***Buffer Overflow Vulnerability Lab***

**Introduction**

In this lab, there are some tasks for getting familiar with buffer-overflow vulnerability. Buffer overflow is a condition that a program attempts to write data beyond the boundaries of pre-allocated fixed-length buffers.

Also, we develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, we learn several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks.

**Task1: Running Shellcode**

Shellcode is the code to launch a shell. It must be loaded into the memory so that we can force the vulnerable program to jump to it. In this task, we learned how to launch a shell by executing a shellcode stored in a buffer.

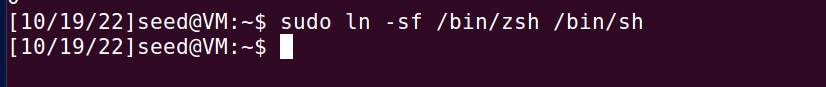
Ubuntu has a security mechanism called address space randomization. It uses this mechanism to randomize the starting address of the heap and stack. This makes guessing the exact addresses difficult. By default, it is on (randomize\_va\_space is 1 or 2 for modern operating systems) we disabled it using the below command:

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

Text

Description automatically generated

The /bin/sh symbolic link points to the /bin/dash shell. The dash shell has a countermeasure that prevents itself from being executed in a Set-UID process, so we will link /bin/sh to another shell that does not have such a countermeasure. This is shown in the below image.



Then, we compiled the following code. This program shows how to launch a shell by executing a shellcode stored in a buffer. we compiled below code using “execstack” option which allows code to be executed from the stack.

Table

Description automatically generated

Compiling shellcode program:

$ gcc -z execstack -o call\_shellcode call\_shellcode.c

And then running the program:

**Text

Description automatically generated**

Shellcode is executed.

**TASK 2: Exploiting the vulnerability**

This program reads input from badfile and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() is only BUF SIZE equals to 300 bytes long, which is less than 517.The buffer overflow will occur because strcpy() does not check boundaries. Since this program is a root-owned Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called badfile. This file is under users’ control. Now, our objective is to create the contents for badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

And then complied the program using below command:

$ gcc -o stack -z execstack -fno-stack-protector stack.c

-fno-stack-protector and "-z execstack" options are for disabling StackGuard and the non-executable stack protection.

Then we made the program a root-owned Set-UID program by changing the ownership of the program to root (1) and then changing the permission to 4755 to enable the Set-UID bit (2).

$ sudo chown root stack (1)

$ sudo chmod 4755 stack (2)

Then created badfile:

$ touch badfile

**Graphical user interface, text, application

Description automatically generated**

**Text

Description automatically generated with medium confidence**

**Text

Description automatically generated**

**Text

Description automatically generated**

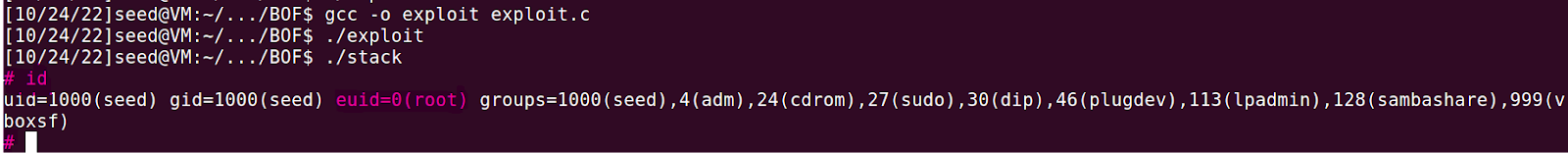
**Text

Description automatically generated**

As shown in the above images, we calculated the return address then we updated the exploit.c according to these values. This program is shown in the below image.

**A picture containing table

Description automatically generated**

****

The above image shows that the effective user ID is root (# means we’ve got the root shell).

**Task3: Defeating dash’s Countermeasure**

The dash shell in Ubuntu drops privileges when the effective user Id does not equal the real UID, so we can change the real user Id of the victim process to zero before invoking the dash program. We can achieve this by invoking setuid(0) before executing execve() in the shellcode.

First I changed the /bin/sh symbolic link, so it points back to /bin/dash:

$ sudo ln -sf /bin/dash /bin/sh

Then 4 instructions were added to the exploit.c according to the following program:

**Text

Description automatically generated with medium confidence**

Then, we created the dash\_shell\_test.c.

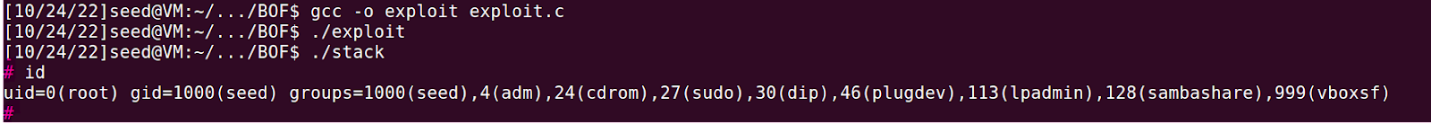
**Text

Description automatically generated**

Then we exploit.c complied and after executing the exploit and stack we see that we have got a root shell.

**Text

Description automatically generated**

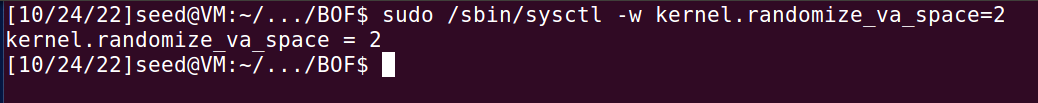
****

**Task4: Defeating Address Randomization**

In this task, we want to defeat the address randomization countermeasure. On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have only 219 = 524,288 possibilities, so we can use the brute-force approach.

First, we turned on address randomization using the below command:

sudo /sbin/sysctl -w kernel.randomize\_va\_space=2



Then we executed brute-force.shell which calls stack program repeatedly.

Text

Description automatically generated

The program stopped and I have got root shell, so my attack was successful.

**Task 5: Turn on the StackGuard Protection**

On this task, we have to turn off address randomization and enable stack guard. For enabling stack guard, I used “-fno-stack-protector” option. After compiling stack.c program, all steps were done according to following images:

**Text

Description automatically generated**

**Text

Description automatically generated**

**Text

Description automatically generated**Graphical user interface, text, application

Description automatically generated

According to the above images, we updated the exploit.c program and run it.

After executing the stack program, we can see the message “stack smashing detected”, and it shows that a buffer overflow attack happened.

**Task 6: Turn on the Non-executable Stack Protection**

On this task, we must turn off address randomization and make the stacks nonexecutable (unlike the previous task). Related commands are shown in the below image. Then we continued other steps according to the following images.

**Text

Description automatically generated**

**Text

Description automatically generated**

**Text

Description automatically generated**

According to the above values we updated the exploit.c program and then we compile and executed it.

Non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks.

**Text

Description automatically generated with medium confidence**