

第四章 软件漏洞

知识点一：溢出漏洞基本概念

知识点二：栈溢出漏洞

知识点三：堆溢出漏洞

知识点四：其它溢出漏洞

知识点一：溢出漏洞基本概念

(1) 漏洞 (Vulnerability)

漏洞也称为脆弱性(Vulnerability)，是计算机系统的硬件、软件、协议在系统设计、具体实现、系统配置或安全策略上存在的缺陷。这些缺陷一旦被发现并被恶意利用，就会使攻击者在未授权的情况下访问或破坏系统，从而影响计算机系统的正常运行甚至造成安全损害。

- 对于漏洞有多种称呼，包括Hole, Error, Fault, Weakness, Failure等，这些称呼都不能涵盖漏洞的含义（脆弱性）。
- 软件漏洞专指计算机系统软件系统漏洞。

(2) 缓冲区溢出漏洞

缓冲区

缓冲区是一块连续的内存区域，用于存放程序运行时加载到内存的运行代码和数据。

缓冲区溢出

缓冲区溢出是指程序运行时，向固定大小的缓冲区写入超过其容量的数据，多余的数据会越过缓冲区的边界覆盖相邻内存空间，从而造成溢出。

缓冲区的大小是由用户输入的数据决定的，如果程序不对用户输入的超长数据作长度检查，同时用户又对程序进行了非法操作或者错误输入，就会造成缓冲区溢出。



缓冲区 溢出攻 击

缓冲区溢出攻击是指发生缓冲区溢出时，溢出的数据会覆盖相邻内存空间的返回地址、函数指针、堆管理结构等合法数据，从而使程序运行失败、或者发生转向去执行其它程序代码、或者执行预先注入到内存缓冲区中的代码。

缓冲区溢出后执行的代码，会以原有程序的身份权限运行。





造成缓冲区溢出的根本原因

是缺乏类型安全功能的程序设计语言（C、C++等）

出于效率的考虑，部分函数不对数组边界条件和函数指针引用等进行边界检查。例如，C 标准库中和字符串操作有关的函数，像strcpy, strcat, sprintf, gets等函数中，数组和指针都没有自动边界检查。

程序员开发时必须自己进行边界检查，防范数据溢出，否则所开发的程序就存在缓冲区溢出的安全隐患，而实际上这一行为往往被程序员忽略或者检查不充分。

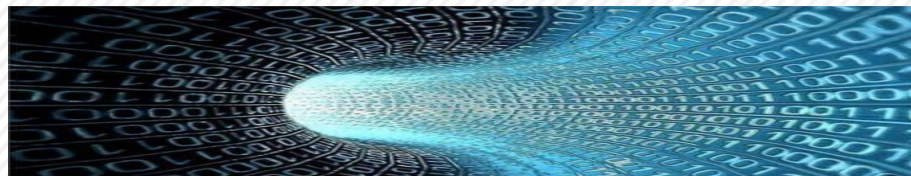
缓冲区溢出通常包括栈溢出、堆溢出、异常处理SEH结构溢出、单字节溢出等

知识点二：栈溢出漏洞

1. 基本概念

栈溢出漏洞：

栈溢出漏洞，即发生在栈区的溢出漏洞。被调用的子函数中写入数据的长度，**大于栈帧的基址到esp之间预留的保存局部变量的空间**时，就会发生栈的溢出。要写入数据的填充方向是**从低地址向高地址增长**，多余的数据就会越过栈帧的基址，覆盖基址以上的地址空间。



下面的程序演示了一个溢出漏洞，代码如下

```
void why_here(void)
{   printf("why u r here?!\\n");
    exit(0);
}

void f()
{   int buff[1];
    buff[2] = (int)why_here;
}

int main(int argc, char * argv[])
{   f();
    return 0;
}
```

如程序所示，主函数将调用函数f，并没有调用why_here函数，但是运行结果如下：



```
C:\Windows\system32\cmd.exe  
why u r here?!  
请按任意键继续. . .
```

在函数f中，所声明的数组buff长度为1，但是由于没有对访问下标的值进行校验，程序中对数组外的内存进行了读写，这是一个典型的溢出漏洞。

```
void f()
{
    int buff[1];
    buff[2] = (int)why_here;
}
```



BUFF



| |
|---------|
| |
| Buff[0] |
| EBP |
| 返回地址 |

```
void why_here(void)
{ printf("why u r here?!\\n");
  exit(0);
}
```

```
void f()
{ int buff; int * p = &buff;
  _____ = (int)why_here;
}
```

```
int main(int argc, char * argv[])
{ f();
  return 0;
}
```

答案: $*(p+2)$ 或者 $p[2]$

p →

| |
|------|
| p |
| buff |
| EBP |
| 返回地址 |

3. 溢出漏洞利用示例

a

修改返回地址

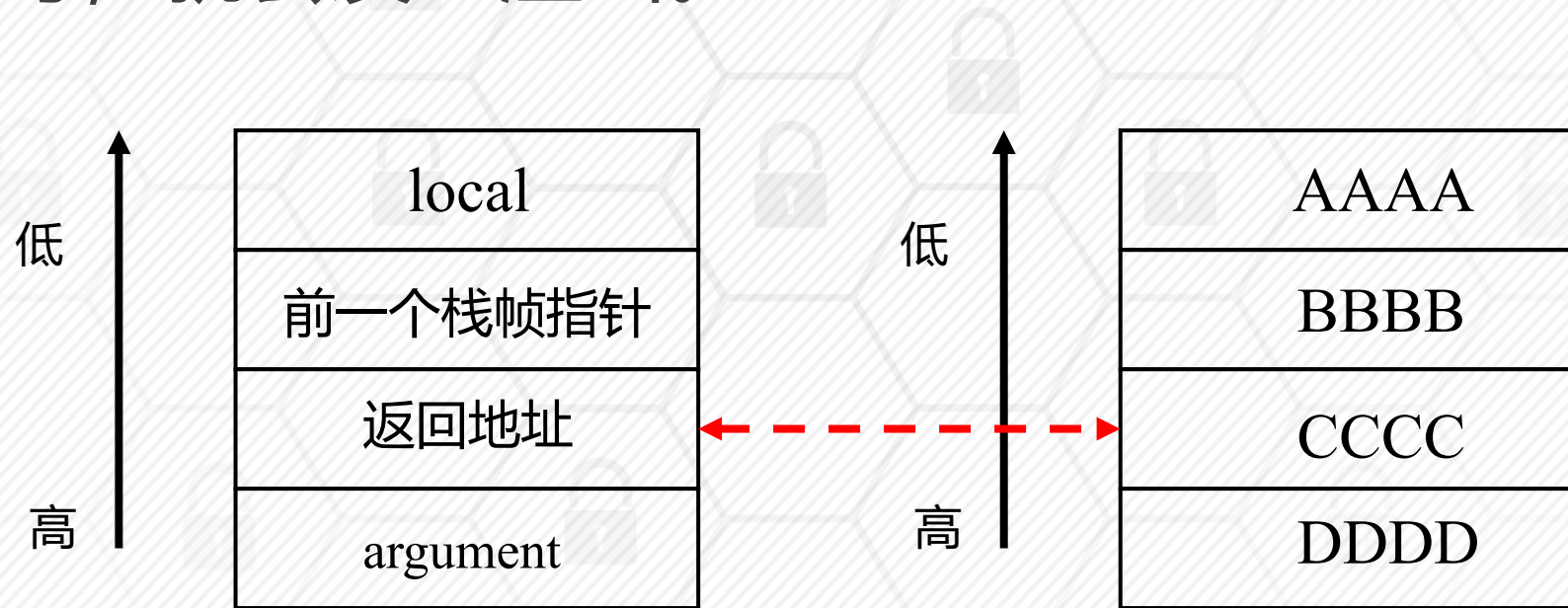
栈的存取采用先进后出的策略，程序用它来保存函数调用时的有关信息，如函数参数、返回地址，函数中的非静态局部变量存放在栈中。**如果返回地址被覆盖，当覆盖后的地址是一个无效地址，则程序运行失败。如果覆盖返回地址的是恶意程序的入口地址，则源程序将转向去执行恶意程序。**

下面以一段程序为例说明栈溢出的原理。

```
void stack_overflow(char* argument)
{
    char local[4];
    for(int i = 0; argument[i];i++)
        local[i] = argument[i];
}
```



函数stack_overflow被调用时堆栈布局如下图所示。图中local是栈中保存局部变量的缓冲区，根据char local[4]预先分配的大小为4个字节，当向local中写入超过4个字节的字符时，就会发生溢出。



如果CCCC地址为攻击代码的入口地址，就会调用攻击代码

b 覆盖临接变量

在第三章，我们**通过修改机器码实现了软件破解**，接下来我们通过在输入上做文章（也是漏洞利用方式），试着**覆盖临近变量的值，以便更改程序执行流程**。

函数的局部变量在栈中一个挨着一个排列。如果这些局部变量中有数组之类的缓冲区，并且程序中存在数组越界的缺陷，那么越界的数组元素就有可能破坏栈中相邻变量的值，甚至破坏栈帧中所保存的EBP值、返回地址等重要数据。

用一个简单例子来说明破坏栈内局部变量对程序的安全性有什么影响（VC6）。

```
void 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;
        }
    }
}
```

```
#include <stdio.h>
#include <iostream>
#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);
    return authenticated;
}
```

观察一下源代码不难发现，authenticated变量的值来源于strcmp函数的返回值，之后会返回给main函数作为密码验证成功与否的标志变量：当authenticated为0时，表示验证成功；反之，验证不成功。

如果我们输入的密码超过了7个字符（注意：字符串截断符NULL将占用一个字节），则越界字符的ASCII码会修改掉authenticated的值。如果这段溢出数据恰好把authenticated改为0，则程序流程将被改变。要成功覆盖临近变量并使其为0，有两个条件：

1

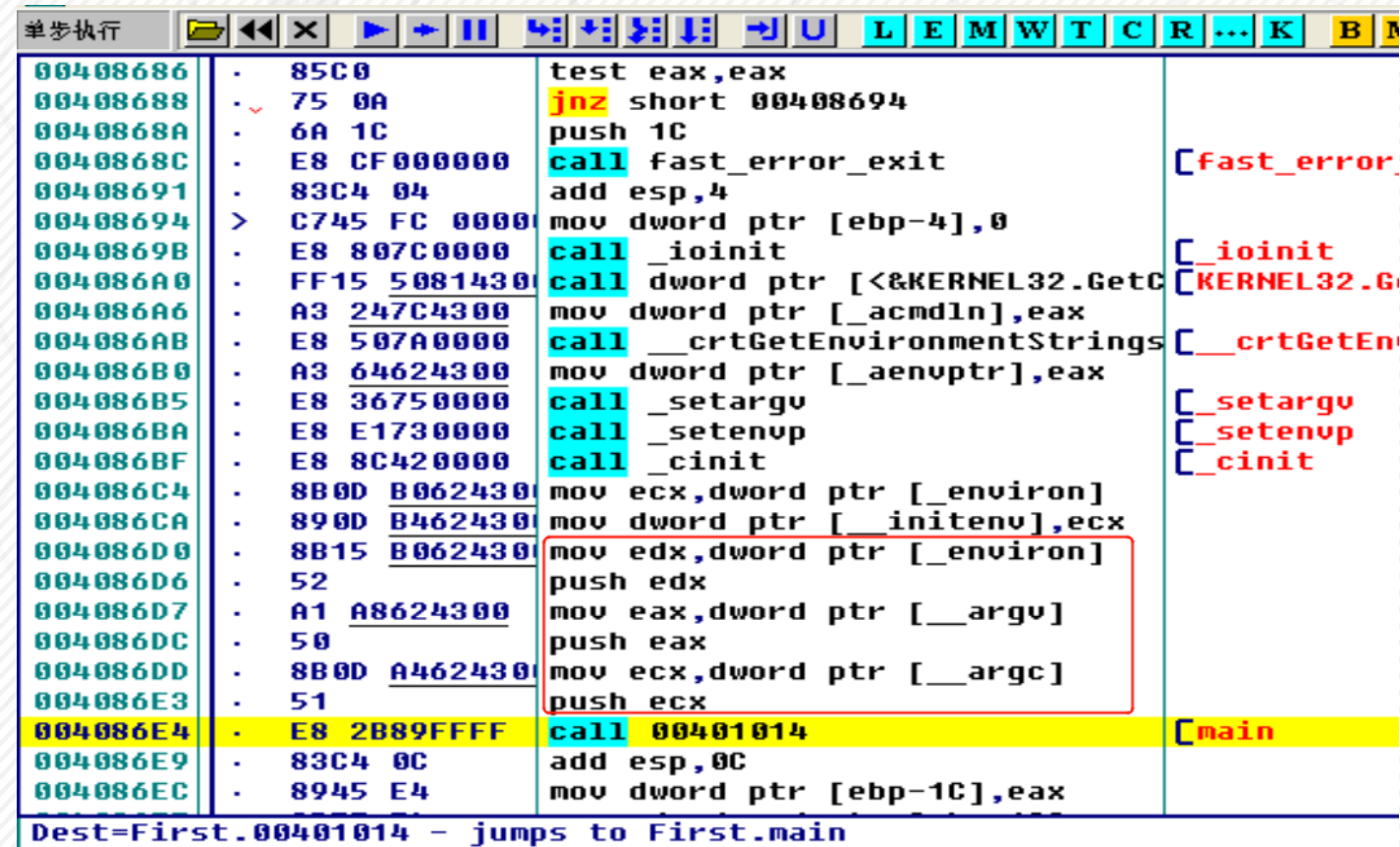
输入一个8位的字符串的时候，比如“22334455”，此时，字符串的结束符恰恰是0，则覆盖变量authenticated的高字节并使其为0；

2

输入的字符串应该大于“12345678”，因为执行strcmp之后要确保变量authenticated的值为1，也就是只有高字节是1，其它字节为0。

实验细节

打开OlllyDBG，装载程序后，会停在程序入口点，单步执行可以定位到主函数：第一，主函数通过OlllyDBG的信息提示区域，会显示main函数信息；第二，Windows控制台程序的主函数参数包含三个，即_argv、_argv和_environ，在函数调用前面的参数入栈环节具有鲜明的特征，截图如下：



| Address | Disassembly | Comment |
|----------|---------------------------------|-------------------|
| 00408686 | test eax, eax | |
| 00408688 | jnz short 00408694 | |
| 0040868A | push 1C | |
| 0040868C | call fast_error_exit | [fast_error_exit] |
| 00408691 | add esp, 4 | |
| 00408694 | mov dword ptr [ebp-4], 0 | |
| 0040869B | call _ioutil | [_ioutil] |
| 004086A0 | call dword ptr [&KERNEL32.GetC | [KERNEL32.GetC] |
| 004086A6 | mov dword ptr [_acmdln], eax | |
| 004086AB | call __crtGetEnvironmentStrings | [__crtGetEn] |
| 004086B0 | mov dword ptr [_aenvptr], eax | |
| 004086B5 | call _setargv | [_setargv] |
| 004086BA | call _setenv | [_setenv] |
| 004086BF | call _cinit | [_cinit] |
| 004086C4 | mov ecx, dword ptr [_environ] | |
| 004086CA | mov dword ptr [__initenv], ecx | |
| 004086D0 | mov edx, dword ptr [_environ] | |
| 004086D6 | push edx | |
| 004086D7 | mov eax, dword ptr [__argv] | |
| 004086DC | push eax | |
| 004086DD | mov ecx, dword ptr [__argc] | |
| 004086E3 | push ecx | |
| 004086E4 | call 00401014 | [main] |
| 004086E9 | add esp, 0C | |
| 004086EC | mov dword ptr [ebp-1C], eax | |

Dest=First.00401014 - jumps to First.main

实验细节

此时，选择步入执行即可转到主函数。之后继续一步步执行程序，会遇到Scanf函数，弹出对话框，接受用户输入，我们输入“22334455”，然后会回到原来程序，继续单步运行，直到调用verify_password函数后，进入该函数代码区域。

| 地址 | 十六进制 | 汇编 | 注释 |
|----------|-------------|----------------------------|---------|
| 0040124F | CC | int3 | |
| 00401250 | 55 | push ebp | 栈帧初始化 |
| 00401251 | 8BEC | mov ebp, esp | |
| 00401253 | 83EC 4C | sub esp, 4C | |
| 00401256 | 53 | push ebx | |
| 00401257 | 56 | push esi | |
| 00401258 | 57 | push edi | |
| 00401259 | 8D7D B4 | lea edi, [ebp-4C] | |
| 0040125C | B9 13000000 | mov ecx, 13 | |
| 00401261 | B8 CCCCCCCC | mov eax, CCCCCCCC | |
| 00401266 | F3:AB | rep stos dword ptr [edi] | |
| 00401268 | 68 1C104300 | push offset 0043101C | |
| 0040126D | 8B45 08 | mov eax, dword ptr [ebp+8] | |
| 00401270 | 50 | push eax | |
| 00401271 | E8 1A720000 | call strcmp | 口令比较 |
| 00401276 | 83C4 08 | add esp, 8 | |
| 00401279 | 8945 FC | mov dword ptr [ebp-4], eax | |
| 0040127C | 8B4D 08 | mov ecx, dword ptr [ebp+8] | |
| 0040127F | 51 | push ecx | |
| 00401280 | 8D55 F4 | lea edx, [ebp-0C] | |
| 00401283 | 52 | push edx | |
| 00401284 | E8 17710000 | call strcpy | 溢出覆盖 |
| 00401289 | 83C4 08 | add esp, 8 | |
| 0040128C | 8B45 FC | mov eax, dword ptr [ebp-4] | 函数返回值处理 |
| 0040128F | 5F | pop edi | |
| 00401290 | 5E | pop esi | |
| 00401291 | 5B | pop ebx | |
| 00401292 | 83C4 4C | add esp, 4C | |

Imm=8

实验细节

在执行完口令比较后，运行完`mov dword part [ebp-4], eax`语句，该语句含义为将EAX寄存器的值（刚执行的`strcmp`函数的返回值）复制给地址`ebp-4`的局部变量。也就是，将口令比较的结果复制给我们定义的局部变量`authenticated`。

通过寄存器窗口，可知当前EBP寄存器值为0x0012FB24，观察此时栈区变化，观察此时`ebp-4`地址处的变量值，同时，我们将数据窗口定位到地址0x0012FB20处（数据窗口区域，点右键，选择“转到□表达式”，出现表达式后，输入0x0012FB20或EBP-4**，然后选择“跟随表达式”），来观察后续的变化，如下：**

| 地址 | 十六进制数据 | | | | | | | | | | | | | | | | ASCII | | |
|----------|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----------|--|--|
| 0012FB10 | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CCCCCCCC | | |
| 0012FB20 | 01 | 00 | 00 | 00 | 80 | FF | 12 | 00 | 1B | 13 | 40 | 00 | 7C | FB | 12 | 00 | 00000001 | | |
| 0012FB30 | FE | FF | FF | FF | 15 | 00 | 00 | 00 | 00 | E0 | FD | 7F | CC | CC | CC | CC | 0040131B | | |
| 0012FB40 | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | 0012FB7C | | |
| 0012FB50 | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | FFFFFFFFE | | |
| 0012FB60 | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | 00000015 | | |
| 0012FB70 | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | CC | 32 | 32 | 33 | 33 | 7FFDE000 | | |

当程序执行到`strcpy`溢出覆盖后的“`mov eax, dword ptr [ebp-4]`”之前，我们可以观察到，溢出成功的覆盖了变量`authenticated`的值为0x00000000。

知识点三：堆溢出漏洞

1. 简单示例

堆溢出漏洞：

堆溢出是指在堆中发生的缓冲区溢出。堆溢出后，数据可以覆盖堆区的不同堆块的数据，带来安全威胁。我们将通过下面一个简单例子，来演示一个简单的堆溢出漏洞：**该漏洞在产生溢出的时候，将覆盖一个目标堆块的块身数据。**

示例：从堆区申请两个堆块，处于低地址的buf1和处于高地址的buf2。buf2存储了一个名为myoutfile 的字符串，用来存储文件名。buf1用来接收输入，同时将这些输入字符在程序执行过程中写入到buf2 存储的文件名myoutfile 所指向的文件中。

```

#define FILENAME "myoutfile"
int main(int argc,char *argv[])
{
    FILE *fd;
    long diff;
    char bufchar[100];

    char* buf1 = (char*)malloc(20);
    char* buf2 = (char*)malloc(20);

    diff = (long)buf2-(long)buf1;

    strcpy(buf2,FILENAME);

    printf("buf1存储地址:%p\n",buf1);
    printf("buf2存储地址:%p,存储内容为文件名:%s\n",buf2,buf2);
    printf("两个地址之间的距离:%d个字节\n",diff);
    if(argc<2){
        printf("请输入要写入文件名\n");
        gets(bufchar);
        strcpy(buf1,bufchar);
    }

```

很明显，往buf1复制，没有边界检查

```

else
    strcpy(buf1,argv[1]);

    printf("buf1存储内容:%s\n",buf1);
    printf("buf2存储内容:%s\n",buf2);
    printf("将%s\n写入文件 %s中\n\n",buf1,buf2);

    fd=fopen(buf2,"a");
    if(fd==NULL){
        fprintf(stderr,"%s 打开错误\n",buf2);
        if(diff<=strlen(bufchar))
            printf("提示:buf1内存溢出!\n");
        getchar();
        exit(1);
    }
    fprintf(fd,"%s\n\n",buf1);
    fclose(fd);
    if(diff<=strlen(bufchar)){
        printf("提示:buf1已溢出，溢出部分覆盖buf2中的myoutfile\n");
        getchar();
        return 0;
    }

```

很明显，往buf1复制，没有边界检查

溢出后，导致buf2可能变成设计的目标文件，而非原始文件

```
#define FILENAME "myoutfile"
int main(int argc,char *argv[])
{
FILE *fd;
long diff;
char bufchar[100];

char* buf1 = (char*)malloc(20);
char* buf2 = (char*)malloc(20);

diff = (long)buf2-(long)buf1;

strcpy(buf2,FILENAME);
```

```
printf("buf1存储地址:%p\n",buf1);
printf("buf2存储地址:%p,存储内容为文件名:%s\n",buf2,buf2);
printf("两个地址之间的距离:%d个字节\n",diff);
if(argc<2){
    printf("请输入要写入文件%s的字符串:\n",buf2);
    gets(bufchar);
    strcpy(buf1,bufchar);
}else
    strcpy(buf1,argv[1]);
```



```
C:\Documents and Settings\Administrator\桌面\漏洞\堆溢
----信息显示----
buf1存储地址:00360910
buf2存储地址:00360958,存储内容为文件名:myoutfile
两个地址之间的距离:72个字节
----信息显示----

请输入要写入文件myoutfile的字符串:
-
```

- ✓ 通过malloc命令，申请了两个堆的存储空间。接着定义了diff变量，它记录了buf1和buf2之间的地址距离，也就是说buf1和buf2之间还有多少存储空间。
- ✓ **输入字符串的长度为大于72个字节**，而且刻意构造一个自定义的字符串“hostility”，是输入为**“72字节填充数据” + “hostility”**。可见buf1的内容长度是超过了72个字节的，而buf2的内容就变成了hostility。

2. 堆溢出漏洞利用

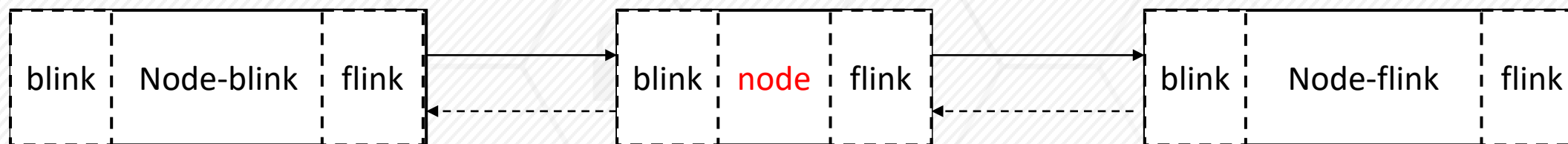
相比于栈溢出，**堆溢出的实现难度更大**，而且往往要求进程在内存中具备特定的组织结构。然而，堆溢出攻击也已经成为缓冲区溢出攻击的主要方式之一。**堆溢出带来的威胁远远不止上面示例演示的那样，结合堆管理结构，堆溢出漏洞可以在任意位置写入任意数据。**

回顾一下第二章介绍的堆管理结构：**在Windows系统中，占有态的堆块被使用它的程序索引，而堆表只索引所有空闲态的堆块。**其中，最重要的堆表有两种：**空闲双向链表freelist（简称空表）和快速单向链表lookaside（简称快表）。**

堆块三类操作：堆块分配、堆块释放和堆块合并，归根结底是对空表链的修改。
这些修改无外乎要向链表里链入和卸下堆块。根据对链表操作的常识，我们可以知道，**从链表上卸载（unlink）一个节点的时候会发生如下操作：**

$\text{node} \rightarrow \text{blink} \rightarrow \text{flink} = \text{node} \rightarrow \text{flink} ;$

$\text{node} \rightarrow \text{flink} \rightarrow \text{blink} = \text{node} \rightarrow \text{blink} ;$



具体的，在Windows堆内存分配时会调用函数RtlAllocHeap，该函数从空闲堆链上摘下一**空闲堆块**，完成双向链表里相关节点的前后指针的变更操作，它会执行如下操作：

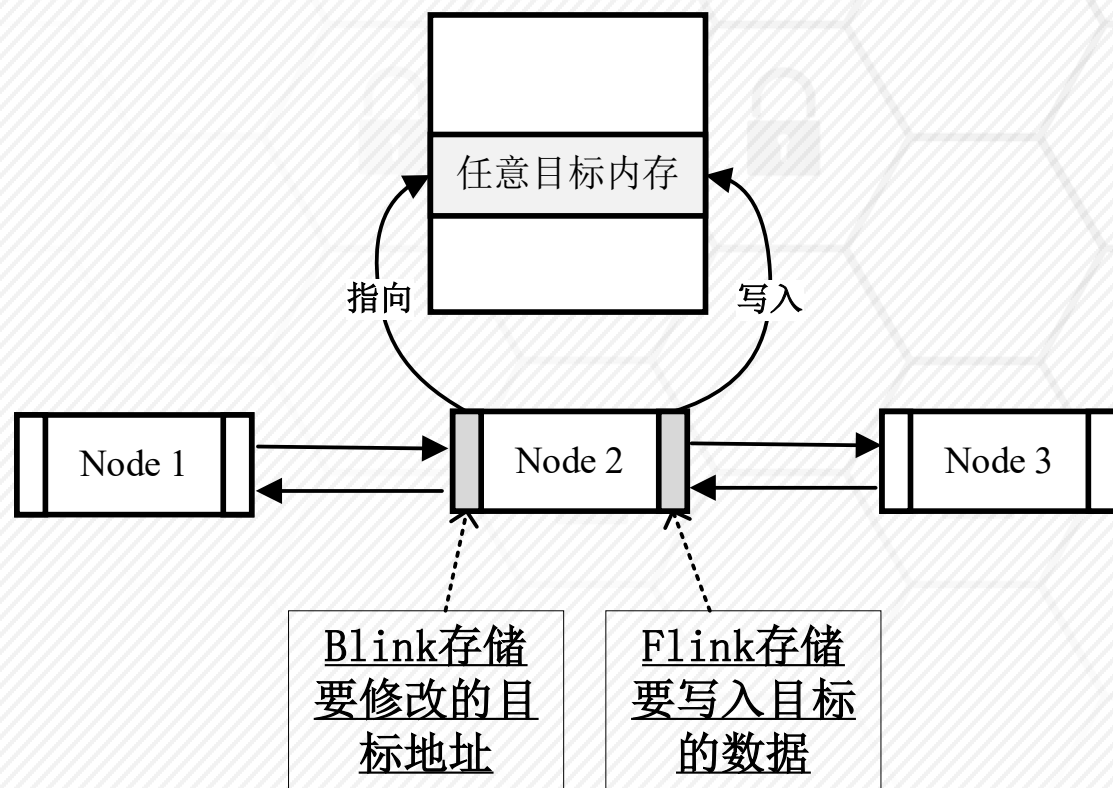
```
mov dword ptr [edi], ecx ;  
mov dword ptr [ecx+4], edi ;
```

其中**ecx**为空闲可分配的堆区块的前向指针，**edi**为该堆区块的后向指针。这两条汇编语句恰好对应了上述两个链表卸载节点对应的前后向指针变化的操作。

空闲堆块的前向指针（数值）写入到
空闲堆块的后向指针（地址）里去

Dword Shoot攻击

如果我们通过堆溢出覆写了一个**空闲堆块**的块首的前向指针flink和后向指针blink，我们可以精心构造一个地址和一个数据，当这个空闲堆块从链表里卸下的时候，就获得一次向内存构造的任意地址写入一个任意数据的机会。这种**能够向内存任意位置写入任意数据的机会称为“Arbitrary Dword Reset”（又称Dword Shoot）**。具体如下图所示。



基于Dword Shoot攻击，攻击者甚至可以劫持进程，运行植入的恶意代码。比如，当构造的地址为重要函数调用地址、栈帧中函数返回地址、栈帧中SEH的句柄等时，写入的任意数据可能就是恶意代码的入口地址。

堆溢出漏洞示例：以下列程序为例，演示堆块分配过程中潜在的Dword Shoot攻击。

实验环境：VC6.0、Windows XP SP3、Debug模式。

在讲这个实验之前，先介绍一下Windows的堆使用。

在Windows里，可以使用Windows缺省堆，也可以用户自己创建新堆：

- 获取缺省堆可以通过GetProcessHeap函数（无参数）得到句柄；
- 创建新堆可以用HeapCreat函数。
- 除了malloc、new等函数外，C/C++也提供了HeapAlloc、HeapFree等函数用于堆的分配和释放。

```
#include <windows.h>
main()
{
    HLOCAL h1, h2, h3, h4, h5, h6;
    HANDLE hp;
    hp = HeapCreate(0, 0x1000, 0x10000); //创建自主管理的堆
    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8); //从堆里申请空间
    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);
    h3 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);
    h4 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);
    h5 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);
    h6 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);

    _asm int 3 //手动增加int3中断指令，会让调试器在此处中断
    //依次释放奇数堆块，避免堆块合并
    HeapFree(hp, 0, h1); //释放堆块
    HeapFree(hp, 0, h3);
    HeapFree(hp, 0, h5); //现在freelist[2]有3个元素

    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);

    return 0;
}
```

整个流程解析：

- ✓ 程序首先创建了一个大小为 0x1000 的堆区，并从其中连续申请了6个块身大小为 8 字节的堆块，加上块首实际上是**6个16字节的堆块**。
- ✓ 释放奇数次申请的堆块是为了防止堆块合并的发生。
- ✓ **三次释放结束后，会形成三个16个字节的空闲堆块放入空表**。因为是16个字节，所以会被依次放入**freelist[2]**所标识的空表，它们依次是h1、h3、h5。
- ✓ 再次申请8字节的堆区内存，加上块首是16个字节，因此**会从freelist[2]所标识的空表中摘取第一个空闲堆块出来，即h1**。
- ✓ 如果我们手动修改h1块首中的前后向指针，能够观察到 DWORD SHOOT 的发生。

实验：调试上述程序，认识堆管理结构

- 通过调试程序来观察堆内存变化
- 调试手段为采用VC6自身的调试器
- 具体了解堆管理过程中的内存变化

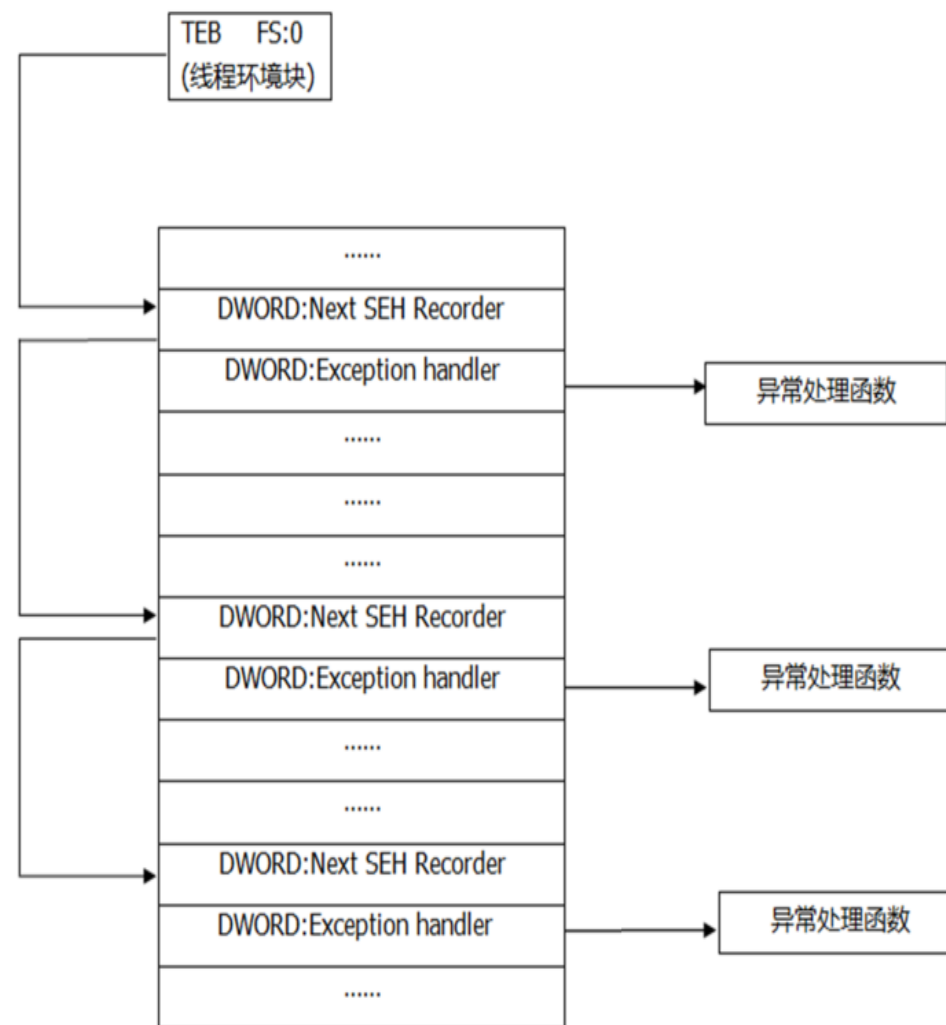
知识点四：其它溢出漏洞

1 SEH结构溢出

为了保证系统在遇到错误时不至于崩溃，仍能够健壮稳定地继续运行下去，Windows会对运行在其中的程序提供一次补救的机会来处理错误，这种机制就是**异常处理机制**。

异常处理结构体SEH是Windows异常处理机制所采用的重要数据结构：

- SHE结构体存放在栈中，栈中的多个SEH通过链表指针在栈内由栈顶向栈底串成单向链表；
- 位于链表最顶端的SEH通过**线程环境块**（TEB, Thread Environment Block）0字节偏移处的指针标识；
- 每个SEH包含两个DWORD指针：**SEH链表指针**和**异常处理函数句柄**，共8个字节。



SEH结构用作异常处理，主要包括如下三个方面：

- 1. 当线程初始化时，会自动向栈中安装一个SEH，作为线程默认的异常处理。**如果程序源代码中使用了_try{}_except{}或者Assert宏等异常处理机制，编译器将最终通过向当前函数栈帧中安装一个SEH来实现异常处理。
- 2. 当异常发生时，操作系统会中断程序，并首先从TEB的0字节偏移处取出距离栈顶最近的SEH，使用异常处理函数句柄所指向的代码来处理异常。**当最近的异常处理函数运行失败时，将顺着SEH链表依次尝试其他的异常处理函数。
- 3. 如果程序安装的所有异常处理函数都不能处理这个异常，系统会调用默认的系统处理程序，**通常显示一个对话框，你可以选择关闭或者最后将其附加到调试器上的调试按钮。如果没有调试器能被附加于其上或者调试器也处理不了，系统就调用ExitProcess终结程序。

SHE攻击

SEH攻击是指通过栈溢出或者其他漏洞，使用精心构造的数据覆盖SEH链表的入口地址、异常处理函数句柄或链表指针等，实现程序执行流程的控制。

因为发生异常的时候，程序会基于SEH链表转去执行一个预先设定的回调函数，攻击者可以利用这个结构进行漏洞利用攻击。

- 由于SEH存放在栈中，利用缓冲区溢出可以覆盖SHE。
- 如果精心设计溢出数据，则有可能把SEH中异常处理函数的入口地址更改为恶意程序的入口地址，实现进程的控制。

SEH链表在栈区的实际分布

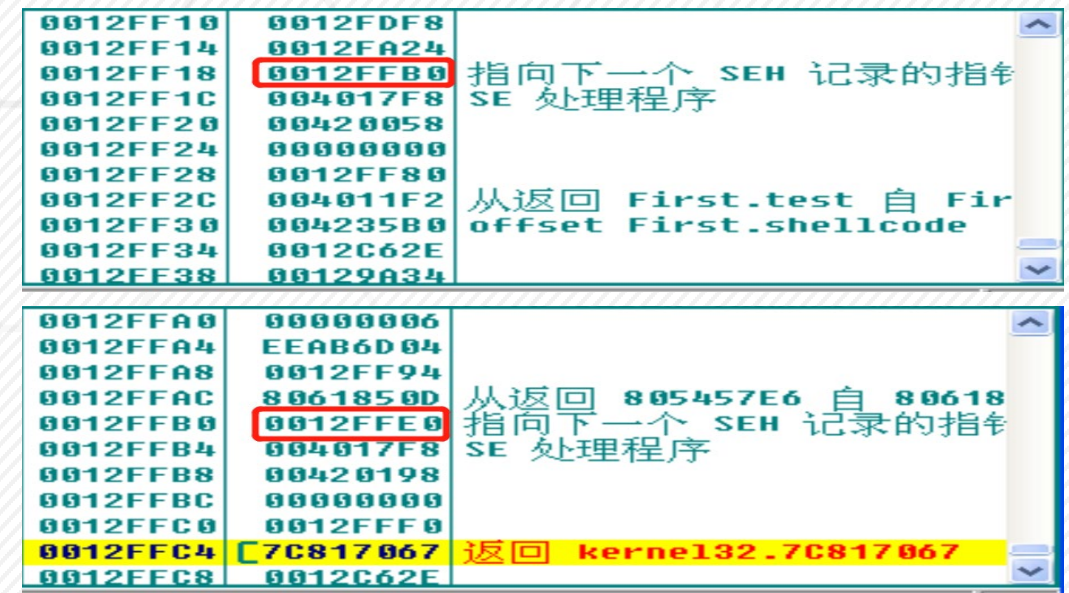
```
char shellcode[] = "";  
void HackExceptionHandler()  
{  
    printf("got an exception, press Enter to kill processn");  
    getchar();  
    ExitProcess(1);  
}  
void test(char* input){  
    char buf[200];  
    int zero = 0;  
  
    __try {  
        strcpy(buf, input);  
        zero = 4 / zero;  
    }__except(HackExceptionHandler())  
    {  
    }  
}  
int main(){  
    test(shellcode);  
    return 0;  
}
```

拖入OllyDBG动态调试，选择View下的SEH chain选项，就能看到当前栈中的SEH表的情况



| 索引 | 类型 | 链接 | 处理程序 |
|----|-----|----------|---------------------------|
| 1 | SEH | 0012F850 | 7C839AC0 |
| 2 | SEH | 0012F930 | 7C839AC0 |
| 3 | SEH | 0012FA44 | 7C9232BC |
| 4 | SEH | 0012FF18 | 00403EC4 _except_handler3 |
| 5 | SEH | 0012FFB0 | 00403EC4 _except_handler3 |
| 6 | SEH | 0012FFE0 | 7C839AC0 |

从图中能看出，0012FF18是离栈顶最近的SHE（此时栈顶为0x0012FFC4）。接着我们在调试的栈窗口去验证存在的SEH链，如下图：



| | | |
|----------|----------|-------------------------|
| 0012FF10 | 0012FDF8 | |
| 0012FF14 | 0012FA24 | |
| 0012FF18 | 0012FFB0 | 指向下一个 SEH 记录的指针 SE 处理程序 |
| 0012FF1C | 004017F8 | |
| 0012FF20 | 00420058 | |
| 0012FF24 | 00000000 | |
| 0012FF28 | 0012FF80 | |
| 0012FF2C | 004011F2 | 从返回 First.test 自 Fir |
| 0012FF30 | 004235B0 | offset First.shellcode |
| 0012FF34 | 0012C62E | |
| 0012FF38 | 00129A34 | |
| 0012FFA0 | 00000006 | |
| 0012FFA4 | EEAB6D04 | |
| 0012FFA8 | 0012FF94 | |
| 0012FFAC | 8061850D | 从返回 805457E6 自 80618 |
| 0012FFB0 | 0012FFE0 | 指向下一个 SEH 记录的指针 SE 处理程序 |
| 0012FFB4 | 004017F8 | |
| 0012FFB8 | 00420198 | |
| 0012FFBC | 00000000 | |
| 0012FFC0 | 0012FFF0 | |
| 0012FFC4 | 7C817067 | 返回 kernel32.7C817067 |
| 0012FFC8 | 0012C62E | |

2

单字节溢出

单字节溢出是指程序中的缓冲区仅能溢出一个字节。

```
void single_func(char *src){  
    char buf[256];  
    int i;  
    for(i = 0;i <= 256;i++)  
        buf[i] = src[i];  
}
```



缓冲区溢出一般是通过覆盖堆栈中的返回地址，使程序跳转到 shellcode 或指定程序处执行。然而在一定条件下，单字节溢出也是可以利用的，它溢出的一个字节必须与栈帧指针紧挨，就是要求必须是**函数中首个变量**，一般这种情况很难出现。

尽管如此，程序员也应该对这种情况引起重视。