

《软件安全》实验报告

姓名：林盛森

学号：2312631

班级：1070

一、实验名称：

堆溢出 Dword Shoot 攻击实验

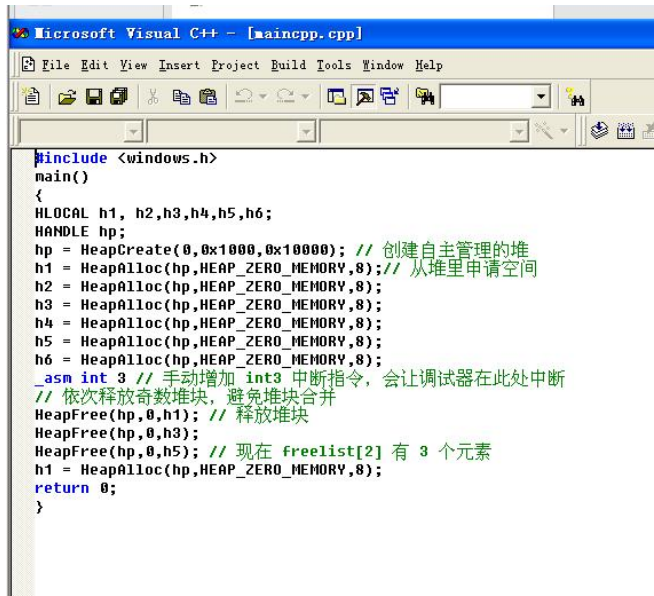
二、实验要求：

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。

三、实验过程：

1. VC6 生成项目：

打开 vc6，新建一个项目，输入实验代码；



```
#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;
}
```

我们先来分析一下这段代码，首先定义了六个变量，用于接收从堆里申请的六块空间，接着创建了一个大小为 0x1000 的堆区，然后申请了六块空间分配给 h1 到 h6，每个块身大小为 8 个字节，加上堆块头，一共 16 个字节，接着添加了一个 int3 指令，这个指令我们在上学期的汇编语言动态逆向分析技术中学到过，是用来让调试器在某一处中断，

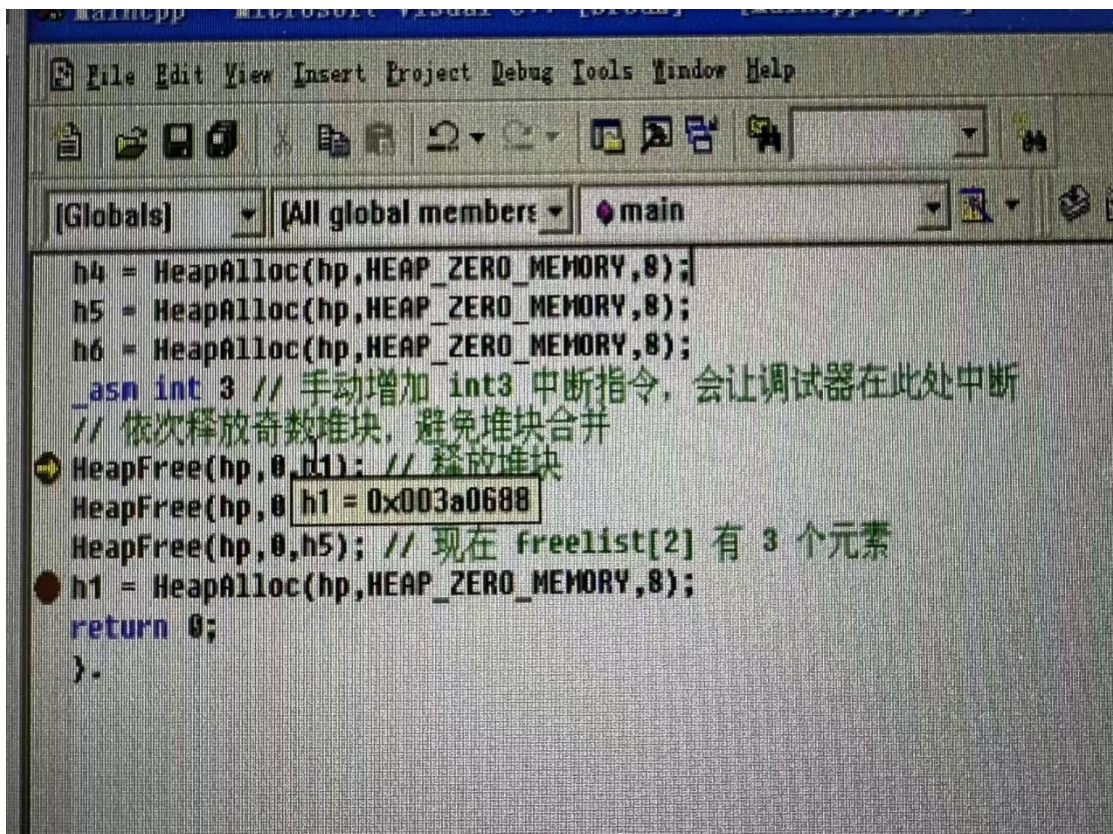
接着把 h1, h3, h5 三个堆块释放掉, 释放掉后会存储被链接在 freelist[2]中, 之后又重新为 h1 申请了空间。

```
(Globals) [All global members] main
#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;
}
```

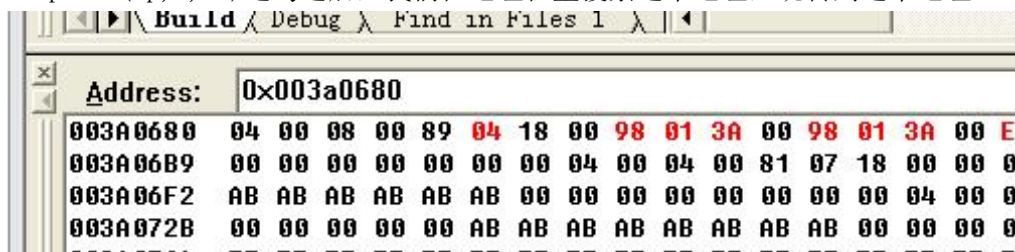
设置断点以便于后面分析;

```
C:\Program Files\Microsoft Visual Studio\MyProjects\test\maincpp.cpp
1:  #include <windows.h>
2:  main()
3:  {
00401010  push     ebp
00401011  mov      ebp,esp
00401013  sub      esp,5Ch
00401016  push     ebx
00401017  push     esi
00401018  push     edi
00401019  lea      edi,[ebp-5Ch]
0040101C  mov      ecx,17h
00401021  mov      eax,0CCCCCCCCh
00401026  rep stos dword ptr [edi]
4:  HLOCAL h1, h2,h3,h4,h5,h6;
5:  HANDLE hp;
6:  hp = HeapCreate(0,0x1000,0x10000); // 创建自主管理的堆
00401028  mov      esi,esp
0040102A  push     10000h
0040102F  push     1000h
00401034  push     0
00401036  call     dword ptr [__imp__HeapCreate@12 (00424140)]
0040103C  cmp      esi,esp
0040103E  call     __chkesp (004011c0)
00401043  mov      dword ptr [ebp-1Ch],eax
7:  h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8); // 从堆里申请空间
00401046  mov      esi,esp
00401048  push     8
0040104A  push     8
0040104C  mov      eax,dword ptr [ebp-1Ch]
0040104F  push     eax
00401050  call     dword ptr [__imp__HeapAlloc@12 (0042413c)]
00401056  cmp      esi,esp
00401058  call     __chkesp (004011c0)
0040105D  mov      dword ptr [ebp-4],eax
8:  h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
00401060  mov      esi,esp
00401062  push     8
00401064  push     8
00401066  mov      ecx,dword ptr [ebp-1Ch]
00401069  push     ecx
0040106A  call     dword ptr [__imp__HeapAlloc@12 (0042413c)]
00401070  cmp      esi,esp
00401072  call     __chkesp (004011c0)
00401077  mov      dword ptr [ebp-8],eax
9:  h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
0040107A  mov      esi,esp
0040107C  push     8
0040107E  push     8
00401080  mov      edx,dword ptr [ebp-1Ch]
00401083  push     edx
00401084  call     dword ptr [__imp__HeapAlloc@12 (0042413c)]
0040108A  cmp      esi,esp
0040108C  call     __chkesp (004011c0)
00401091  mov      dword ptr [ebp-0Ch],eax
```

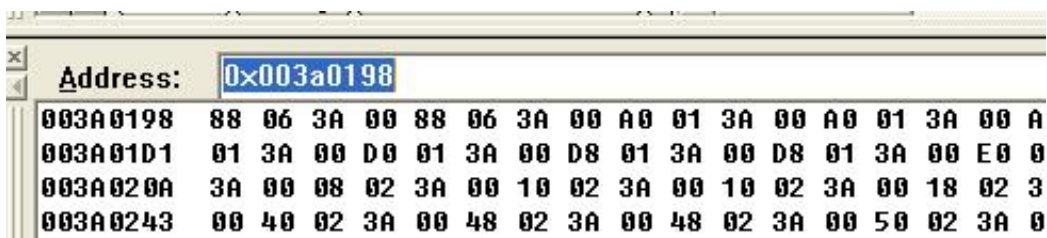
右击可以点击 go to assembly 可以获得汇编代码;



首先键盘点击 f10 单步调试, 运行到 `HeapFree(hp, 0, h1)` 这句, 我们将鼠标悬停到 `h1` 上发现 `h1` 指向 `0x003a0688` 这个地址, 这是 `h1` 块身的首地址, 由于每个堆块有一个大小为 8 字节的块首, 所以我们可以知道块首的地址为 `0x003a0680`, 在执行完 `HeapFree(hp, 0, h1)` 这句之后, 我们在地址栏里搜索这个地址, 跳转到这个地址。



前八个字节为块首信息, 后八个字节为块身信息, 分别是 `fblink` 和 `blink` 指针, 我们可以看到这两个指针都指向了 `003a0198` 这个地址, 由于刚把 `h1` 这个块给释放掉, 推测实现了 `h1` 与 `freelist[2]` 的双向连接, 我们可以推测出 `003a0198` 这个地址就是 `freelist[2]` 的地址, 我们可以跳转到这个地址进行观察。



跳转到 `003a0198` 发现, `freelist[2]` 的 `blink` 和 `fblink` 指针, 指向同一个地址 `003a0688`, 就是被释放的 `h1` 块身的地址, 说明在 `freelist` 链中我们成功链接上了 `h1` 这个节点。

Build Debug Find in Files 1	
Address:	0x003a0680
003A0680	04 00 08 00 89 04 18 00 C8 06 3A 00 98 01 3A 00
003A0689	00 00 00 00 00 00 00 04 00 04 00 81 04 18 00 98
003A06F2	AB AB AB AB AB AB 00 00 00 00 00 00 00 04 00
003A072B	00 00 00 00 00 AB AB AB AB AB AB AB 00 00 00

在执行过 `HeapFree(hp, 0, h3)` 这句之后，释放了第三个堆块，我们跳转到 `h1` 块首地址 `003a0680`，其中 `blink` 指针指向 `003a0198`，`flink` 指针指向 `003a06c8`，可以猜测 `003a06c8` 就是第三个堆块块身的地址，也就是说，在 `h1` 之后成功链接上了 `h3`。

Address:	0x003a06c8
003A06C8	98 01 3A 00 88 06 3A 00 EE FE EE FE EE FE EE FE
003A0701	00 04 00 B9 07 18 00 00 00 00 00 00 00 00 AB
003A073A	00 00 00 00 00 00 18 01 04 00 EE 14 EE 00 78 01
003A0773	EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE

跳转到 `003a06c8` 这个地址后，我们发现第三个块身的 `blink` 指针指向 `003a0688`，也就是第一个堆块块身地址，`flink` 指针指向 `003a0198`，并且此时 `freelist[2]` 的 `blink` 指针指向 `h3`，实现了 `freelist[2]` 与 `h3` 的双向连接。

Address:	0x003a06c8
003A06C8	08 07 3A 00 88 06 3A 00 EE FE EE FE EE FE EE FE
003A0701	00 04 00 B9 04 18 00 98 01 3A 00 C8 06 3A 00 EE
003A073A	00 00 00 00 00 00 18 01 04 00 EE 14 EE 00 78 01
003A0773	FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE
003A07AC	EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE EE

再执行 `HeapFree(hp, 0, h5)` 这句，释放第五个堆块，跳转到第三个堆块块身地址，可以看到 `blink` 指针指向 `003a0688`，与 `h1` 相连，`flink` 指针指向 `003a0708`，也就是说这个地址就是 `h5` 的地址。

Address:	0x003a0708
003A0708	98 01 3A 00 C8 06 3A 00 EE FE EE FE EE FE EE FE
003A0741	01 04 00 EE 14 EE 00 78 01 3A 00 78 01 3A 00 EE
003A077A	EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
003A07B3	FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE

跳转到 `003a0708` 这个地址，其 `blink` 指针指向 `003a06c8`，就是 `h3` 的块身地址，此时完成了 `freelist[2]→h1→h3→h5` 的双向连接，并且 `h5` 的 `flink` 指针指向 `003a0198`，也就是 `freelist[2]` 的地址，并且此时 `freelist[2]` 的 `blink` 指针指向 `h5`，于是实现了 `h5` 与 `freelist[2]` 的双向连接。

Address:	0x003a0198
003A0198	C8 06 3A 00 08 07 3A 00 A0 01 3A 00 A0 01 3A 00
003A01D1	01 3A 00 D0 01 3A 00 D8 01 3A 00 D8 01 3A 00 E0
003A020A	3A 00 08 02 3A 00 10 02 3A 00 10 02 3A 00 18 02
003A0243	00 40 02 3A 00 48 02 3A 00 48 02 3A 00 50 02 3A

最后我们执行 `h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8)` 这一句，重新为 `h1` 分配空间，

我们可以猜测其将从空表中被删除掉，也就是 freelist[2] 的 flink 指针应该改成指向 h3。我们跳转到 freelist[2] 的地址，其 blink 指针指向 003a0708，即 h5 的地址，flink 指针指向 003a06c8，就是 h3 的块身地址，印证了我们的猜想。并且 h3 的 blink 指针也指向 003a0198。这也就实现了把 h1 的后向指针的值写入到前向指针所指向的地址，前向指针的值写入了后向指针所指向的地址，如果我们手动修改 h1 块首中的前后向指针，使其指向恶意代码的入口地址，即可观察到 DWORD SHOOT 的发生，就是堆溢出漏洞。

四、心得体会：

更加熟练使用 vc6 中断点设置、反汇编、语句逐步执行、跳转到某一个地址的操作，并且学会了如何分析一个地址所指向的信息，对小端序有了更好的理解；

学会了如何创建堆块、释放堆块，对堆块有了更好的了解，其由块首和块身组成，在为块身分配空间的时候，块首会自动分配；

通过分析不同存储地址的一些信息，对堆块合并、摘除时其 blink 指针、flink 指针以及空表的结构如何变化有了更好的了解；

通过本次实验，对 DWORD SHOOT 的攻击原理有了更深刻的理解，若是把卸下的堆块的前后向指针指向恶意代码的入口地址，后果不堪设想，使我加深了对堆溢出的理解，也提醒我在写程序时要注意到溢出安全的问题。