black hat EUROPE 2022

DECEMBER 7-8, 2022

BRIEFINGS

Good Motive but Bad Design: Pitfalls in MPU Usage in Embedded Systems in the Wild

Wei Zhou, Zhouqi Jiang, Le Guan

華中科技大學

UNIVERSITY OF GEORGIA

About Us

Wei Zhou

- Associate Professor at Huazhong University of Science and Technology
- Research interest: IoT Security and Program Analysis
- Published at: ACM CCS, USENIX Security, ESCRIOS, etc.

華中科技大學

Zhouqi Jiang

- Graduate student at Huazhong University of Science and Technology
- Research interest: IoT Security and Trust Computing

華中科技大學

Le Guan

- Assistant professor at the University of Georgia
- Research interest: Systems Security and IoT Security
- Published at: ACM CCS, USENIX Security, NDSS, IEEE S&P, ICSE, etc.

UNIVERSITY OF GEORGIA

Agenda

- **Introduction to Memory Protection Unit (MPU)**
- MPU adoption in the wild
- Common pitfalls and limitations in using MPU
- Mitigation suggestions
- Summary and disclosure

What is Memory Protection Unit (MPU)

- The Memory Management Unit (MMU), a standard feature in commodity computing platforms, is absent in resource-restricted microcontroller units (MCUs)
- As a stripped-down version of MMU**,** the **Memory Protection Unit (MPU)** provides basic security functions for MCUs, e.g., Arm Cortex-M series MCUs
- How MPU works?
  - For a limited number of configurable memory regions, MPU assigns **access permissions (e.g., R/W)** based on the current **privilege level** of the execution
  - A fault happens when a memory access violates the access permission
  - MPU can **only** be configured by privileged code

How to Program MPUs (PMSAv7)?

- Setting The Enable bit (LSB) in MPU Control Register (CTRL) to enable the MPU.
- Region Base Address Register (RBAR): address/size information of a memory region
- Region Attribute/Size Register (RASR): access permission/attributes of a memory region
  - The **XN bit** in RASR also provides **eXecure Never (XN) capability**
  - **Attributes (e.g., cacheability and shareability)** of each region can be configured by TEX, C and B fields in RASR
  - Large regions can be further divided **into eight equally sized sub-regions**, but it inherits the same permissions with parent regions
- The PRIVDEFENA bit in MPU Control Register (CTRL) can be used to enable the default memory map as **a background region for privileged access**.
- **Constrains on memory regions**

(1) At least 32 bytes

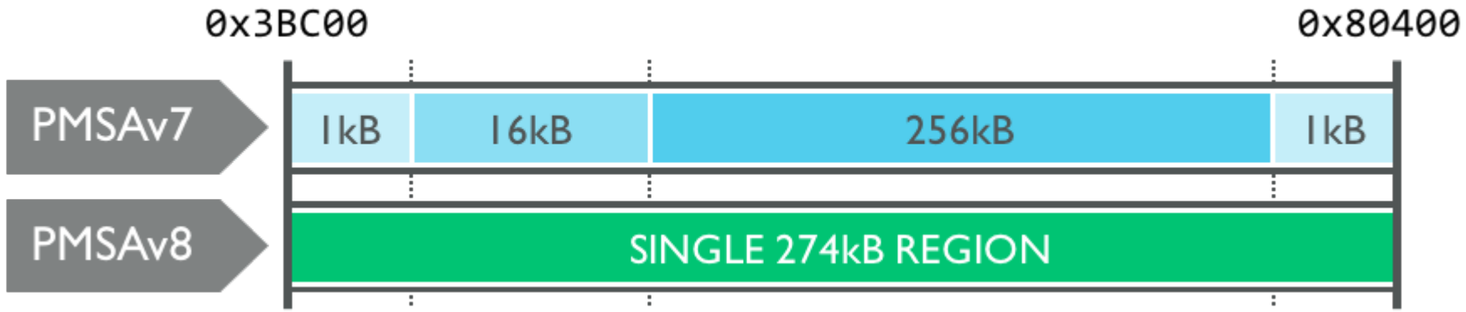
(2) Power of two

(3) Must be aligned with 32 bytes

(4**) Limited region numbers (M0+/M3/M4 up to 8 and M7 up to 16)**

What's new in PMSAv8?

- More MPU regions (up to 16 regions for both normal and secure world in M23 and M33)
- Use Start and Limit (end) address via separated MPU registers to define memory regions, but still must be 32-byte aligned



- PMSAv8 also introduces a new memory attribute indirection register (MPU_MAIR), making it easier for multiple regions to share the same attribute, while at the same time maintaining their own access permissions

Agenda

- Introduction to Memory Protection Unit (MPU)
- **MPU adoption in the wild**
- Common pitfalls and limitations in using MPU
- Mitigation suggestions
- Summary and disclosure

MPU-enabled security functions

- **Code Integrity Protection (CIP):** Code regions can be set as non-writable by unprivileged code to prevent code injection and manipulation.
- **Data Execution Prevention (DEP):** Data regions like stack or heap can be set non-executable
- **Stack Guard (SG):** An inaccessible memory region can be placed at the stack boundary to detect stack overflows
- **Kernel Memory Isolation (KMI):** User mode (unprivileged) code cannot access any memory belonging to the kernel space without invoking system calls
- **User Task Memory Isolation (TMI):** User mode (unprivileged) tasks can only access its own memory except explicitly shared memory regions that belong to other tasks or kernel

MPU adoption in popular MCU systems

	OS	MPU Support	MPU Support: CIP	DEP	KMI	TSI	SG	PI
Open-source	Contiki	None	-	-	-	-	-	-
	RIoT	Optional	Default-off	-	-	-	Default-off	-
	Mynewt	None	-	-	-	-	-	-
	LiteOS	None	-	-	-	-	-	-
	Zephyr	Optional	Default-on	Default-on	Default-off	Default-off	Default-off	Default-off
	TinyOS	None	-	-	-	-	-	-
	FreeRTOS	None	-	-	-	-	-	-
	FreeRTOS-MPU	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory	-	-
	MbedOS	Optional	Default-on	Default-on	-	-	-	-
	TizenRT	Optional	Default-off	Default-off	Default-off	Default-off	Default-off	-
	CMSIS-Keil RTX	Optional	Default-off	Default-off	-	Default-off	-	Default-off
	Azure RTOS ThreadX	Optional	Default-off	Default-off	Default-off	-	-	-
Proprietary	embOS	Optional						
	Integrity RTOS	Mandatory						
	NXP MQX RTOS	Optional						
	Nucleus RTOS	Optional						
	SafeRTOS	Mandatory						
	µC/OS-III	Optional						
	VxWorks	None						

We try to find out the reason in this work.

- Only a few MCU OSs use MPU, especially for the open-source OSs
- Even if MPU is supported, only a few security features are enabled by default

Agenda

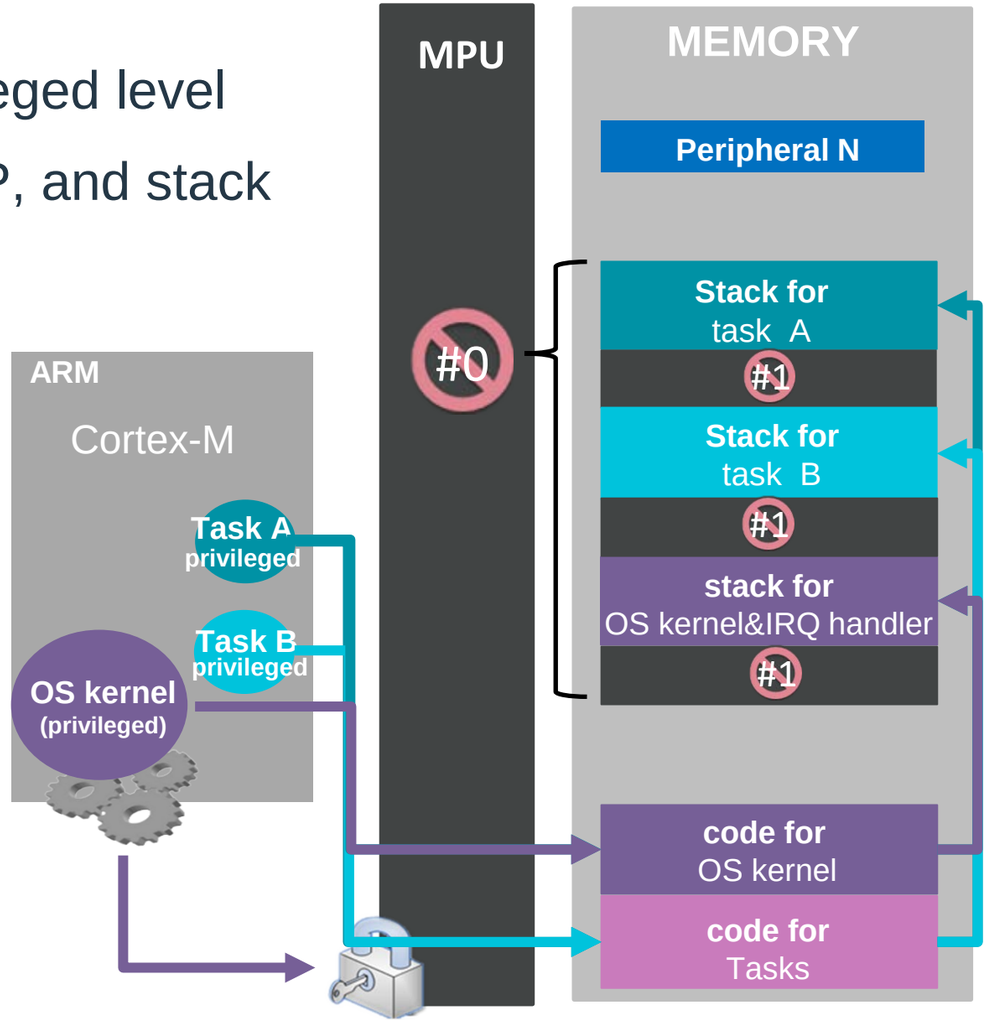
- Introduction to Memory Protection Unit (MPU)
- MPU adoption in the wild
- **Common pitfalls and limitations in using MPU**
- Mitigation suggestions
- Summary and disclosure

Common pitfalls in using MPU

- Weak protection
  - **Case study: Bypassing MPU protection in RIoT-MPU**
  - Case study: Privileged escalation in FreeRTOS-MPU
- Incomplete protection
- Prohibitive overhead
- Conflict with existing system designs

Case Study : MPU-enabled RIoT

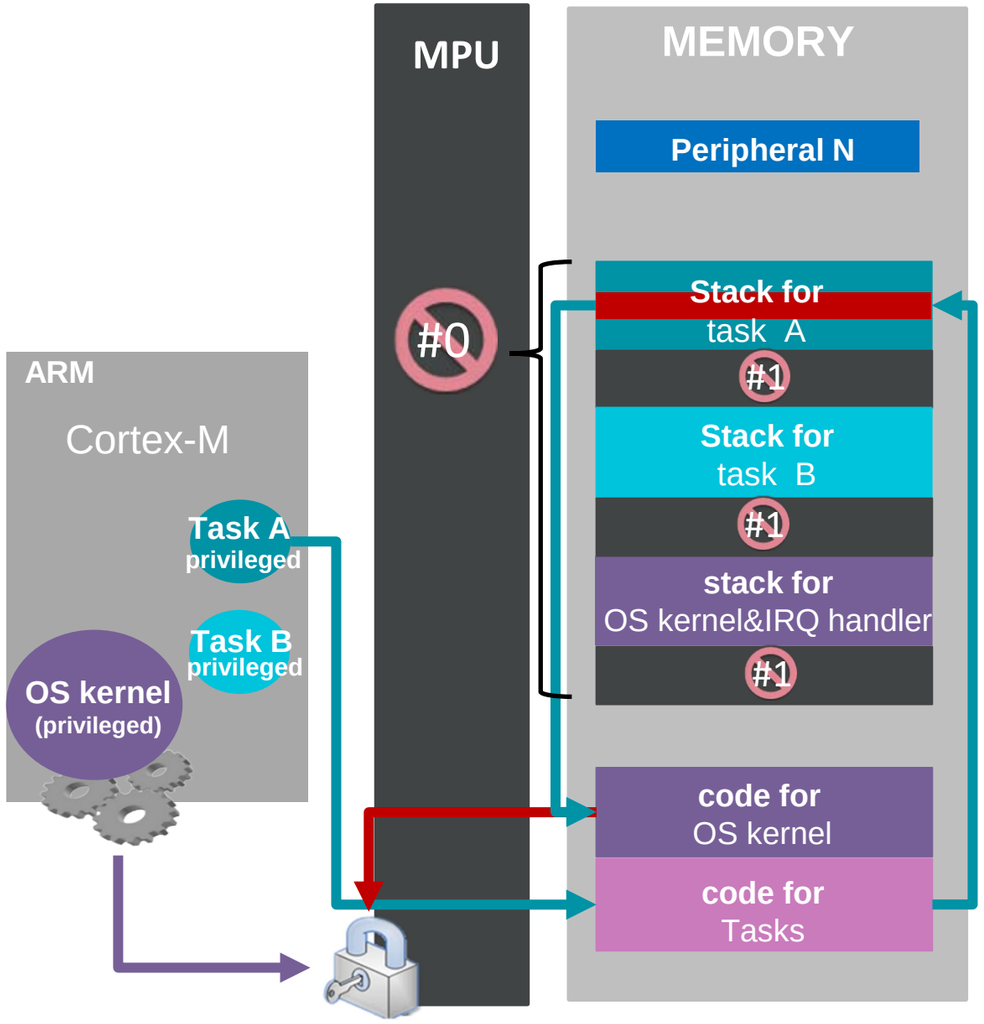
- Some MCU OSs like RIoT run all the code under privileged level
- They only provide some basic protections such as DEP, and stack guard (SG) with MPU
- Data Execution Prevention (DEP): RIoT enables the MPU region number 0 to cover the whole RAM region as non-executable
- Stack Guard (SG): RIoT defines the permission of the last 32 bytes (the smallest MPU region) of the main stack as read-only via the MPU region number 1. Similarly, when switching to another task, RIoT configures the last 32 bytes of the target task stack as read-only via the MPU region number 1.
  - Cannot detect stack overflow of individual stack frames
  - Cannot detect control flow hijacking attack



Bypassing MPU in MPU-enabled RIoT

- Bug: MPU can be disabled by control flow hijacking attack (e.g., ROP)
- Cause: MPU control registers (e.g., MPU_CTRL) are located in the system peripheral region, which can be accessed by any privileged code. RIoT also provides an easy-to-use driver APIs for MPU configurations (e.g., mpu_enable and mpu_disable driver APIs).

```
int mpu_disable(void) {
#if __MPU_PRESENT
    MPU->CTRL &= ~MPU_CTRL_ENABLE_Msk;
    return 0;
#else
    return -1;
#endif
}
```



Attack Demo

Visual Studio Code — Dec 6 09:17

main.c - riot-stack-overflow - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER

RIOT-STACK-OVERFLOW
- .vscode
- bin
  - alientek-pandora
  - asm.S
  - input.txt
  - out.asm
- .gdbinit
- .gitignore
- .input_value.sh.swp
- input_value.sh
- main.c
- Makefile
- Makefile.ci
- openocd.cfg
- README.md
- vulnerable.c
- vulnerable.h

OUTLINE

TIMELINE

vulnerable.c | main.c | stdlib.h | input_value.sh | vulnerable.h | README.md

main.c > receive_handler(int, char **)

```c
void vulnerable_function(void) {
    char buffer[32] = {0};
    printf("Please input your name: \n");
    scanf("%[^\n]", buffer);
    printf("Your name is: %s\n", buffer);
}

#include <stdlib.h>
int receive_handler(int argc, char **argv) {
    (void) argc; (void) argv;
    char s[32] = {0};
    uint32_t mpu_ctrl = *(volatile uint32_t *)0xe000ed94;
    my_itoa(mpu_ctrl, s, 2);
    printf("mpu_ctrl: 0x%08ld (0b%s)\r\n", mpu_ctrl, s);
    vulnerable_function();
    return 0;
}

int main(void)
{
#ifdef MODULE_NETIF
    gnrc_netreg_entry_t dump = GNRC_NETREG_ENTRY_INIT_PID(GNRC_NETREG_DEMUX_CTX_ALL,
                                                          gnrc_pktdump_pid);
    gnrc_netreg_register(GNRC_NETTYPE_UNDEF, &dump);
#endif

    (void) puts("Welcome to RIOT!");

    commands[0] = receive_command;
    commands[1] = mpu_disable_command;

    char line_buf[SHELL_DEFAULT_BUFSIZE];
    shell_run(commands, line_buf, SHELL_DEFAULT_BUFSIZE);

    return 0;
}

#include "mpu.h"
int mpu_disable_handler(int argc, char **argv) {
```

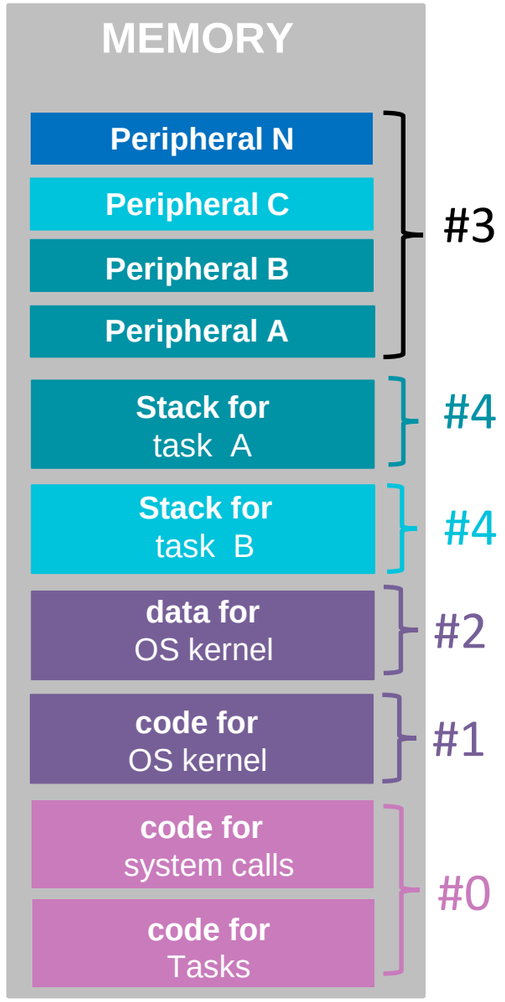
pandora-board* ⊗0 ⚠0 — Ln 77, Col 27 — Spaces: 4

Common pitfalls in using MPU

- Weak protection
  - Case study: Bypassing MPU protection in RIoT-MPU
  - **Case study: Privileged escalation in FreeRTOS-MPU**
- Incomplete protection
- Prohibitive overhead
- Conflict with existing system designs

Case Study : FreeRTOS-MPU

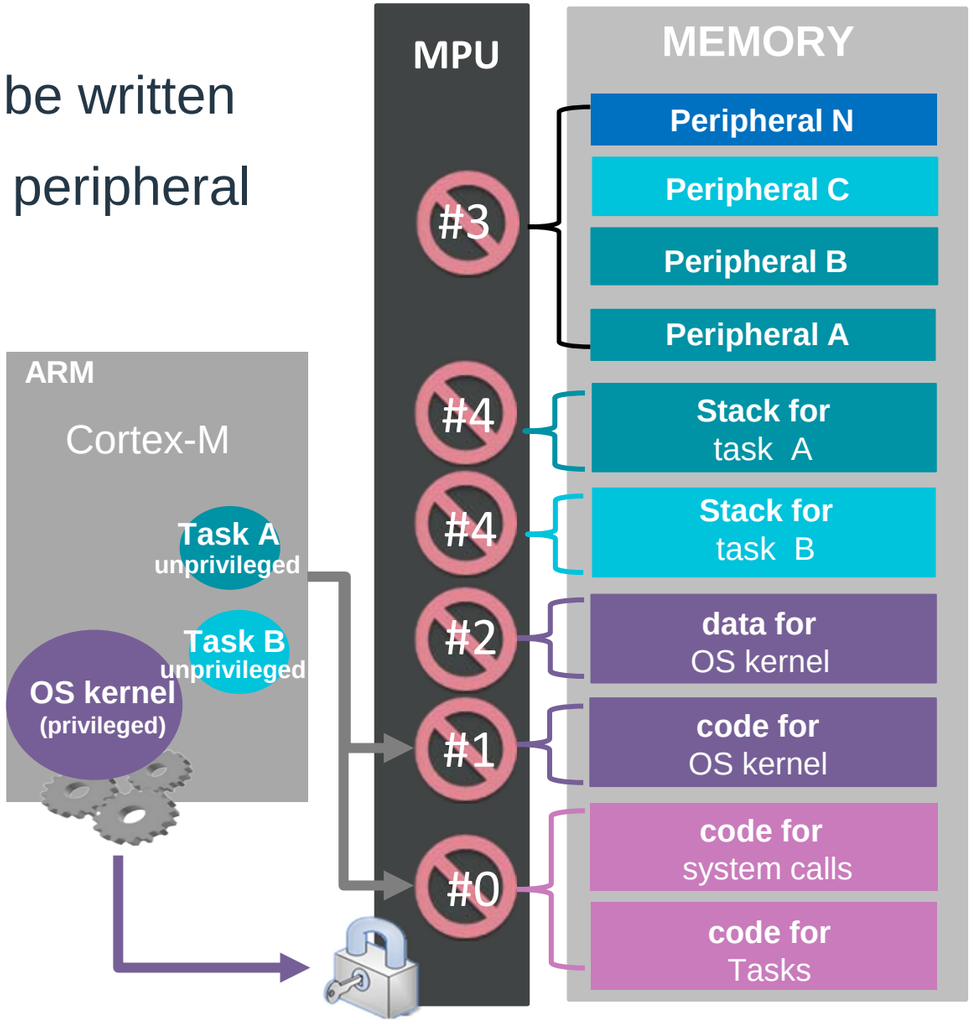
Region No.	Range	Usage	Privilege Level	Permission
0	flash_segement_start -flash_segemnt_end	Non-writeable code segment for task and system call	Privileged Unprivileged	r-x r-x
1	privileged_functions_start -privileged_functions_end	Kernel APIs Isolation	Privileged Unprivileged	r-x ∅
2	privileged_data_start -privileged_data_end	Kernel Data Isolation	Privileged Unprivileged	rwx ∅
3	0x40000000-0x5fffffff	Non-executable Peripherals	Privileged Unprivileged	rw- rw-
4	User Task Stack	User Task Stack Isolation	Privileged Unprivileged	rw- rw-
5-N	User-defined	E.g. peripheral isolation	-	-



- Background region in grey is enabled for privileged access only

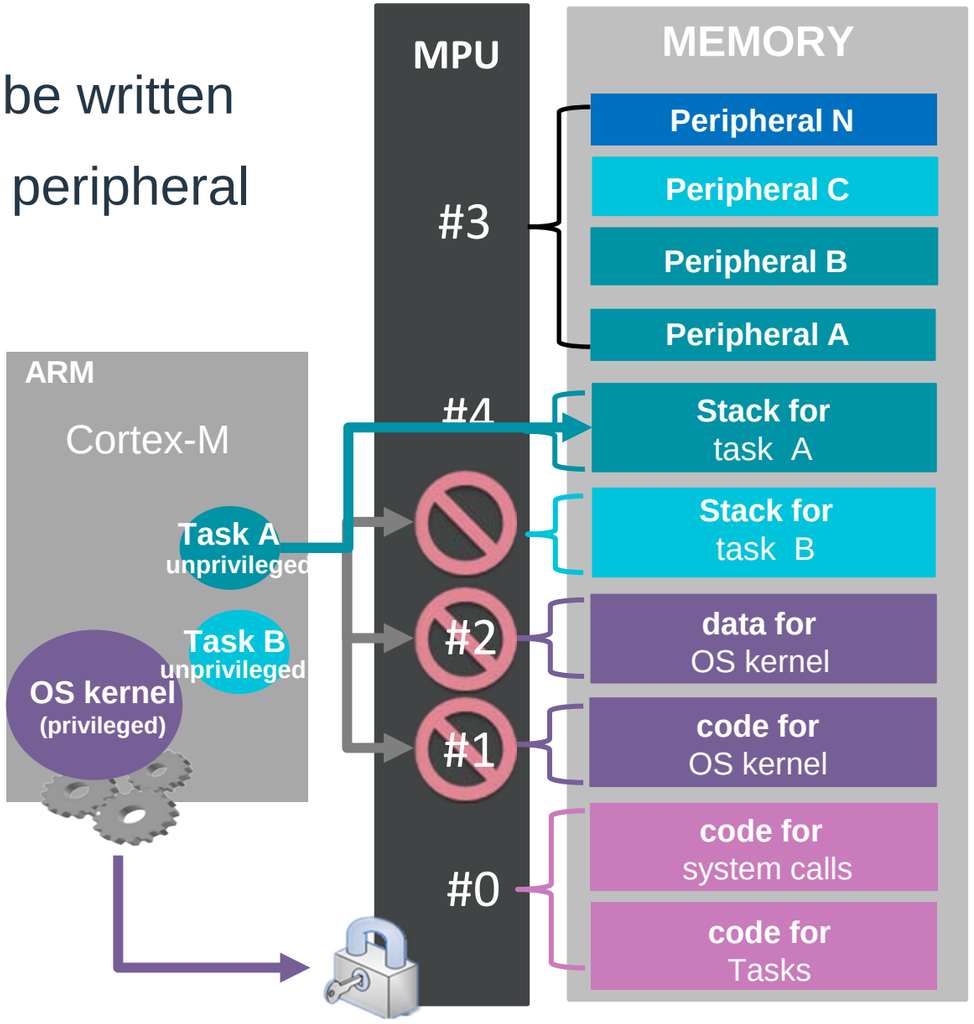
Security features in FreeRTOS-MPU

- Code Integrity Protection (CIP): All code region cannot be written
- Data Execution Prevention (DEP): All data regions and peripheral regions are non-executable



Security features in FreeRTOS-MPU

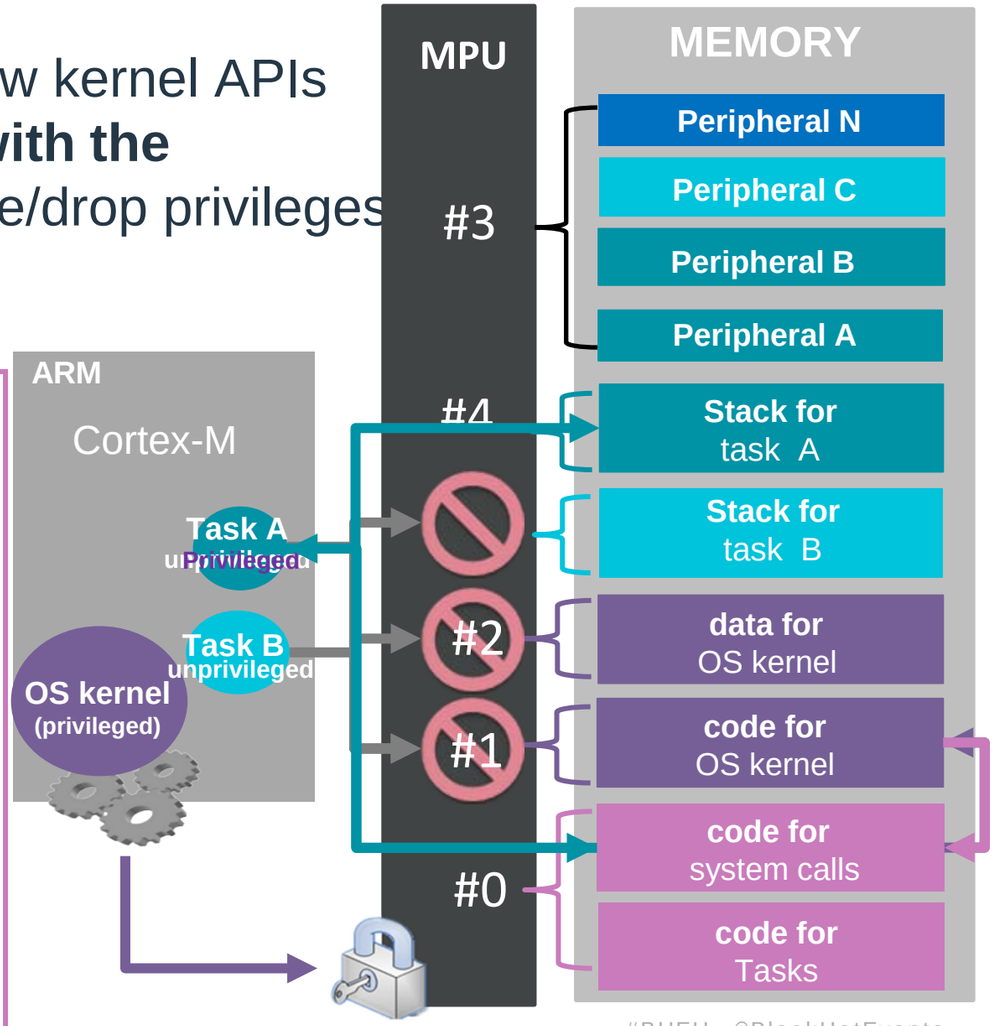
- Code Integrity Protection (CIP): All code region cannot be written
- Data Execution Prevention (DEP): All data regions and peripheral regions are non-executable
- User Task Memory Isolation (TMI): Unprivileged tasks can only access their own stack and up to three user definable memory regions (three per task)
- Kernel Memory Isolation (KMI): The FreeRTOS kernel API and data are located in a region of Flash that can only be accessed while the microcontroller is in privileged mode (calling as system call causes a temporary switch to privileged mode)



Look deeper in system call implementation

- For compatibility, FreeRTOS MPU does not provide new kernel APIs for system calls, but **wraps the original kernel APIs with the xPortRaisePrivilege and vPortResetPrivilege** to raise/drop privileges

```
void MPU_vTaskDelay(TickType_t xTicksToDelay){
    BaseType_t xRunningPrivileged = xPortRaisePrivilege();
    vTaskDelay(xTicksToDelay);
    vPortResetPrivilege(xRunningPrivileged);
}
BaseType_t xPortRaisePrivilege(void){
    BaseType_t xRunningPrivileged;
    xRunningPrivileged = portIS_PRIVILEGED();
    /*If the CPU is not privileged, raise privilege.*/
    if (xRunningPrivileged == pdFALSE){
        portRAISE_PRIVILEGE();
    }
    return xRunningPrivileged;
}
#define portRAISE_PRIVILEGE() __asm volatile ("svc %0 \n" ::"i" (portSVC_RAISE_PRIVILEGE) : "memory");
void prvSVCHandler(uint32_t* pulParam){
    ...
    case portSVC_RAISE_PRIVILEGE:
        if ((ulPC >= _syscalls_flash_start_) && (ulPC <= _syscalls_flash_end_)){
            __asm {
                /*Obtain control value.*/
                mrs ulReg, control
                /*Set privilege bit.*/
                bic ulReg, #1
                /*Write back control value*/
                msr control, ulReg
            }
        }
    break;
    ...
}
```



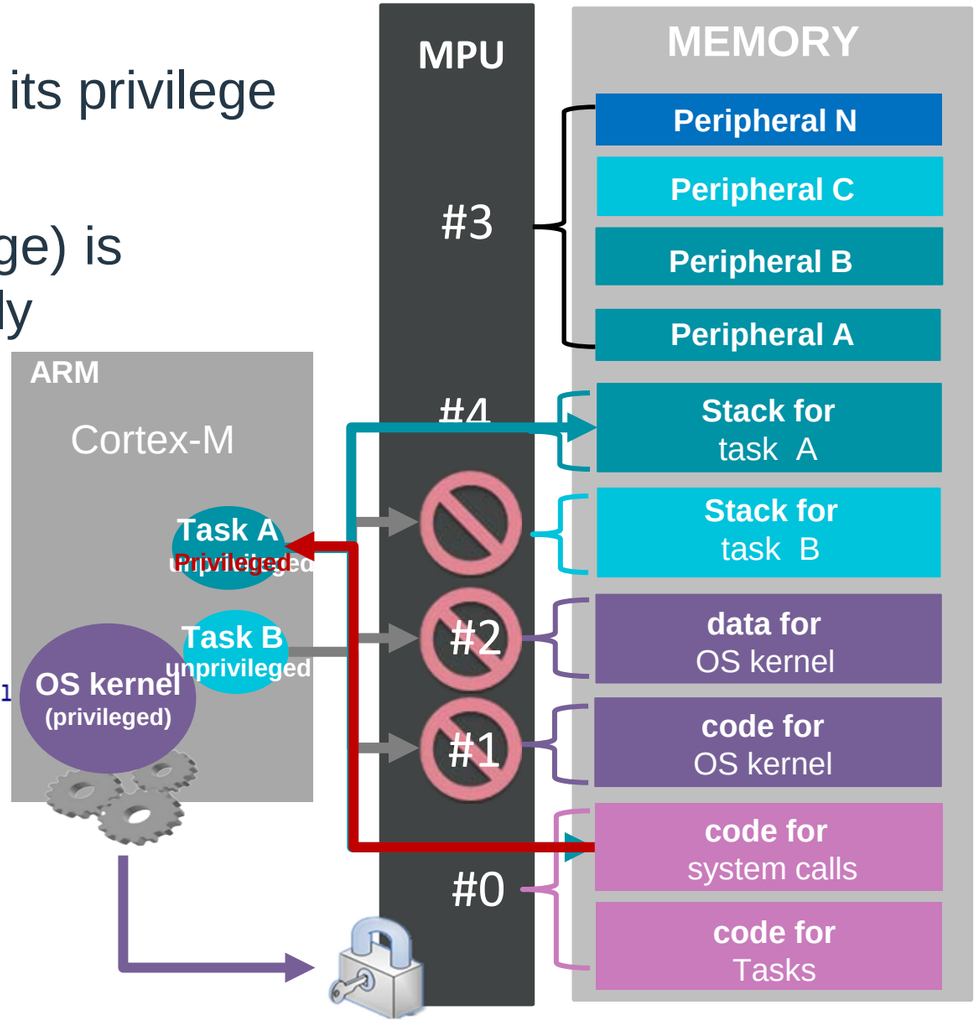
Privilege escalation in FreeRTOS-MPU

- Bug1 (v10.4.5 and before): An unprivileged task can raise its privilege by calling the internal function xPortRaisePrivilege
- Cause: Privilege escalation function (xPortRaisePrivilege) is separated with kernel function and can be called directly
- Patch (v10.4.6): Change xPortRaisePrivilege and vPortResetPrivilege as macros.

Is problem solved?

```
#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();
    }
}
```

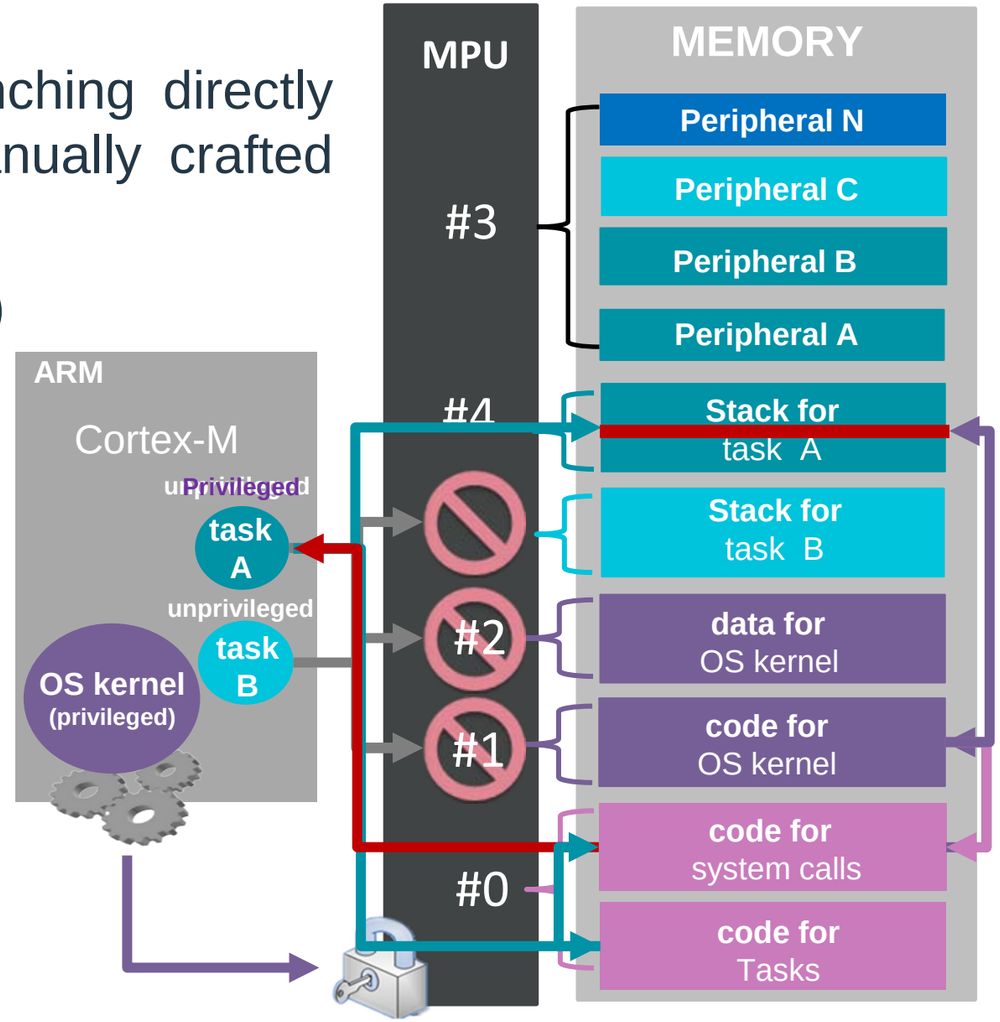
```
void __cdecl MPU_vTaskDelay(TickType_t xTicksToDel
{
  BaseType_t v5; // [sp+0h] [bp-10h]
  TickType_t v6; // [sp+4h] [bp-Ch]

  v6 = xTicksToDelay;
  v5 = xIsPrivileged();
  if ( !v5 )
    __asm { SVC     2 }
  vTaskDelay(v6);
  if ( !v5 )
    vResetPrivilege();
}
```



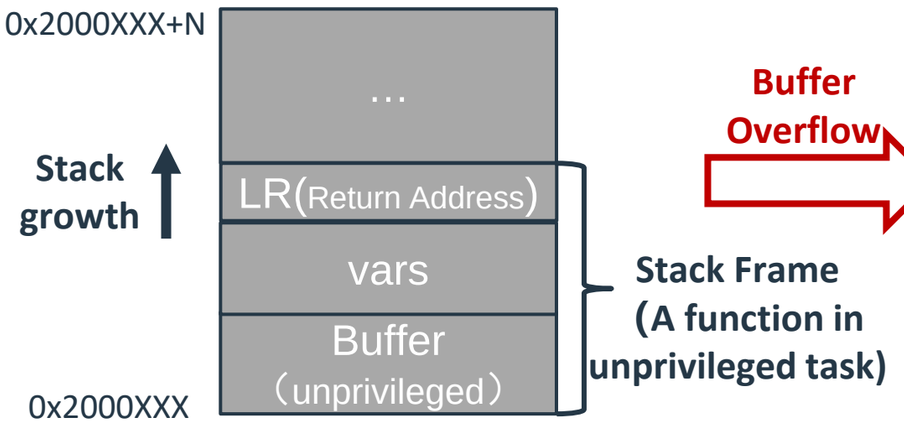
Privilege escalation in FreeRTOS-MPU

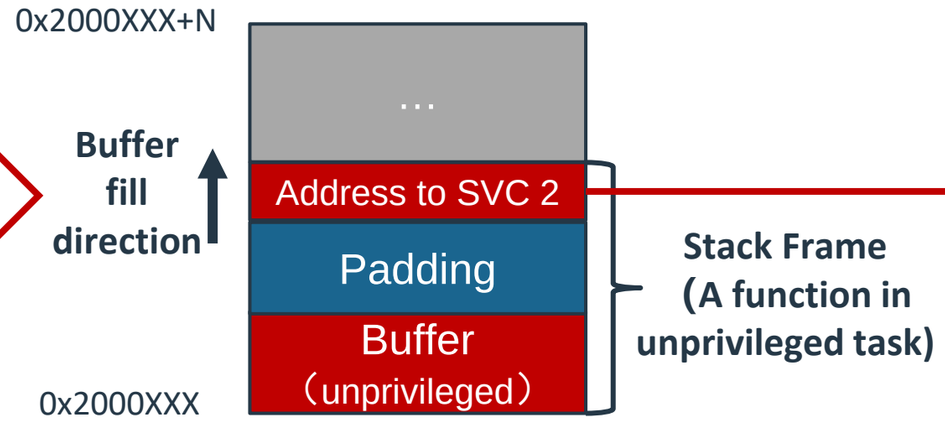
- Bug2 (v10.4.6 and before): Privilege escalation by branching directly inside system calls (MPU wrapper APIs) with a manually crafted stack frame
- Causes: Privilege escalation operation (SVC interrupt) is separated with kernel API and uses stack to store the original privilege level

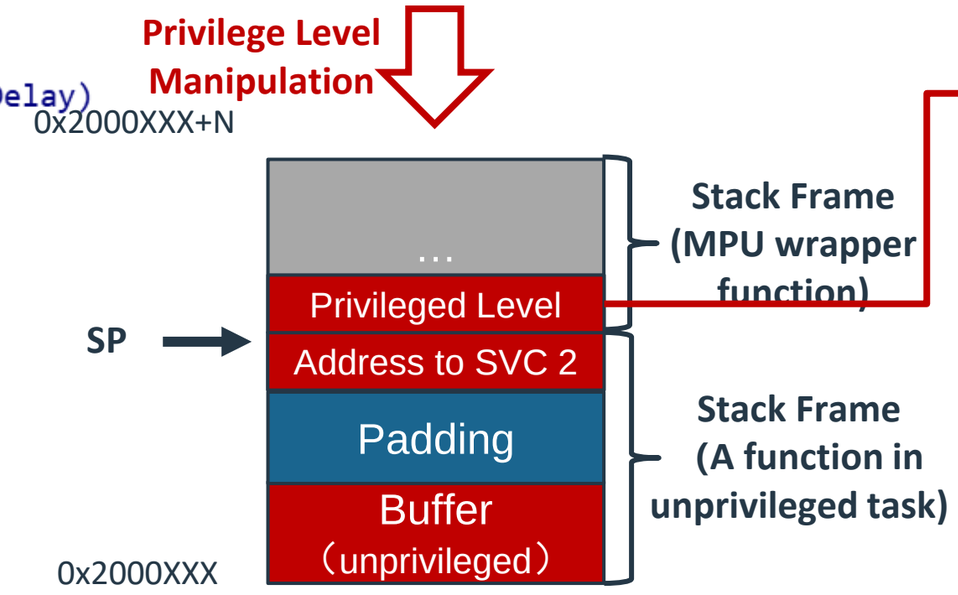
```
void __cdecl MPU_vTaskDelay(TickType_t xTicksToDelay)
{
  BaseType_t v5; // [sp+0h] [bp-10h]
  TickType_t v6; // [sp+4h] [bp-Ch]

  v6 = xTicksToDelay;
  v5 = xIsPrivileged(); ①
  if ( !v5 )
    __asm { SVC     2 } ②
  vTaskDelay(v6); ③
  if ( !v5 ) ④
    vResetPrivilege(); ⑤
} ⑥
```



Exploitation Steps








```
void __cdecl MPU_vTaskDelay(TickType_t xTicksToDelay)
{
  BaseType_t v5; // [sp+0h] [bp-10h]
  TickType_t v6; // [sp+4h] [bp-Ch]

  v6 = xTicksToDelay;
  v5 = xIsPrivileged(); ①
  if ( !v5 )
    __asm { SVC     2 } ②
  vTaskDelay(v6); ③
  if ( !v5 ) ④
    vResetPrivilege(); ⑤
} ⑥
```

```
                EXPORT MPU_vTaskDelay
MPU_vTaskDelay
                PUSH    {R7,LR}
                SUB     SP, SP, #8
                STR     R0, [SP,#4]
                BL      xIsPrivileged
                STR     R0, [SP]
                LDR     R0, [SP]
                CBNZ    R0, loc_800815E
                B       loc_800815A
; ---------------------------------------
loc_800815A
                SVC     2
                B       loc_800815E
; ---------------------------------------
loc_800815E
                LDR     R0, [SP,#4]
                BL      vTaskDelay
                LDR     R0, [SP]
                CBNZ    R0, loc_8008170
                B       loc_800816A
; ---------------------------------------
loc_800816A
                BL      vResetPrivilege
                B       loc_8008170
; ---------------------------------------
loc_8008170
                ADD     SP, SP, #8
                POP     {R7,PC}
; End of function MPU_vTaskDelay
```

Attack Demo

D:\IoTSecLab\jmpsvc2-freertos\Keil\JmpSvc2.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

vtaskdelay

JmpSvc2

Project

- Project: JmpSvc2
  - JmpSvc2
    - App
      - app_main.c
      - app_main.h
      - jmpsvc2.c
      - jmpsvc2.h
      - retarget_io.c
    - Startup
      - startup_stm32l475xx.s
      - memfault_handler.c
    - FreeRTOS
      - croutine.c
      - event_groups.c
      - list.c
      - queue.c
      - stream_buffer.c
      - tasks.c
      - timers.c
      - mpu_wrappers.c
      - port.c
      - portmacro.h
      - heap_4.c
    - Config
      - FreeRTOSConfig.h
    - Core
      - main.c
      - stm32l4xx_it.c
      - stm32l4xx_hal_msp.c
      - stm32l4xx_hal_timebase_tim
    - Drivers/STM32L4xx_HAL_Driver
      - stm32l4xx_hal_dfsdm.c
      - stm32l4xx_hal_i2c.c
      - stm32l4xx_hal_i2c_ex.c
      - stm32l4xx_hal_qspi.c

Project | Books | {} Functio... | Templa...

jmpsvc2.c | mpu_wrappers.c | startup_stm32l475xx.s | main.c | stm32l4xx_hal_uart.c | port.c | stm32l4xx_hal_pwr_ex.c | tasks.c

```c
10  void serial_scanf(UART_HandleTypeDef *huart, const char* pattern, uint8_t *str_out);
11
12  void ReceiveFunction() {
13    char receiveBuffer [ 32 ]; /* This buffer can be overflowed */
14    char *buffer = receiveBuffer;
15
16    serial_scanf(&huart1, "%s", (uint8_t *)buffer);
17  }
18
19  void UnprivilegedTask( void * pvParameters ) {
20    ( void ) pvParameters; /* Unused parameters. */
21
22    if ( portIS_PRIVILEGED() == pdFALSE ) {
23      printf("Current task is not privileged.\r\n");
24    }
25
26    ReceiveFunction();
27
28    if ( portIS_PRIVILEGED() == pdTRUE ) {
29      printf("Current task is privileged.\r\n");
30      printf("Attack successful!\r\n");
31    }
32
33    for( ; ; )
34    {
35      vTaskDelay( pdMS_TO_TICKS( 1000 ) );
36    }
37  }
38
39  void vStartJmpSvc2Demo( void )
40  {
41    static StackType_t TaskStack[ configMINIMAL_STACK_SIZE ] __attribute__( ( aligned( configMINIMAL_STACK_SIZE * sizeof( StackType_t ) ) ) );
42    TaskParameters_t TaskParameters =
43    {
44      .pvTaskCode   = UnprivilegedTask,
45      .pcName       = "UnprivilegedTask",
```

Build Output

```
Batch-Build summary: 0 succeeded, 0 failed, 1 skipped - Time Elapsed: 00:00:00
Load "Debug\\JmpSvc2.axf"
Erase Done.
Programming Done.
Verify OK
```

Build Output | Find In Files | All References of 'tskIDLE_PRIORITY'

ST-Link Debugger | L:11 C:1 | CAP NUM SCRL OVR R/W

10°C 多云 | 搜索 | 英 | 15:40 2022/12/7

Patch

- Decide the original privilege level at the beginning with control register
- Introduced the portMEMORY_BARRIER macro to prevent instruction re-ordering when GCC link time optimization is used

```
void MPU_vTaskDelay( TickType_t xTicksToDelay ) /* FREERTOS_SYSTEM_CALL */ {
    BaseType_t xRunningPrivileged;
    if( portIS_PRIVILEGED() == pdFALSE ) {
        portRAISE_PRIVILEGE();
        portMEMORY_BARRIER();

        xPortRaisePrivilege( xRunningPrivileged );
        vTaskDelay( xTicksToDelay );
        vPortResetPrivilege( xRunningPrivileged );
        portMEMORY_BARRIER();

        portRESET_PRIVILEGE();
        portMEMORY_BARRIER();
    }
    else
    {
        vTaskDelay( xTicksToDelay );
    }
}
```

Privilege escalation in FreeRTOS-MPU

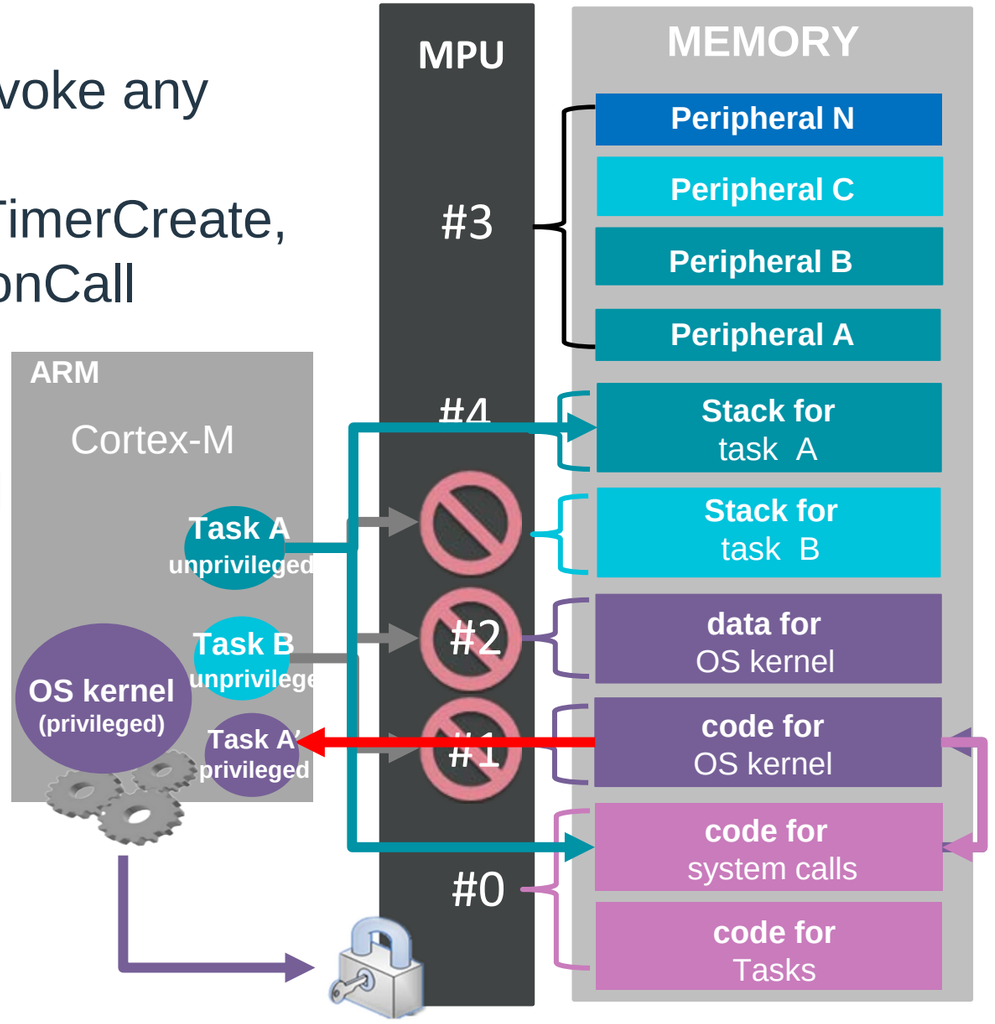
- Bug3 (v10.4.6 and before): An unprivileged task can invoke any function with privilege by passing it as a parameter to MPU_xTaskCreate, MPU_xTaskCreateStatic, MPU_xTimerCreate, MPU_xTimerCreateStatic, or MPU_xTimerPendFunctionCall
- Cause: Privileged and unprivileged tasks can be created with the same kernel API (xTaskCreate) with different parameters (*uxPriority*) which is also wrapped within many system call functions

uxPriority

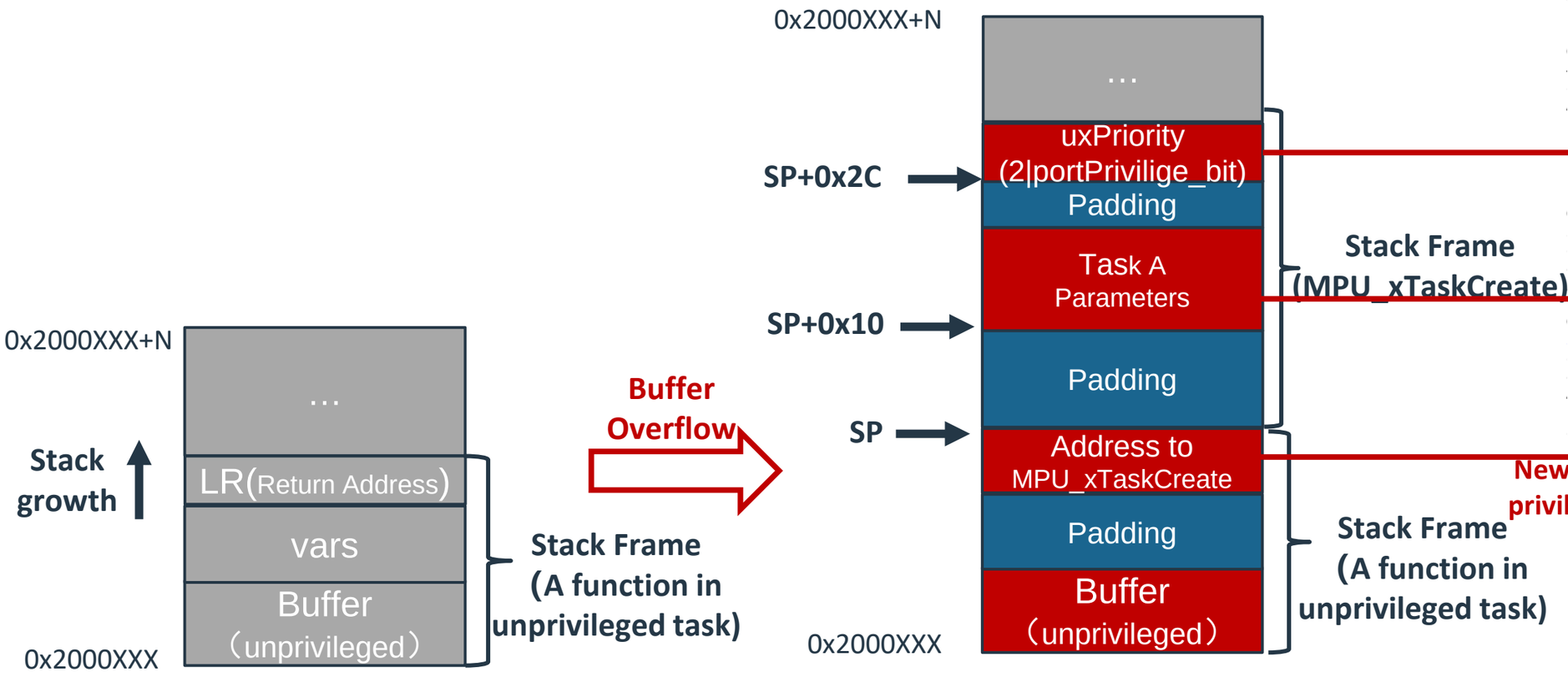
The priority at which the created task will execute.

Systems that include MPU support can optionally create a task in a privileged (system) mode by setting the bit `portPRIVILEGE_BIT` in `uxPriority`. For example, to create a privileged task at priority 2 set `uxPriority` to (2 | `portPRIVILEGE_BIT`).

Priorities are asserted to be less than `configMAX_PRIORITIES`. If `configASSERT` is undefined, priorities are silently capped at (`configMAX_PRIORITIES` - 1).



Exploitation Steps




```
MPU_xTaskCreate

                PUSH    {R4,LR}
                SUB     SP, SP, #0x20
                LDR.W   R12, [SP,#0x2C]
                LDR.W   R12, [SP,#0x28]
                STR     R0, [SP,#0x1C]
                STR     R1, [SP,#0x18]
                STRH.W  R2, [SP,#0x16]
                STR     R3, [SP,#16]
                BL      xIsPrivileged
                CBNZ    R0, loc_800804E
                B       loc_800801E
; ---------------------------------------------------------------------------

loc_800801E                                     ; CODE XREF: MPU_x
                SVC     2
                LDR     R0, [SP,#0x28+arg_0]
                BIC.W   R0, R0, #0x80000000
                STR     R0, [SP,#0x28+arg_0]
                LDR     R0, [SP,#0x1C]          ; pxTaskCode
                LDR     R1, [SP,#0x18]          ; pcName
                LDRH.W  R2, [SP,#0x16]          ; usStackDepth
                LDR     R3, [SP,#0x10]          ; pvParameters
                LDR.W   R12, [SP,#0x28]         ; pxCreatedTask
                LDR     R4, [SP,#0x2C]          ; uxPriority
                MOV     LR, SP
                STR.W   R4, [LR,#4]             ; pxCreatedTask
                STR.W   R12, [LR]               ; uxPriority
                BL      xTaskCreate
                STR     R0, [SP,#0x28+var_1C]
                BL      vResetPrivilege
                B       loc_8008070
; ---------------------------------------------------------------------------

loc_800804E                                     ; CODE XREF: MPU_x
                LDR     R0, [SP,#0x1C]          ; pxTaskCode
                LDR     R1, [SP,#0x18]          ; pcName
                LDRH.W  R2, [SP,#0x16]          ; usStackDepth
                LDR     R3, [SP,#0x10]          ; pvParameters
                LDR.W   R12, [SP,#0x28]         ; pxCreatedTask
                LDR     R4, [SP,#0x2C]          ; uxPriority
                MOV     LR, SP
                STR.W   R4, [LR,#4]             ; pxCreatedTask
                STR.W   R12, [LR]               ; uxPriority
                BL      xTaskCreate
                STR     R0, [SP,#0x28+var_1C]
                B       loc_8008070
; ---------------------------------------------------------------------------

loc_8008070                                     ; CODE XREF: MPU_x
                                                ; MPU_xTaskCreate+
                LDR     R0, [SP,#0x28+var_1C]
                ADD     SP, SP, #0x20
                POP     {R4,PC}
```

New task with privileged level

Attack Demo

D:\IoTSecLab\jmpsvc2-freertos\Keil\JmpSvc2.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

MPU_xTaskDelay

JmpSvc2

Project

- Project: JmpSvc2
  - JmpSvc2
    - App
      - app_main.c
      - app_main.h
      - jmpsvc2.c
      - jmpsvc2.h
      - retarget_io.c
    - Startup
      - startup_stm32l475xx.s
      - memfault_handler.c
    - FreeRTOS
      - croutine.c
      - event_groups.c
      - list.c
      - queue.c
      - stream_buffer.c
      - tasks.c
      - timers.c
      - mpu_wrappers.c
      - port.c
      - portmacro.h
      - heap_4.c
    - Config
      - FreeRTOSConfig.h
    - Core
      - main.c
      - stm32l4xx_it.c
      - stm32l4xx_hal_msp.c
      - stm32l4xx_hal_timebase_tin
    - Drivers/STM32L4xx_HAL_Driver
      - stm32l4xx_hal_dfsdm.c
      - stm32l4xx_hal_i2c.c
      - stm32l4xx_hal_i2c_ex.c
      - stm32l4xx_hal_qspi.c

Project | Books | {} Functio... | {}, Templa...

jmpsvc2.c | mpu_wrappers.c | startup_stm32l475xx.s | main.c | stm32l4xx_hal_uart.c | port.c | stm32l4xx_hal_pwr_ex.c | tasks.c

```c
void PrintMyName() {
  uint8_t receiveBuffer [ 32 ]; /* This buffer can be overflowed. */

  printf("Please input your name:\r\n");
  serial_scanf(&huart1, "%s", receiveBuffer);
  printf("Your name: %s\r\n", receiveBuffer);
}

void UnprivilegedTask( void * pvParameters ) {
  ( void ) pvParameters; /* Unused parameters. */
  uint8_t paddingBuffer [ 64 ];

  printf("UnprivilegedTask running.\r\n");

  if ( portIS_PRIVILEGED() == pdFALSE ) {
    printf("Current task is unprivileged.\r\n");
    PrintMyName();
    taskYIELD();
  } else {
    printf("Current task is privileged.\r\n");
    printf("Attack successful!\r\n");
  }

  for( ; ; )
  {
    MPU_vTaskDelay( pdMS_TO_TICKS( 1000 ) );
  }
}

void vStartJmpSvc2Demo( void )
```

Build Output

```
Flash Load finished at 00:05:31
Load "Debug\\JmpSvc2.axf"
Erase Done.
Programming Done.
Verify OK
```

Build Output | Find In Files | All References of 'tskIDLE_PRIORITY'

ST-Link Debugger | L:40 C:13 | CAP NUM SCRL OVR R/W

7°C 阴 | 搜索 | 英 | 0:22 2022/12/7

Common pitfalls in using MPU

- Weak protection
  - Case study: Bypassing MPU protection in RIoT-MPU
  - Case study: Privileged escalation in FreeRTOS-MPU
- **Incomplete protection**
- **Prohibitive overhead**
- **Conflict with existing system designs**

Incomplete protection

- No protection for interrupt handlers
  - Exception vector reads from the Vector Address Table always use the default system address map and are not subject to an MPU check
  - Interrupt handlers (handle mode) run in the privileged mode, which can access any resources
- Incomplete protection for peripherals
  - Any load, store or instruction fetch transactions to the PPB, within the range 0xE0000000-0xE00FFFFF (system peripherals), are not subject to an MPU check.
  - Due to the programming constrains (e.g., at least 32B and alignment), MPU is not suitable for protecting peripherals with small regions

Incomplete protection

- Incomplete permissions assignment
  - No execute-only (XO) permission
  - Privileged permission ≥ Unprivileged permissions

Prohibitive overhead

- To leverage MPU to realize kernel/task isolation, invocation to kernel APIs has to go through context switch twice
  - Our experiment shows that one thousand privilege switches in a FreeRTOS-MPU system **takes 3.5ms** on average on the MPS2+ FPGA prototyping system broad (Cortex-M4 AN386) with 25MHZ CPU clock frequency.
- MPU regions need to be re-configured for different tasks and applications.
  - FreeRTOS has to reset MPU regions #5-7 during an application switch
  - Tizen has to reset MPU regions #3-7 during an app (including multiple tasks) switch and #6 and #7 during a task switch

Conflict with exiting system design

- Limited MPU regions for real world applications
  - Very few available user-defined regions for peripheral isolation
    - No OS provides peripheral isolation by default.
  - Very few available regions shared between two tasks
    - No OS provides shared memory protection by default.
  - Impossible to enable too many security features at same times
    - E.g.: When activating all MPU features provided by Tizen, there is no more available MPU regions on ARM Cortex-M0+/M3/M4 based MCUs which only support eight MPU regions
- Porting software leveraging MPU may cause compatibility issues
  - Only 30% manufacturers implement MCU hardware security features into current design

Agenda

- Introduction to Memory Protection Unit (MPU)
- MPU adoption in the wild
- Common pitfalls and limitations in using MPU
- **Mitigation suggestions**
- Summary and disclosure

Minimizing pitfalls

- Be careful about permission overlap
  - **Observation:** Most OSs use lower-number MPU regions for kernel protections (All open-source OS except for latest FreeRTOSv10.5).
  - **Risk:** Developer could configure those higher numbered user-defined MPU regions to override kernel protections.
  - **Recommendation:** System and general protection (e.g., KMI, DEP,CIP) should use higher-number MPU regions.

Minimizing pitfalls

- Be careful about privilege switch during system call
  - **Observation:** OSs wrap the kernel APIs with separated privilege switch function as system calls (e.g., FreeRTOSv10.5.0 before).
  - **Risk:** Privilege escalation with control flow hijacking attack or a manipulable stack.
  - **Recommendation:** MCU OS should use individual system calls for kernels API with software interrupts like Linux or additional caller checks should be performed before system call invocations, and the kernel should make sure the privilege is dropped after system calls.

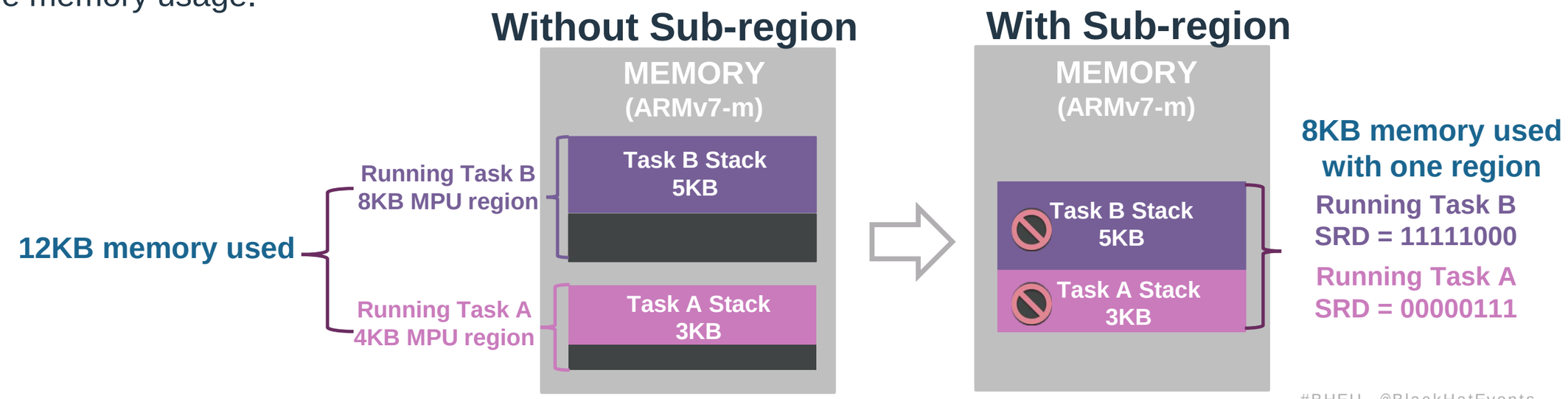
Minimizing pitfalls

- Privilege separation is also needed for general protections
  - **Observation:** OSs which only provide protections like Stack Guard, DEP and CIP, always run the whole system at the privileged level like RIoT.
  - **Risk:** Disabling the desired protections by reconfiguring MPUs with control flow hijacking attack.
  - **Recommendation:** System should drop privilege immediately after MPU configuration.

Region usage optimization

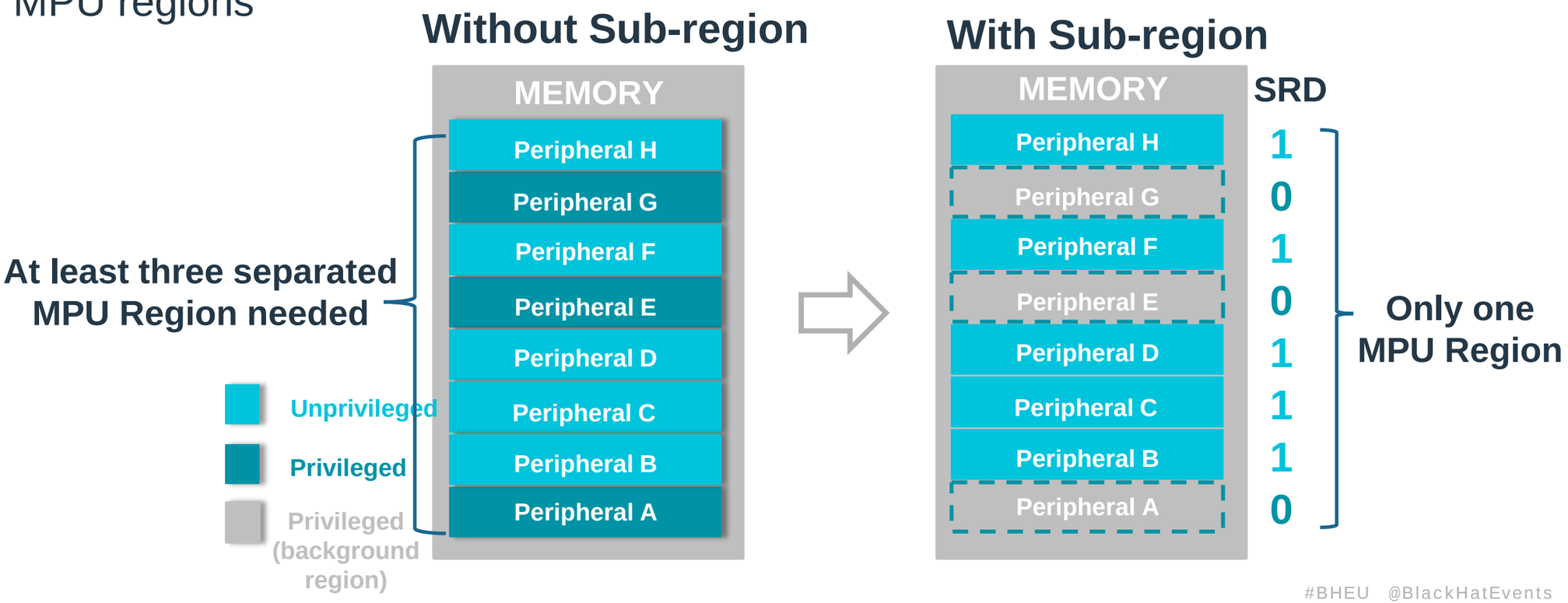
- Be aware of the default ARMv7-M address map permissions.
  - Default memory access permissions/attributes of memory regions is enforced by ARM without MPU
  - E.g., non-executable for standard and system peripheral regions

- Taking advantage of Sub-regions
  - Save memory usage.



Region usage optimization

- Taking advantage of Sub-regions
  - Saving memory usage
  - Saving MPU regions



Software workaround

- Protecting the user-defined sensitive resource (i.e., ensuring code can only access its required data) rather than OS itself
  - Minor (NDSS 2018) isolates tasks with memory view switches (task and kernel are all running on unprivileged level) to avoid privilege escalation
  - ACES (USENIX Security 2018) isolates compartments based on code functionality.

Hardware Retrofitting

- A redesigned MPU can addresses the insecurity and inflexibility in a lightweight way.
  - ARMv8-M architecture extends TrustZone technology to Cortex-M series. The secure regions can be used as additional regions and be assigned with higher privileged level beyond privileged level in normal world.
  - Trustlite proposed execution-aware MPU which the not only validates data accesses (read/write/execute) but additionally considers the currently active instruction pointer as the subject performing the access.

Agenda

- Introduction to Memory Protection Unit (MPU)
- MPU adoption in the wild
- Common pitfalls and limitations in using MPU
- Mitigation suggestions
- **Summary and disclosure**

Summary

- To our surprise, we found **that MPU as a ready-to-use security feature** for protecting microcontroller **is rarely used in real-world products**
- We studied the source code of multiple MCU OSs to find explanations for this situation and eventually **identified some common pitfalls**.
- Some of the flaws **are fundamental and not remedial in a short term**
- We give **recommendations for better use of MPU**

Disclosure

- All bugs we demonstrated has been patched in latest FreeRTOS kernel
  - Security update Reference: https://www.freertos.org/security/security_updates.html
- RIoT developer team has acknowledged our finding, but the benefit of disabling access to the MPU or the `mpu_disable()` function without a userspace / kernelspace split is quite limited, only mildly increases the attack surface in the context of the attack model RIOT assumes.

black hat EUROPE 2022

DECEMBER 7-8, 2022

BRIEFINGS

Thank you

Wei Zhou, Zhouqi Jiang, Le Guan

華中科技大學

UNIVERSITY OF GEORGIA