

# 《软件安全》实验报告

姓名：陆皓喆      学号：2211044      班级：信息安全

## 实验名称：

堆溢出Dword Shoot攻击实验

## 实验要求：

以第四章示例4-4代码为准，在VC IDE中进行调试，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。

## 实验过程：

### 1.进入VC反汇编

我们输入源码，利用Windows的HeapCreate函数来创建堆，进行实验。源码内容如下所示。

```
#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;
}
```

我们观察程序，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。

接着是释放奇数次申请的堆块，这是为了防止堆块的合并。

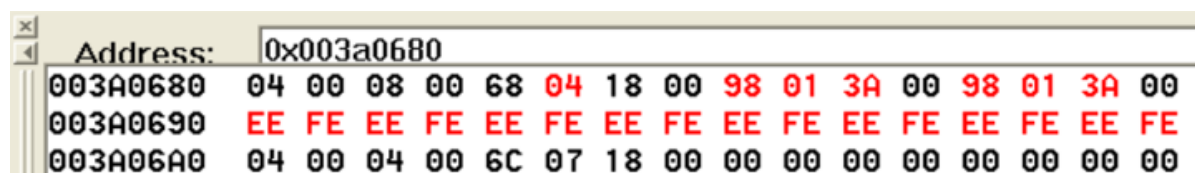
三次释放结束后，会形成三个16个字节的空闲堆块放入空表。因为是16个字节，所以会被依次放入 `freelist[2]` 所标识的空表，它们依次是 `h1`、`h3`、`h5`。

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`。之后我们只需要通过修改 `h1` 的前后向指针就可以观察到 `Dword Shoot` 攻击。下面，我们通过设置断点来观察整个攻击的两个指针的变化流程。

## 2.堆管理过程中的内存具体变化

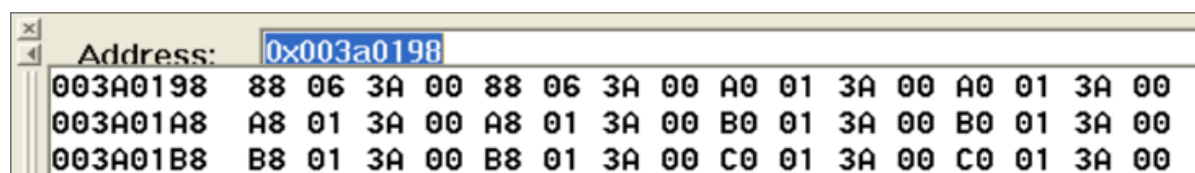
我们通过在代码中设置断点，来观察内存的变化。

将鼠标移到第一个块身 `h1` 处，观察到其地址为 `0x003a0688`，因为这是块身的起始地址，再减去8就是块首的地址 `0x003a0680`。当执行完 `h1` 堆块的释放后，我们跳转到这个地址观察。



Address:	0x003a0680
003A0680	04 00 08 00 68 04 18 00 98 01 3A 00 98 01 3A 00
003A0690	EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
003A06A0	04 00 04 00 6C 07 18 00 00 00 00 00 00 00 00 00

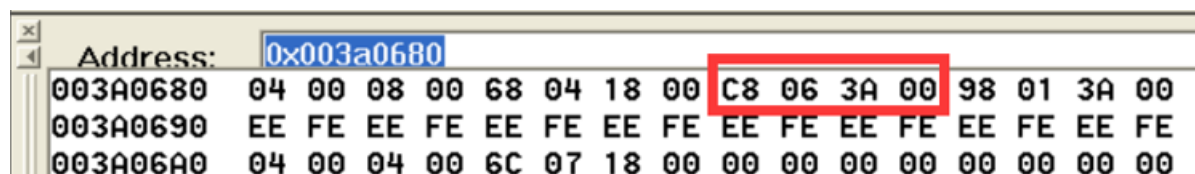
观察 `0x003a0680` 开始的内存，前八个字节是块首的一些信息。后八个字节分别是 `fblink` 和 `blink` 对应的内容，可以看到他们都指向了 `0x003a0198`。根据堆块空表的管理方式，我们可以推测出这个地址实际上就是 `freelist[2]` 的地址。我们可以跳转到这个地址观察即可。



Address:	0x003a0198
003A0198	88 06 3A 00 88 06 3A 00 A0 01 3A 00 A0 01 3A 00
003A01A8	A8 01 3A 00 A8 01 3A 00 B0 01 3A 00 B0 01 3A 00
003A01B8	B8 01 3A 00 B8 01 3A 00 C0 01 3A 00 C0 01 3A 00

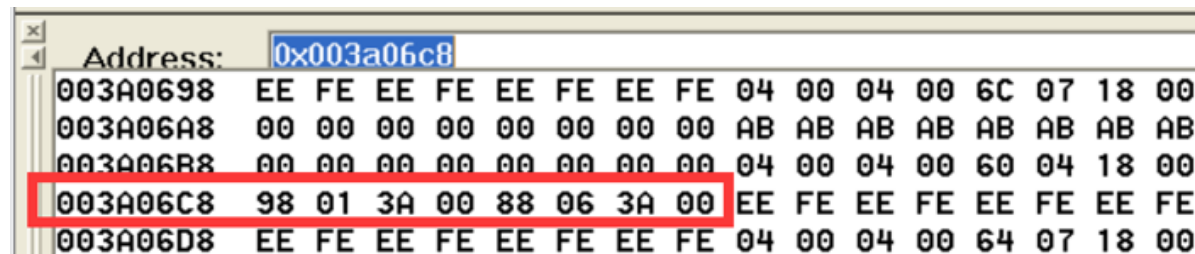
我们发现，`freelist[2]` 的 `fblink` 和 `blink` 均指向了 `0x003a0688`，这就是刚才释放的 `h1` 的地址，同时也说明此时这条链上只链入了 `h1` 一个空闲堆块，前向指针和后向指针都指向 `h1`。

接着，我们来完成释放 `h3` 的操作，释放后，继续跳转到 `h1` 的内存处查看。



Address:	0x003a0680
003A0680	04 00 08 00 68 04 18 00 C8 06 3A 00 98 01 3A 00
003A0690	EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
003A06A0	04 00 04 00 6C 07 18 00 00 00 00 00 00 00 00 00

我们发现，与上一步相比，`h1` 的 `fblink` 指针位置发生了改变，说明在空表上 `freelist[2]` 中又链入了一个空闲堆块，即 `h3`。`h1` 的前向指针此时指向的 `0x003a06c8` 就是 `h3` 堆块的块身地址。我们跳转到 `h3` 所在的地址 `0x003a06c8` 查看即可。



Address:	0x003a06c8
003A0698	EE FE EE FE EE FE EE FE 04 00 04 00 6C 07 18 00
003A06A8	00 00 00 00 00 00 00 00 AB AB AB AB AB AB AB AB
003A06B8	00 00 00 00 00 00 00 00 04 00 04 00 60 04 18 00
003A06C8	98 01 3A 00 88 06 3A 00 EE FE EE FE EE FE EE FE
003A06D8	EE FE EE FE EE FE EE FE 04 00 04 00 64 07 18 00

这里，前四个字节是 `fblink` 的值，指向的就是 `freelist[2]`，后四个字节是 `blink` 的值，指向的是 `h1` 堆块的块身。这说明此时空表结构为：`freelist[2]` 链接 `h1` 链接 `h3`，`h3` 又和 `freelist[2]` 双向链接。

我们继续进行 `h5` 的释放，释放完之后，跳转到 `0x003a06c8`，即 `h3` 的块身地址查看。

Address:	0x003a06c8
003A0698	EE FE EE FE EE FE EE FE 04 00 04 00 6C 07 18 00
003A06A8	00 00 00 00 00 00 00 00 AB AB AB AB AB AB AB AB
003A06B8	00 00 00 00 00 00 00 00 04 00 04 00 60 04 18 00
003A06C8	08 07 3A 00 88 06 3A 00 EE FE EE FE EE FE EE FE
003A06D8	EE FE EE FE EE FE EE FE 04 00 04 00 64 07 18 00

发现他的前向指针 `flink` 发生了改变，指向了 `0x003a0708`，这是新链入的 `h5` 的地址。我们跳转到该地址查看。

Address:	0x003a0708
003A06D8	EE FE EE FE EE FE EE FE 04 00 04 00 64 07 18 00
003A06E8	00 00 00 00 00 00 00 00 AB AB AB AB AB AB AB AB
003A06F8	00 00 00 00 00 00 00 00 04 00 04 00 58 04 18 00
003A0708	98 01 3A 00 C8 06 3A 00 EE FE EE FE EE FE EE FE
003A0718	EE FE EE FE EE FE EE FE 04 00 04 00 5C 07 18 00

可以发现，`flink` 指针所指向的地址依然是 `freelist[2]`，后面的 `blink` 指向的是 `h3` 的地址，此时，我们已经完成了三个堆块的释放，我们分析出此时块表内存的存储结构应该是：`freelist[2]` 链接 `h1` 链接 `h3` 链接 `h5`，`h5` 又和 `freelist[2]` 双向链接。

综上所述，此时各个堆块的 `flink` 和 `blink`：

	flink	blink
Freelist[2]	0x003a0688(h1)	0x003a06708(h5)
h1	0x003a06c8(h3)	0x003a0198(freelist[2])
h3	0x003a0708(h5)	0x003a0688(h1)
h5	0x003a0198(freelist[2])	0x003a06c8(h3)

最后我们重新分配一个块身为8字节的堆。取下第一个空闲堆块 `h1`。这时我们先跳转到 `0x003a0198`，即 `freelist[2]` 处，我们发现他的 `flink` 变成了 `0x003a06c8`（链接 `h3`），即发生了将 `h1` 前向指针的值写入到了 `h1` 后向指针所指的地址内存里。

Address:	0x003a0198
003A0168	00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
003A0178	48 07 3A 00 48 07 3A 00 80 01 3A 00 80 01 3A 00
003A0188	88 01 3A 00 88 01 3A 00 90 01 3A 00 90 01 3A 00
003A0198	C8 06 3A 00 08 07 3A 00 A0 01 3A 00 A0 01 3A 00
003A01A8	A8 01 3A 00 A8 01 3A 00 B0 01 3A 00 B0 01 3A 00

跳转到 `0x003a06c8`，即 `h3` 处，我们发现，我们发现 `h3` 堆块的 `blink` 变为了 `0x003a0198`（`freelist[2]`），即发生了将 `h1` 后向指针的值写入到 `h1` 前向指针所指的地址内存里。

以上就是完成了 `Dword Shoot` 攻击原理的展示。卸下一个堆块的时候，会将其前向指针和后向指针的值写入到其指向的内存当中，因此我们可以利用这个来实现一次 `Dword Shoot` 攻击。

# 心得体会：

1. 在本次实验中，我学会了如何在VC6中建立堆、释放堆。
2. 在本次实验中，通过跟踪堆块内存位置，我深入理解了堆表的内存管理形式，理解了堆表的合并、空表的链接等知识点，明白了两个指针（fblink、blink）的变化形式。
3. 理解了 Dword Shoot 的攻击原理，即**精心构造一个地址和一个数据，当这个空闲堆块从链表里卸下的时候，就获得一次向内存构造的任意地址写入一个任意数据的机会。**
4. 通过本节课的学习，我了解了栈溢出、堆溢出的危害，其中包括之前自己写的一些C代码，回过头看，其实都有一些缺陷与漏洞。
5. 回想到上学期的《汇编语言与逆向技术》中的第七次实验课程，我们也是通过逆向分析来破解代码，当时是输入6位密码来进行破解，但是当我输入36位以上的密码时，就不需要进行破解，就可以破解密码，当时还不知道为什么会这样，以为是代码出了问题，但是现在想想，应该是出现了栈溢出的问题。我回看了上次实验的内容，发现输出函数的地址与输入challenge的地址之差为36，所以一旦输入36位以上的内容时，就会出现栈溢出的问题，导致无论如何，都会输出密码正确。下图是当时我的实验报告。

经过查看内存空间，我发现：当challenge的长度大于等于36时，会使得输出结果的内存空间被覆盖，导致其直接输出1，使最后的结果不管怎么样都会输出正确的“Congratulations, you made it! ”。这其实是该程序的一个漏洞，可以通过限制其输入长度或者改变输出结果内存存储位置去解决这个问题。

.data:004030B1	Str	db 0	; DATA XREF: start+Efo
.data:004030B1			; start+21fo
.data:004030B2	Str_1	db 0	; DATA XREF: start+71fr
.data:004030B3	byte_4030B3	db 0	; DATA XREF: start+B0fr
.data:004030B4	Str_3	db 0	; DATA XREF: start+78fr
.data:004030B5	byte_4030B5	db 0	; DATA XREF: start+81fr
.data:004030B6		db 0	
.data:004030B7		db 0	
.data:004030B8		db 0	
.data:004030B9		db 0	
.data:004030BA		db 0	
.data:004030BB		db 0	
.data:004030BC		db 0	
.data:004030BD		db 0	
.data:004030BE		db 0	
.data:004030BF		db 0	
.data:004030C0		db 0	
.data:004030C1		db 0	
.data:004030C2		db 0	
.data:004030C3		db 0	
.data:004030C4		db 0	
.data:004030C5		db 0	
.data:004030C6		db 0	
.data:004030C7		db 0	
.data:004030C8		db 0	
.data:004030C9		db 0	
.data:004030CA		db 0	
.data:004030CB		db 0	
.data:004030CC		db 0	
.data:004030CD		db 0	
.data:004030CE		db 0	
.data:004030CF		db 0	
.data:004030D0		db 0	
.data:004030D1	dword_4030D1	dd 0	; DATA XREF: start+101fw
.data:004030D1			; start:loc_40111D7r

（如图，查看输出结果的内存空间）