1 链表“拆卸”中的问题

堆管理系统的三类操作：堆块分配、堆块释放和堆块合并归根结底都是对链表的修改。例如，分配就是将堆块从空表中“卸下”；释放是把堆块“链入”空表；合并稍微复杂点，但也可以看成是把若干个堆块先从空表中“卸下”，修改块首信息（大小），之后把更新后的新块“链 入”空表。

所有“卸下”和“链入”堆块的工作都发生在链表中，如果我们能伪造链表结点的指针，在“卸下”和“链入”的过程中就有可能获得一次读写内存的机会。

堆溢出利用的精髓就是用精心构造的数据去溢出下一个堆块的块首，改写块首中的前向指针（flink）和后向指针（blink），然后在分配、释放、合并等操作发生时伺机获得一次向内存任意地址写入任意数据的机会。

我们把这种能够向内存任意位置写入任意数据的机会称为“DWORD SHO OT”。

通过 DWORD SHOOT，攻击者可以进而劫持进程，运行 shellcode

注意：DWORD SHOOT 发生时，我们不但可以控制射击的目

标（任意地址），还可以选用适当的子弹（4 字节恶意数据）。

表 5-3-1

点射目标（Target）	子弹（payload）	改写后的结果
栈帧中的函数返回地址 she	llcode 起始地址	函数返回时，跳去执行 shellcode
栈帧中的 S.E.H 句柄 she	llcode 起始地址	异常发生时，跳去执行 shellcode
重要函数调用地址 she	llcode 起始地址	函数调用时，跳去执行 shellcode

本节将重点讲解 DWORD SHOOT 发生的原理，下节将介绍怎样利用 DWORD SHOOT 劫持进程，执行 shellcode。

这里举一个例子来说明在链表操作中 DWORD SHOOT 究竟是怎样发生的。将一个结点从双向链表中“卸下”的函数很可能是类似这样的。

```
int remove (ListNode * node)
{
node -> blink -> flink = node -> flink;
node -> flink -> blink = node -> blink;
return 0;
}
```

按照这个函数的逻辑，正常拆卸过程中链表的变化过程如图 5.3.1 所示。

当堆溢出发生时，非法数据可以淹没下一个堆块的块首。这时，块首是可以被攻击者控制的，即块首中存放的前向指针（flink）和后向指针（blink）是可以被攻击者伪造的。当这个堆块被从双向链表中“卸下”时， node -> blink -> flink = node -> flink 将把伪造的 flink 指针值写 入伪造的 blink 所指的地址中去，从而发生 DWORD SHOOT。这个过程如图 5.3.2 所示。

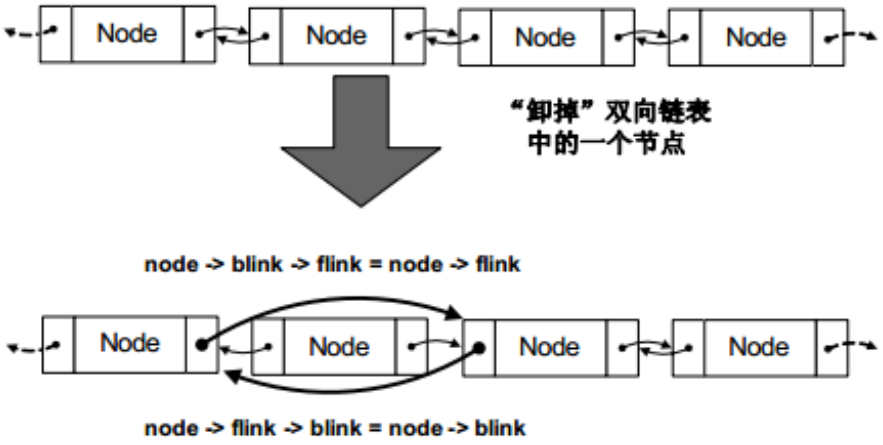

图 5.3.1　空闲双向链表的拆卸

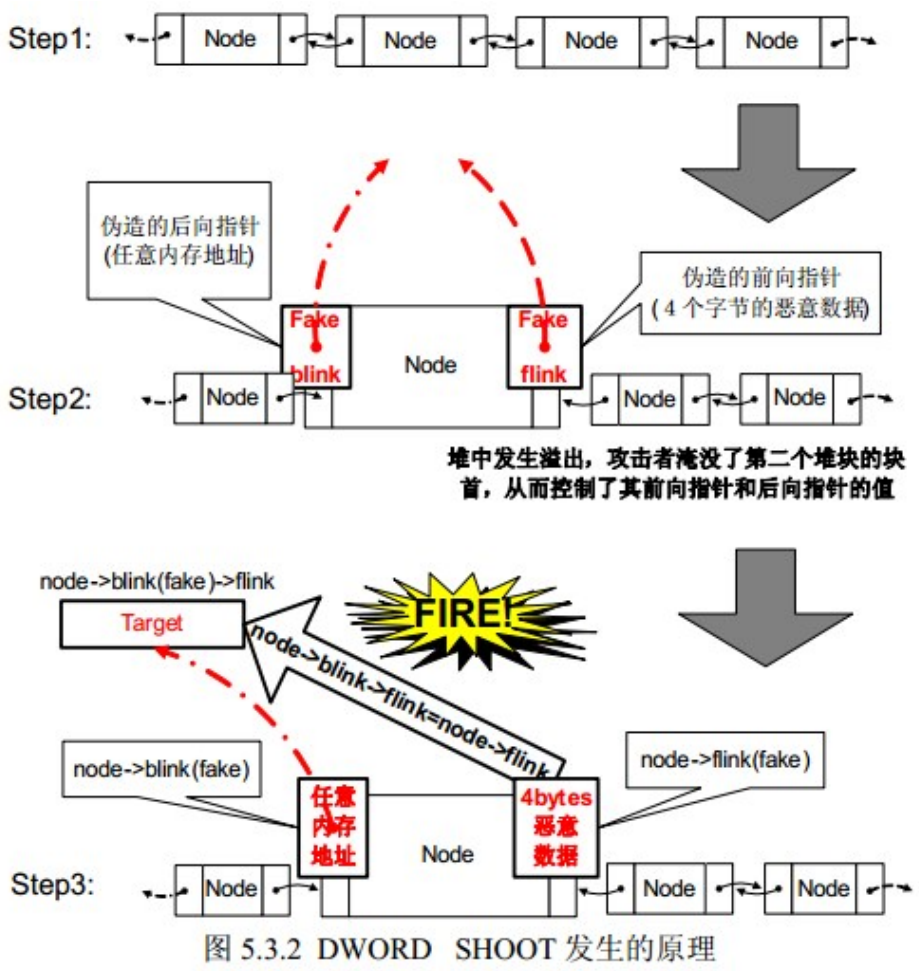

图 5.3.2 DWORD SHOOT 发生的原理

2 在调试中体会“DWORD SHOOT”

我们通过一个简单的调试过程来体会前面的 DWORD SHOOT 技术。用于调试的代码如下。

```
#include <windows.h>
main()
{
HLOCAL h1, h2,h3,h4,h5,h6;
HANDLE hp;
hp = HeapCreate(0,0x1000,0x10000);
```

```
h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
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//used to break the process
//free the odd blocks to prevent coalesing
HeapFree(hp,0,h1);
HeapFree(hp,0,h3);
HeapFree(hp,0,h5); //now freelist[2] got 3 entries
//will allocate from freelist[2] which means unlink the last entry
//(h5)
h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
return 0;
}
```

在这段程序中应该注意：

（1）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 个大小为 8 字节

的堆块（加上块首实际上是 16 字节），这应该是从初始的大块中“切”下来的。

（2）释放奇数次申请的堆块是为了防止堆块合并的发生。

（3）三次释放结束后， freelist[2]所标识的空表中应该链入了 3 个空闲堆块， 它们依次是 h1、h3、 h5。

（4）再次申请 8 字节的堆块，应该从 freelist[2]所标识的空表中分配，这意味着最后一个堆块 h5 被从空表中“拆下”。

（5）如果我们手动修改 h5 块首中的指针，应该能够观察到 DWORD SHOOT 的发生。

为了调试真正状态的堆，应该直接运行程序，让其在_asm int 3 处自己中断，然后在附上调试器。

三次内存释放操作结束后，直接在内存区按快捷键 Ctrl+G 察 0x00520688 处的堆块状况

如图 5.3.3 所示。

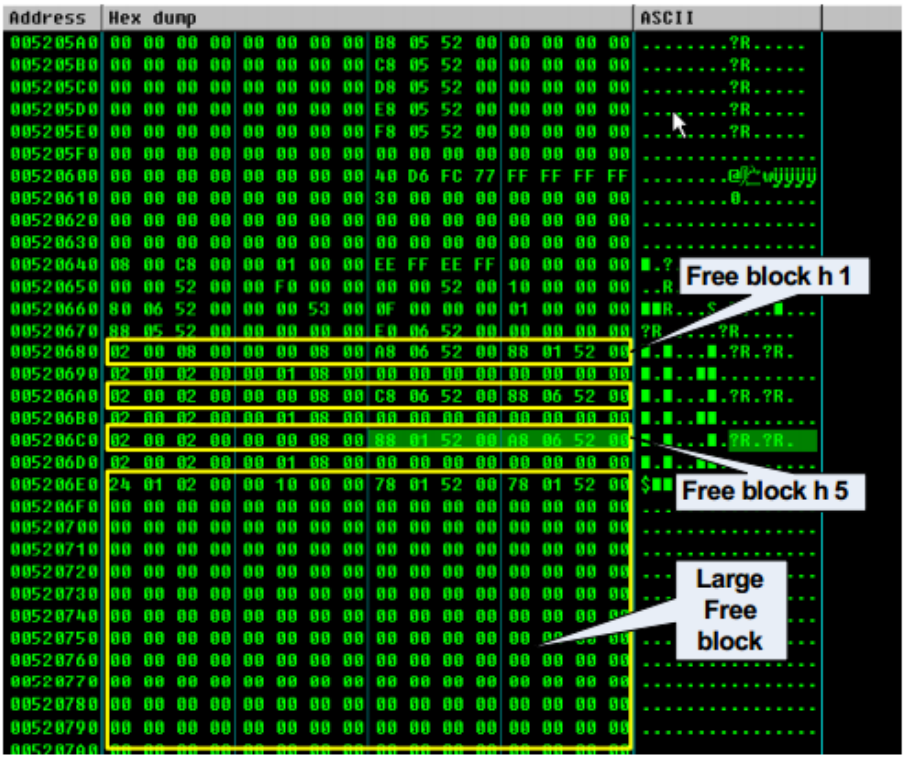

图 5.3.3 DWORD SHOOT 前堆块的状态

从 0x00520680 处开始，共有 9 个堆块

表 5-3-3 堆块情况

	起始位置	Flag	Size 单位：8bytes	前向指针 flink	后向指针 blink
h1	0x00520680	空闲态 0x00	0x0002	0x005206A8	0x00520188
h2	0x00520690	占用态 0x01	0x0002	无	无
h3	0x005206A0	空闲态 0x00	0x0002	0x005206C8	0x00520688
h4	0x005206B0	占用态 0x01	0x0002	无	无
h5	0x005206C0	空闲态 0x00	0x0002	0x00520188	0x005206A8
h6	0x005206D0	占用态 0x01	0x0002	无	无
尾块	0x005206E0	最后一项（0x10）	0x0124	0x00520178 (freelist[0])	0x00520178 (freelist[0])

空表索引区的状况如图 5.3.4 所示。

除了 freelist[0]和 freelist[2]之外，所有的空表索引都为空（指向自身）。

综上所述，整条 freelist[2]链表的组织情况如图 5.3.5 所示。这时，最后一次 8 字节的内存请求会把

freelist[2]的最后一项（原来的 h5）分配出去，这意味着将最后一个结点从双向链表中“卸下”。

如果我们现在直接在内存中修改 h5 堆块中的空表指针（当然攻击发生时是由于溢出而改写的），那么应该能够观察到 DWORD SHOOT 现象，如图 5.3.6 所示。

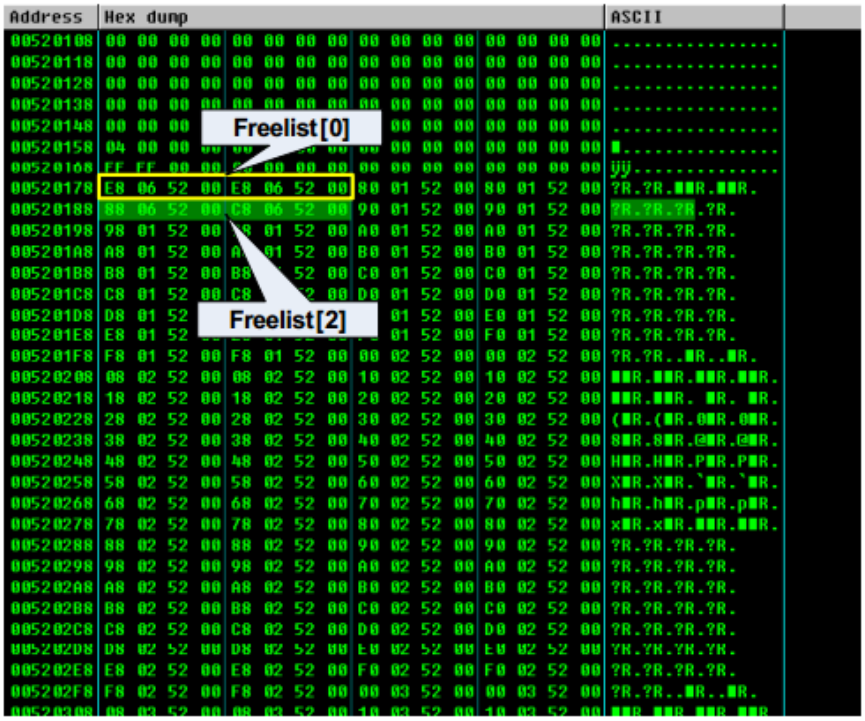

图 5.3.4 DWORD SHOOT 前堆表的状态

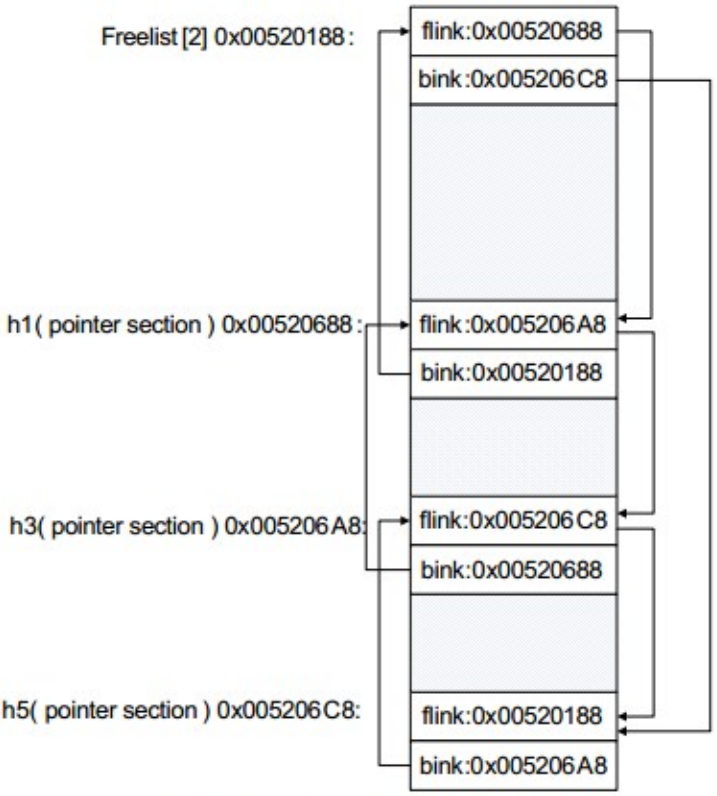

图 5.3.5　空闲双向链表示意图

如图 5.3.6 所示，直接在调试器中手动将 0x005206C8 处的前向指针改为 0x44444444，后向指针改为 0x00000000。当最后一个分配函数被调用后，调试器被异常中断，原因是无法将0x44444444 写入 0x00000000。当然，如果我们把射击目标定为合法地址，这条指令执行后，0x44444444 将会被写入目标。

图 5.3.6　制造 DWORD SHOOT

以上只是引发 DWORD SHOOT 的一种情况。事实上，堆块的分配、释放、合并操作都能引发 DWORD SHOOT（因为都涉及链表操作），甚至快表也可以被用来制造 DWORD SHOOT。由于其原理上基本一致，故不一一赘述，您可以利用本节的理论分析和调试技巧自己举一反三。