[[1]](#footnote-1)

Return-to-libc Attack LabDavid Warren, CS428, dlwarren2@crimson.ua.edu

*Abstract*—This lab is about Return-to-libc vulnerabilities. This attack occurs similarly to a buffer-overflow attack, but can subvert the protections given via a non-executable stack. The return-to-libc attack does not need an executable stack and uses no shell code. It causes the the program to jump to existing code such as the system() function in libc, which can then give a root shell.

# INTRODUCTION and Lab Definition

T

HIS lab is an exploration of the concepts of the return-to-libc attack and some specific countermeasures. It allows the students to walk through how a return-to-libc attack is conducted 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. Non-executable stack defenses naturally are not effective against this attack as no code is being run in the stack [1]. The exploit is achieved from given code in the lab definition with small modifications to the “badfile” file by inserting a new return address for /bin/sh, the system() function, and the exit() function.

# 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 exploit.c, and retlib.c. Address randomization is first turned off for Task 1. Finally, retlib.c is compiled with the “noexecstack” flag and without StackGuard protection. We are now ready to begin writing our exploit in Task 1.

# Task 1

## Summary of task

In this task, our end goal is create the “badfile” that will be read into retlib.c which contains a return-to-libc vulnerability in the bof() function. After we have made modifications to the retlib.c file and found the correct buffer indices and correct address for each of those indices, we may begin our attack.

## Process of task

The first step we need to do is find the addresses of /bin/sh, system(), and exit(). /bin/sh is found via a helper program named in our instance helper.c, seen below.

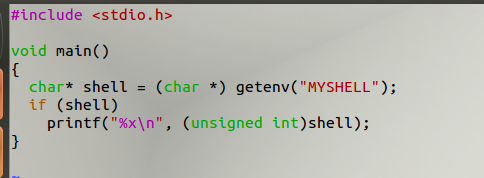


Fig. 1. Helper function to find /bin/sh address.

To use this program, we must first set an environmental variable of $MYSHELL to “/bin/sh” which we can accomplish with the export command “export MYSHELL=”/bin/sh”. We confirm this with a simple echo of $MYSHELL and proceed to run the program. The results of said program may be seen below.



Fig. 2. Helper function results of /bin/sh address.

Now we have our address for /bin/sh and can place this in our first area of the buffer. Next, we must find the address for the system() and exit() calls and place those in the same buffer in exploit.c. This is achieved through the use of gdb. We debug retlib and find the address of system() and exit(), mark those and then place them in the buffer. Results of gdb may be seen below.

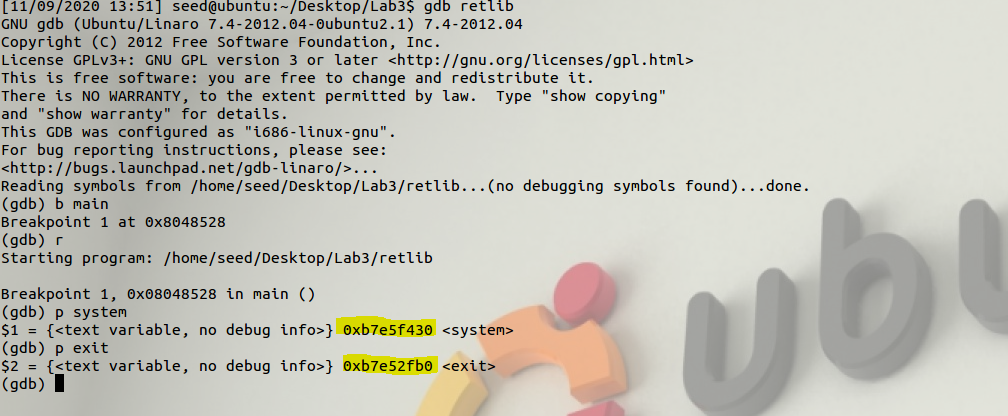


Fig. 3. Using gdb to find system() and exit() addresses.

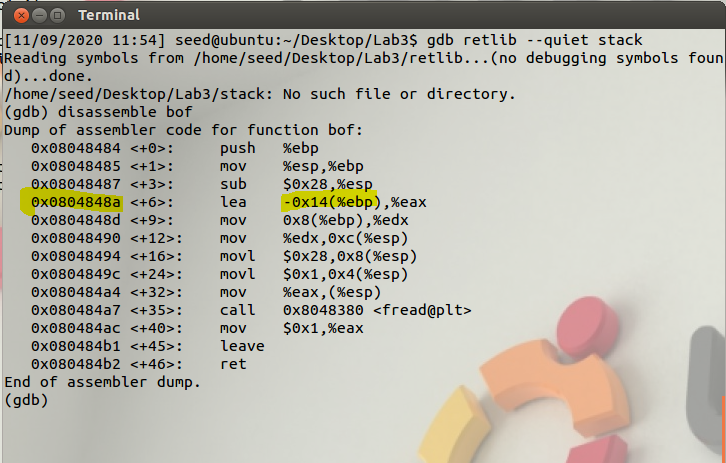
After placing all of these addresses in exploit.c, we must now find the appropriate indices to place them. We can find the beginning of where to place them similar to the buffer-overflow attack. We use gdb to find the address of $ebp, add the length of the base pointer (4 bytes), and place our system() call. Since each of the found addresses are 4 bytes, we just offset each by 4 bytes in the order system(), /bin/sh, and then exit(). To find the base pointer, we disassemble bof from the retlib program, find where it is initialized in the machine code, and then convert the offset to decimal to see where to place it in the buffer. This can be seen below. 

Fig. 4. Using gdb to find system() and exit() addresses.

After we see that the base pointer is being offset by 0x14 bytes, we convert that to decimal (20) and then add the length of the base pointer (4 bytes) to reach our initial system() call index of 24. After this, we just increase the index by 4 each time, placing our /bin/sh address first, and then ending with exit(). The final state of the exploit.c can be seen below.

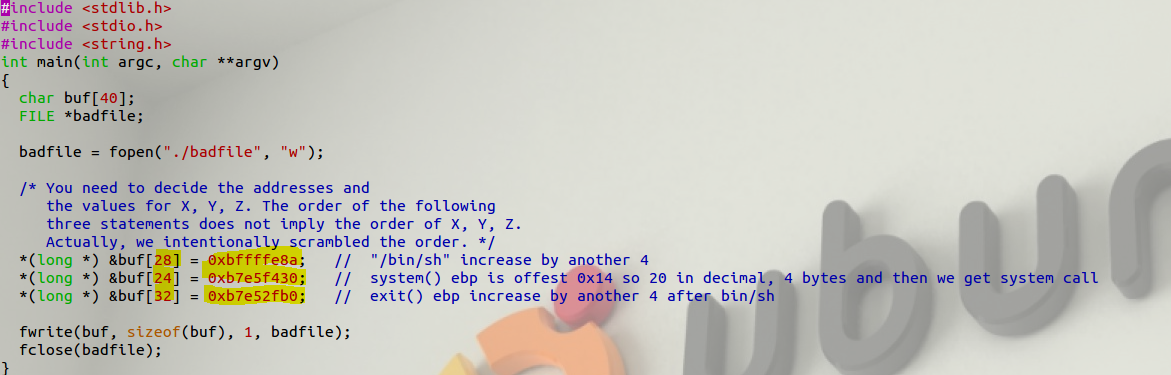


Fig. 5. Final state of exploit.c

Now that we have finished the exploit.c file, we may run our exploit.

## Result of exploit

The exploit was successful and we achieved a root shell. Results can be seen below.

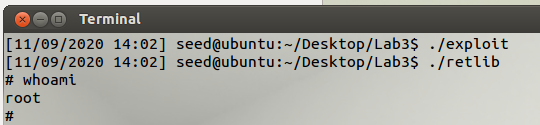


Fig. 6. Success of exploit

We have relatively quickly been able to reach root shell with the return-to-libc attack when there is not address randomization or StackGuard enabled. This may prove to be a very effective attack against a program with a buffer-overflow vulnerability with only a non-executable stack countermeasure enacted.

# Task 2

## Summary of task

Now, we will attempt the same attack but with address randomization set to 2. I predict this will be successful as the addresses have changed and our previous prediction of where /bin/sh, system(), and exit() have most likely changed.

## Process of task

First, we reenable address randomization with a value of 2. See command below.



Fig. 7. Reenable address virtualization.

Next, we generate the bad file and retry the exploit.

## Analysis of countermeasure

This countermeasure successfully prevented the attack and the program lead to a segmentation fault. I attempted what was previously done in the buffer-overflow lab and attempted to brute force the attack by looping the exploit. Command can be seen below.



Fig. 8. Attempted a brute force.

The brute force failed as well. This was expected as the address randomization changed the addresses of the three variables which we had to find in Task 1. Address virtualization appears to be a successful countermeasure against the return-to-libc attack. Results can be seen below.

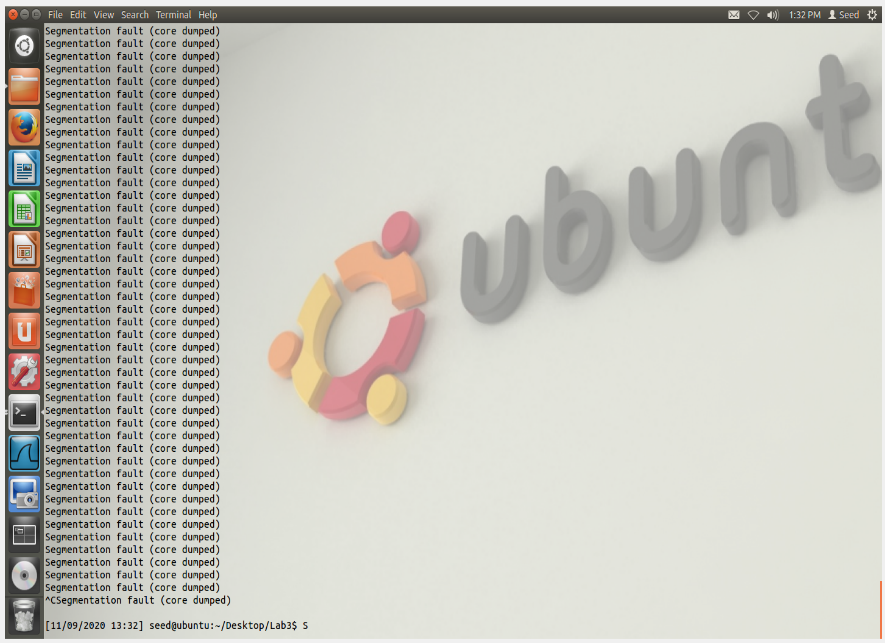


Fig. 9. Failure of brute force.

# Task 3

## Summary of task

Now we will attempt the same exploit with address randomization turned off, and StackGuard Protection enabled. I suspect this countermeasure will also be effective. Attempting a call to system() will result in program termination when in the stack.

## Process of task

After return the address randomization value back to 0, we recompile retlib as root wit StackGuard enabled. Setup command can be seen below.

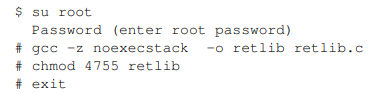


Fig. 10. Compile with StackGuard enabled.

Now we may attempt to run our exploit again.

## Analysis of countermeasure

Even with address randomization disabled, the return-to-libc was still prevented with StackGuard. The program detected stack smashing and then reached a segmentation fault. The result can be seen below.

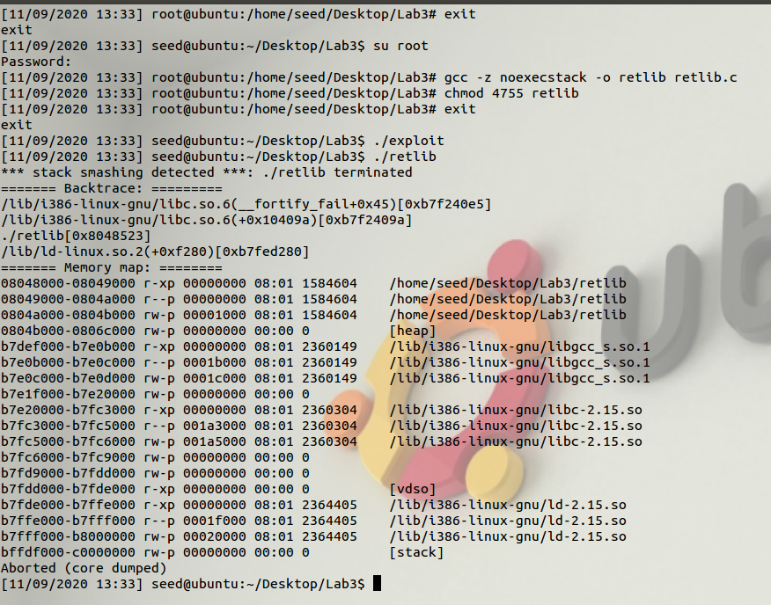


Fig. 10. Stack smashing detected and exploit fails.

As predicted, the system() call was not able to be made and the program crashed. StackGuard Protection is an effective countermeasure against the return-to-libc attack.

# Results and Conclusion

The exploit was successful in one out of three instances. Address randomization was successful as the probability of the same addresses we added in exploit.c is near zero. This would be one effective way in protecting a return-to-libc attack. StackGuard Protection was also a successful countermeasure against the return-to-libc attack. The system() call was not allowed to execute and the program terminated. However, the non-executable stack countermeasure in Task 1 was not effective in preventing the attack. We were able to get a root shell just by manipulating the buffer to point to code that was already in the program rather than adding in our own new shell code like in the buffer-overflow vulnerability lab. Overall, the return-to-libc attack can be successfully completed in certain circumstances but as long as the program takes the proper precautions and has enabled address randomization and StackGuard, return-to-libc will not be successful. Overall, the attack vector for this circumstance is relatively narrower than with the general buffer-overflow, but their still poses an alternative to use this buffer-overflow vulnerability via the return-to-libc attack.

References

[1] Return-to-libc Attack Lab. Available:

http://www.cis.syr.edu/~wedu/seed/Labs\_12.04/Software/Return\_to\_libc/Return\_to\_libc.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/.

1. [↑](#footnote-ref-1)