CIS 643 Computer Security

Lab 5 Return-to-libc Attack Lab

**Instructor: Wenliang (Kevin) Du**

Contents

Initialization Routine 2

The Vulnerable Program 2

Turn off memory randomization 2

Task 1: Exploiting the Vulnerability 2

Find return address 2

Find addresses of library functions 3

Using environment variable to set parameter for system() 3

Create badfile 4

Begin attack 4

Task 2: Protection in /bin/bash 5

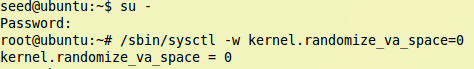
Task 3: Address Randomization and Stack Smash Protection 5

## Initialization Routine

### The Vulnerable Program

### Macintosh HD:Users:Ider:Desktop:Screen Shot 2011-10-17 at 10.51.25 PM.png

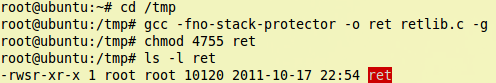
### Turn off memory randomization



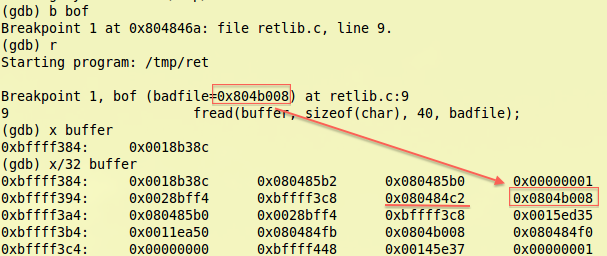
## Task 1: Exploiting the Vulnerability

### Find return address

To make the lab easier, compile the vulnerable program for debug, and also turn off the stack protector

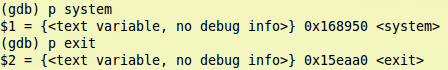


Login as normal user, then use gdb to debug the Set-UID program



Now we could see that return address has 24 bytes offset of buffer

### Find addresses of library functions

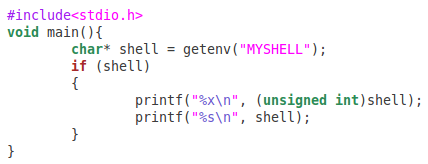
In gdb, print out the addresses of system() and exit()

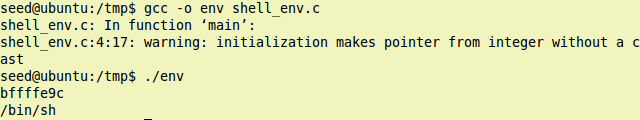
### Using environment variable to set parameter for system()

Set environment variable and export for sub program use

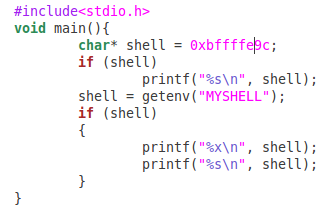
Macintosh HD:Users:Ider:Desktop:Screen Shot 2011-10-17 at 11.15.59 PM.png

Use getenv() to find the address

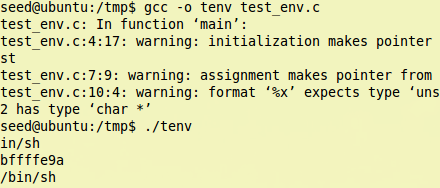


Compile above program and run it

Now we get the address is 0xbffffe9c. So we use this address to test if we could get “/bin/sh”



Compile the test program and run it



Unfortunately, we only could get partial string of MYSHELL, and in the test program, it looks like the address for the environment variable is a little different: it has several bytes offset.

So we need to debug the vulnerable program again to find the real address of MYSHELL in the program. Although we are not sure where it is, but we could tell that it must be near 0xbffffe9c.

Macintosh HD:Users:Ider:Desktop:Screen Shot 2011-10-17 at 11.28.26 PM.png

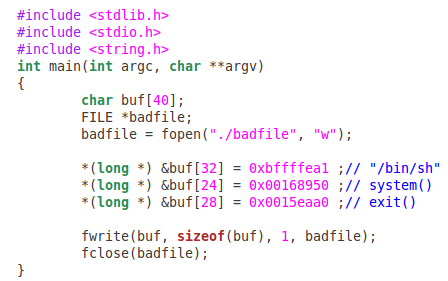
We print out the value at 0xbffffe9c, the result is output is part of “MYSHELL=/bin/sh”. So we could guess that real address would be 0xbffffea1.

Macintosh HD:Users:Ider:Desktop:Screen Shot 2011-10-17 at 11.31.02 PM.png

The output also proves our guess.

### Create badfile

Since we have all the addresses, we could set program to create the badfile



The return address should be overwrite with system() address, so it should be located begin at 24, which is offset from buffer address.

The “return address” for system() should be exit() address, so it begin at 28, which is just after return address.

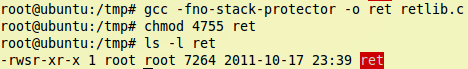
After that, we put the parameter for system().

Compile it and run, we could get the badfile.

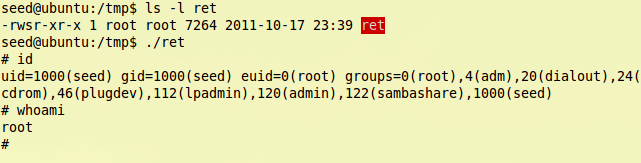
Macintosh HD:Users:Ider:Desktop:Screen Shot 2011-10-17 at 11.41.55 PM.png

### Begin attack

Use root to compile vulnerable program without debug flag



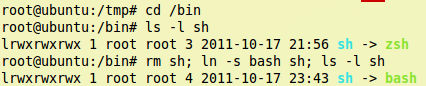
Use normal user to run the program, see if we could run the library function and get root permission:



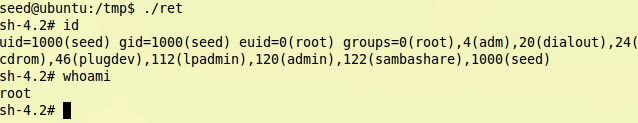
From the result, we see that we get root permission successfully.

## Task 2: Protection in /bin/bash

Make /bin/sh link to /bin/bash instead of /bin/zsh



Leave everything unchanged, and run the vulnerable program again

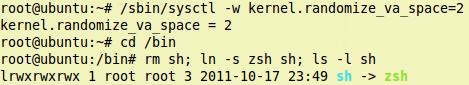


This time, we still could get root permission.

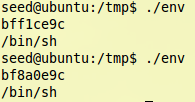
As we observed in Buffer-Overflow lab, the /bin/bash shell in new version of Ubuntu has no protection for preventing Set-UID from reaching bash with effective id, so it is just like zsh.

## Task 3: Address Randomization and Stack Smash Protection

Turn on the address randomization and change /bin/sh back to /bin/zsh

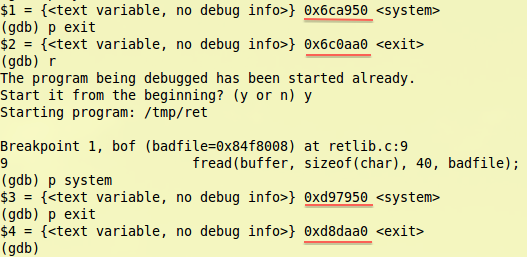


Before run the vulnerable program, let’s find the address of MYSHELL environment variable



We run the program, which will output the address of MYSHELL twice, and find that the address is totally different.

If we debug the vulnerable program, and output the addresses of system and exit



We see that addresses for them are also different.

Although we could run the program many times in Buffer-Overflow lab, as in that lab we only need to guess one address, and NOP would also help us to reach the malicious code. In this task, if we want guess, we need three addresses hit target.

From above picture, we see that difference in address between system() and exit() are both 0x9EB0, so maybe we could use the same offset to guess three addresses.

Anyway, the statistic probability of successful attack would be very low.