**Lab 2.3 Buffer Overflow Vulnerability**

**Overview**

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into actions. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, you will be given a program with a buffer-overflow vulnerability; your task is to develop a scheme to exploit the vulnerability and finally to gain the root privilege. It uses Ubuntu VM created in Lab 2.1. Ubuntu 12.04 is recommended.

1. Initial Setup

**Linux Security Mechanisms**

**Address Space Randomization**. Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses

difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable

these features using the following commands:

$ su root

Password: (enter root password)

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



**ExecShield Protection**. Fedora linux implements a protection mechanism called ExecShield by default, but Ubuntu systems do not have this protection by default. ExecShield essentially disallows executing any code that is stored in the stack. As a result, buffer-overflow attacks will not work. To disable ExecShield in Fedora, you may use the following command.

$ su root

Password: (enter root password)

# sysctl -w kernel.exec-shield=0



**GCC Security Mechanisms**

**The StackGuard Protection Scheme**. The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection when you are comiling the program using the switch -fno-stack-protector. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

$ gcc -fno-stack-protector example.c

**Non-Executable Stack**. 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, i.e., they need to mark 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 marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable. To change that, use the following option when compiling programs:

For executable stack:$ gcc -z execstack -o test test.c

For non-executable stack:$ gcc -z noexecstack -o test test.c

1. Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

#include <stdio.h>

int main( ) {

char \*name[2];

name[0] = ''/bin/sh'';

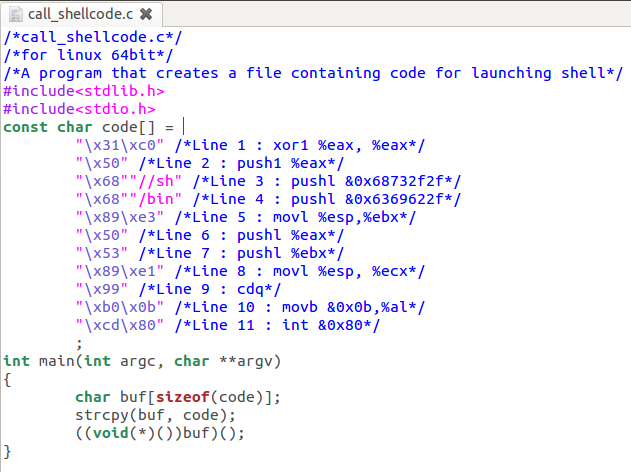
name[1] = NULL;

execve(name[0], name, NULL);

}

The shellcode that we use is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer. Please compile and run the following code, and see whether a shell is invoked.

For 32bit, Insert command “vi call\_shellcode.c” and write the following code.

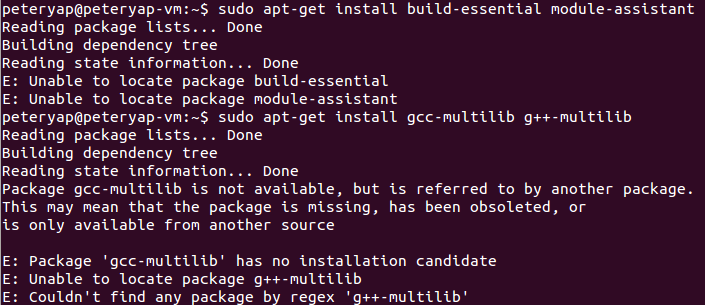


1. 安装一些必要的组件，并设置禁止地址空间的随机分配，通过命令

$ sudo apt-get install build-essential module-assistant

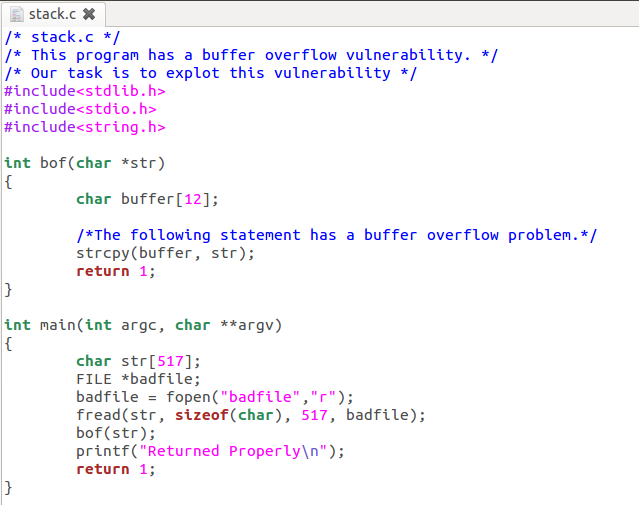
$ sudo apt-get install gcc-multilib g++-multilib

来安装32位的编译方式



1. Write a program “stack.c”.

Insert command “vi stack.c” and write the following code.



1. Compile the Vulnerable Program and make it set-root-uid. You can achieve this by compiling it in the root account, and chmod the executable to 4755: (32bit)

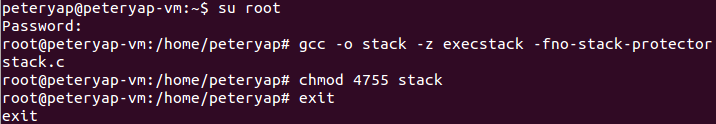
$ su root

Password (enter root password)

# gcc -m32 -g -z execstack -fno-stack-protector -o stack stack.c

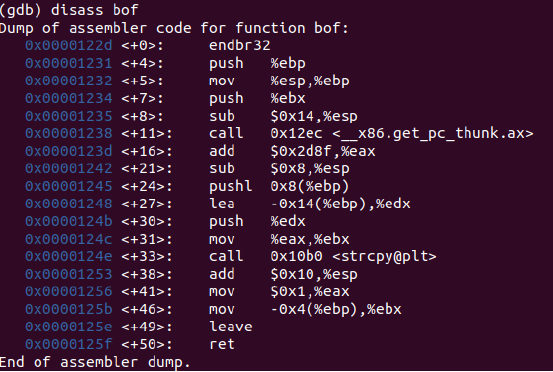
# chmod 4755 stack

# exit

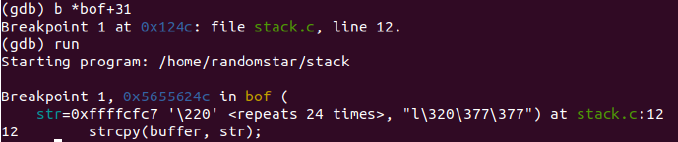


1. A partially completed exploit code called "exploit.c". The goal of this code is to construct contents for "badfile". In this code, the shellcode is given and I need to develop the rest.
2. 通过gdb stack 获取shellcode在内存中的地址，使用disass command来查看汇编代码。

这里我们关注函数bof的汇编代码



从这可看到strcpy这个函数传入了两个参数str和buffer指针，buffer对应的地址可以在0x0000124c中的寄存器eax中找到，因此通过b \*bof+31设置断点，然后通过r command来启动运行。



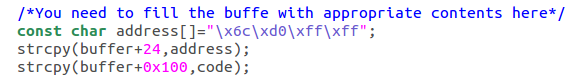
遇到breakpoint时，通过command “i r eax”来查看寄存器中的值，得到0xffffcf6c，也可以用”i r esp”来获得函数的返回地址0xffffcf80+4 ，这个+4是因为ebp被压入栈，因此需要+4





计算差值可得地址的偏移量为0x18，也就是24，如果需要buffer覆盖原来的地址，只需要修改buffer+24的返回地址，堆栈增长的方向是bof函数参数，返回地址，old ebp，buffer的参数(从高地址到低地址)，而返回的地址指向shellcode存放的地址，我们假设存放的地方在buffer+0x100处，则实际的地址是0xffffcf6c+0x100=0xffffd06c

因此在exploit.c文件的代码填写处中应该填写的内容是



执行exploit.c





1. We get the root shell successfully. Lesson complete.