Crypto\_Encryption/.

[4] Aafer, Yousra. Notes on Non-Executable Stack. Available: https://seedsecuritylabs.org/Labs\_16.04/Software/Buffer\_Overflow/files/NX.pdf.

1. [↑](#footnote-ref-1)