Emmanuel Taylor

November 7, 2014

CMSC414 0201

Dr. Shankar

**Project 3: Return-to-libc Attack Lab**

**Purpose:**

The purpose of this lab is to gain a first-hand experience on a style of attack called buffer-overflow. This attack bypasses existing protections implemented in major Linux operating systems. In this lab, we are given a program with a buffer-overflow vulnerability and gain the root privilege. We will learn several protection schemes that have been implemented in Ubuntu to counter against these buffer-overflow attacks.

**Task 0: Initial Setup:**

In order to make the buffer-overflow attacks able to work on our Ubuntu Virtual Machine, we have to disable the security mechanisms that are implemented by Linux distributions and the Virtual Machine. This is done by disabling address space randomization so that the buffer-overflow attack will have no problem guessing the address. This can be done with the following command:

$ su root

Password: (enter root password)

#sysctl –w kernel.randomize\_va\_space=0

Next, we have to disable GCC’s “Stack Guard” security mechanism. We’ll use the following command in the Makefile to bypass this mechanism:

$ gcc –fno-stack-protector example.c

The objective of this lab is to show that the non-executable stack protection does not work so we’ll compile our program using the following command:

$ gcc –z noexecstack –o test test.c

Finally, we’ll compile the retlib.c program that was given to us and make it set-root-uid. This will be accomplished using the following commands:

$ su root

Password: (enter root password)

# gcc –fno-stack-protector –z noexecstack –o retlib retlib.c

# chmod 4755 retlib

# exit

The program above is vulnerable to buffer overflow attacks. It reads an input of 40 bytes from a file called “badfile” into a buffer of size 12 which causes the overflow. We must create contents for “badfile” so that a root shell can be created.

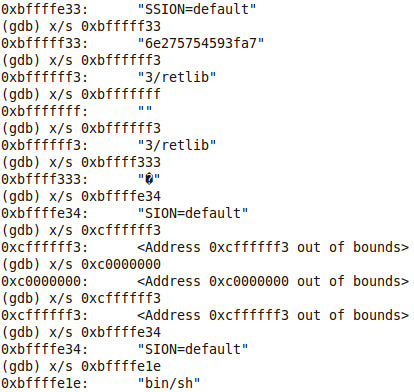
**Task 1: Exploiting the Vulnerability**

For this task, we created the badfile from a given framework. In the file, we were asked to find the address space and values for the system() and exit() functions as well as the string “/bin/sh”. Using the GDB Debugger, I was able to find the values for system() and exit() pretty easily. Below is a screenshot of the commands used to find the address values.



*Figure 1: Screenshot of system() and exit() address values*

Next, I used trial and error to find the address value of “bin/sh”. Below is another screenshot of the commands I used to complete this.



*Figure 2: Screenshot of address values of the string “bin/sh”*

This turned out not to be the completely correct value for the address of “bin/sh”. Through trial and error, I found the correct address to be 0xbffffeab. In order to find values for X, Y, and Z, I used the stack pointer to find the return addresses of each of the system functions in the frame info of the bof() function. It turns out that the first 16 bytes did not matter. Before launching my attack, I made sure to change the following parameters in as a root user: I changed directories to /bin and removed sh. After this I typed the command ln –s /bin/zsh /bin/sh. After launching my attack, I successfully spawned a root shell as you can see below.



*Figure 3: Root Shell*

After successfully spawning a root shell, I tried change the name of the file retlib.c to newretlib.c in order to see if the attack would still be successful. After following the previous steps, the attack was not successful. I think this is because changing the name of the vulnerable program causes the address to change, especially when the number of characters in the file name is different. This is what appeared on th screen of my VM after I tried to run the new executable.



*Figure 4: Failed Attempt at Attack*

**Task 2: Address Randomization**

In this task, we turned on Ubuntu’s address randomization protection and attempted to run the same type of attack that was previous run in Task 1. This was done by using the following command:

$ su root

Password: (enter root password)

# /sbin/sysctl –w kernel.randomize\_va\_space=2

My attack failed when I tried to run my program with Ubuntu’s address randomization protection. I ended up getting a segmentation fault which I think happens because I am trying to access memory that I don’t have rights to. This could be because Ubuntu’s address randomization protection changes the address of the system functions as well as the placement of the “bin/sh” string. For this very reason, address randomization makes the return-to-libc attack very difficult because there is only small chance that you can guess the addresses correctly since they are always changing.

**Task 3: Stack Guard Protection**

In this task, we turned on Ubuntu’s Stack Guard protection. This was done by using the following commands:

$ su root

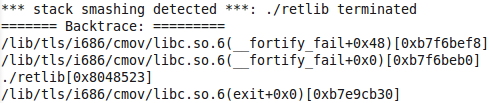
Password: (enter root password)

# gcc –z noexecstack –o retlib retlib.c

# chmod 4755 retlib

#exit

After attempting to run the same attack from Task 1, I was unable to get a shell. Instead, the executable terminated because of stack smashing. Below is a screenshot of the terminal screen.



*Figure 4: Stack Smashing Detection*

Stack Guard makes protection makes it hard to carry out a return-to-libc attack because it protects the return address on the stack from being changed. By placing a “canary” next to the return address, the Stack Guard protection can terminate the program whenever the “canary” is changed.