《漏洞利用及渗透测试基础》实验报告

林晖鹏 2312966

March 2025

1 实验名称

Dword Shoot 攻击

2 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。

3 实验过程

3.1 实验流程

1. 我们先创建大小为 0x1000 的初始堆，并从其中连续申请 6 个大小为 8 字节的堆（加上块首实际上是 16 字节）。
2. 依次释放第一个第三个第五个堆块。(不释放相邻的是因为防止堆块合并，这样这些释放的堆块大小都为 16，都会放在 Freelist[2] 中。)
3. 释放结束后会生成三个 16 字节的空闲堆块放进 Freelist 中，都是 16 字节，结合前面所学，会放进 Freelist[2] 中。
4. 然后我们再次申请 8 字节的空间，此时会从 Freelist[2] 中选取堆块，就会把之前第一个堆块取下来
5. 取下来的时候会修改 Freelist[2] 上前后堆块的指针指向。
6. 同时也会修改第一个堆块的前后指针，使其指向将要修改数据的内存区域以及存放修改数据的内存区域。

3.2 实验代码解析

```
#include <windows.h>
main()
{
    HLOCAL h1,h2,h3,h4,h5,h6;
    HANDLE hp;
    //0x1000 是初始化堆的大小，0x10000 是堆的最大大小。
    hp=HeapCreate(0,0x1000,0x10000);          //申请初始堆
    h1=HeapAlloc(hp,HEAP_ZERO_MEMORY,8);      //连续申请六个堆块
    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                                //手动加断点
    HeapFree(hp,0,h1);                        //释放 h1,h2,h3
    HeapFree(hp,0,h3);
    HeapFree(hp,0,h5);

    h1=HeapAlloc(hp,HEAP_ZERO_MEMORY,8);      //重新申请空间，观察过程
    return 0;
}

```

3.3 代码执行过程解析

3.3.1 初始化堆并连续申请内存

我们可以看到申请到的地址，h1 ~ h6 每个间隔 16 字节，是连续的。

Name	Value
h6	0x003a0728
h5	0x003a0708
h4	0x003a06e8
h3	0x003a06c8
h2	0x003a06a8
h1	0x003a0688
hp	0x003a0000

图 1: 地址展示

3.3.2 释放前后 h1 堆块的变化

在释放之前，h1 块首的地址为 0x003a0680，我们从下图可以看到前八个字节存的是块首即堆块信息，后面八个字节都为 00，存的是堆块的数据，因为刚刚申请，数据为空。

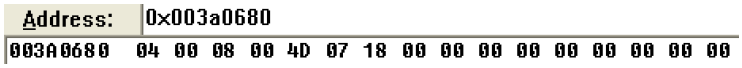

图 2: 释放前 h1 的字节

在释放之后，我们可以看到 h1 的变化：

1. 首先是块首的内容发生变化，修改了堆块的状态信息，表示空闲堆块；
2. 其次是块身的八字节变成了 Freelist 上的前驱指针和后驱指针，因为此时 Freelist[2] 中只有 h1 这个堆块，所以前后指针都指向链表头，即 Freelist[2]，指针值都为 00 98 01 3A。

图 3: 释放后 h1 堆块变化

再者，此时我们来查看这个链表表头的情况：

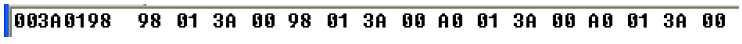

图 4: 链表表头变化前

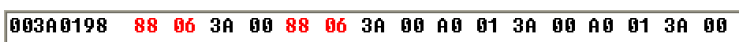

图 5: 链表表头变化后

可以看到，在释放之后，表头的前后指针都指向了 h1。

3.3.3 释放 h1h3h5 后 Freelist 中链表状态

在观察完单个释放的变化之后，其余的释放其实都一样，都是改变前驱后驱指针，将这个堆块插入 Freelist[2] 中。下面我们来观察释放完之后的链表情况：

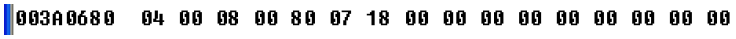

图 6: h1 前

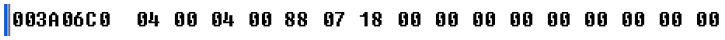

图 7: h2 前

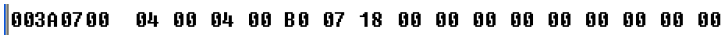

图 8: h3 前

在释放完后，状态变成下面图中所示：

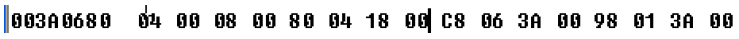

图 9: h1 后

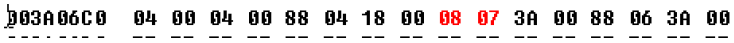

图 10: h2 后

图 11: h3 后

在释放之后，h1、h2、h3 的前驱后驱指针变为：

堆块	前驱指针	后驱指针
h1	C8 06 3A 00	98 01 3A 00
h3	08 07 3A 00	88 06 3A 00
h5	98 01 3A 00	C8 06 3A 00

可以看到 h1 的后驱和 h5 的前驱都指向 h2；h1 的前驱和 h5 的后驱都指向链表头的地址；而 h3 的前后指向 h1 和 h5。

3.3.4 重新申请 h1 后空闲链表状态变化

再重新申请内存之后，会拿 Freelist[2] 中第一个堆块，也就是原本 h1 的堆块。此时链表头后驱变成指向 h3 原本的堆块；h3 原本的堆块变成指向链表头。而申请出来的堆块，也就是 h1 原本的堆块，堆首的状态改变，堆身的内存初始化为 0：

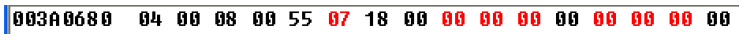

图 12: 重新申请 h1

3.4 Dword Shoot 攻击思路

本质上，这个调用空闲堆的时候，就是把后驱指向的写到前驱的地址处，如果我们更改前后驱的指针，这样就能把修改后后驱指向的地址里的数据写到前驱指向的地址，得到一个任意写入数据的机会。

4 Dword Shoot 攻击尝试

下面我们尝试用 OllDbg 修改，试着把 1111 写到 h2 的堆身里面：可以将 h1 的前驱改成 h2 的地址，然后后驱改成 [11111111]。

尝试失败。一开始打算在 OllDby 里面修改它的 mov 的值，但是愣是单步遍历汇编代码半天，找不到书中的那个调换的代码在哪。然后我尝试先在 VC6 里面找修改值的地方大概在哪里，由于没有学过汇编，最后看晕了也没找着。

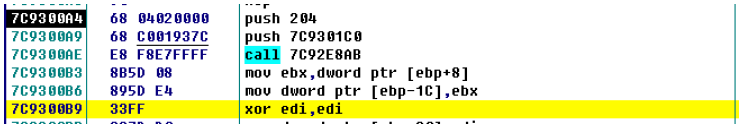

图 13: 在这个 call 调用里面，然后进去好多层（晕）

5 心得体会

- 掌握了堆的申请和释放的底层流程，同时在这个过程中，进一步熟悉用 VC6 进行反汇编代码分析。

- 尝试用 OlyDbg 来进行 Dword shoot 攻击，虽然失败了，但是在过程中不仅熟悉了 OlyDbg 的使用，还深入思考了 Dword Shoot 的原理并进行设计。