HeapOverflow

1. 堆

（1）.堆是程序运行时动态分配的内存。所谓动态是指所需内存的大小在程序设计时不能预先决定的，需要在程序运行时参考用户的反馈。

（2）.堆在使用时需要程序员使用专用的函数进行申请，如C语言中的malloc等函数、C++中的new函数等都是最常见的分配堆内存的函数。堆内存申请有可能成功，也有可能失败，这与申请内存的大小、机器性能和当前运行环境有关。

（3）.一般用一个堆指针来使用申请的内存，读、写、释放都是通过这个指针来完成。

（4）.使用完毕后要通过堆释放函数进行回收这片内存，否则会造成内存泄漏。如free,delete等

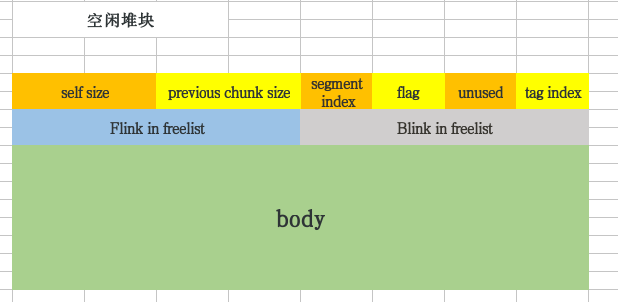
1. 堆数据结构

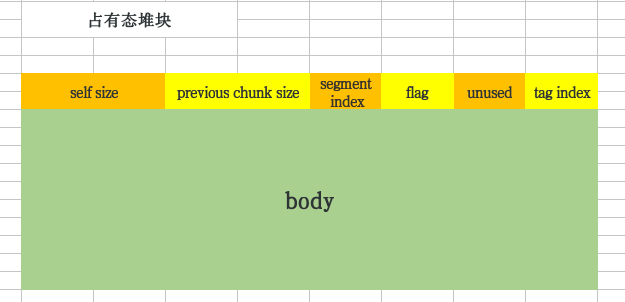
对于管理系统来说，响应程序的内存使用申请就意味着要在**杂乱**的堆区中**辨别**哪些内存是正在被使用的，哪些内存是空闲的，并最终寻找到一片**恰当**的空闲内存区域，以指针形式返回给程序。

1. 堆块：堆区额内存按不同大小组织成块，以堆块为单位进行标识，而不是传统的按字节标识。一个堆块包括两个部分：块首和块身。块首是一个堆块头部的几个字节，用来标识这个块首自身的信息，例如，大小、空闲或占用。块身是紧跟在块首后面的部分，也是最终分配给用户使用的数据区。

注意：块管理系统返回的指针一般是块身的起始位置，连续申请内存就是发现返回的内存之间存在“空隙”，那就是块首。

flag：01使用，10即将使用（尾块）





1. 堆表：堆表一般位于堆区的起始位置，用于检索堆区中所有堆块的总要信息，包括堆块的位置、堆块的大小、空闲或占用等。 堆表的数据结构决定了整个堆区的组织方式。堆表往往不知一种数据结构：如平衡二叉树等。

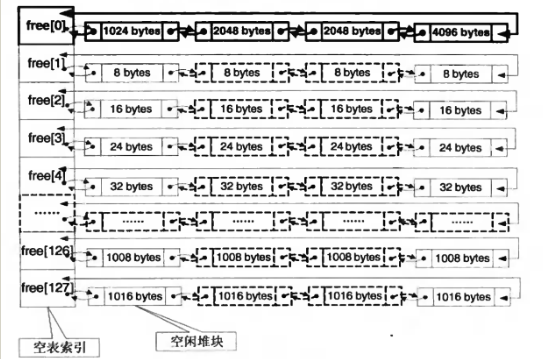
在Windows中占用态的堆被使用它的程序管理，堆表只是管理空闲态的堆块。

1. 堆表类型
2. 空闲双向链表Freelist（空表）

空闲堆块块首包含一对指针，这对指针把空闲堆块组织成双向链表。按照堆块大小的不同，空表总共被分成128条。堆区一开始的堆表区中有一个128项的指针数组，被称作空表索引。该数组每一项包含两个指针，用于标识一条空表。

把空闲堆块链入不同的空表，可以方便管理。空表第一项free[0]链入所有大于等于1024字节的堆块（小于512K）。这些堆块按照各自的大小在零号空表中升序排列。

不足8byte按8byte算（不包括块首）



HLOCAL h1,h2,h3,h4,h5,h6;

HANDLE hp;

hp = HeapCreate(0,0x1000,0x10000);

\_\_asm int 3

h1 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,3);//HEAP\_ZERO\_MEMORY：堆块初始大小为8byte

h2 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,5);

h3 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,6);

h4 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,8);

h5 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,19);

h6 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,24);

//free block and prevent coaleses

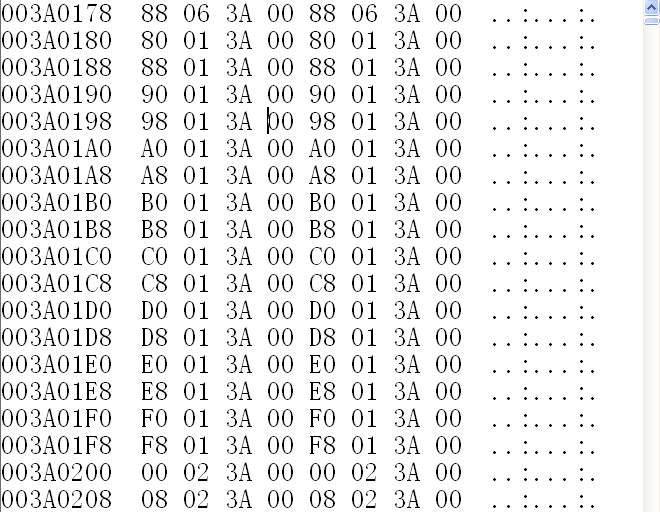
HeapFree(hp,0,h1); //free to free[2]

HeapFree(hp,0,h3); //free to free[2]

HeapFree(hp,0,h5); //free to free[4]

HeapFree(hp,0,h4); //coalese h3,h4,h5,link the large block to free[8]

空闲双向链表free(偏移0x178)



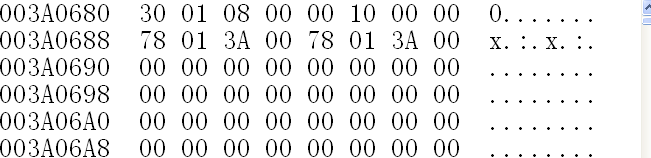
3E0178-->free[0]:3A0688;3A0688

3E0180-->free[1]:3A0180;3A0180

3E0188-->free[2]:3A0188;3A0188

....

free[0]空闲堆块：



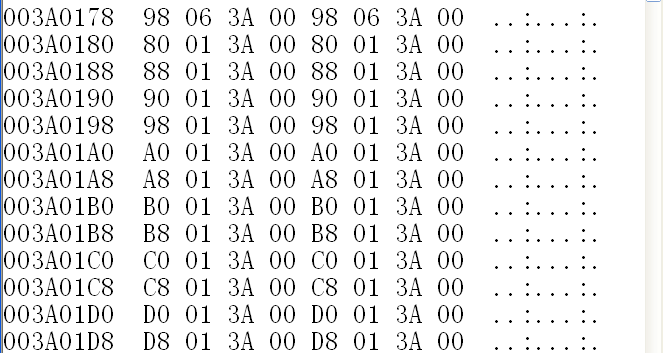
self size：0x130（0x130\*8=0x980字节=2432byte）

flag:10(未使用)

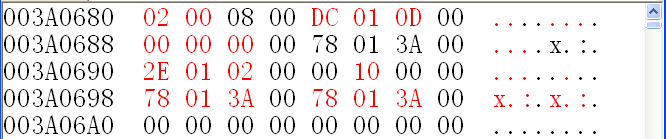
Flink：0x3A0178(free[0])

Blink:0x3A0178(free[0])

申请第一块h1



3E0178-->free[0]:3A0698;3A0698



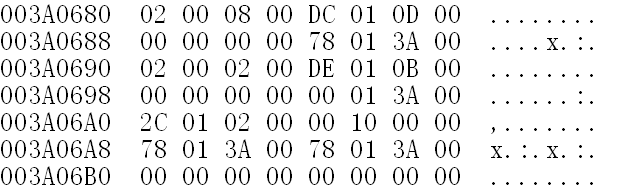
self size：0x12E（0x12E\*8=0x970字节=2416byte）

flag:10(未使用)

Flink：0x3A0178(free[0])

Blink:0x3A0178(free[0])

申请第二块h2



self size：0x128（0x128\*8=0x940字节=2368byte）

flag:10(未使用)

Flink：0x3A0178(free[0])

Blink:0x3A0178(free[0])

申请第三块h3



self size：0x12A（0x12A\*8=0x950字节=2384byte）

flag:10(未使用)

Flink：0x3A0178(free[0])

Blink:0x3A0178(free[0])

申请第四块h4



self size：0x128（0x128\*8=0x940字节=2368byte）

flag:10(未使用)

Flink：0x3A0178(free[0])

Blink:0x3A0178(free[0])

申请第五块h5



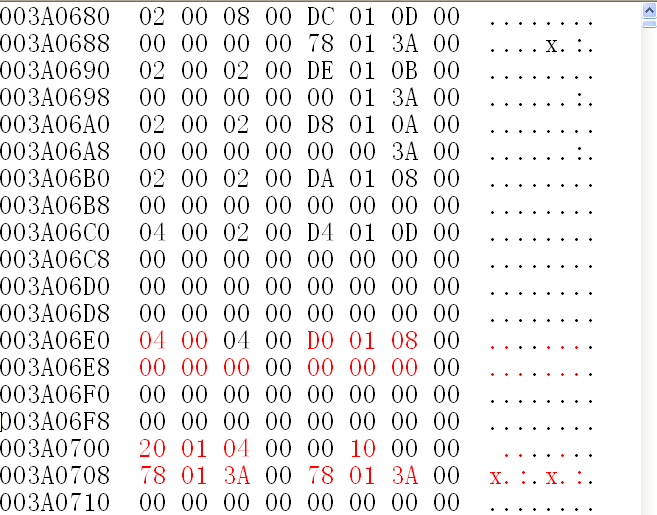
self size：0x124（0x124\*8=0x920字节=2336byte）

flag:10(未使用)

Flink：0x3A0178(free[0])

Blink:0x3A0178(free[0])

申请第六块h6



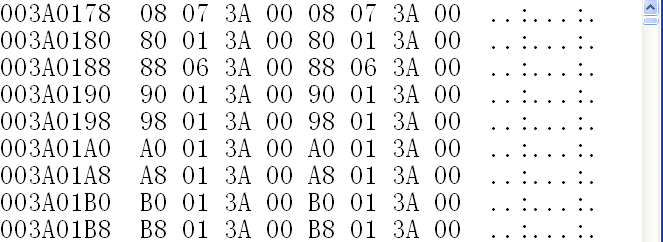
self size：0x114（0x120\*8=0x900字节=2304byte）

flag:10(未使用)

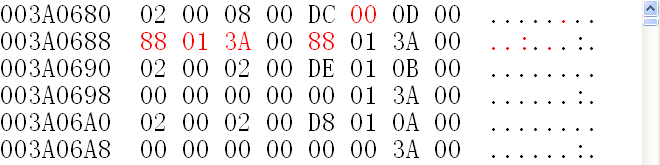
Flink：0x3A0178(free[0])

Blink:0x3A0178(free[0])

释放h1至free[2]



3A0188-->free[2]:3A0688;3A0688

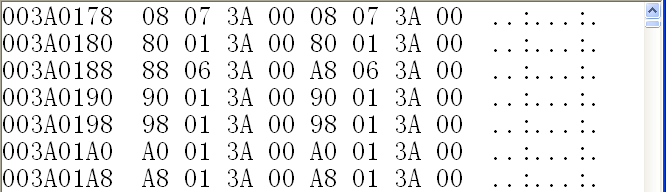


self size：0x2（0x2\*8=0x10字节=16byte）

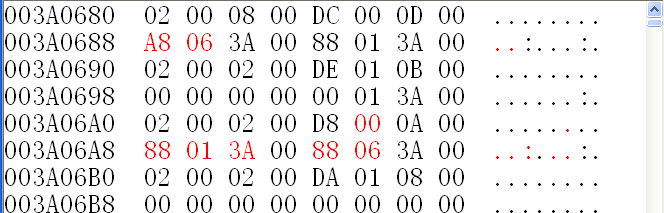
Flink：0x3A0188(free[2])

Blink:0x3A0188(free[2])

释放h3至free[2]



3A0188-->free[2]:3A0688;3A06A8

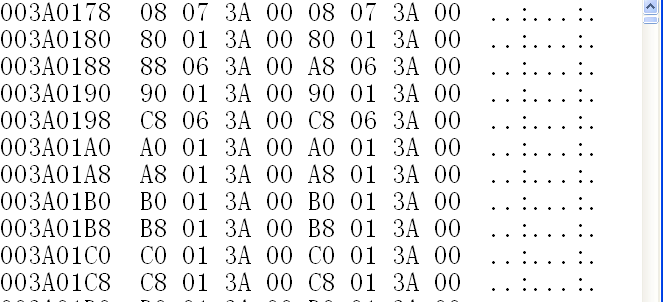


self size：0x2（0x2\*8=0x10字节=16byte）

Flink：0x3A0188(free[2])

Blink:0x3A0688

释放h5至free[4]



3E0198-->free[4]:3A06C8;3A06C8

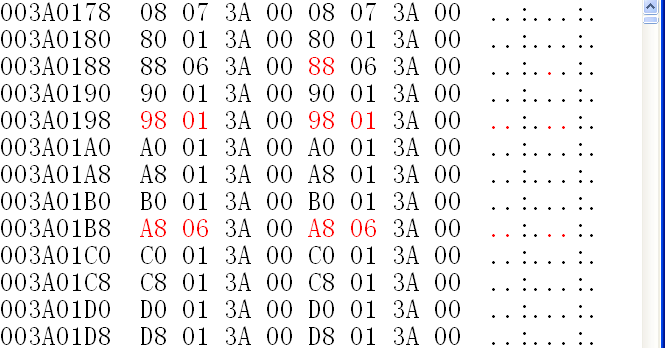


self size：0x4（0x4\*8=0x20字节=32byte）

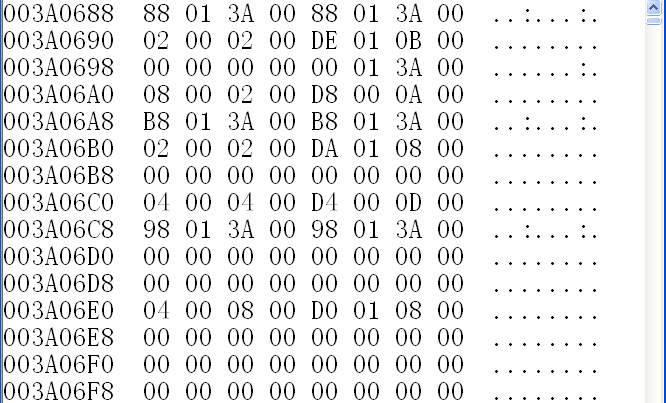
Flink：0x3A0198(free[3])

Blink:0x3A0198(free[3])

释放h4,合并h3，h4，h5至free[8]



3E01B8-->free[8]:3A06A8;3A06A8



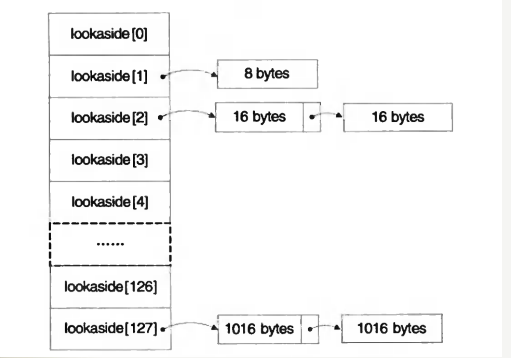
self size：0x8（0x8\*8=0x40字节=64byte）

Flink：0x3A01B8(free[8])

Blink:0x3A01B8(free[8])

1. 快速单项链表Lookaside（快表）

快表是Windows用来加速分配而采用的一种堆表。快表也有128条，组织结构与空表类似，只是堆块按单链表组织，而且每条快表最多只有4个节点。快表空闲块被置为占用态，所以不会发生堆块合并操作。.快表只有精确分配时才会分配。.分配与失败有限使用快表，失败用空表。



HLOCAL h1,h2,h3,h4,h5,h6,h7;

HANDLE hp;

hp = HeapCreate(0,0,0); //堆创建带有快表的

\_\_asm int 3

h1 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,8); //申请内存,以构成快表

h2 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,8);

h3 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,16);

h4 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,24);

HeapFree(hp,0,h1); //释放掉申请的内存，以构成快表（因为快表初始化是空的，而且不会发生堆块合并）

HeapFree(hp,0,h2);

HeapFree(hp,0,h3);

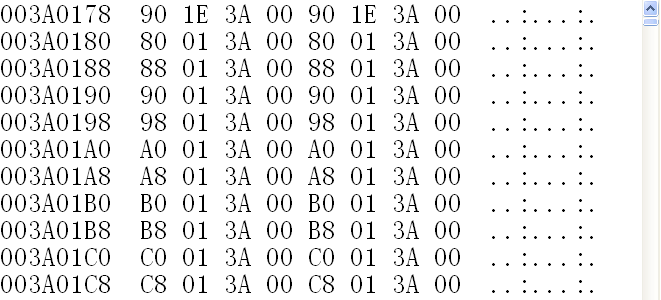
HeapFree(hp,0,h4);

//堆分配顺序的验证

h2 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,16); //再申请,会从快表中分配

HeapFree(hp,0,h2);

快速单项链表



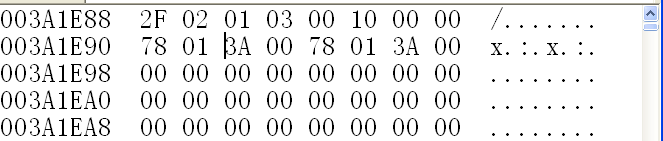
3A0178-->free[0]:3A1E90;3A1E90

3A0180-->free[1]:3A0180;3A0180

3A0188-->free[2]:3A0188;3A0188

...

free[0]空闲堆块

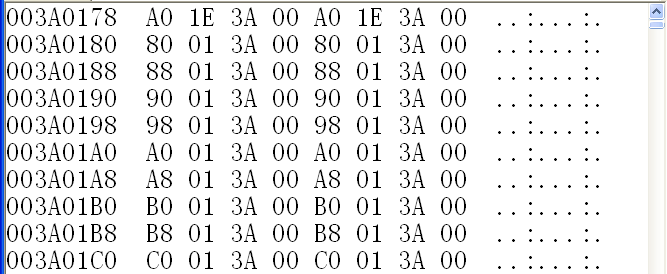


self size：0x22F（0x22F\*8=0x1178字节=4472byte）

Flink：0x3E0178(free[0])

Blink:0x3E0178(free[0])

分配h1



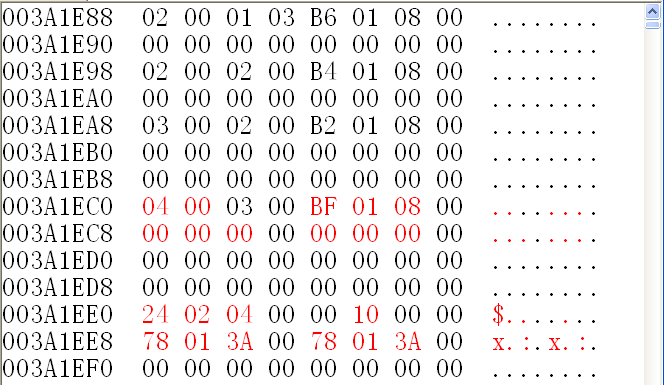
3A0178-->free[0]:3A1EA0;3A1EA0



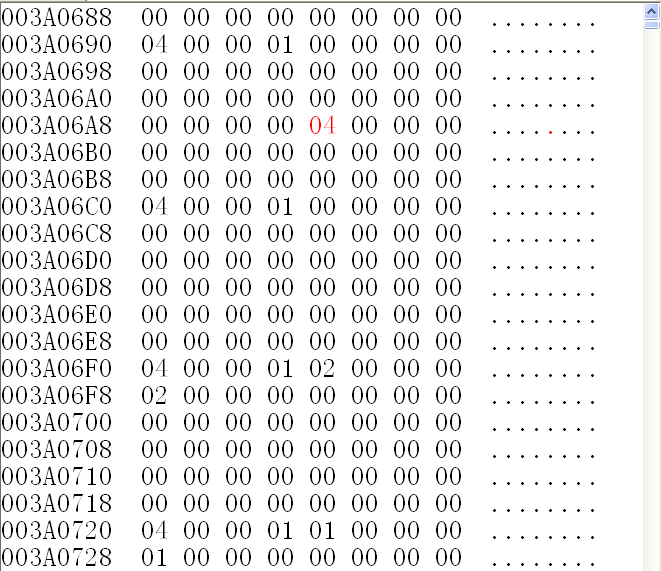
self size：0x22D（0x22D\*8=0x1168字节=4456byte）

link:0x3A0178(free[0])

分配h1,h2,h3,h4



快表（偏移0x688）



3A0688--->lookaside[0]

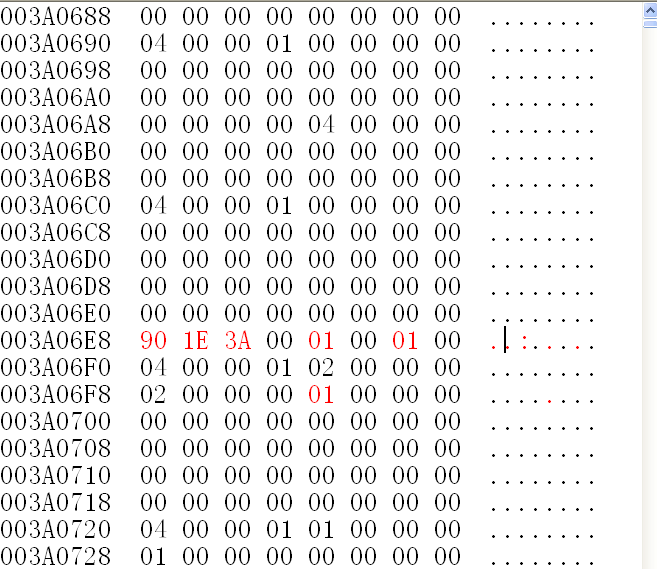
3A06B8--->lookaside[1]

3A06E8-->lookaside[2]

.....

释放h1,h2,h3,h4

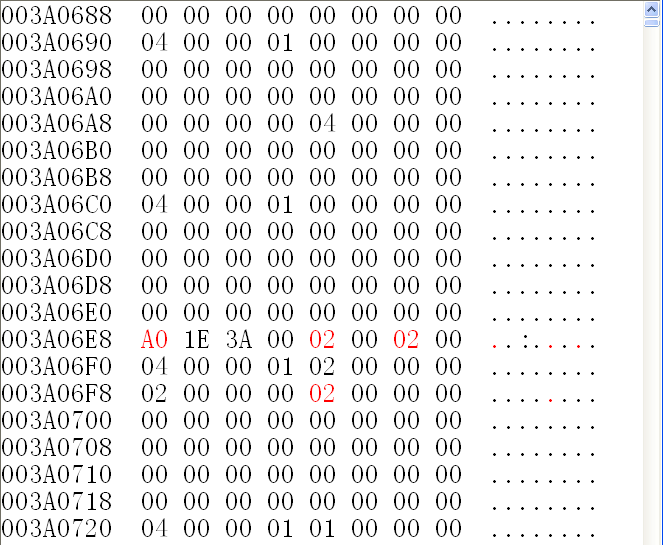
(1)



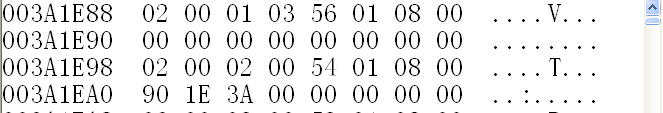
lookaside[2]:0x3A1E90

25

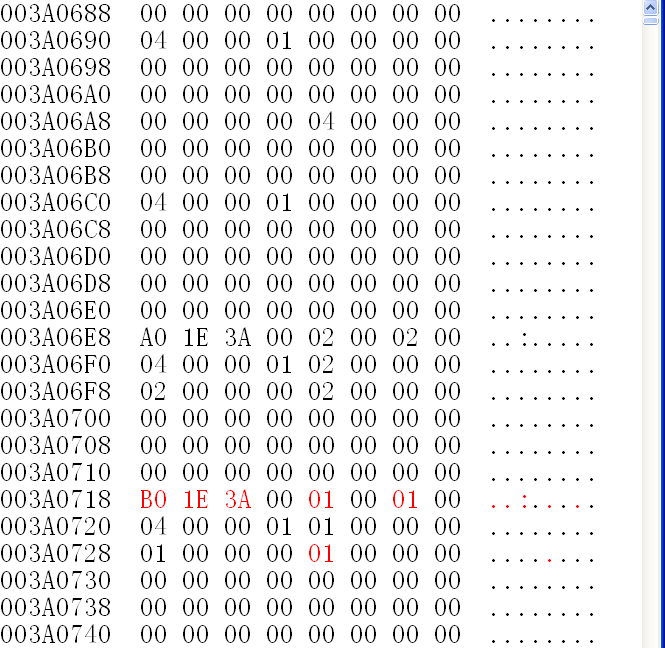
(2)



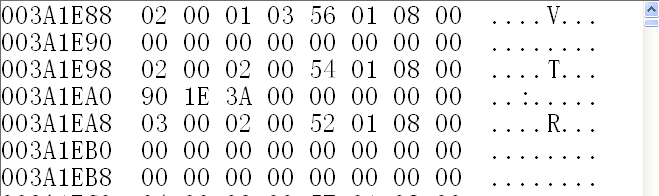
lookaside[2]:3A1EA0



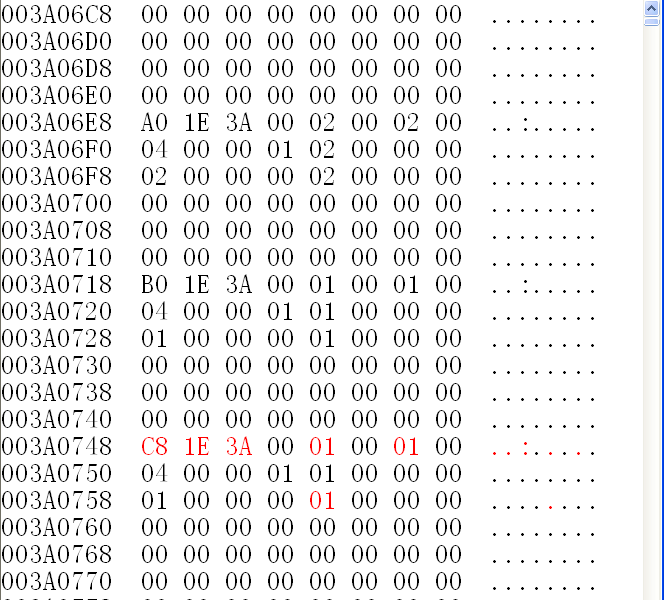
(3)



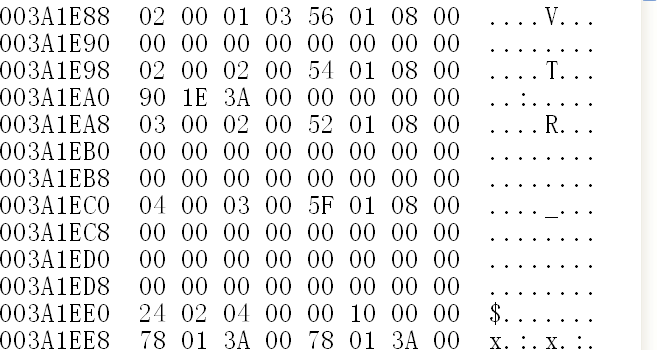
lookaside[3]:3A1EB0



(4)



lookaside[4]:3A1EC8



1. 堆溢出

（1）DWORD SHOOT

堆溢出的利用的精髓就是精心构造的数据溢出下一个堆块的块首，改写块首的前向指针和后向指针，然后在分配、释放、合并等操作发生时获得一次向内存任意地址读写任意数据的机会。

node->blink->flink = node -> flink;

node->flink->blink= node ->blink;

HLOCAL h1, h2,h3,h4,h5,h6;

HANDLE hp;

hp = HeapCreate(0,0x1000,0x10000);

h1 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,8);

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 //used to break the process

//free the odd blocks to prevent coalesing

HeapFree(hp,0,h1);

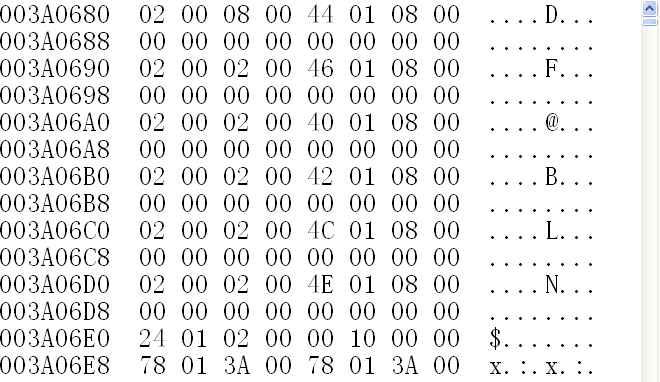
HeapFree(hp,0,h3);

HeapFree(hp,0,h5); //now free[2] got 3 entries

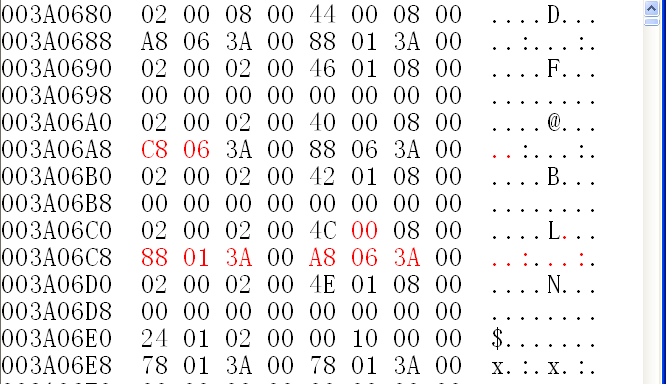
//will allocate from free[2] which means unlink the last entry (h5)

h1 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,8);

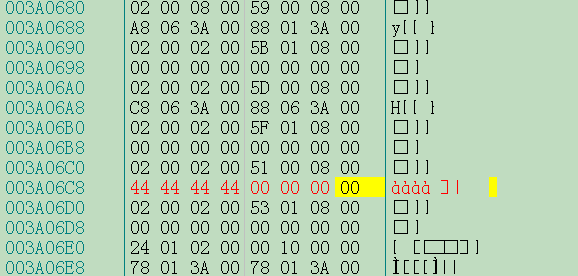
分配h1～h6



释放h1，h3，h5



申请h1，往0x00000000中写入0x44444444



（2）堆溢出可以改变

1.内存变量：修改能影响程序执行的重要标志变量，例如更改身份验证函数的返回值。

2.代码逻辑：修改代码段重要函数关键逻辑，如程序分支处的判断逻辑。

3.函数返回地址：堆溢出也可以利用DWORD　SHOOT更改函数返回地址。

4.攻击异常处理：程序产生异常，Windows转入异常处理机制，包括SEH等。

5.函数指针：如C++的虚函数调用。改写这些指针后，函数调用往往就可以劫持进程。

1. 堆的保护机制

(1)safe Unlink:在堆块拆卸时，先进行验证

未启用前：

int remove (ListNode \*node){

node -> blink -> flink = node -> flink;

node -> flink -> blink = node -> blink;

return 0;

}

启用后:

int remove (ListNode \*node){

if((node -> blink -> flink == node)&&(node -> flink -> blink == node)){

node -> blink -> flink = node -> flink;

node -> flink -> blink = node -> blink;

return 0;

}else{

//异常处理

}

}

(2)heap coookie:与栈的security cookie类似,在堆块的块首中布置security cookies(原来的segment table)

heap.xlsx

(3)元数据加密：堆块的块首一些重要的数据会与一个随机的4位数进行异或运算