CM32M4xxR PMP Privacy Protection隐私保护应用样例说明

Create Time	@July 21, 2021 6:49 PM
End Time	
Start Date	@July 21, 2021
Tags	2021年7月 已完成

0.功能说明

该应用样例展示了如何利用PMP的内存隔离特性，

通过本样例将展示如何进入用户模式，配置用户模式访问权限、用户栈及系统栈的切换及通过系统调用的方式进行实现隐私API的访问；

1.运行环境

环境

Name	版本
软件开发环境	NucleiStudio
Untitled	

2.外设资源

资源

Name	配置	描述
串口	UART5（TX-PE8, RX-PE9）	115200-8-1-n

3.功能描述

该应用例程中存在一处关键数据及运算，位于privacy.c中；该文件主要实现了敏感数据的加密功能，通常部分内容特别是加密密钥为隐私数据，应不被第三方应用或其他开发者访问；

为了更好的展示对隐私数据的保护及调用方式，特此设计了用户接口模块user_app.c；该模块中共实现了两种调用方式：

- 直接调用，系统启动后，输入数字‘0’触发；

💡 在用户代码中直接调用隐私保护加密api；用于隐私保护api在特定的地址区域，并被PMP保护，因此直接访问模式将出发PMP异常；

- 系统调用，系统启动后，输入数字'1'触发；

💡 通过系统调用触发异常，实现特权模式的变更；

为更好的展示RISC-V特权模式及PMP特性，本实例在系统启动后，将切换至UserMode运行用户代码，同时为了更好的保护隐私数据不出现泄露，特此将系统栈及用户栈进行隔离，在UserMode下使用用户栈处理；同时异常进入MachineMode时，进行堆栈的切换；

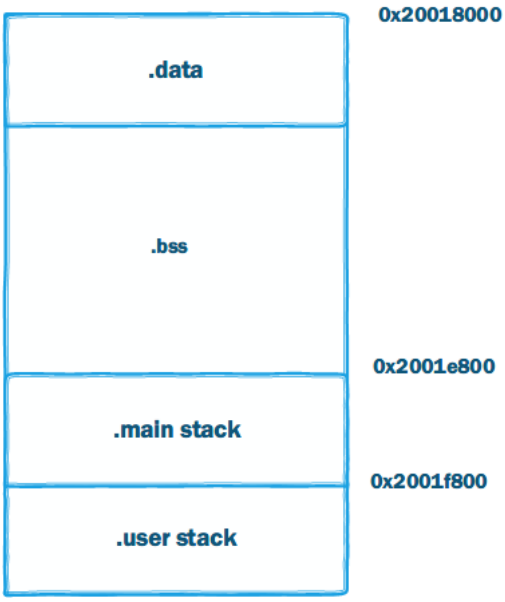
运行日志如下：

```
PMP Example Start!!!!!!

Please enter either 0 or 1 to select the call mode, 0 for direct call and 1 for API call !!!!!!

Enter 0!!!!!!

enter user app

enter user_app_direct

PMP Exception Occures Run at 0x20012010, Fetching Memory at 0x20012010


PMP Example Start!!!!!!

Please enter either 0 or 1 to select the call mode, 0 for direct call and 1 for API call !!!!!!

Enter 1!!!!!!

enter user app

enter user_app_api!

PMP Exception Occures Run at 0x20011028, Enter System Call !!!

Enter Privacy System CALL

user_app_api success!
```

4.示例设计说明

4.1 堆栈隔离

虽然隐私保护逻辑可实现采用隔离的方式，通过系统调用间接访问加密密钥，阻止用户直接调用，获取隐私内容；但在系统调用过程中隐私内容依旧可能通过栈中留有痕迹，为了进一步的降低隐私泄漏的风险，特此将系统栈及用户栈进行隔离，将系统栈的访问权限设置为None;




```
MEMORY
{
  ...
  main_stack(wxa!ri): ORIGIN = 0x2001e800, LENGTH = 0x800
  user_stack(wxa!ri): ORIGIN = 0x2001f800, LENGTH = 0x800
}
.mainstack :
  {
    . = ALIGN(4);
    *(.mainstack)
    *(.mainstack.*)
    . = ALIGN(4);
  } > main_stack
```

为了方便在特权模式变更时切换栈指针，在进入用户模式前，将系统栈指针置于mscratch暂存寄存器中；

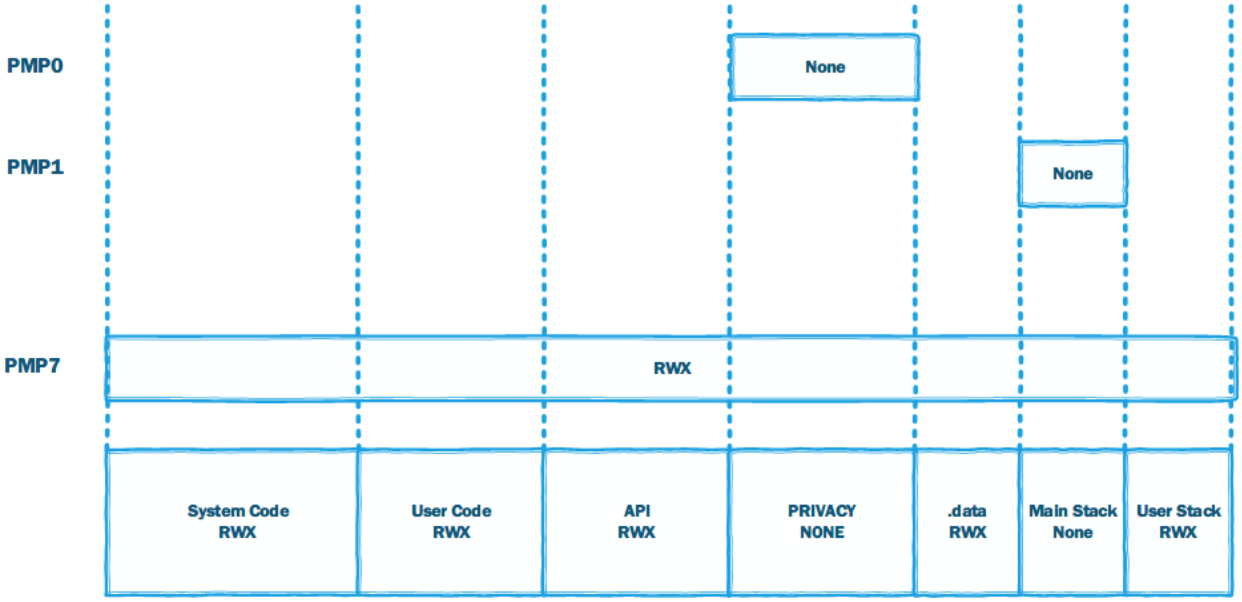
```
  configMainStack:
    csrw CSR_MSCRATCH, a0
    jr ra
```

同时在进入异常处理或异常退出时进行堆栈的切换及还原；

```
.macro SWTICH_STACK
  csrrw sp,CSR_MSCRATCHCSW, sp
.endm

exc_entry:

    SWTICH_STACK
    /* Save the caller saving registers (context) */
    SAVE_CONTEXT
    /* Save the necessary CSR registers */
    SAVE_CSR_CONTEXT

    csrr a0, mcause
    mv a1, sp

    call core_exception_handler
    /* Restore the necessary CSR registers */
    RESTORE_CSR_CONTEXT
    /* Restore the caller saving registers (context) */
    RESTORE_CONTEXT

    SWTICH_STACK
```

```
    /* Return to regular code */
    mret
```

4.2 PMP配置



在启动初期，定义了一个背景PMP配置项（PMP7）支持所有空间的读写权限；并针对隐私保护区及系统堆栈区设置为禁止权限；因此在进入用户模式后，该部分区域将不可访问；

💡 RISC-V中PMP的配置匹配区域支持重叠覆盖，PMP的序号越低优先级越高；

💡 在机器模式下运行，PMP的配置权限通常不会匹配生效，除非PMP配置项锁定或MPRV域为1；
在用户模式下运行时，若PMP未匹配的地址空间均无法访问；

💡 同时由于RT-Thread当前运行在Machine Mode下，默认PMP的配置项对MachineMode不起作用，为此修改Mstatus的MPRV位1，生效MachineMode的PMP匹配；

```
#define USER_CODE_REGION_BASE 0x20010000
#define API_CODE_REGION_BASE 0x20011000
#define USER_STACK_REGION_BASE 0x2001f800
#define PRIVACY_CODE_REGION_BASE 0x20012000
#define MAIN_STACK_REGION_BASE 0x2001e800

  PMP_Region_InitTypeDef pmp_init;
  pmp_init.Number = PMP_REGION_NUMBER0;
  pmp_init.Enable = PMP_REGION_ENABLE;                  // Enable Configuration
  pmp_init.Lock = PMP_REGION_UNLOCK;                    // Configuration Lock, may not modify unless reset, and also match in Machine Mode
  pmp_init.BaseAddress = PRIVACY_CODE_REGION_BASE;      //Seting test_array Base
  pmp_init.Size = PMP_REGION_SIZE_2KB;                  //Setting array size
```

```
  pmp_init.AddressMatching = PMP_ADDRESS_MATCHING_NAPOT; //Setting PMP Size to NAPOT mode -> 2^n
  pmp_init.AccessPermission = PMP_ACCESS_NONE;           //Setting array permission is Read Only

  PMP_ConfigRegion(&pmp_init);

  pmp_init.Number = PMP_REGION_NUMBER1;
  pmp_init.Enable = PMP_REGION_ENABLE;                   // Enable Configuration
  pmp_init.Lock = PMP_REGION_UNLOCK;                     // Configuration Lock, may not modify unless reset, and also match in Machine Mode
  pmp_init.BaseAddress = MAIN_STACK_REGION_BASE;       //Seting test_array Base
  pmp_init.Size = PMP_REGION_SIZE_2KB;                   //Setting array size
  pmp_init.AddressMatching = PMP_ADDRESS_MATCHING_NAPOT; //Setting PMP Size to NAPOT mode -> 2^n
  pmp_init.AccessPermission = PMP_ACCESS_NONE;           //Setting array permission is Read Only

  PMP_ConfigRegion(&pmp_init);

  pmp_init.Number = PMP_REGION_NUMBER7;
  pmp_init.Enable = PMP_REGION_ENABLE;                   // Enable Configuration
  pmp_init.Lock = PMP_REGION_UNLOCK;                     // Configuration Lock, may not modify unless reset, and also match in Machine Mode
  pmp_init.BaseAddress = 0;       //Seting test_array Base
  pmp_init.Size = PMP_REGION_SIZE_4GB;                   //Setting array size
  pmp_init.AddressMatching = PMP_ADDRESS_MATCHING_NAPOT; //Setting PMP Size to NAPOT mode -> 2^n
  pmp_init.AccessPermission = PMP_ACCESS_RWX;          //Setting array permission is Read Only

  PMP_ConfigRegion(&pmp_init);
```

4.3用户模式

用户模式为RISC-V的一种特权模式，该模式下对CSR寄存器的访问将受到约束，同时仅能访问PMP配置中允许的空间范围，并且无权修改PMP的配置；

从 Machine Mode 切换到 User Mode ，需先将 mstatus 的 MPP 域置为0，通过执行 mret 指令触发；执行mret指令后，PC指针将跳转至mepc处执行；具体实现可参见system_config.S;

```
.global enterUserMode
enterUserMode:
  li t0, MSTATUS_MPP
  csrc mstatus, t0
  csrw mepc, a0
  mv a0,a1
  mret
```

4.4系统调用

由于隐私保护密钥及加密算法在用户模式下PMP配置为无权限，因此用户无法直接访问隐私保护API，为此需通过系统调用的方式进入异常处理实现特权模式的切换；

> 💡 CM32M4xxR内核从 User Mode 切换到 Machine Mode 只能通过异常、中断或者 NMI；
> 通常可利用ecall指令，主动触发软件异常，类似ARM Cortex-M内核的SVC系统调用；

本示例中实现了一个三参数的系统调用接口（privacy_api.c），以完成系统调用号及参数的传递；

```
#define SYSTEM_CALL_PRIVACY        1

#define SYSTEM_ECALL(__num, __a0, __a1, __a2)                                  \
  ({                                                                    \
    register unsigned long a0 asm("a0") = (unsigned long)(__a0);  \
    register unsigned long a1 asm("a1") = (unsigned long)(__a1);  \
    register unsigned long a2 asm("a2") = (unsigned long)(__a2);  \
    register unsigned long a7 asm("a7") = (unsigned long)(__num); \
    asm volatile("ecall"                                            \
           : "+r"(a0)                                           \
           : "r"(a1), "r"(a2), "r"(a7)                          \
           : "memory");                                         \
    a0;                                                             \
  })

uint8_t privacy_encrypt_api(uint8_t *src, uint16_t src_len, uint8_t * dst)
{
```

```
  SYSTEM_ECALL(SYSTEM_CALL_PRIVACY,src,src_len,dst);
  return 0;
}
```

上述示例中，__num为系统调用号，用户可根据自身业务需求进行扩展，本示例中为了演示仅实现了隐私接口的调用号相关实现；

为了结果异常调用，可在启动初期，注册PMP软件异常的处理接口；

```
Exception_Register_EXC(EXC_ECALL_U, PMP_SYSCALL_Handler);
```

并在软件异常处理函数中实现隐私保护处理的接口调用，用于该接口执行在异常处理函数中，并以进入机器模式，因此访问控制将不受PMP的权限配置；因此调用能够顺利执行；

```
void PMP_SYSCALL_Handler(unsigned long mcause, unsigned long sp) {
  printf("PMP Exception Occures Run at 0x%lx, Enter System Call !!!\r\n", __RV_CSR_READ(CSR_MEPC));
  uint32_t saved_regs = sp;
  uint32_t mepc = ((uint32_t *)saved_regs)[12];
  uint32_t funcId = ((uint32_t *)saved_regs)[15];
  switch(funcId)
  {
    case SYSTEM_CALL_PRIVACY:
      printf("Enter Privacy System CALL\r\n");
      uint8_t *src = (uint8_t *)((uint32_t *)saved_regs)[5];
      uint16_t src_len = (uint16_t)((uint32_t *)saved_regs)[6];
      uint8_t *out = (uint8_t *)((uint32_t *)saved_regs)[7];
      privacy_encrypt(src, src_len, out);

    break;
    default:
      printf("System Call not Defined\r\n");
      while(1);
      return;
  }

  mepc += 4;
  ((uint32_t *)saved_regs)[12] = mepc;
}
```

💡 特别需要注意的时，由于通过ecall指令进入异常后，mepc指针将被设置为ecall指令位置，因此在异常退出后，程序仍将跳转至ecall指令处，从而重复进入异常，为此为了实现顺序执行，应在异常退出前调整mepc指针的位置；

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/73ee4e01-de85-4593-b583-28a70e75be26/PMP.vsdx