[[1]](#footnote-1)

Return-to-libc Attack Lab 3

Stephen Sottosanti, CS 428, and smsottosanti@crimson.ua.edu

*Abstract*— These instructions give you guidelines for preparing to run a return-to-libc attack. This attack exploits the non-executable stack vulnerability. Countermeasures on how to prevent such an attack will also be discussed. Return-to-libc is another approach at taking advantage of a program with a buffer overflow vulnerability. The objective of this lab is to gain the first-hand experience on a return-to-libc attack. Text is written to a file which when read in by the vulnerable program will execute our malicious code instead what the program was intended to do.

# Task One

***Lab Definition*—This is very similar to a buffer overflow attack. One security implementation for the buffer overflow attack is a non-executable stack. When the stack is set to executable it makes it easy to implement a buffer overflow. However, when the stack is set to non-executable it makes it more difficult to implement a successful overflow attack. This attack is a return to libc attack as it takes advantage of the system function call that is placed on to the stack because most programs use the c library and when that is the case the c library gets called and placed on to the stack. Such a vulnerability can lead to severe damage. In this lab return-to-libc will be used to get root access on a system by taking advantage of a vulnerable program. An application called VirtualBox is first installed on to your computer. A virtual machine is then installed from seed security labs website; this is because the virtual machine already contains the necessary programs and settings in order easily complete this lab.**

***A.*** *Buffer Overflow Recap*

Buffer overflow attack is a very similar attack and is the approach taken before the return-to-libc attack. If there is a buffer overflow vulnerability, then your first attack is most likely going to be a buffer overflow attack. This attack, if successful, will overflow the buffer that is vulnerable and change the return address. The return address will be re-directed close to where your malicious code is. The return address will then run the malicious code that you placed on to the stack. This only works if the stack is executable, meaning the code that is placed on the stack is allowed to be run. The alternative is a nonexecutable stack. This will not allow code on the stack to be executed. If a buffer overflow attack is attempted this way the program will seg fault and the attack will fail.

*Figure 1.1*



*B. What is Return-to-lib-c*

Return to libc is the next approach when buffer overflow is not going to work because the stack is set to be non-executable. Return to libc still changes the return address but instead of returning the address to the stack the return address needs to point to another function that when passed with the string “/bin/sh” will return a root shell. The vulnerable program shown in figure 1.1 will read text from a file called badfile and within that badfile will contain addresses pointing at functions that will run code the program did not intend to run. After debugging the program, you will get an offset of 20 and the return address is offset value + 4 so we know the return address is at 24 which will be very important when trying to execute the attack.

*C. Return-to-libc*

The ultimate goal is to run the shell code and since it can no longer run on the stack it needs to be ran on something else.

As we know, we can no longer run the shell code on the stack as it will cause a seg fault. There are other options to which we can point the return address. There is more existing code that we have at our disposal if we take a closer look. We have our own code, libraries, and OS code. Unfortunately, the kernel code is protected for good reason and cannot be accessed so our best option is to make use of the library code. The program uses the c library which means it is somewhere in memory and can be used to help with executing the return-to-libc attack. Functions used in the vulnerable program such as print, read, and open are standard library functions that are going to be from the libc library. There are a lot of functions within the libc library which that can be useful. The one that is going to help us successful execute this attack is the system() function as it will allow us to run the “/bin/sh” command.

*System(“/bin/sh”)*

This is going to be a little bit more difficult than a buffer overflow attack. There are a few more steps involved. The first step is to find the address of the system() function as this will be the function that will get us the root shell. After that address is found the next step is to find a way to pass the argument “/bin/sh” to the system function. We have to input that string into memory, locate the address, and then find a way to pass that string to the system() function.

## D. Disable Security Measures

Ubuntu and other Linux distributions have implemented several security measures that will prevent the return-to-libc attack from happening. They will be turned off to make it easier to complete the return-to-libc attack.

*Figure 1.2 Turn off Randomization*

A picture containing text

Description automatically generated

The first security measure to be turned off is the address space randomization. This randomizes the starting address of the heap and the stack. This makes it difficult to locate the return address since it is changing. You will need root permission to run such a command. The command to turn off address space randomization is shown in figure 1.2 and is as follows:

*$sudo sysctl -w kernel.randomize\_va\_space=0*

The GCC complier implements a security mechanism called “Stack Guard” to prevent overflows. When this is active, buffer overflow will not work. Stack guard will be disabled by compiling the vulnerable program with the following arguments:

*$ gcc -fno-stack-protector <vulnerable program>*

Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not. They do so by marking a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This lab will show how to run an attack on a program with a buffer overflow vulnerability with non-executable stack enabled. The vulnerable program will be compiled with the following arguments in addition to the previous one involving the stack protector:

For non-executable stack:

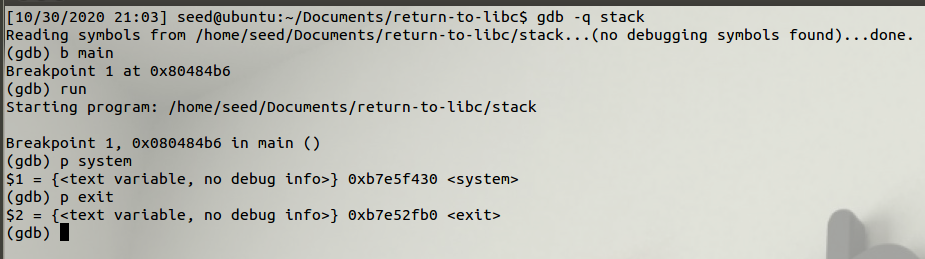
*$ gcc -z nonexecstack -o test test.c*

*E. Change Privileges*

Change the owner of the vulnerable program object file (in my case it is called retlib) to root using the following commands:

*$ sudo chown root retlib*

*$sudo chmod 4755 retlib*

*Figure 1.3*

*F. Address of System() and Exit()*

The system() function is going to be found the in the libc library which is loaded into memory because the vulnerable program uses that library. The exit() function will also be found in the libc library. A debugger will be used to locate both addresses. As you can see from figure 1.3 the following command will be used to run the debugger:

*$gdb -q <object filename for vulnerable program>*

Since the system function is within the libc library we can use a debugger to locate the address of the function. We can also use the debugger to locate the address of the exit() function. Within the debugger a breakpoint at the main() function is set so when the debugger runs it is going to stop when it hits main(). Once the program is started the libc library is loaded into memory. Using the gdb you can then refer to figure 1.3 or use the following commands to print out the address of both the system function and the exit function:

*(gdb) p system*

*(gdb p exit*

These addresses are important and are saved as they will be used to execute the return-to-libc attack.

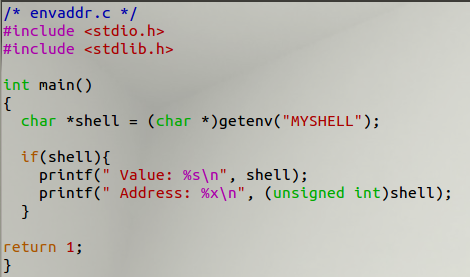
*G. How to pass “/bin/sh”*

After finding the addresses for both the system and the exit function the next step is to add “/bin/sh” to memory and then figure out how to pass that string to the system function. We are going to have to push the string into the vulnerable program ourselves. It is possible that the string might already be in memory but more likely than not we are going to have to put it in memory. The set-UID runs from a parent shell, so inside the parent shell you can define a new environment variable. When the set-UID program is run all the environment variables that have been defined within the parent process will be copied into the child process. The following command was used to export “///bin/sh” as an environment variable:

*$ export MYSHELL=”///bin/sh”*

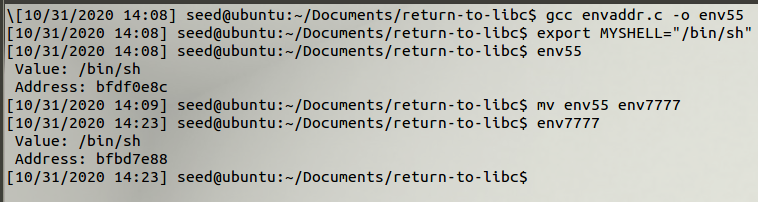
Three /// needed to be used due to variable length that was setting it off.

*Figure 1.4*



After you export the environment variable to the memory you can a simple program such as the one in figure 1.4 and it will return the address of the string “/bin/sh” which will be used in the exploit program.

*Figure 1.5*



The name of the object file of the program that finds the address of the MYSHELL environment is important to take note of. If the length of the object file is changed then the address of the environment variable also changes. This is because before pushing all the environment variables onto the stack the program’s name gets put on the stack. The program name is actually above the environment variable so if the length of the program file changes then the location of the environment variable will be pushed down the stack a little, as you can see in figure 1.5.

*H. How to pass arguments to system()*

Now that we have all of the addresses the next step is to run the exploit program and to pass the /bin/sh string to the system function. We know where the buffer is and where the ebp points to. When we jump to a system function the ebp value changes. Once we are in the systems functions stack frame and we know where the ebp points to, we know that the first argument is ebp + 8 and the return address is at ebp + 4.

The systems ebp gets found from using the debugger. When the vulnerable program ebp value returns, the ebp value changes. The ebp value changes when the function exists, and it also changes when it enters a function. So, the system() function return address is the system() address + 4 bytes. That is where the exit function will go. If we put nothing there, then a random number is most likely going to be put there and the program will most likely crash.

*I. Root Shell*

*Figure 1.6*

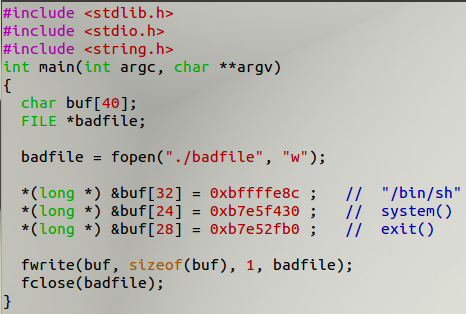


Figure 1.6 (exploit.c) shows the location within the buf variable the different addresses are going to be. Using this program, we can generate a bad file that will be read to the vulnerable program and return a root shell. The following commands are used to compile and set up the bad file for the vulnerable program:

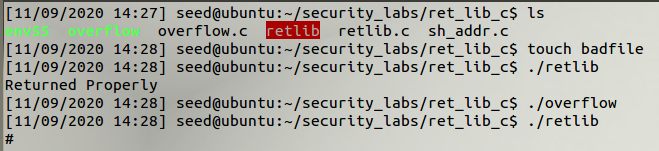
*$ gcc -o exploit explot.c*

*$ touch badfile*

*$ ./exploit*

*$./retlib*

*Figure 1.7*



Exploit will fill the file “badfile” with the addresses of the system function, exit function, and the /bin/sh string address. After running the vulnerable program, a root shell is returned as you can see in figure 1.7.

II. Task Two

***Lab Definition*- There are many implementations within a system to help prevent a buffer overflow attack. task two comprised of turning Address space randomization back on and seeing what the outcome is.**

The main purpose of Address Space Layout Randomization is to make it more difficult for the attacker to launch the attack even if there is a buffer overflow vulnerability. In order to launch the attack, you need to know the ebp value and the buffer value to find the offset. We previously used gdb to debug the program and the find the ebp address along with the offset value, system function, and exit function. After turning the randomization back on I was not able to get a root shell. The address of the functions and the environment variable kept changing every time the program was ran, which is the intention of address space layout randomization.

III. Task Three

***Lab Definition –* task three wanted to see the affects of the stack guard.**

The stack guard will add a marking between the buffer and the return address so in case you have overflowed the buffer, and this endangers the return address the program will automatically stop. The exploit did not work as the stack guard prevented the overflow from happening. The size of the buffer was 12 so anything over that sets the stack guard off and causes the program to seg fault.

IV. Discussion

VirtualBox was installed and used to run the vm that had the necessary programs and settings to complete this lab. The countermeasures that were put in place by default were taken off to help with successfully completing this lab.

V. Conclusion

In conclusion, the offset of the vulnerable program was found using a debugger. This is important ass the offset + 4 bytes is where the return address is located. The address of the system function and the exit function were also found using the debugger. The system function allowed us to run the /bin/sh command and get root access. The exit call was needed to make sure the program did not crash. The location for the first system function argument was the systems address + 8 bytes and then the return address is the systems address + 4 bytes

to find the return address. The address combined with the offset value were used to create a file that when read by the vulnerable program returned to us a root shell.

This lab showed us that even when the stack is non-executable that there is another way to exploit a vulnerable program with a buffer overflow vulnerability. Address randomization and stack guard make it difficult to exploit that vulnerability. They are good countermeasures when trying to protect programs with a overflow vulnerability.

1. [↑](#footnote-ref-1)