《软件安全》实验报告

姓名：刘星宇　学号：2212824　班级：信息安全法学双学位班

实验名称：

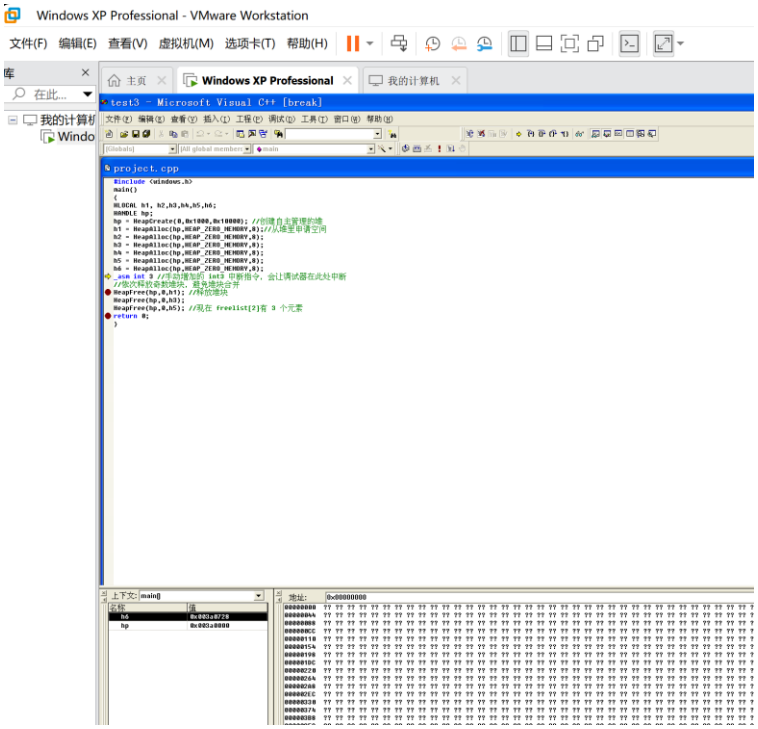
堆溢出 Dword Shoot 模拟实验

实验要求：

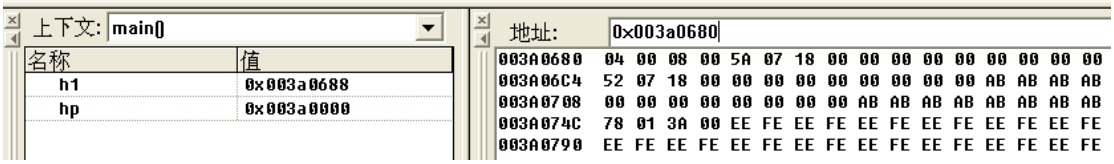
以第四章示例 4-4 代码为准，在 VC IDE 中进行调试，记录 Unlink 节点时的双向空闲链表的状态变化，了解堆溢出漏洞下的 Dword Shoot 攻击。

实验过程：

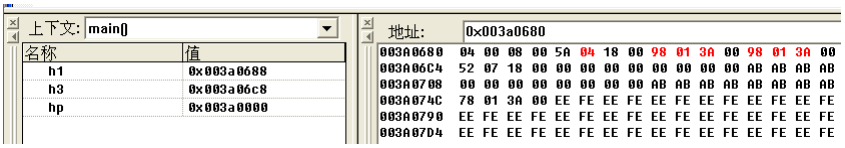
1. 进入 VC，在 VC IDE 中进行调试



2. 过调试程序来观察堆内存变化，采用 VC6 自身的调试器，具体了解堆管理过程中的内存变化



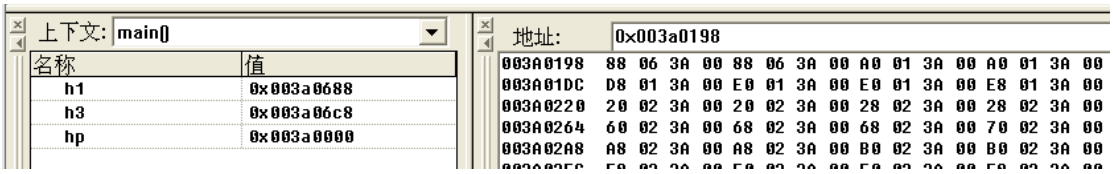
（1） 执行 HeapFree(hp,0,h1)语句时 hp 为 0x003a0000, h1 为 0x003a0688，根据堆块结构，h1 堆块的块身起始位置为 0x003a0688，块首起始位置为 0x003a0680。语句执行后，内存变化如下：



首状态变化，0x003a0688 开始的块身位置的前 8 个字节（Flink 和 Blink） 发生了变化，由 0x000000 变为具体的有效地址。

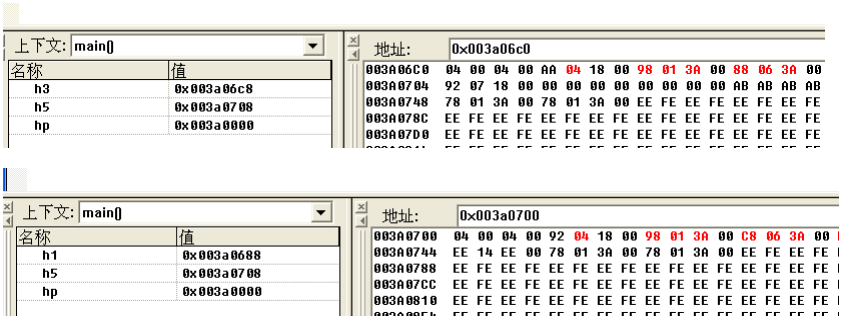
这个是第一个 16 字节的堆块释放，将被链入到 freelist[2]空表中，而此时 Flink 和 Blink 的值都是 0x003a0198，也是 freelist[2] 的地址。

我们转到 0x003a0198 处，如下：



可见，freelist[2]的 Flink 和 Blink 都是 0x003a0688。意味着，当前 freelist[2]唯一后 继节点就是刚刚空闲的 h1 块（地址为 0x003a0688），而 h1 块是唯一前继节点是 freelist[2]。 其它地址（freelist[3]、freelist[4]、freelist[5]）的 Flink 和 Blink 均指向自身，说明都是空表。

（2） 依次执行 HeapFree(hp,0,h3)和 HeapFree(hp,0,h5)后 可知，此时 freelist[2]链表状态为：freelist[2]<=>h1<=>h3<=>h5。

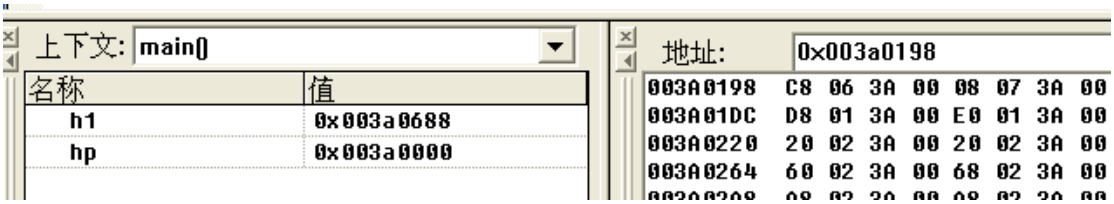


（3） 执行 HeapAlloc(hp,HEAP_ZERO_MEMORY,8)语句时

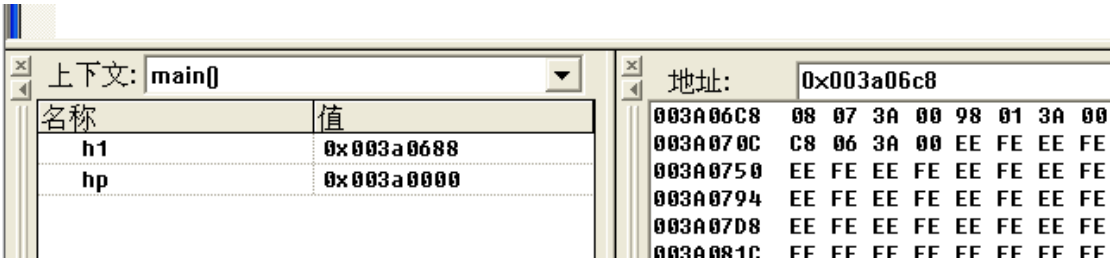
此时，当再次分配空间的时候，从 freelist[2]的双向链表里摘下一块大小为 16 字节的堆块， 首先摘得 h1（地址为 0x003a0688）。

此时 freelist[2]（地址为 0x003a0198）所存储的信息为：Flink（前 4 个字节）为 0x003a0688，Blink（后 4 个字节）为 0x003a0708；h1（地址为 0x003a0688）所存储的信息为：Flink 为 0x003a06c8，Blink 为 0x003a0198； h3（地址为 0x003a06c8）所存储的信息为：Flink 为 0x003a0708，Blink 为 0x003a0688。

摘走 h1 之后，内存变为： ”freelist[2]（地址为 0x003a0198）的前 4 个字节变为 0x003a06c8，实际发生了将 h1 后向 指针（值为 0x003a0198）地址处的值写为 h1 前向指针的值。



h3（地址为 0x003a06c8）的 Blink 变为 h1->Blink，即 0x003a0198，实际发生了将 h1 前向 指针（值为 0x003a06c8）地址处的值写为 h1 后向指针的值。



3． Dword Shoot 攻击。假设在执行该语句之前，h1 的 Flink 和 Blink 被改写为特定地址和特定数值，那么就完成一 次 Dword Shoot 攻击。

心得体会：

通过本次试验，我对堆内存管理、堆溢出漏洞以及 Dword Shoot 攻击有了更深入的理解。

首先，我深刻认识到了堆内存管理的复杂性。在程序运行过程中，堆管理器需要不断地分配和释放内存块，同时还要维护一个双向链表来跟踪空闲的内存块。这个过程中涉及到很多指针操作和内存布局的细节，稍有不慎就可能导致内存泄漏、越界访问等安全问题。

其次，我对堆溢出漏洞的危害性有了更加直观的认识。通过精心构造输入数据，攻击者可以破坏堆内存的结构，进而控制程序的执行流程。在本次试验中，利用 Dword Shoot 攻击改写指针，我能够观察到堆内存结构的变化，并理解攻击者如何利用这些变化来执行恶意操作。