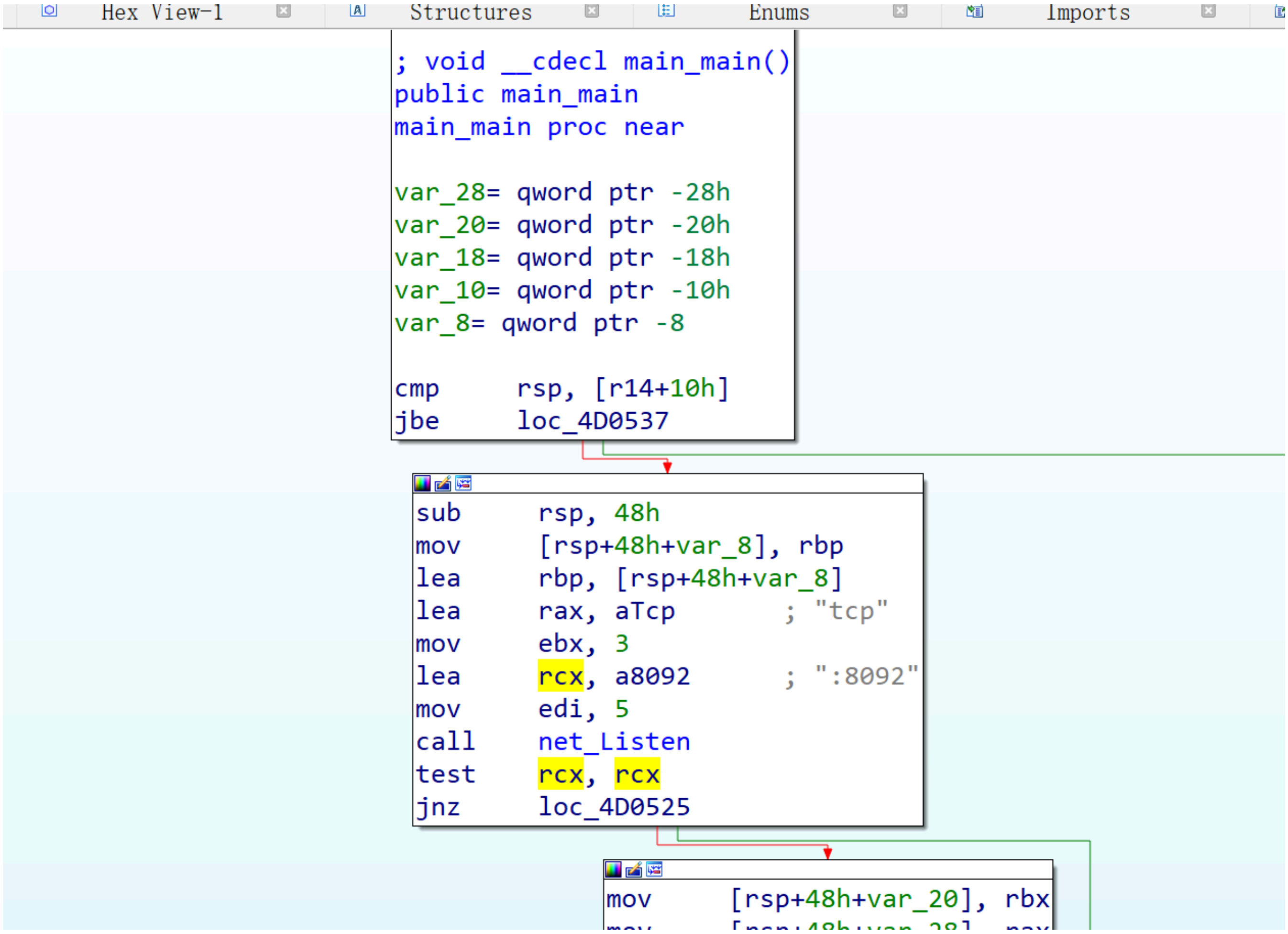
1. 查看main函数，发现调用了net_Listen函数并且参数为“tcp”和“:8082“，可以推测出该题目监听了本地的8082端口用来接收tcp连接。




2. 接下来调用了函数runtime_newproc，参数为函数 main_main_func1，可以推测是新建了goroutine来运行函数main_main_func1。

```
mov     [rsp+48h+var_10], rax
mov     [rsp+48h+var_18], rbx
lea     rax, unk_4E5520
call    runtime_newobject
lea     rcx, main_main_func1
mov     [rax], rcx
mov     rdx, [rsp+48h+var_10]
mov     [rax+8], rdx
cmp     dword ptr cs:runtime_writeBarrier.enabled, 0
jnz     short loc_4D0504
```

```
mov     rdx, [rsp+48h+var_18]
mov     [rax+10h], rdx
jmp     short loc_4D04AA
```

```
loc_4D0504:
lea     rdi, [rax+10h]
mov     rdx, [rsp+48h+var_18]
call    runtime_gcWriteBarrierDX
jmp     short loc_4D04AA
```

```
loc_4D04AA:
call    runtime_newproc
mov     rax, [rsp+48h+var_28]
mov     rbx, [rsp+48h+var_20]
```

3. main_main_func1函数中调用了main_handle_connection函数，在此函数中首先调用了ReadBytes，又调用了main_handle_input函数，最后会输出字符串"You win"，所以主要判断逻辑应该是在函数main_handle_input中。

```
while ( 1LL )
{
  bufio___Reader__ReadBytes(*(__uint8 *)v10, v18, v20, v22);
  if ( io_EOF.tab == (runtime_itab_0 *)v2 VALUE MAY BE UNDEFINED; _BYTE
  {
    v23.len = v7;
    v23.array = (interface_{}_0 *)10LL;
    runtime_ifaceeq((bool)v12, v15, v17, v19.tab);
    if ( v8 )
      break;
  }
  main_handle_input();
  runtime_convI2I(v12, (runtime_interfacetype_0 *)v15, v17);
  v2 = 8LL;
  fmt_Fprintf(v11, v14, v19, v21, v23);
}
```

```
loc_4D0382:
call    main_handle_input
lea     rax, unk_4E3520
mov     rbx, [rsp+120h+var_C0]
call    runtime_convI2I
mov     rbx, [rsp+120h+conn.data]
lea     rcx, aYouWin    ; "You win\n"
mov     edi, 8
xor     esi, esi
xor     r8d, r8d
mov     r9, r8
call    fmt_Fprintf
mov     rcx, [rsp+120h+var_D8]
```

4. main_handle_input函数中调用了handle_tcp_Unpack_tcp_reply函数，并将结果写到了[rsp+220h+var_1A0+offset]中。

```
call    letsGO_interal_handle_tcp_Unpack_tcp_reply
mov     word ptr [rsp+220h+var_1A0], ax
mov     word ptr [rsp+220h+var_1A0+2], bx
mov     dword ptr [rsp+220h+var_1A0+4], ecx
mov     dword ptr [rsp+220h+var_1A0+8], edi
mov     word ptr [rsp+220h+var_1A0+0Ch], si
mov     word ptr [rsp+220h+var_1A0+0Eh], r8w
```

5. main_handle_input之后以"127.0.0.1"为参数调用了两次handle_tcp_Ip2int函数，handle_tcp_Ip2int函数中对字符串进行了一些分割、Atoi和位移操作，实现的功能就是将ip字符串以字符"."分割然后转化为数字。

```
mov     [rsp+220h+var_60], rbx
mov     [rsp+220h+var_1D8], rax
lea     rax, a127001    ; "127.0.0.1"
mov     ebx, 9
call    letsGO_interal_handle_tcp_Ip2int
mov     [rsp+220h+var_1EC], eax
mov     ebx, 9
lea     rax, a127001    ; "127.0.0.1"
call    letsGO_interal_handle_tcp_Ip2int
mov     rcx, [rsp+220h+var_1C8]
mov     rdx, [rsp+220h+var_1D8]
sub     rcx, rdx
f_raw = rcx             ; __uint8
```

```
    runtime_morestack_noctxt();
  strings_genSplit(v6, v9, v11, v13, v15);      // "."
  v14 = v0;
  v3 = 0LL;
  v4 = 0;
  while ( v0 > v3 )
  {
    v16 = v3;
    v12 = v4;
    v17 = v2;
    strconv_Atoi(v7, v8, v10);
    if ( ((3LL - v16) & 0x1000000000000000LL) != 0LL )
      runtime_panicshift();
    v4 = v12 + ((unsigned int)(8LL * (3LL - v16)) < 0x20LL ? v5 << (8 * (3 - v16)) : 0LL);
    v0 = v14;
    v3 = v16 + 1LL;
    v2 = v17 + 16LL;
  }
}
```

6. main_handle_input接下来构造了tcp pseudo header用来计算checksum，计算checksum之后过了一系列判断，以此可以推算出它接收的tcp包中header的各个参数: lport和rpotr分别为99和233、seqnum为0xdeadbeef、标志位为2（syn）其他按照标准tcp包来构造。

```
  letsGO_interal_handle_tcp_Pack_tcp_pseudo_header();
  letsGO_interal_handle_tcp_Calculate_checksum(v24, v30);
  if ( v13 != v40 )
    goto LABEL_30;                          // checksum err
  if ( v45 != 0xE90063 )
    goto LABEL_29;                          // port err
  if ( (_QWORD)v46 != 0xDEADBEEFLL )
    goto LABEL_28;                          // num err
  if ( (BYTE8(v46) & 63LL) != 2 )
    goto LABEL_27;                          // flag err
  if ( v41 < 4LL )
    goto LABEL_26;                          // data too short
```

7. 使用python写一份简单的交互脚本来测试，脚本可以构造tcp包并发送：

```python
import struct
def ip2int(ip):
    ip = ip.split('.')
    result = 0
    result = result + int(ip[3])
    result = result + (int(ip[2]) << 8)
    result = result + (int(ip[1]) << 16)
    result = result + (int(ip[0]) << 24)
    return result

def int2ip(i):
    i = hex(i)[2:]
    i = i.rjust(len(i) + (len(i) % 2), '0')
    result = ''
    result = result + str(int(i[:2], 16)) + '.'
    result = result + str(int(i[2:4], 16)) + '.'
    result = result + str(int(i[4:6], 16)) + '.'
```

```python
    result = result + str(int(i[6:8], 16))
    return result

def pack_tcp_pseudo_header(data, laddr, raddr):
    pseudo = struct.pack(
        '!IIBBH',
        laddr,
        raddr,
        0,
        6,
        len(data)
    ) + data
    if len(pseudo)%2 !=0:
        pseudo += b'\x00'
    return pseudo
def calculate_checksum(tcp):
    highs = tcp[0::2]
    lows = tcp[1::2]
    checksum = ((sum(highs) << 8) + sum(lows))
    while True:
        carry = checksum >> 16
        if carry:
            checksum = (checksum & 0xffff) + carry
        else:
            break
    checksum = ~checksum & 0xffff
    return checksum
def pack_tcp_request(data):
    lport = 99
    rport = 233
    seqnum = 0xdeadbeef
    acknum = 0
    flag = 0x8002
    window = 0xffff
    checksum = 0
    urgentpointer = 0
    option = [0x02, 0x04, 0x05, 0xb4, 0x00, 0x00, 0x00, 0x00, 0x00 ,0x00 ,0x00 ,0x00]
    tcp_pack = struct.pack(
        "!HHIIHHHH",
        lport,
        rport,
        seqnum,
        acknum,
        flag,
        window,
        checksum,
        urgentpointer
    ) + bytes(option)
    laddr = ip2int("127.0.0.1")
```

```python
    raddr = ip2int("127.0.0.1")
    tcp_pack +=data
    checksum = calculate_checksum(pack_tcp_pseudo_header(tcp_pack,laddr,raddr))
    print(checksum)
    tcp_pack = struct.pack(
        "!HHIIHHHH",
        lport,
        rport,
        seqnum,
        acknum,
        flag,
        window,
        checksum,
        urgentpointer
    ) + bytes(option) + data
    return tcp_pack
if __name__ == "__main__":
    payload = pack_tcp_request(b'test')
    print(payload)
    from pwn import *
    context.log_level = "debug"
    sh = remote("127.0.0.1", 8092)
    sh.sendline(payload)
    sh.interactive()
```

运行题目之后再运行脚本的到如下报错：

```
ubuntu@VM-16-6-ubuntu:~$ ./letsGO
panic: [*] length err!

goroutine 18 [running]:
main.handle_input({0xc0000c2000, 0x25, 0x25})
        /Users/uuuuu/Desktop/WorkSpace/MkPuzzles/Ichunqiu/Drafts/letsGO/main.go:67 +0x545
main.handle_connection({0x521e50?, 0xc00009c038})
        /Users/uuuuu/Desktop/WorkSpace/MkPuzzles/Ichunqiu/Drafts/letsGO/main.go:84 +0x187
created by main.main
        /Users/uuuuu/Desktop/WorkSpace/MkPuzzles/Ichunqiu/Drafts/letsGO/main.go:100 +0x4f
```

说明程序运行到了最后的比对阶段：

```c
    if ( data.len != 28LL )
      goto LABEL_24;                          // length err!
    for ( result = 0LL; result < 28LL; ++result )
    {
      if ( res[result] != data.array[result] )
      {
```

8. 通过最后的比对阶段，可以定位到比对数据res和经过处理的输入数据data，在对data溯源之前首先来科普一下golang中数组的存储方式和参数存储方式，golang数组会以一个18bytes大小的结构体进行存储，域中分别是指向真正数据的array指针、代表当前用到的数组长度len和数组的总共可用长度cap。然后golang的参数一般都会保存在rbp+正offset的位置，所以可以直接通过查看main_handle_input函数的栈桢定位我们的input数据。

```
000000008 input          dq ?
000000010 input_len      dq ?
000000018 input_cap      dq ?
```

9. 溯源data来源，可以发先data由data0赋值，而data0是input加上了某些进行了奇怪的运算的出来的偏移。

```
data_length = input_cap - head_size;
data_0 = (char *)(input + (head_size & ((int)(head_size - input_cap) >> 63LL)));
data_length_too = v2 - head_size;
_40b.data = (void *)(data_length_too - 1LL);
runtime_convTslice(v22, v29);
encoding_binary_Write(v23, v33, v36, _40b);
if ( v54[3LL] > (unsigned int)v54[1LL] )
  goto LABEL_31;
v53 = *v54;
letsGO_interal_handle_tcp_Ip2int();
letsGO_interal_handle_tcp_Ip2int();
letsGO_interal_handle_tcp_Pack_tcp_pseudo_header();
letsGO_interal_handle_tcp_Calculate_checksum(v24, v30);
if ( v13 != v40 )
  goto LABEL_30;                          // checksum err
if ( v45 != 0xE90063 )
  goto LABEL_29;                          // port err
if ( (_QWORD)v46 != 0xDEADBEEFLL )
  goto LABEL_28;                          // num err
if ( (BYTE8(v46) & 63LL) != 2 )
  goto LABEL_27;                          // flag err
if ( v41 < 4LL )
  goto LABEL_26;                          // data too short
sub_462F74();
s.array = (uint8 *)&v50;
s.len = 256LL;
s.cap = 256LL;
key.array = (uint8 *)&data_0[(data_length_too - 4LL) & (-(data_length - data_length_too + 4LL) >> 63LL)];
key.len = 3LL;
key.cap = data_length - data_length_too + 4LL;
data.array = (uint8 *)data_0;
data.len = v14;
data.cap = data_length;
```

10. 溯源head_size，发现它是由tcp包中的flag的前四个bit（tcp包头部中的偏移）乘以4得来的，也就是tcp包头部的总长度。所以head_size-input_cap为负数，而负数向右位移63位就基本全是1了，所以data0=input+(head_size&0xffffffff)，也就是data0=input+head_size，所以data就是tcp包头部数据后面的所有数据。

```
WORD1(v37._type) = v6;
head_size = (unsigned __int16)(4 * (flag >> 12LL));
option = input + (((int)(20LL - input_cap) >> 63LL) & 20LL);
runtime_newobject(v17, v26);
v54 = v12;
runtime_convTnoptr(v18, v27, v31.tab);
encoding_binary_Write(v19, v31, v34, v37);
runtime_convTslice(v20, v28);
encoding_binary_Write(v21, v32, v35, v38);
data_length = input_cap - head_size;
data_0 = (char *)(input + (head_size & ((int)(head_size - input_cap) >> 63LL)));
data_length_too = v2 - head_size;
```

11. 溯源data.len（ida的伪代码没有正确识别），发现是从rdx寄存器传过来的，而rdx是从[rsp+220h+var_1E8]赋值过来的，而[rsp+220h+var_1E8]的值是input_length-head_size-1也就是data0的长度-1，之所以-1是因为data0的最后一个字符为换行。

```
mov     rdx, qword ptr [rsp+220h+var_1E8]
nop     word ptr [checksum_calculate+checksum_calculate+00000000h]
nop
cmp     rdx, 4
jl      loc_4D0138
```

```
lea     rdi, [rsp+220h+var_170]
nop     word ptr [checksum_calculate+checksum_calculate+00000000h]
nop     dword ptr [checksum_calculate+checksum_calculate+00h]
mov     [rsp+220h+var_230], rbp
lea     rbp, [rsp+220h+var_230]
call    sub_462F74
mov     rbp, [rbp+0]
lea     rsi, [rsp+220h+var_170]
mov     [rsp+220h+s.array], rsi
mov     [rsp+220h+s.len], 100h
mov     [rsp+220h+s.cap], 100h
mov     rsi, [rsp+220h+data_length]
mov     r8, [rsp+220h+head_size]
mov     r9, rsi
sub     rsi, r8
add     rsi, 4
mov     r10, rsi
neg     rsi
sar     rsi, 3Fh
add     r8, 0FFFFFFFFFFFFFFFCh
and     rsi, r8
mov     r8, [rsp+220h+data_0]
add     rsi, r8
mov     [rsp+220h+key.array], rsi
mov     [rsp+220h+key.len], 3
mov     [rsp+220h+key.cap], r10
mov     [rsp+220h+data.array], r8
mov     [rsp+220h+data.len], rdx
```

FF91 00000000004CFF91: main_handle_input+3B1

```
mov     rcx, [rsp+220h+input_cap]
mov     rdx, [rsp+220h+head_size]
sub     rcx, rdx
raw_data_0 = rcx          ; __uint8
mov     [rsp+220h+data_length], raw_data_0
mov     rbx, raw_data_0
neg     raw_data_0
sar     rcx, 3Fh
and     rcx, rdx
mov     rsi, [rsp+220h+input]
lea     rax, [rsi+rcx]
mov     [rsp+220h+data_0], rax
mov     rcx, [rsp+220h+input_len]
sub     rcx, rdx
mov     [rsp+220h+head_size], rcx
lea     rdx, [rcx-1]
raw_data_length_0 = rdx ; __int64
mov     qword ptr [rsp+220h+var_1E8], raw_data_length_0 ;
```

12. 依照上面的经验可以看出key.array = &data0[data_length_too - 4]（这个data_length_too变量就是上图中的[rsp+220h+head_size]），即data0中的后四个比特，而key.length=3代表key只使用前三个字节（最后一个字节为换行符）

```
key.array = (uint8 *)&data_0[(data_length_too - 4LL) & (-(data_length - data_length_too + 4LL) >> 63LL)];
key.len = 3LL;
key.cap = data_length - data_length_too + 4LL;
```

13. 在初始化了长度为256的数组s和长度为3的数组k之后，函数又调用了两个加密函数，特征很明显，一眼就能看出来是rc4的init和crypt。

```
key.array = (uint8 *)&data_0[(data_length_too - 4LL) & (-(data_length - data_length_too + 4LL) >> 63LL)];
key.len = 3LL;
key.cap = data_length - data_length_too + 4LL;
data.array = (uint8 *)data_0;
data.len = v14;
data.cap = data_length;
for ( i = 0LL; i < v14; ++i )
{
  if ( (unsigned __int8)(data_0[i] - 32) > 0x5EuLL )
    goto LABEL_25;
}
letsGO_interal_handle_enc_Enc1(v25._type, v25.data, v37);
letsGO_interal_handle_enc_Enc2(v26, v33, v38);
```

Enc1:

```
for ( i = 0LL; i < 256LL; i = v8 + 1LL )
{
  if ( i >= (unsigned int)v5[1LL] )
    runtime_panicIndex();
  *(_BYTE *)(*v5 + i) = i;
  if ( !v6 )
    runtime_panicdivide();
  v8 = i;
  v9 = i % v6;
  if ( v9 >= v3[1LL] )
    runtime_panicIndex();
  v15[v8] = *(_BYTE *)(v9 + *v3);
}
v10 = 0LL;
v11 = 0;
while ( v10 < 256LL )
{
  v12 = v5[1LL];
  v13 = *v5;
  if ( v10 >= v12 )
    runtime_panicIndex();
  v14 = *(_BYTE *)(v10 + v13);
  v11 += v15[v10] + v14;
  if ( v12 <= v11 )
    runtime_panicIndex();
  *(_BYTE *)(v13 + v10) = *(_BYTE *)(v11 + v13);
  if ( v5[1LL] <= (unsigned int)v11 )
    runtime_panicIndex();
  *(_BYTE *)(*v5 + v11) = v14;
  ++v10;
}
```

Enc2:

```
while ( v4 > (int)v6 )
{
  ++v7;
  v9 = v3[1LL];
  v10 = *v3;
  if ( v9 <= v7 )
    runtime_panicIndex();
  v11 = *(_BYTE *)(v10 + v7);
  v8 += v11;
  if ( v9 <= v8 )
    runtime_panicIndex();
  *(_BYTE *)(v10 + v7) = *(_BYTE *)(v10 + v8);
  if ( v3[1LL] <= (unsigned int)v8 )
    runtime_panicIndex();
  *(_BYTE *)(*v3 + v8) = v11;
  v12 = v3[1LL];
  v13 = *v3;
  if ( v12 <= v7 )
    runtime_panicIndex();
  if ( v12 <= v8 )
    runtime_panicIndex();
  v14 = *(_BYTE *)(v13 + v8) + *(_BYTE *)(v13 + v7);
  if ( v6 >= v5[1LL] )
    runtime_panicIndex();
  if ( v12 <= v14 )
    runtime_panicIndex();
  *(_BYTE *)(*v5 + v6) = *(_BYTE *)(v13 + v14) ^ *(_BYTE *)(v6 + *v5);
  ++v6;
}
```

15. 加密完成之后，我们数据长度要为28，然后和res数组进行比较，res的值已经写在函数开头了。所以大概的逻辑就是给指定端口发送一个tcp包，包含了数据data，以data的后三个字节为密钥对data进行rc4加密，加密出来的结果就是res。

```
if ( data.len != 28LL )
  goto LABEL_24;                              // length err!
for ( result = 0LL; result < 28LL; ++result )
{
  if ( res[result] != data.array[result] )
  {
```

16. 有了逻辑就可以编写脚本解题了，三个字节的密钥不是很长，可以将res数据提取出来后进行爆破。脚本如下：

```python
from Crypto.Cipher import ARC4

enc = [6, 116, 180, 226, 73, 13, 145, 54, 149, 157, 122, 254, 199, 169, 164, 161, 240, 246, 3, 86, 144, 250, 26, 50, 167, 109, 57, 238]
key = 0
for i in range(16777215):
    key = i.to_bytes(3, "little")
    rc4 = ARC4.new(key)
    m = rc4.decrypt(bytes(enc))
    # print(key)
    if m[-3:] == key:
        print(b"target is : "+key)
        print(b"after decrypt: "+m)
```

运行完成之后有两个结果。

```
❯ python3 crack.py
b'target is : BQ*'
b'after decrypt: \x93\xff9r\xd6\xcb\xa9i&n*\xfb)\xf5\x17\xc0\xbca\xb3\xf1\x95(\xcb\xa0LBQ*'
b'target is : rd}'
b'after decrypt: flag{Go_1an9_1s_n07_s0_Hard}'
```

仔细观察一下源程序发现还有一个限制，即data必须是可见字符。所以即可得出正确的flag

```c
data.cap = data_length;
for ( i = 0LL; i < v14; ++i )
{
  if ( (unsigned __int8)(data_0[i] - 32) > 0x5EuLL )
    goto LABEL_25;
}
```

17. 用交互脚本验证

将打包的数据改成flag的值，将题目运行起来后运行脚本。

```python
payload = pack_tcp_request(b'flag{Go_1an9_1s_n07_s0_Hard}')
print(payload)
from pwn import *
context.log_level = "debug"
sh = remote("127.0.0.1", 8092)
sh.sendline(payload)
sh.interactive()
```

打印出You win，验证成功。

```
❯ python3 exp.py
30840
b'\x00c\x00\xe9\xde\xad\xbe\xef\x00\x00\x00\x00\x80\x02\xff\xffxx\x00\x00\x02\x04\x05\xb4\x00\x00\x00\x00\x00\x00\x00\x00flag{Go_1an9_1s_n07_s0_Hard}'
[+] Opening connection to 127.0.0.1 on port 8092: Done
[DEBUG] Sent 0x3d bytes:
    00000000  00 63 00 e9  de ad be ef  00 00 00 00  80 02 ff ff  │·c··│····│····│····│
    00000010  78 78 00 00  02 04 05 b4  00 00 00 00  00 00 00 00  │xx··│····│····│····│
    00000020  66 6c 61 67  7b 47 6f 5f  31 61 6e 39  5f 31 73 5f  │flag│{Go_│1an9│_1s_│
    00000030  6e 30 37 5f  73 30 5f 48  61 72 64 7d  0a           │n07_│s0_H│ard}│·│
    0000003d
[*] Switching to interactive mode
[DEBUG] Received 0x8 bytes:
    b'You win\n'
You win
$
```