《软件安全》实验报告

姓名：郭子涵　学号：2312145　班级：信息安全、法学双学位班

1　实验名称：

堆溢出`Dword Shoot`模拟实验。

2　实验要求：

以第四章示例4-4代码为准，在VC IDE中进行调试，观察堆管理结构，记录Unlink节点时的双向空闲链表的状态变化，了解堆溢出漏洞下的`Dword Shoot`攻击。

3　实验过程：

3.1　使用VC6建立项目

在VC6中建立本项目：

```
#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 //手动增加的 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;
```

```
}
```

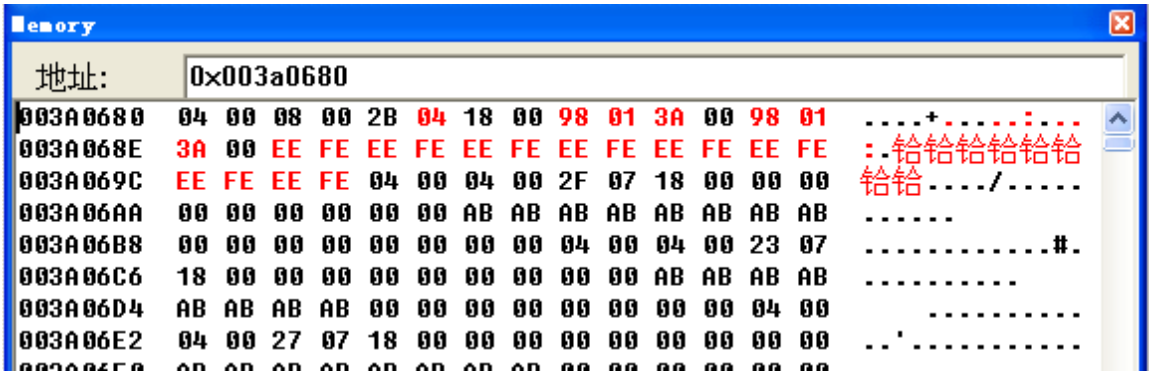
代码利用`Windows`的`HeapAlloc`和`HeapFree`函数控制堆块分配，制造一个特定的堆布局：

1. **创建堆**：使用`HeapCreate(0, 0x1000, 0x10000)`创建一个私有堆。
2. **分配堆块**：依次分配六个 8 字节的小堆块（`h1`到`h6`）。
3. **触发调试中断**：`_asm int 3`插入调试中断，便于调试器在此处暂停，观察堆的状态。
4. **释放部分堆块**：释放`h1`、`h3`、`h5`，形成`freelist[2]`中的 3 个空闲块，避免堆合并。
5. **重新分配堆块**：重新申请`h1`，它可能会占用`freelist[2]`中的一个空闲块。

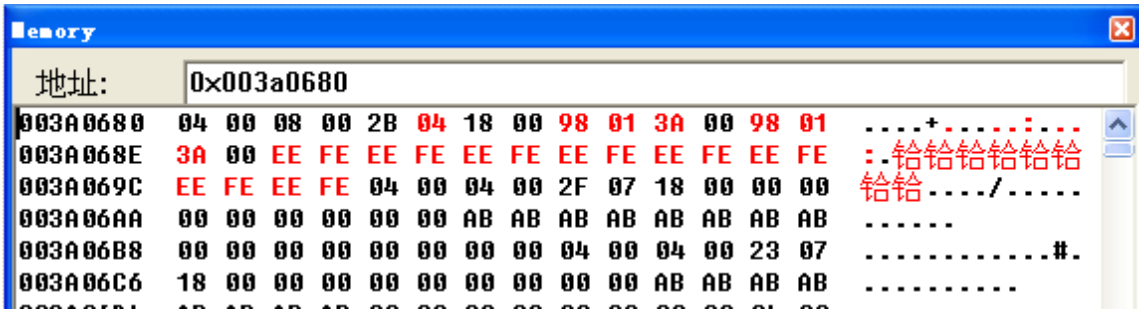
利用**精确控制堆块的分配和释放**，为后续的堆溢出攻击或漏洞利用创造特定的内存布局。

3.2 堆溢出Dword Shoot模拟实验

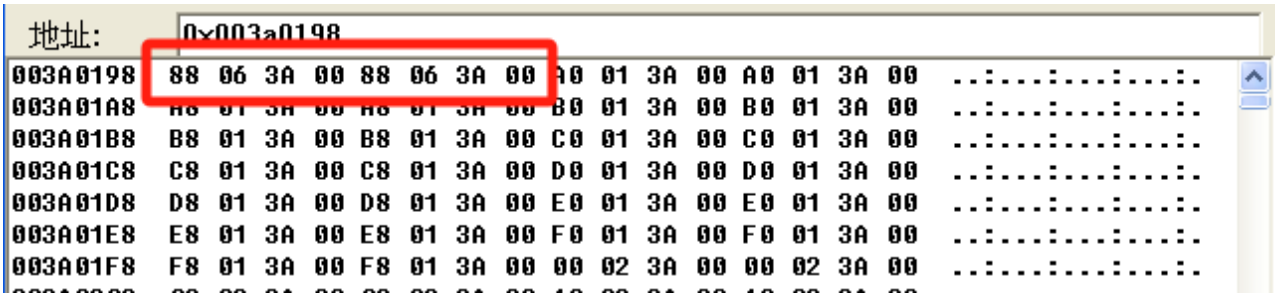
对代码设置断点进行调试。块`h1`的块身地址为`0x003a0688`，减去八字节即为块首地址`0x003a00680`



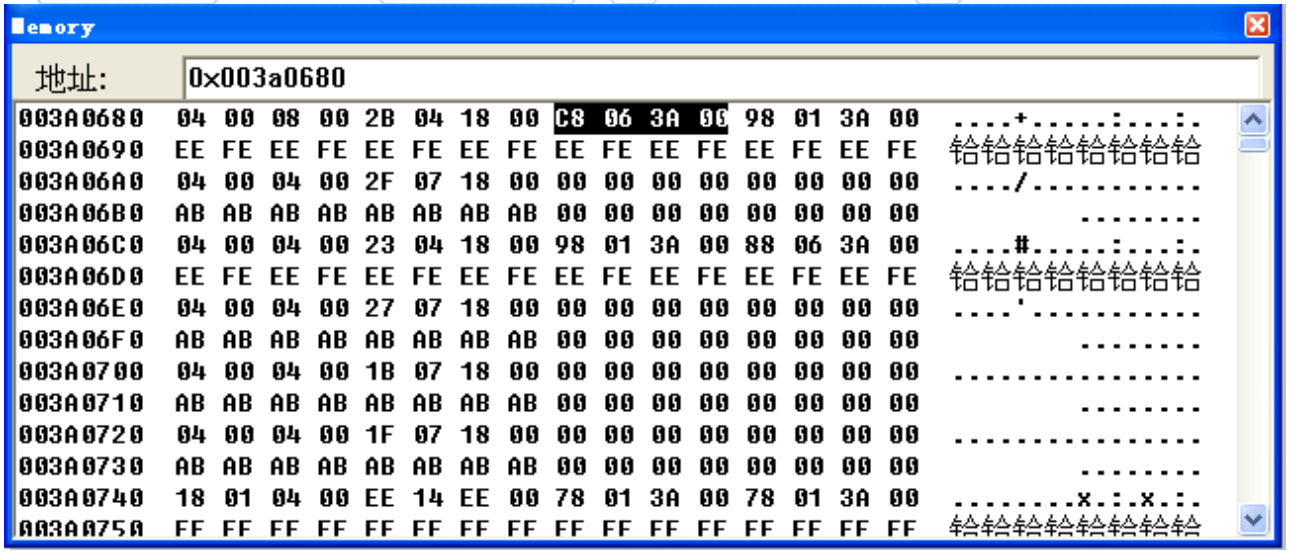
释放`h1`之后，在块首八个字节之后的两个地址相同为`0x00310198`均为指针`flink`和`blink`所指向的地址，`0x00310198`即为`freelist[2]`的地址。



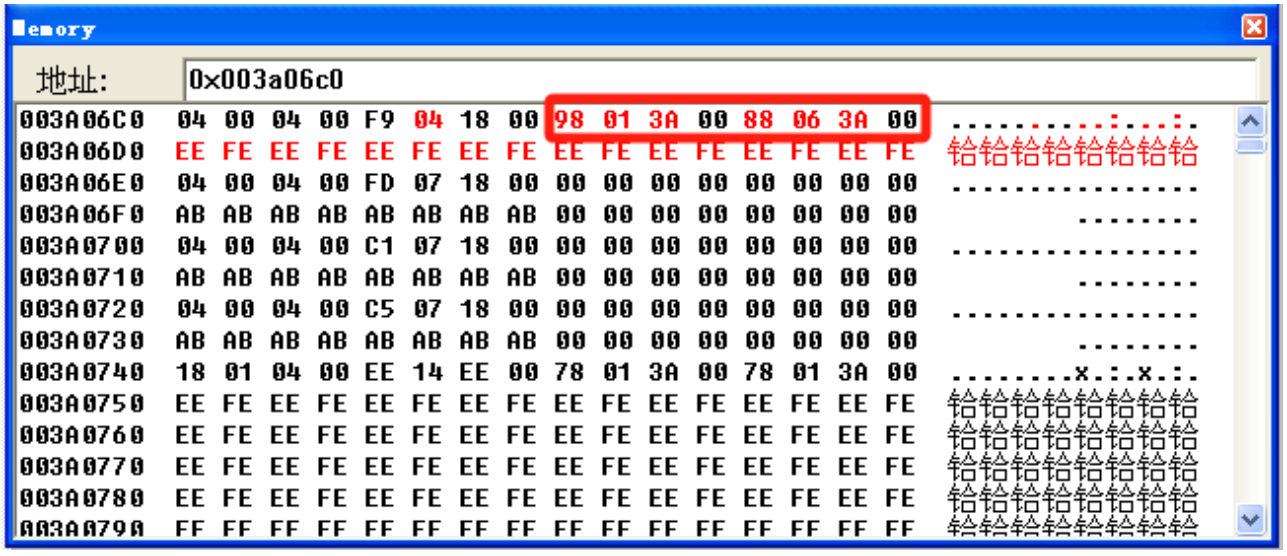
观察`0x003a0198`处信息，`freelist[2]`的`flink`和`blink`均指向了`0x003a0688`，表明此时`freelist[2]`只链入`h1`一个空闲堆块。



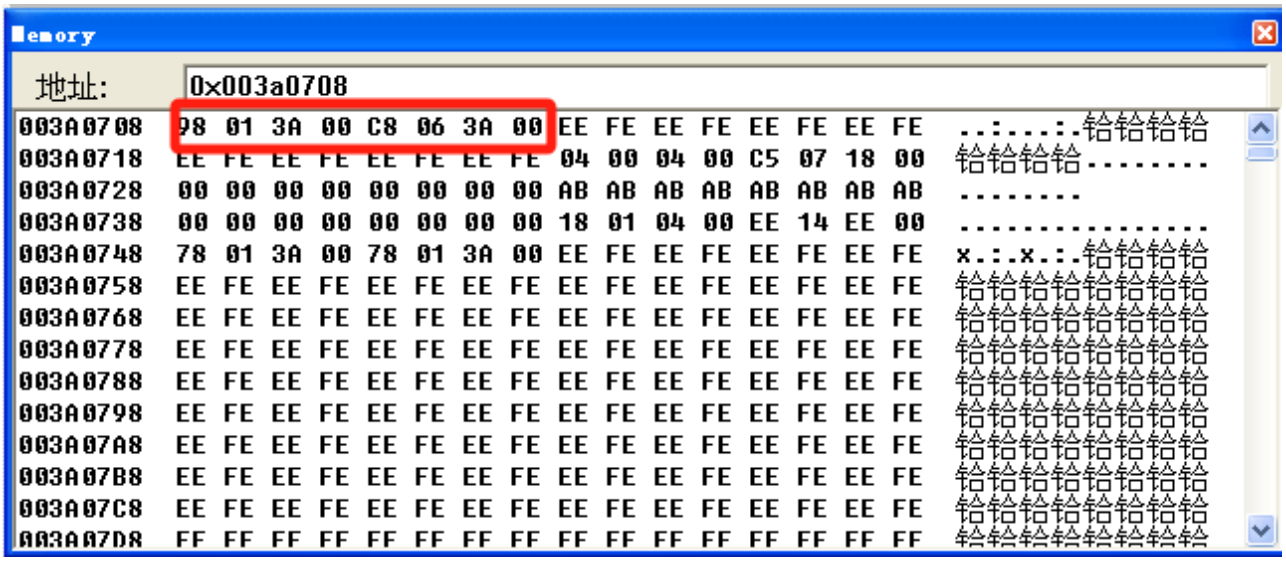
执行 F10 释放 h3，观察 h1 的 flink 指针位置变为 0x003a06c8， blink 指针还是指向 0x003a0198，说明此时 freelist[2] 在 h1 后链入新的块即为 h3



释放 h3，其 flink 指针位置为 0x003a0198,指向 freelist[2]， blink 指针指向 0x003a0688,即 h1 块身的位置，链接在 h1 后。



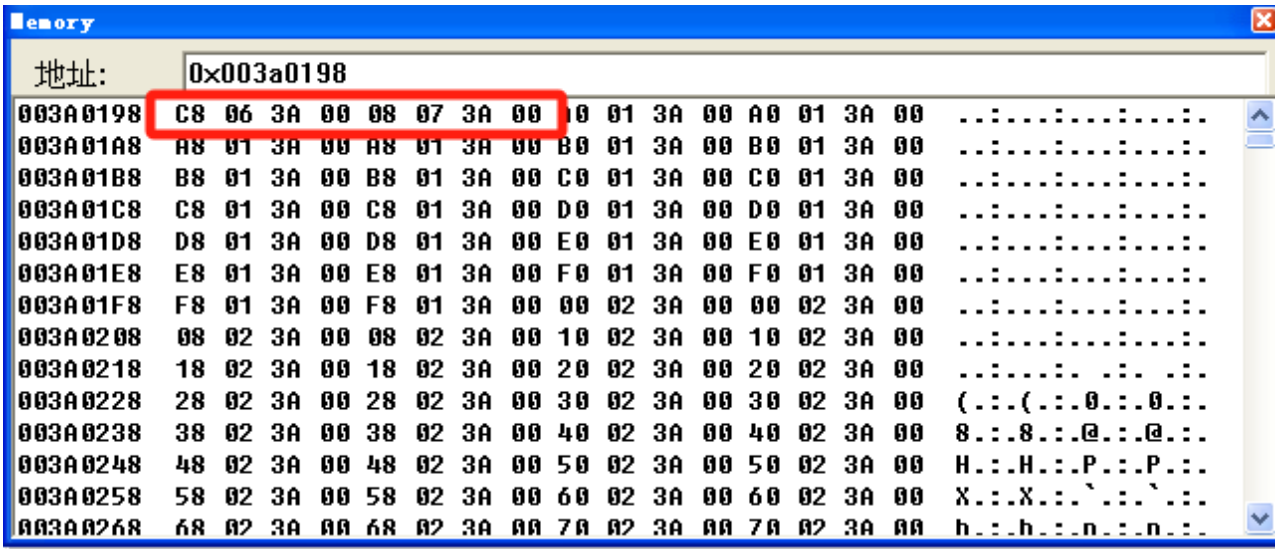
执行 F10 释放 h5，其 flink 指针位置为 0x003a0198,指向 freelist[2]， blink 指针指向 0x003a06c8,即 h3 块身的位置，链接在 h3 后。



最终我们得知此双向链表`Freelist[2] ⇄ h1 ⇄ h3 ⇄ h5 ⇄ Freelist[2]`具体地址如下

```
Freelist[2]
  ├─ flink: 0x003a0688 → h1
  └─ blink: 0x003a06708 → h5
h1
  ├─ flink: 0x003a06c8 → h3
  └─ blink: 0x003a0198 → Freelist[2]
h3
  ├─ flink: 0x003a0708 → h5
  └─ blink: 0x003a0688 → h1
h5
  ├─ flink: 0x003a0198 → Freelist[2]
  └─ blink: 0x003a06c8 → h3
```

取下块`h1`观察`freelist[2]`，其`flink`指针位置为`0x003a06c8`,指向`h3`，即奖`h1`的前向指针的值写入到`h1`后向指着所致的地址内存中。双向链表变为：`Freelist[2] ⇄ h3 ⇄ h5 ⇄ Freelist[2]`



此时就可以通过修改`h1`的两个指针`blink`存储要修改的目标地址，`Flink`存储要写入目标的数据，获得一次像内存构造的任意地址写入一个任意数据的机会（`Dword Shoot`攻击）。

4 心得体会:

本次实验通过精确控制 Windows 堆的分配和释放，模拟了Dword Shoot 堆溢出漏洞下的`Dword Shoot`攻击原理。当空闲队块从链表李卸下时就可以实现一次像任意地址写入任意数据的（`Dword Shoot`攻击）的机会。这一过程也加深了我对 Windows 堆管理机制的理解，特别是空闲链表的行为，`flink`和`blink`两个指针在加入链表和退出链表的变化形式及其对漏洞利用的影响。

实验中的`HeapAlloc`和`HeapFree`操作展示了如何有意制造堆碎片化，使得特定的堆块可以被**可控数据**重新填充，这对于后续的溢出攻击至关重要。

总的来说，本次实验让我更加深刻地认识到堆内存管理的细节及其潜在安全风险，也提醒我在开发过程中要避免堆溢出等常见漏洞，从而提高软件的安全性。