

《软件安全》实验报告

姓名：李雅帆 学号：2213041 班级：信安班

一、实验名称：

堆溢出 Dword Shoot 模拟实验

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.流程解析：

- (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。
- (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，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。
- (4) 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。
- (5) 如果我们手动修改 h1 块首中的指针，应该能够观察到 DWORD SHOOT 的发生。

2.实验过程

在 VC6 中创建新的工程， 并进入 DeBug 模式。

```
Work3.cpp *
#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;
}
```

(1) 执行 HeapFree(hp,0,h1)语句时。

执行 HeapFree(hp,0,h1)前， h1 块身的首地址为 0x003a0688， 对应的块首的起始地址为 0x003a0680。

地址:	0x003A0688
003A0688	00 00 00 00 00 00 00 00 AB AB AB AB AB AB AB 00 00
003A06C3	00 B4 07 18 00 00 00 00 00 00 00 00 00 AB AB AB AB AB
003A06FE	00 00 04 00 04 00 8C 07 18 00 00 00 00 00 00 00 00
003A0739	00 00 00 00 00 00 00 18 01 04 00 EE 14 EE 00 78 01 3A
003A0774	EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
003A07AF	FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE

地址:	0x003A0680
003A0680	04 00 08 00 BC 07 18 00 00 00 00 00 00 00 AB AB AB AB AB AB AB 00 00 00
003A06BB	00 00 00 00 00 00 04 00 04 00 B4 07 18 00 00 00 00 00 AB AB AB AB AB AB
003A06F6	AB AB 00 00 00 00 00 00 00 00 04 00 04 00 8C 07 18 00 00 00 00 00 00 AB
003A0731	AB AB AB AB AB AB AB 00 00 00 00 00 00 18 01 04 00 EE 14 EE 00 78 01 3A
003A076C	EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
003A07A7	FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE

当执行 HeapFree(hp,0,h1)后， 块身的前 8 个字节发生了变化， 前 8 个字节分表示的是 Flink 和 Blink， 二者的值都是 0x003a0198， 即 2 是 freelist[2]的地址。我们转到 freelist[2]的地址处可以发现， freelist[2]的 Flink 和 Blink 都指向了 0x003a0688， 这就是我们释放的 h1 的块身的首地址， 由于 freelist[2]中目前只有 h1 一个堆块， 因此两个值是完全相同的。

[illegible]

(2) 依次执行 `HeapFree(hp,0,h3)`和 `HeapFree(hp,0,h5)`后。

地址:	0x003A0198																							
003A0198	88	06	3A	00	C8	06	3A	00	A0	01	3A	00	A0	01	3A	00	A8	01	3A	00	A8	01		
003A01BC	B8	01	3A	00	C0	01	3A	00	C0	01	3A	00	C8	01	3A	00	C8	01	3A	00	D0	01		
003A01E0	E0	01	3A	00	E0	01	3A	00	E8	01	3A	00	E8	01	3A	00	F0	01	3A	00	F0	01		
003A0204	00	02	3A	00	08	02	3A	00	08	02	3A	00	10	02	3A	00	10	02	3A	00	18	02		
003A0228	28	02	3A	00	28	02	3A	00	30	02	3A	00	30	02	3A	00	38	02	3A	00	38	02		
003A024C	48	02	3A	00	50	02	3A	00	50	02	3A	00	58	02	3A	00	58	02	3A	00	60	02		
003A0270	70	02	3A	00	70	02	3A	00	78	02	3A	00	78	02	3A	00	80	02	3A	00	80	02		

地址:	0x003A0198																																																																																																																																																															
003A0198	88	06	3A	00	88	06	3A	00	A0	01	3A	00	A0	01	3A	00	A8	01	3A	00	A8	01	003A01BC	B8	01	3A	00	C0	01	3A	00	C0	01	3A	00	C8	01	3A	00	C8	01	3A	00	D0	01	003A01E0	E0	01	3A	00	E0	01	3A	00	E8	01	3A	00	E8	01	3A	00	F0	01	3A	00	F0	01	003A0204	00	02	3A	00	08	02	3A	00	08	02	3A	00	10	02	3A	00	10	02	3A	00	18	02	003A0228	28	02	3A	00	28	02	3A	00	30	02	3A	00	30	02	3A	00	38	02	3A	00	38	02	003A024C	48	02	3A	00	50	02	3A	00	50	02	3A	00	58	02	3A	00	58	02	3A	00	60	02	003A0270	70	02	3A	00	70	02	3A	00	78	02	3A	00	78	02	3A	00	80	02	3A	00	80	02

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

执行完语句，可以看到，freelist[2]的 Blink 变成了 h3 块身的首地址，h3 的 Flink 变成了 freelist[2]的地址，h1 块身中目前没有数据，全部为 0。

地址:	0x003A0198
003A0108	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
003A012C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
003A0150	00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00
003A0174	00 00 00 00 48 07 3A 00 48 07 3A 00 80 01 3A 00
003A0198	C8 06 3A 00 08 07 3A 00 A0 01 3A 00 A0 01 3A 00
003A01BC	B8 01 3A 00 C0 01 3A 00 C0 01 3A 00 C8 01 3A 00

地址:	0x003A0688
003A05F8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40
003A061C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
003A0640	08 00 C8 00 00 01 00 00 EE FF EE FF 00 00 00 00 00 00
003A0664	00 00 3B 00 0F 00 00 00 01 00 00 00 88 05 3A 00 00
003A0688	00 00 00 00 00 00 00 00 AB AB AB AB AB AB AB AB 00

地址:	0x003A06C8
003A0638	00 00 00 00 00 00 00 00 08 00 C8 00 00 01 00 00 EE FF EE FF
003A065C	10 00 00 00 00 06 3A 00 00 00 3B 00 0F 00 00 00 01 00 00 00
003A0680	04 00 08 00 A8 07 18 00 00 00 00 00 00 00 00 00 AB AB AB AB
003A06A4	AC 07 18 00 00 00 00 00 00 00 00 00 AB AB AB AB AB AB AB AB
003A06C8	08 07 3A 00 98 01 3A 00 EE FE EE FE EE FE EE FE EE FE EE FE
003A06EC	00 00 00 00 AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00

(4) 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。

四、心得体会：

实验涉及到了堆内存管理的一些基本概念和细节，通过手动操作堆块的申请、释放和指针修改，可以观察到堆内存管理的一些特性和可能出现的问题。

首先，实验中提到了创建大小为 0x1000 的堆区，并连续申请了 6 个 8 字节的堆块，这是为了形成一定的堆结构，并且观察后续的释放操作。

接着，通过释放奇数堆块，形成了三个 16 字节的空闲块，这些空闲块会被

加入到相应的空闲块链表中。这个步骤展示了堆内存管理中的空闲块合并机制，即相邻的空闲块可以合并成一个更大的空闲块，提高了内存利用率。

然后，在重新申请 8 字节的堆内存时，程序会从相应大小的空闲块链表中选择一个空闲块返回给程序使用，这展示了堆内存管理中的分配算法，通常是选择合适大小的空闲块以尽量减少内存碎片。

最后，通过手动修改堆块的指针，触发了 DWORD SHOOT，这说明了堆内存管理的一个重要问题，即堆块的越界访问或者指针篡改可能导致程序的崩溃或者安全漏洞。

实验通过简单的堆内存管理操作，展示了堆内存的结构和管理方式，以及可能出现的问题，有助于理解堆内存管理机制和进行相关安全分析。