第5章

漏洞利用

学习要求：掌握漏洞利用的核心思想、shellcode 的概念；理解 shellcode 的编写过程，掌握 shellcode 编码技术；掌握 Windows 安全防护技术相关的 ASLR、GS Stack Protection、DEP、SafeSEH、SEHOP 等概念，了解其局限性；掌握跳板攻击、堆喷洒、返回导向编程等漏洞利用技术，理解 API 函数自搜索技术的原理。

课时：4 课时。

分布：[漏洞利用概念—代码植入示例][shellcode 编写—shellcode 编码][Windows 安全防护—地址定位技术][API 函数自搜索技术—绕过其他安全防护]。

5.1 概念及示例

5.1.1 漏洞利用

1. 概念

漏洞利用是指针对已有的漏洞，根据漏洞的类型和特点而采取相应的技术方案，进行尝试性或实质性的攻击。exploit 的意思是利用，它在黑客眼里就是漏洞利用。有漏洞不一定就有 exploit，但是有 exploit 就肯定有漏洞。

假设，刚刚发现了一个 MiniShare 最新版的 0day 漏洞。MiniShare 是一款文件共享软件，该 0day 漏洞是一个缓冲区溢出漏洞，这个漏洞影响之前的所有版本。当用户向服务器发送的报文长度过大（超过堆栈边界）时就会触发该漏洞。得到该漏洞后，可以做点什么呢？善意点的，可以对同学或者朋友的计算机搞恶作剧，让他的计算机弹出个对话框之类的。恶意的话，可以利用这个漏洞向目标机器植入木马，窃取用户个人隐私等。那么，到底如何能达成这些目的呢？

2. 漏洞利用的手段

1996 年，Aleph One 在 *Underground* 发表了著名论文 *Smashing the Stack*

for Fun and Profit，其中详细描述了 Linux 系统中栈的结构和如何利用基于栈的缓冲区溢出。在这篇具有划时代意义的论文中，Aleph One 演示了如何向进程中植入一段用于获得 shell(实际上，shell 是一个命令解释器，它解释由用户输入的命令并且把它们送到内核)的代码，并在论文中称这段被植入进程的获得 shell 的代码为 shellcode。现在，shellcode 已经表达的是广义上的植入进程的代码，而不是狭义上的仅仅用来获得 shell 的代码。

漏洞利用的核心就是利用程序漏洞劫持进程的控制权，实现控制流劫持，以便执行植入的 shellcode 或者达到其他的攻击目的。控制流劫持是一种危害性极大的攻击方式，攻击者能够通过它来获取目标机器的控制权，甚至进行提权操作，对目标机器进行全面控制。当攻击者掌握了被攻击程序的内存错误漏洞后，一般会考虑发起控制流劫持攻击。早期的攻击通常采用代码植入的方式，通过上载一段代码，将控制转向这段代码执行。在栈溢出漏洞利用过程中，攻击的目的是覆盖返回地址，以便劫持进程的控制权，让程序跳转去执行 shellcode。

3. 漏洞利用的结构

要完成控制流劫持和达到不同攻击的目的，exploit 最终是需要执行 shellcode 的，但 exploit 中并不仅仅是 shellcode。exploit 要想达到攻击目标，需要做的工作更多，如对应的触发漏洞、将控制权转移到 shellcode 的指令一般均不相同，而且这些语句通常独立于 shellcode 的代码。这些能实现特定目标的 exploit 的有效载荷，称为 payload。

一个经典的比喻，将漏洞利用的过程可以比作导弹发射的过程：exploit、payload 和 shellcode 分别是导弹发射装置、导弹和弹头。exploit 是导弹发射装置，针对目标发射导弹(payload)；导弹到达目标之后，释放实际危害的弹头(类似 shellcode)爆炸；导弹除了弹头之外的其余部分用来实现对目标进行定位追踪、对弹头引爆等功能，在漏洞利用中，对应 payload 的非 shellcode 部分。

总的来说，exploit 是指利用漏洞进行攻击的动作；shellcode 用来实现具体的功能；payload 除了包含 shellcode 之外，还需要考虑如何触发漏洞并让系统或者程序执行 shellcode。

5.1.2 覆盖邻接变量示例

在 4.1.2 节，已经演示过如何利用栈溢出漏洞覆盖邻接变量、控制程序执行流程、实现漏洞利用，完成软件破解。本节的实验，通过一个外部输入文件，再简单回顾一下利用的过程。

假设已知一个系统的注册机验证过程的漏洞，程序举例如示例 5-1。

【示例 5-1】

```
#include<stdio.h>
#include<windows.h>
#define REGCODE "12345678"
```

```
int verify (char * code)
{
    int flag;
    char buffer[44];
    flag=strcmp(REGCODE, code);
    strcpy(buffer, code);
    return flag;
}
```

假设其主程序启动时要校验注册码：

```
void main()
{
    int vFlag=0;
    char regcode[1024];
    FILE * fp;
    LoadLibrary("user32.dll");
    if (!(fp=fopen("reg.txt","rw+")))
        exit(0);
    fscanf(fp,"%s", regcode);
    vFlag=verify(regcode);
    if (vFlag)
        printf("wrong regcode!");
    else
        printf("passed!");
    fclose(fp);
}
```

verify 函数的缓冲区为 44 字节，对应的栈帧状态如图 5-1 所示。

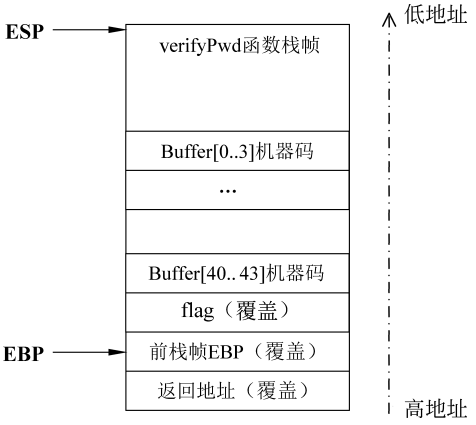

图 5-1　程序对应的堆栈结构图

利用这个漏洞可以破解该软件，让注册码无效。

注意：能成功破解有两个要素。第一是注册码字符串（前 8 字节）要小于 REGCODE，确保 flag 值为 1；第二是通过结束符覆盖 flag 的高位 1，得到使其值变为 0 的效果。

这是一种控制流劫持的漏洞利用手段。

只需要覆盖 flag 状态位使其变为 0。设计要求：buffer（44 字节）＋字节（整数 0）。对应的实现：①在 reg.txt 中写入 45 字节（前 8 字节小于 REGCODE），最后 1 字节为 0；②在 reg.txt 中写入 44 字节（前 8 字节小于 REGCODE），fscanf 读的时候自动添加结束符 0。

我们采用第一个方式，为了对 reg.txt 写入二进制数据，利用 UltraEdit 打开 reg.txt，并在该文件中写入 12341234123412341234123412341234123412341234123412341。需要将最后 1 字节由 ASCII 1 改为 0x00。

单击工具栏的“切换至十六进制模式”，如图 5-2 所示，更改最后 1 字节为 0 即可。

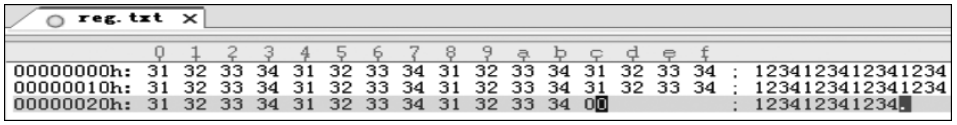

图 5-2　切换至十六进制模式更改最后 1 字节

此时，运行所生成的 exe 程序执行成功。

5.1.3　代码植入示例

通过覆盖返回地址让进程执行植入的 shellcode 是最传统的漏洞利用方式。

shellcode 往往需要用汇编语言编写，并转换成二进制机器码，其内容和长度经常还会受到很多苛刻限制，故开发和调试的难度很高。

植入代码之前需要做大量的调试工作，例如，弄清楚程序有几个输入点，这些输入将最终会当作哪个函数的第几个参数读入内存的那一个区域，哪个输入会造成栈溢出，在复制到栈区时对这些数据有没有额外的限制等。调试之后还要计算函数返回地址距离缓冲区的偏移并覆盖，选择指令的地址，最终制作出一个有攻击效果的“承载”着 shellcode 的输入字符串。

我们将以前面的程序为例，向其植入一段代码，使其达到可以覆盖返回地址，该返回地址将执行一个 MessageBox 函数，弹出窗体。这个代码植入完成攻击的过程就是漏洞利用，也就是 exploit；含有 shellcode 的输入字符串就是 payload；弹出对话框的机器码就是 shellcode。

【实验 5-1】 基于示例 5-1，向其植入一段代码，弹出 MessageBox 窗体。在 Windows XP 环境下，基于 VC 6.0 进行实验。

为了能覆盖返回地址，需要在 reg.txt 中至少写入：buffer（44 字节）＋flag（4 字节）＋前 EBP 值（4 字节），也就是 53～56 字节才是要覆盖的地址。

让程序弹出一个消息框只需要调用 Windows 的 API 函数 MessageBox。

我们将写出调用这个 API 的汇编代码，然后翻译成机器码，用十六进制编辑工具填入 reg.txt 文件。

用汇编语言调用 MessageBoxA 需要 3 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 Console 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 4 个参数。

为了让植入的机器码更加简洁明了，在实验准备中构造漏洞程序时已经人工加载了 user32.dll 库，所以第(1)步操作不用在汇编语言中考虑。

第一步：获得函数入口地址。

有多种方式可以获得函数入口地址，下面介绍两种。

基于工具来获得函数入口地址。MessageBoxA 的入口地址可以通过 user32.dll 在系统中加载的基址和 MessageBoxA 在库中的偏移地址相加得到。具体可以使用 VC 6.0 自带的小工具 Dependency Walker 获得这些信息。可以在 VC 6.0 安装目录下的 Tools 下找到它。

运行 Dependency Walker 后，随便拖曳一个有图形界面的 PE 文件进去，就可以看到它所使用的库文件。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址。

如图 5-3 所示，user32.dll 的基址为 0x77D10000，MessageBoxA 的偏移地址为 0x000407EA。基址加上偏移地址就得到了 MessageBoxA 函数在内存中的入口地址：0x 77D507EA。

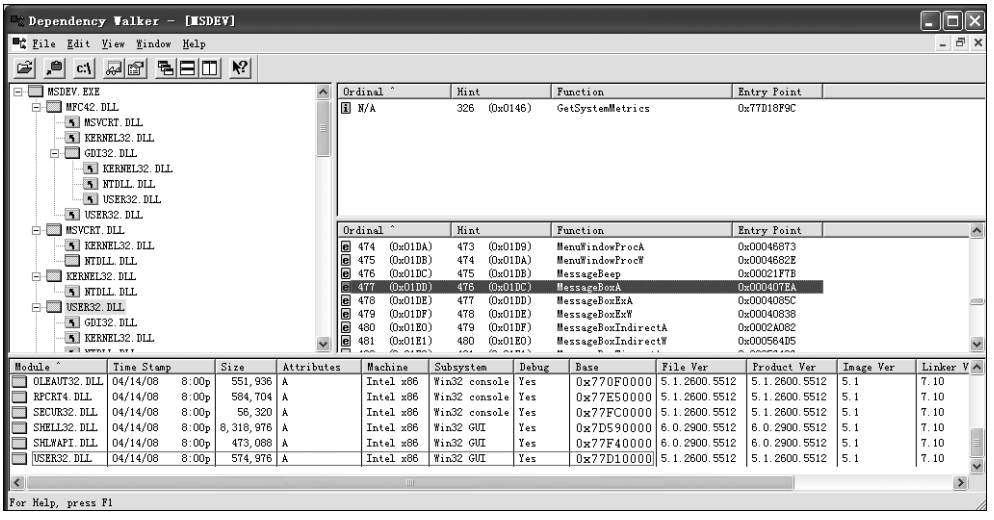

图 5-3　运行 Dependency Walker 后，打开一个 PE 文件

注意：user32.dll 的基址和其中导出函数的偏移地址与操作系统版本号、补丁版本号等诸多因素相关，故用于实验的计算机上的函数入口地址很可能与这里不一致。一定注意要在当前实验的计算机上重新计算函数入口地址，否则后面的函数调用会出错。

使用代码来获取相关函数地址。在 C/C++ 语言中，GetProcAddress 函数检索指定的动态连接库(Dynamic Linked Library，DLL)中的输出库函数地址。如果函数调用成功，返回值是 DLL 中的输出函数地址。函数原型如下：

```
FARPROC GetProcAddress(
  HMODULE hModule,          //DLL 模块句柄
  LPCSTR lpProcName         //函数名
);
```

参数 hModule 包含此函数的 DLL 模块的句柄。LoadLibrary、AfxLoadLibrary 或者 GetModuleHandle 函数可以返回此句柄。参数 lpProcName 是包含函数名的以 NULL 结尾的字符串，或者指定函数的序数值。如果此参数是一个序数值，它必须在一个字的低字节，高字节必须为 0。FARPROC 是一个 4 字节指针，指向一个函数的内存地址，GetProcAddress 的返回类型就是 FARPROC。如果要存放这个地址，可以声明以一个 FARPROC 变量来存放。

```
#include<windows.h>
#include<stdio.h>
int main()
{
        HINSTANCE LibHandle;
        FARPROC ProcAdd;
        LibHandle = LoadLibrary("user32");
        //获取 user32.dll 的地址
        printf("user32 = 0x%x \n", LibHandle);
        //获取 MessageBoxA 的地址
        ProcAdd=(FARPROC)GetProcAddress(LibHandle,"MessageBoxA");
        printf("MessageBoxA = 0x%x \n", ProcAdd);
        getchar();
        return 0;
}
```

运行上述代码后，同样可以得到 MessageBoxA 函数在内存中的入口地址：0x77D507EA。

第二步：编写函数调用汇编代码。

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。首先把字符串 westwest 压入栈区，消息框的文本和标题都显示为 westwest，只要重复压入指向这个字符串的指针即可；第 1 个和第 4 个参数这里都将设置为 NULL。

写出的汇编指令所对应的机器码如表 5-1 所示。

表 5-1　汇编指令所对应的机器码

机器码(十六进制)	汇 编 指 令	注　　释
33 DB	xor　ebx,ebx	将 ebx 的值设置为 0
53	push　ebx	将 ebx 的值入栈
68 77 65 73 74	push　74736577	将字符串 west 入栈
68 77 65 73 74	push　74736577	将字符串 west 入栈
8B C4	mov　eax,esp	将栈顶指针存入 eax(栈顶指针的值就是字符串的首地址)
53	push　ebx	入栈 MessageBox 的参数——类型
50	push　eax	入栈 MessageBox 的参数——标题
50	push　eax	入栈 MessageBox 的参数——消息
53	push　ebx	入栈 MessageBox 的参数——句柄
B8 EA 07 D5 77	mov eax, 0x77D507EA	调用 MessageBoxA 函数,注意,每个机器的该函数的入口地址不同,按实际值写入
FF D0	call eax	

得到的 shellcode：33 DB 53 68 77 65 73 74 68 77 65 73 74 8B C4 53 50 50 53 B8 EA 07 D5 77 FF D0。

第三步：注入 shellcode 代码。

将这段 shellcode 写入 reg.txt 文件,且在返回地址处写 buffer 的地址,如图 5-4 所示。

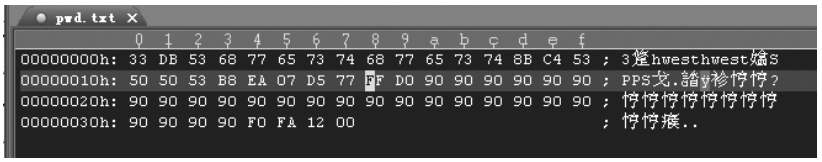

图 5-4　shellcode 写入 reg.txt

Buffer 的地址可以通过 OllyDbg 查看得到,也可以通过 VC 6.0 转到反汇编方式得到,即 0012faf0(该地址跟随环境不同可能会发生变化)。

攻击成功效果如图 5-5 所示。

注意：Windows XP 环境下静态 API 的地址是准确的,但是 Windows XP 之后的操作系统版本增加了 ASLR(Address Space Layout Randomization)保护机制,地址就不准确了,需要动态获取,利用地址定位技术或者通用型 shellcode 编写可以解决这个问题。

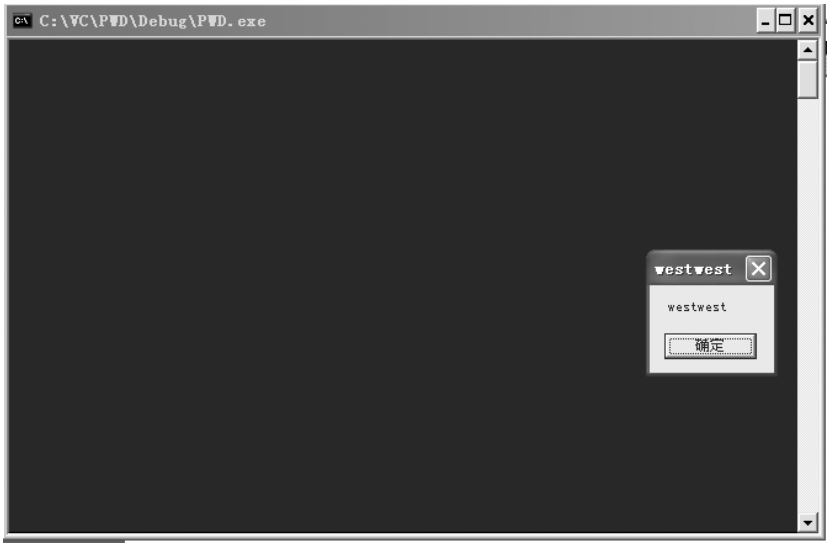

图 5-5 攻击成功

5.2 shellcode 编写

漏洞利用中最关键的是 shellcode 的编写。

上面演示了一个通过汇编语言编写 shellcode 的例子，但是，直接用汇编语言编写很麻烦，而且还需要查表来获得其机器码，很容易出错。此外，即使我们可以熟练地用汇编语言编写 shellcode 代码，但还需要对一些特定字符进行转码。例如，对于 strcpy 等函数造成的缓冲区溢出，会认为 NULL 是字符串的终结，所以 shellcode 中不能有 NULL，如果有需要则要进行变通或编码。

shellcode 获取的工具。除了手动编写 shellcode，可以利用 Metasploit 框架下的 msfvenom 生成 shellcode，还有一些工具有助于获取 shellcode，如 cobaltstrike 等。

本节重点介绍 shellcode 编写和代码提取的方法和思路。

5.2.1 提取 shellcode 代码

由于 shellcode 必须以机器码的形式存在，因此，如何得到机器码是一个关键技术。

一种简单编写并提取 shellcode 的方法如下。

1. 用 C 语言书写要执行的 shellcode

使用 VC 6.0 编写程序，如示例 5-2 所示。

【示例 5-2】

```
#include<stdio.h>
#include<windows.h>
```

```
void main()
{
    MessageBox(NULL,NULL,NULL,0);
    return;
}
```

2. 换成对应的汇编代码

利用调试功能，找到其对应的汇编代码，如图 5-6 所示。

```
1:    #include<stdio.h>
2:    #include<windows.h>
3:    void main()
4:    {
00401010   push        ebp
00401011   mov         ebp,esp
00401013   sub         esp,40h
00401016   push        ebx
00401017   push        esi
00401018   push        edi
00401019   lea         edi,[ebp-40h]
0040101C   mov         ecx,10h
00401021   mov         eax,0CCCCCCCCh
00401026   rep stos    dword ptr [edi]
5:        MessageBox(NULL,NULL,NULL,0);
00401028   mov         esi,esp
0040102A   push        0
0040102C   push        0
0040102E   push        0
00401030   push        0
00401032   call        dword ptr [__imp__MessageBoxA@16 (0042428c)]
00401038   cmp         esi,esp
0040103A   call        __chkesp (00401070)
6:        return;
```

图 5-6　得到汇编代码

直接得到的汇编语言通常需要进行再加工。对于 push 0 而言，可以通过“xor ebx，ebx”之后执行 push ebx 来实现(push 0 的机器码会出现 1 字节的 0，对于直接利用需要解决字节为 0 的问题，因此转换为 push ebx)。具体地，在工程中编写汇编语言如示例 5-3 所示。

【示例 5-3】

```
#include<stdio.h>
#include<windows.h>
void main(){
    LoadLibrary("user32.dll"); //加载 user32.dll
    _asm
{
    xor ebx,ebx
    push ebx                    //push 0
    push ebx
    push ebx
    push ebx
    mov eax, 77d507eah          //77d507eah 是 MessageBox 函数在系统中的地址
    call eax
```

```
}
return;
}
```

push 0 不建议直接使用，因此采用了“xor ebx,ebx”之后执行 push ebx 来代替。

3. 根据汇编代码，找到对应地址中的机器码

同样，在第一行汇编代码处打断点，利用调试定位具体内存中的地址，如图 5-7 所示。

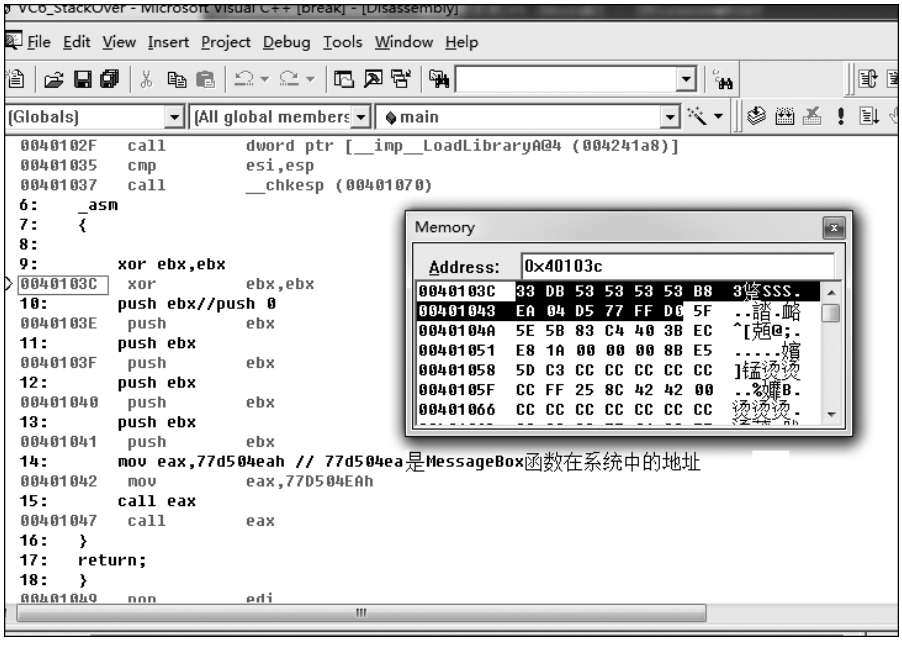

图 5-7 定位具体内存中的地址

注意：实际调试时，MessageBox 函数的入口地址需要根据自己的计算机重新计算。

这样，在 Memory 窗口就可以找到对应的机器码：33 DB 53 53 53 53 B8 EA 04 D5 77 FF D0。

接下来就可以利用这个 shellcode 来实现漏洞利用了，一个 VC 6.0 测试程序如示例 5-4 所示。

【示例 5-4】

```
#include<stdio.h>
#include<windows.h>
char ourshellcode[]="\x33\xDB\x53\x53\x53\x53\xB8\xEA\x07\xD5\x77\xFF\xD0";
void main()
{
    LoadLibrary("user32.dll");
    int * ret;
```