第 4 章

缓冲区溢出

内容提要

缓冲区溢出是一种常见的软件漏洞形式，可被用于实现远程植入、本地提权、信息泄露、拒绝服务等攻击目的，具有极大的攻击力和破坏力。学习缓冲区溢出原理和利用有助于巩固自身安全，加强系统防御。本章包含六个实验，涵盖了缓冲区溢出原理和利用两部分内容，前者包括栈溢出、整型溢出、UAF（Use After Free）类型缓冲区溢出实验，后者通过覆盖返回地址、覆盖函数指针和覆盖 SHE（Structured Exception Handler）链表实验学习溢出利用技术。

本章重点

- 缓冲区溢出原理及实践；
- 常见缓冲区溢出利用方式及实践。

4.1　概述

缓冲区一词在软件中指的是用于存储临时数据的区域，一般是一块连续的内存区域，如 char Buffer[256]语句就定义了一个 256 B 的缓冲区。缓冲区的容量是预先设定的，但是如果往里存入的数据大小超过了预设的区域，就会形成所谓的缓冲区溢出。例如，memcpy（Buffer, p, 1024）语句，复制的源字节数为 1024 B，已经超过了之前 Buffer 缓冲区定义的 256 B。

由于缓冲区溢出的数据紧随源缓冲区存放，必然会覆盖到相邻的数据，从而产生非预期的后果。从现象上看，溢出可能会导致：

（1）应用程序异常；

（2）系统服务频繁出错；

（3）系统不稳定甚至崩溃。

从后果上看，溢出可能会造成：

（1）以匿名身份直接获得系统最高权限；

（2）从普通用户提升为管理员用户；

（3）远程植入代码执行任意指令；

（4）实施远程拒绝服务攻击。

产生缓冲区溢出的原因有很多，如程序员的疏忽大意，C 语言等编译器不做越界检查等。学习缓冲区溢出的重点在于掌握溢出原理和溢出利用两方面的内容。

4.2　缓冲区溢出原理及利用

下面介绍缓冲区溢出原理和缓冲区溢出利用两部分内容。

4.2.1　缓冲区溢出原理

栈溢出、整型溢出和 UAF（Use After Free）类型缓冲区溢出是缓冲区溢出常见的三种溢出类型，下面分别介绍它们的原理。

1．栈溢出原理

“栈”是一块连续的内存空间，用来保存程序和函数执行过程中的临时数据，这些数据包括局部变量、类、传入/传出参数、返回地址等。栈的操作遵循后入先出（Last In First Out，LIFO）的原则，包括出栈（POP 指令）和入栈（PUSH 指令）两种。栈的增长方向为从高地址向低地址增长，即新入栈数据存放在比栈内原有数据更低的内存地址，因此其增长方向与内存的增长方向正好相反。

有三个 CPU 寄存器与栈有关：

（1）SP（Stack Pointer，x86 指令中为 ESP，x64 指令中为 RSP），即栈顶指针，它随着数据入栈出栈而变化；

（2）BP（Base Pointer，x86 指令中为 EBP，x64 指令中为 RBP），即基地址指针，它用于标示栈中一个相对稳定的位置，通过 BP，可以方便地引用函数参数及局部变量；

（3）IP（Instruction Pointer，x86 指令中为 EIP，x64 指令中为 RIP），即指令寄存器，在调用某个子函数（call 指令）时，隐含的操作是将当前的 IP 值（子函数调用返回后下一条语句的地址）压入栈中。

当发生函数调用时，编译器一般会形成如下程序过程：

（1）将函数参数依次压入栈中；

（2）将当前 IP 寄存器的值压入栈中，以便函数完成后返回父函数；

（3）进入函数，将 BP 寄存器值压入栈中，以便函数完成后恢复寄存器内容至函数之前的内容；

（4）将 SP 值赋值给 BP，再将 SP 的值减去某个数值用于构造函数的局部变量空间，其数值的大小与局部变量所需内存大小相关；

（5）将一些通用寄存器的值依次入栈，以便函数完成后恢复寄存器内容至函数之前的内容，此时栈的布局如图 4.1 所示。

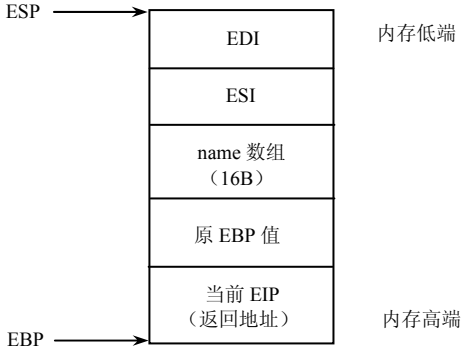

图 4.1　栈布局图

（6）开始执行函数指令；

（7）函数完成计算后，依次执行程序过程（5）、（4）、（3）、（2）、（1）的逆操作，即先恢复通用寄存器内容至函数之前的内容，接着恢复栈的位置，恢复 BP 寄存器内容至函数之前的内容，再从栈中取出函数返回地址之后返回父函数，最后根据参数个数调整 SP 的值。

栈溢出指的是向栈中的某个局部变量存放数据时，数据的大小超出了该变量预设的空间大小，导致该变量之后的数据被覆盖破坏。由于溢出发生在栈中，所以被称为栈溢出。

防范栈溢出需要从以下几方面入手：

（1）编程时注意缓冲区的边界；

（2）不使用 strcpy、memcpy 等危险函数，仅使用它们的替代函数；

（3）在编译器中加入边界检查；

（4）在使用栈中重要数据之前加入检查，如 Security Cookie 技术。

2．整型溢出原理

在数学概念中，整数指的是没有小数部分的实数变量；而在计算机中，整数包括长整型、整型和短整型，其中每一类又分为有符号和无符号两种类型。如果程序没有正确的处理整型数的表达范围、符号或者运算结果时，就会发生整型溢出问题，这一般又分为三种类型。

（1）宽度溢出。由于整型数都有一个固定的长度，存储其的最大值是固定的，如果该整型变量尝试存储一个大于这个最大值的数，将会导致高位被截断，引起整型宽度溢出。

（2）符号溢出。有符号数和无符号数在存储的时候是没有区别的，如果程序没有正确地处理有符号数和无符号数之间的关系，比如将有符号数当做无符号数对待，或者将无符号数当做有符号数对待时，就会导致程序理解错误，引起整型符号溢出问题。

（3）运算溢出。整型数在运算过程中常常发生进位，如果程序忽略了进位，就会导致运算结果不正确，引起整型运算溢出问题。

整型溢出是一种难以杜绝的漏洞形式，其大量存在于软件中。要防范该溢出问题除了注意正确编程外，还可以借助代码审核工具来发现问题。另外整型溢出本身并不会带来危害，只有当错误的结果被用到了如字符串复制、内存复制等操作中才会导致严重的栈溢出等问题，因此也可以从防范栈溢出、堆溢出的角度进行防御。

3．UAF 类型缓冲区溢出原理

UAF 类型缓冲区溢出是目前较为常见的漏洞形式，它指的是由于程序逻辑错误，将已释放的内存当做未释放的内存使用而导致的问题，多存在于 Internet Explorer 等使用了脚本解释器的浏览器软件中，因为在脚本运行过程中内部逻辑复杂，容易在对象的引用计数等方面产生错误，导致使用已释放的对象。

4.2.2　缓冲区溢出的利用

缓冲区溢出会造成程序崩溃，但要达到执行任意代码的目的，还需要做到如下两点：一是在程序的地址空间里安排适当的代码，这些代码可以完成攻击者所需的功能；二是控制程序跳转到第一步安排的代码去执行，从而完成指定的功能。

1）*在程序的地址空间里安排适当的代码*

在程序的地址空间里安排适当的代码包括植入法和利用已经存在的代码两种方法。

（1）植入法：一般是向被攻击程序输入一个过长的字符串作为参数，而程序将该字符串不加检查地放入缓冲区。这个字符串里包含了由攻击者精心构造的一段 Shellcode。Shellcode 实质上就是机器指令序列，可以完成攻击者所需的功能。

（2）利用已经存在的代码：有时候攻击者所需要的代码已经在被攻击的程序中，攻击者可以不必自己再去写烦琐的 Shellcode，而只需控制程序跳转至该段代码并执行，然后给相应的函数调用传递一些参数。

2）*控制程序跳转的方法*

（1）覆盖返回地址：每当发生一个函数调用时，栈中都会保存函数结束后的返回地

址。攻击者通过改写返回地址使之指向攻击代码，这类缓冲区溢出被称做“stack smashing attack”。

（2）覆盖函数或者对象指针：函数指针可以用来定位任何地址空间，如果攻击者在能够溢出的缓冲区附近找到函数指针，那么就可以通过溢出该缓冲区来改变函数指针。在之后的某一时刻，当程序调用该函数时，程序的流程就按照攻击者的意图跳转了。

（3）覆盖 SEH 链表：有的函数在使用函数指针和返回地址之前做了检测，一旦发现更改就会做相应的处理来避免遭受溢出攻击，从而使以上两种方法无法成功，而若通过覆盖 Windows 系统下的结构化异常处理（SEH）链表则可以较好地绕过防护完成攻击。

下面介绍这三种缓冲区溢出利用技术。

1．覆盖返回地址

通过覆盖返回地址来控制程序流程是栈溢出最常见的利用技术。从前面介绍的栈溢出原理可以看出，返回地址处于栈中较高内存的位置，很容易被超长的局部变量所覆盖，程序最终执行至被覆盖的地址处指令时发生错误。由于该地址来自局部变量，而局部变量又来自用户输入即程序参数，因此只需要修改程序参数就可以控制程序的流程。注意，当程序出错时，ESP 寄存器的值正好指向程序参数中的某个位置，因此要利用该漏洞，可以将该处填充为 shellcode，并将程序参数中被覆盖的返回地址的 4 个字节修改为内存的某个指令地址，该地址的指令为 jmp esp（16 进制为 0xff 0xe4）。此时覆盖返回地址时的栈布局如图 4.2 所示。

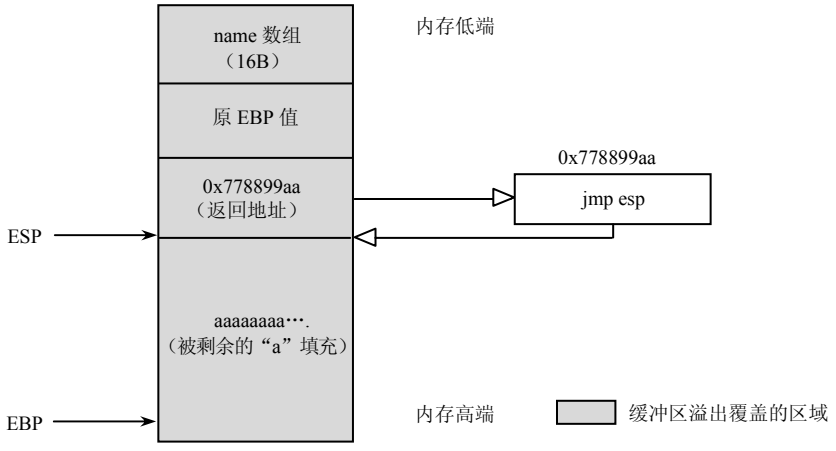

图 4.2　覆盖返回地址时的栈布局

2．覆盖函数或对象指针

函数指针是一种特殊的变量，它用于保存函数的起始地址。当调用函数指针时，程序会转向该起始地址执行代码。如果函数指针被保存在缓冲区之后（更高地址），当发生缓冲区溢出时，函数指针就会被覆盖，之后如果调用了该函数指针，就可以控制程序的流程了。

3．覆盖 SEH 链表

首先简单介绍一下 Windows 结构化异常处理机制。结构化异常处理是一种对程序异常的处理机制，它把错误处理代码与正常情况下所执行的代码分开。当系统检测到软件发生异常时，执行线程立即被中断，并将控制权交给异常调度程序，它负责从结构化异常处理（SEH）链表中查找处理异常的方法。

SEH 链表按照单链表的结构组织，链中所有节点都存储在栈空间。每个链中的节点由两个字段组成，第一个字段是指向下一个节点的指针，第二个字段是异常处理回调函数的指针。而最后一个节点的 Next 指针为 0xFFFFFFFF，如图 4.3 所示。

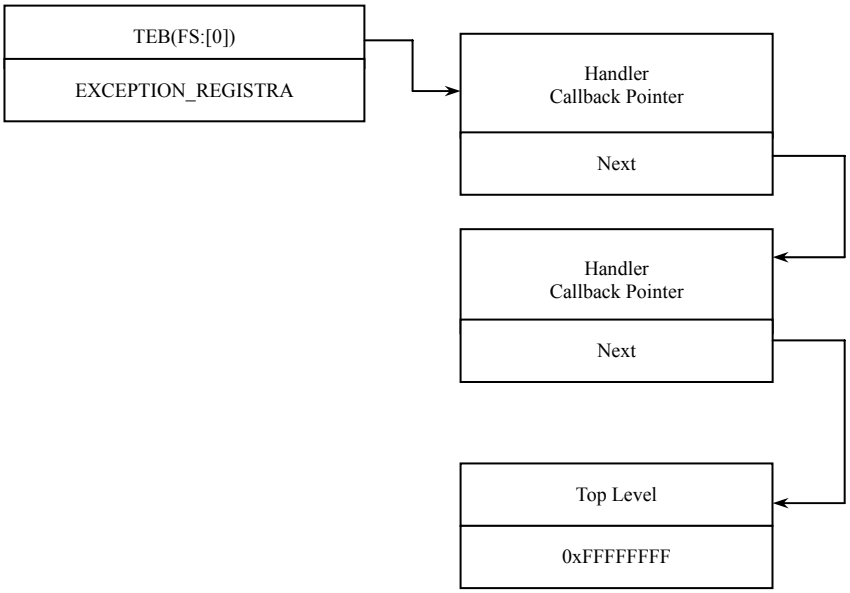

图 4.3　SEH 链表结构图

SEH 链表的插入操作采用头插法，当有新的结构加入链中时，通常会看到下面的操作：

```
pushxxxxxxxx
moveax,fs:[0]
pusheax
movdwordptrfs:[0],esp
```

其中，fs:[0]始终指向链中的第一个节点，而 push xxxxxxxx 所做的工作就是把回调函数指针压入栈中。接着通过后面三条汇编指令修改两个指针，完成节点的插入操作。当线程中发生异常时，操作系统需要从头节点遍历 SEH 链表，调用第一个回调函数来处理异常；如果异常已被处理则停止遍历，否则调用下一个回调函数。依此类推，如果所有回调函数都不能处理异常，则使用最后一个——默认的异常处理节点，弹出出错的对话框，然后中止进程的执行。

栈溢出时，如果在函数返回之前，发生了对返回地址的检查，或者是由于栈中的局部变量遭到破坏，导致程序发生异常，这些情况下都不能使用返回地址来进行溢出利用。

考虑到大多数情况下栈的内容被破坏时（其中也包括了 SEH 链表上的节点），结构

化异常处理是程序执行过程中另一个隐蔽的流程，因此可以通过修改 SEH 链表节点来控制程序流程。

4.3　栈溢出实验

4.3.1　实验目的

本实验要求了解栈的内存布局和工作过程，掌握栈溢出原理。

4.3.2　实验内容及环境

1．实验内容

本实验通过调试器跟踪栈溢出发生的整个过程，验证和掌握栈溢出原理。

2．实验环境

（1）靶机系统环境为 Windows XP SP3 32 位。

（2）OllyDbg：是一款动态调试工具。OllyDbg 将 IDA 与 SoftICE 结合起来，是 Ring 3 级调试器，非常容易上手掌握。

（3）Visual C++ 6.0（VC 6.0）：Visual C++是微软推出的一款 C++编译器，是一款功能强大的可视化软件开发工具。自 1993 年微软公司推出 Visual C++ 1.0 后，随着其新版本的不断问世，Visual C++就已成为专业程序员进行软件开发的首选工具。Visual C++ 6.0 由许多组件组成，包括编辑器、调试器及程序向导、类向导等开发工具。

4.3.3　实验步骤

1．编译代码

通过 VC 6.0 将以下代码编译成 debug 版的.exe 文件。

```
1    intmain(intargc, char* argv[])
2    {
3        char name[16];
4        strcpy(name, (const char*)argv[1]);
5        printf("%s\n", name);
6        return 0;
7    }
```

2．加载程序

生成.exe 文件并使用 OllyDbg 加载.exe 文件，设置程序参数为 30 个“a”，按“F9”键直接运行到 main 函数入口处，如图 4.4 所示。

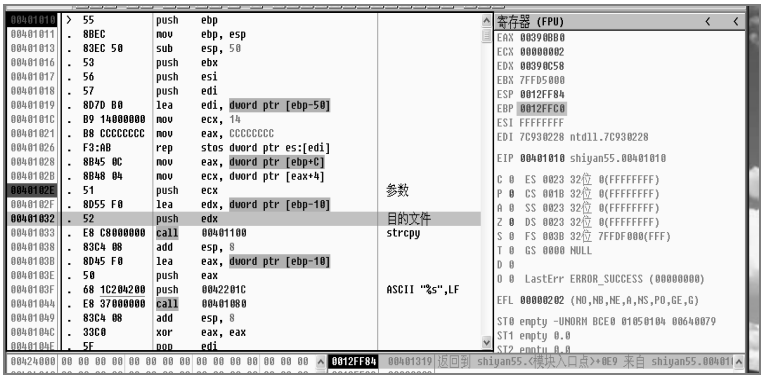

图 4.4　程序停在 main 函数入口点

3．观察参数入栈

使程序单步运行到 strcpy 函数之前，观察栈内变化：首先压入返回地址和原 EBP 值，之后留出 0x50 B 大小的局部变量空间并进行初始化（内容初始化为 0x0C），再压入 EBX、ESI、EDI 三个寄存器的值，之后将输入参数地址和目的文件地址压入栈中，如图 4.5 所示。

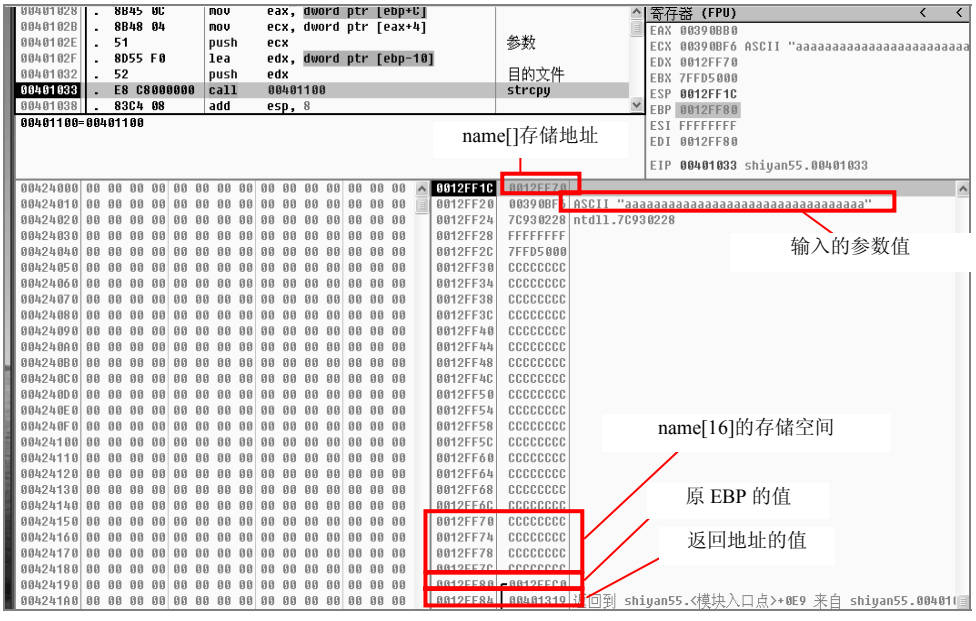

图 4.5　栈中参数的分布

4．观察缓冲区

通过汇编指令可知 EDX 指向 name[16]的起始地址 0x0012ff70，该缓冲区范围从 0x0012ff70～0x0012ff7c 共 16B，即为 name[16]分配的空间。该起始地址也是之后 strcpy 函数的第一个参数，即目的缓冲区地址。需要注意的是，栈中紧挨着 name[16]的是原 EBP

的值（0x0012ffc0）和原 EIP 的值（0x00401319），即当前函数的返回地址。

5．跟踪 strcpy 函数

单步步过 strcpy 函数，观察栈内变化，如图 4.6 所示。

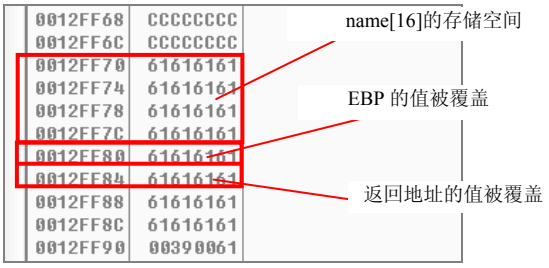

图 4.6　发生栈溢出

可见 name 的空间即 0x0012ff70～0x0012ff7c 都被复制了“a”，但是由于源字符串长度过长，导致顺着内存生长方向继续复制“a”，最终原 EBP 的值和返回地址都被“a”覆盖，造成了缓冲区溢出。

4.3.4　实验要求

使用 OllyDbg 跟踪栈溢出的全过程，并画出其过程中栈的变化图。

4.4　整型溢出实验

4.4.1　实验目的

本实验要求掌握整型溢出的原理，了解宽度溢出和符号溢出的发生过程。

4.4.2　实验内容及环境

1．实验内容

本实验使用 VC 6.0 的源码调试功能，尝试不同的程序输入，并跟踪变量和内存的变化，以观察不同整型溢出的原理。

2．实验环境

（1）靶机系统环境为 Windows XP SP3 32 位；

（2）VC 6.0，具体详见本书 4.3 节的介绍。

4.4.3　实验步骤

1．编译代码

通过 VC 6.0 将以下两段代码分别编译成 debug 版的 t1.exe 和 t2.exe。

```
//整型宽度溢出
```

```
1    intmain(intargc, char *argv[]){
2        unsigned short s;
3        inti;
4        char buf[10];
5        i = atoi(argv[1]);
6        s = i;
7        if(s >= 80){
8            printf("错误！输入不能超过 10！\n");
9            return -1;
10       }
11       memcpy(buf, argv[2], i);
12       buf[i] = '\0';
13       printf("%s\n", buf);
14       return 0;
15   }
```

//整型符号溢出

```
1    int    main(intargc, char *argv[]){
2        char kbuf[800];
3        int size = sizeof(kbuf);
4        intlen = atoi(argv[1]);
5        if(len> size){
6            printf("错误！输入不能超过 800！\n");
7            return 0;
8        }
9        memcpy(kbuf, argv[2], len);
10   }
```

2．加载程序

使用 VC 6.0 调试 t1.exe，在程序参数栏填入“100aaaaaaaaaaaaaaa”，如图 4.7 所示，按“Ctrl”+“F5”组合键运行。

图 4.7　参数设置

3．检查参数

由于参数 i 的值大于 10，不能通过第 7 行的条件判断，程序运行显示“错误！输入不能超过 10!”后退出。

4．修改参数

修改参数 i 的值为 65537，并在第 6 行设置一个断点，按“F5”键运行。

5．观察运行环境

程序停在断点处，观察 VC 6.0 程序运行的上下文窗口，注意此时“i=0x00010001 (65537)”，如图 4.8 所示。

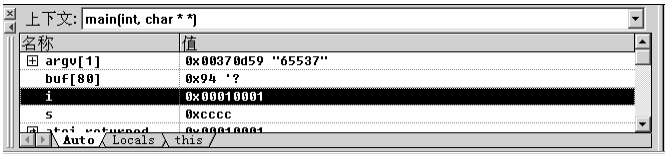

图 4.8　参数 i 的值

6．宽度溢出

按“F10”键单步运行，注意“s=0x0001”，此时“i”的高位被截断了，发生了整型宽度溢出，如图 4.9 所示。

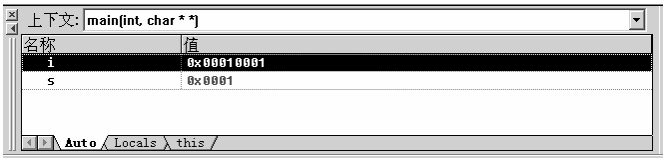

图 4.9　发生整型宽度溢出

7．缓冲区溢出

由于 s 的值小于 10，通过了第 7 行的条件判断，进入到第 11 行的 memcpy 函数。而复制的长度 i=65537 又远大于 buf 缓冲区的值 10，导致缓冲区溢出，所以程序提示出错，如图 4.10 所示。

图 4.10　程序提示出错

8．加载程序

调试 t2.exe，在程序参数栏填入“1000aaaaaaaaaaaaaaa”，按“Ctrl”+“F5”组合键运行。

9．检查参数

由于此时参数 i 的值为 1000，大于限定 size=800，所以不能通过第 5 行的条件判断，程序提示：“错误！输入不能超过 800!”后退出。

10．修改参数

修改参数 i 的值为–1，在第 5 行设置断点，按“F5”键运行。

11．观察运行环境

程序停在断点处，观察 VC 6.0 程序运行的上下文窗口，注意此时 len=0xffffffff（即为–1），而 size=0x00000320，如图 4.11 所示。

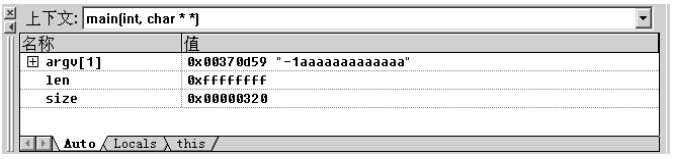

图 4.11　参数 len 的值

12．符号溢出

由于 len 的定义是有符号数 int，所以此时 len=–1，小于 size 的值，通过第 5 行条件判断，执行 memcpy 函数。但是 memcpy 函数的第三个参数定义为无符号数的 size_t，因而会将 len 作为无符号数对待，由此发生整型符号溢出错误。此时 len=0xffffffff（即 4294967295），远大于目的缓冲区 kbuf 的值 800，继续运行会发生错误。

4.4.4　实验要求

使用 VC 6.0 程序跟踪发生宽度溢出和符号溢出的全过程，并给出过程中相关参数和内存的变化情况。

4.5　UAF 类型缓冲区溢出实验

4.5.1　实验目的

本实验要求掌握 UAF 类型缓冲区溢出的原理，了解 UAF 类型缓冲区溢出的发生过程。

4.5.2　实验内容及环境

1．实验内容

本实验使用 VC 6.0 的源码调试功能，观察内存块 p1 的创建和释放过程，并观察内存块 p1 释放后再次使用的情况，以了解 UAF 类型缓冲区溢出的原理。

2．实验环境

（1）靶机系统环境为 Windows XP SP3 32 位；

（2）VC 6.0：具体详见本书 4.3 节实验工具介绍。

4.5.3　实验步骤

1．编译代码

通过 VC 6.0 将以下代码分别编译成 debug 版的.exe 文件。

```
1   typedef VOID (WINAPI *MYFUNC)();
2   void WINAPI myfunc()
3   {
4       printf("this is func\n");
5   }
6   typedefstructmyclass{
7       intlen;
8       char str[12];
9       MYFUNC func;
10  }MYCLASS;
11
12  intmain(intargc, char* argv[])
13  {
14      MYCLASS *p1 = (MYCLASS*)malloc(sizeof(MYCLASS));
15      p1->func = myfunc;
16      p1->func();
17      free(p1);
18      char *p2 = (char*)malloc(100);
19      strcpy(p2,argv[1]);
20      p1->func();
21      return ();
22  }
```

2．观察内存块 p1

设置程序参数为“aaaaaaaaaaaaaaaaaaaa”，调试源代码，在第 15 行设置断点，按“F5”键执行到断点处。观察内存块 p1 已被分配了内存地址 0x003707b8（地址可能会有变化），此时 p1->func 的值仍然为未初始化的“0xcdcdcdcd”，如图 4.12 所示。

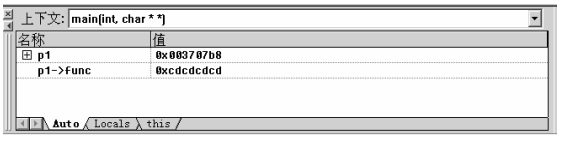

图 4.12　内存块 p1 的地址

3．观察函数地址

单步执行 1 次，p1->func 的值为 myfunc 函数的地址 0x00401005，如图 4.13 所示。

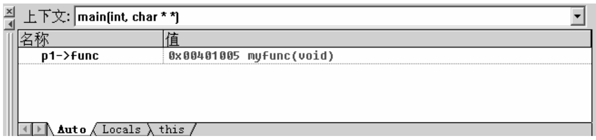

图 4.13　p1->func 的值

4．释放 p1

继续单步执行到第 18 行，注意到虽然内存块 p1 的地址已被释放，但仍指向地址 0x003707b8，同时其数据已被 0xfeeefeee 覆盖，如图 4.14 所示。

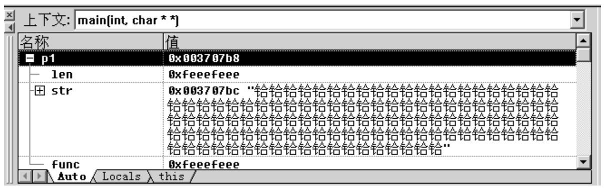

图 4.14　内存块 p1 的地址被释放

5．观察内存块 p2

单步执行 1 次，内存块 p2 的地址已被分配，注意其地址 0x003707b8 与内存块 p1 的地址一样，如图 4.15 所示。

图 4.15　内存块 p2 的地址与内存块 p1 的地址相同

6．破坏内存

单步执行 1 次，字符串“aaaaaaaaaaaaaaaaaaaa”被复制到内存块 p2 所指向的内存中，同时 p1->func 的地址也被修改为 0x61616161，如图 4.16 所示。

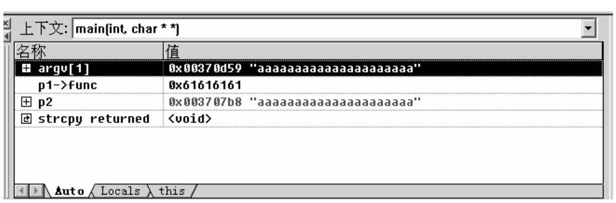

图 4.16　p1->func 的地址被修改

7．程序出错

再次单步执行，由于内存块 p2 与内存块 p1 的地址指向同一块内存，且该内存的内容已被修改，当调用已释放的 func 函数时，程序出错崩溃，指令地址指向 0x61616161。可见发生 UAF 类型缓冲区溢出漏洞有 3 个条件：

（1）旧对象被释放；

（2）申请的新对象恰好能覆盖到旧对象区域；

（3）使用旧对象。

4.5.4　实验要求

使用 VC 6.0 跟踪 UAF 类型溢出的全过程，并给出过程中相关参数和内存的变化情况。

4.6　覆盖返回地址实验

4.6.1　实验目的

本实验要求了解通过覆盖返回地址进行缓冲区溢出利用的原理，掌握覆盖返回地址的过程。

4.6.2　实验内容及环境

1．实验内容

本实验在栈溢出实验的基础上，通过观察返回地址被覆盖后的后续流程，了解和掌握通过覆盖返回地址进行缓冲区溢出利用的技术。

2．实验环境

（1）靶机系统环境为 Windows XP SP3 32 位；

（2）OllyDbg：具体详见本书 4.3 节实验工具介绍；

（3）VC 6.0：具体详见本书 4.3 节实验工具介绍。

4.6.3　实验步骤

1．重复实验 4.3

完成实验 4.3，使发生栈溢出。

2．观察 retn 指令

继续单步执行到程序中心的 retn 指令，如图 4.17 所示。

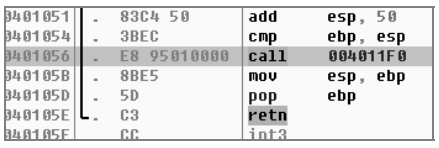

图 4.17　执行到程序中 retn 指令

此时栈顶寄存器（ESP）的值指向地址 0x0012ff84，即返回地址。retn 指令的内部操作过程如下：

（1）将栈顶数据值取出，赋给 EIP 寄存器；

（2）跳转至 EIP 寄存器地址指向的指令继续执行。

因此，retn 指令将 0x0012ff84 地址的值 0x61616161 赋给 EIP 寄存器地址，之后运行 0x61616161 地址处的指令。

3．程序出错

由于 0x61616161 地址的内容不可读，导致访问错误，程序崩溃，如图 4.18 所示。

图 4.18　程序出错

4.6.4　实验要求

使用 OllyDbg 跟踪栈溢出后的流程，并说明程序出错的原因。

4.7　覆盖函数指针实验

4.7.1　实验目的

本实验要求了解通过覆盖函数指针进行缓冲区溢出利用的原理，掌握覆盖函数指针的过程。

4.7.2　实验内容及环境

1．实验内容

本实验使用 VC 6.0 的源代码调试功能，跟踪函数指针 myfunc()的变化，了解和掌握通过覆盖函数指针进行缓冲区溢出利用的技术。

2．实验环境

（1）靶机系统环境为 Windows XP SP3 32 位；

（2）VC 6.0：具体详见本书 4.3 节实验工具介绍。

4.7.3　实验步骤

1．编译代码

通过 VC 6.0 将以下代码编译成 debug 版的.exe 文件。

```
1    typedef VOID (WINAPI* FUNC)(void);
2    void func()
3    {
4        printf("this is func\n");
5    }
6    intmain(intargc, char* argv[])
7    {
8        FUNC myfunc = (FUNC)func;
9        printf("myfunc is store at %08x\n",&myfunc);
10       myfunc();
11       char name[16];
12       strcpy(name,argv[1]);
13       myfunc();
14       return ();
```

2．加载程序

调试.exe 程序，在程序参数栏输入“bbbbbbbbbbbbbbbbbbbbbbbbbbbb”，在第 10 行设置断点，按“F5”键运行到断点处，观察此时 myfunc 已被赋值为 0x00401005，且其被保存在 0x0012ff7c 地址处，如图 4.19 所示。

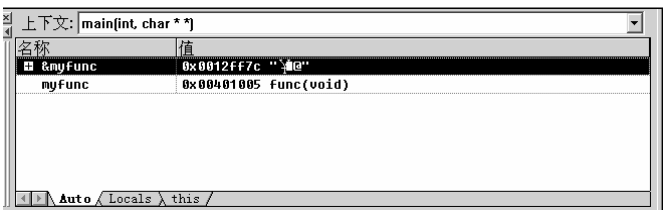

图 4.19　myfunc 的地址和值

3．观察函数地址

单步执行到第 12 行，发现 name[16]数组被保存在地址 0x0012ff6c 处，在 myfunc()函数地址 0x0012ff7c 的低地址处，距离相差 16B，如图 4.20 所示。

图 4.20　name[16]的地址

4．覆盖函数地址

单步执行 1 次，由于 strcpy 语句导致了 name[16]数组的缓冲区溢出，因而往后覆盖 myfunc()的值为 0x62626262，如图 4.21 所示。

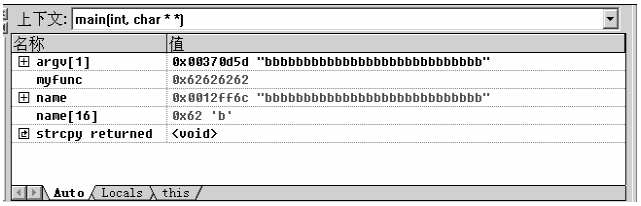

图 4.21　myfunc 的值被覆盖

5．程序出错

再次单步执行 myfunc()函数时发生错误，指令地址为 0x62626262，这与覆盖返回地址类似，如果要控制程序的流程，需要将 myfunc()函数指针修改为某个跳转地址。在这个例子中，程序出错时按“ALT”+“5”组合键调出寄存器窗口，可以观察到此时 EAX=0x0012ff6c，正好指向 name[16]的地址，因此可以将 myfunc 的值改为 jmp eax 指令所在的地址。覆盖函数指针完成跳转如图 4.22 所示。

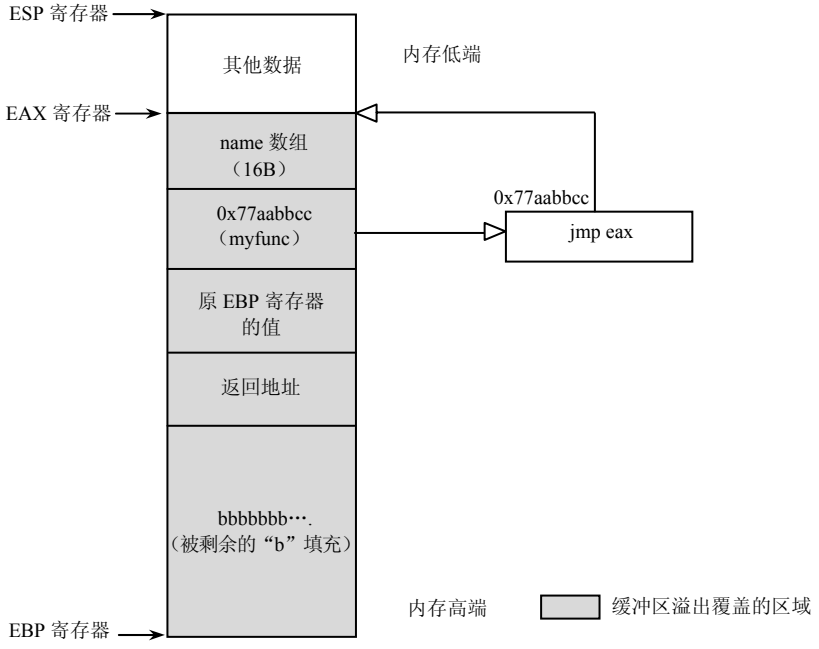

图 4.22　覆盖函数指针完成跳转

4.7.4　实验要求

使用 VC 6.0 跟踪覆盖函数地址的全过程，并给出过程中相关参数和内存的变化情况，