1. Goal of the lab

We want to show:

- Kernel thread can touch “secret” memory ✅
- User thread is *blocked* by MPU ❌
- When user thread tries anyway → Zephyr raises MemManage fault → kills that thread

This proves memory isolation: just because you’re running code on the same MCU doesn’t mean you get full access.

Perfect story for teaching secure firmware, privilege separation, safety RTOS.

---

2. Lab flow (what students do / see)

1. Define a “sensitive buffer” in a special memory section.
2. Give kernel thread access to it.
3. Create a user-mode thread (unprivileged).
4. DO NOT grant that user thread permission to this buffer.
5. Have user thread try to read it.
6. Watch it crash with a memory protection fault while the kernel thread continues running fine.

In class you say:

> “Congratulations. You just saw userspace die without taking the system down. That’s RTOS-grade process isolation.”

---

3. Kconfig / prj.conf requirements

Your `prj.conf` **must** enable:

```
# basic RTOS
CONFIG_MAIN_STACK_SIZE=2048
CONFIG_HEAP_MEM_POOL_SIZE=4096

# threads + scheduling
CONFIG_THREAD_NAME=y

# userspace / MPU support
CONFIG_USERSPACE=y
CONFIG_ARM_MPU=y
CONFIG_EXCEPTION_STACK_TRACE=y
CONFIG_PRINTK=y
CONFIG_LOG=y
CONFIG_LOG_DEFAULT_LEVEL=3
```

Why each matters:

- `CONFIG_USERSPACE` → lets us create unprivileged threads and control memory domains.
- `CONFIG_ARM_MPU` → actually programs the Cortex-M MPU regions.
- `CONFIG_EXCEPTION_STACK_TRACE` → we get a helpful backtrace when it blows up, good for demo.
- `CONFIG_HEAP_MEM_POOL_SIZE` → needed because `k_thread_create()` with user mode needs memory from a pool Zephyr can manage safely.

---

4. Full code

File: `src/main.c`

```
/*
 * Demo: Memory Protection Fault in Zephyr
 *
 * Scenario:
 *  - kernel_thread: privileged, can read secret_data
 *  - user_thread: unprivileged, NOT granted access, tries to read
-> boom
 *
 * Works on Cortex-M with MPU, e.g. STM32F407 + Zephyr.
 */

#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>
#include <zephyr/sys/util.h>
#include <zephyr/app_memory/app_memdomain.h>
#include <zephyr/app_memory/app_memdomain_defs.h>
#include <zephyr/sys/mem_manage.h>

/* ----------------------------
 * 1. Sensitive data region
 * ----------------------------
 *
 * We place this in its own app memory partition so we can decide
 * who can touch it. Mark it as "APP_SHARED" but DO NOT map it
```

```c
 * into the user thread's memory domain.
 *
 * On ARM-M MPU, each partition becomes an MPU region with specific
 * permissions per thread/domain.
 */

__aligned(32) __attribute__((section(".secret_data")))
static uint8_t secret_data[32] = "TOP_SECRET_KEY_MUST_NOT_LEAK";

/* Create a memory partition descriptor for secret_data */
K_APPMEM_PARTITION_DEFINE(secret_partition);
APP_MEMORY_REGION(secret_partition, secret_data);

/*
 * Explanation:
 * - K_APPMEM_PARTITION_DEFINE(...) creates metadata Zephyr uses
 *   to build MPU regions.
 * - APP_MEMORY_REGION ties our buffer to that partition.
 *
 * We will later add ONLY the kernel thread to this partition.
 * The user thread will *not* get mapped to it.
 */
```

```
/* ----------------------------
 * 2. Thread stacks
 * ----------------------------
 *
 * For user threads we MUST use K_THREAD_STACK_DEFINE(), not a raw
array,
 * because Zephyr needs to mark that memory as user-accessible.
 */

#define KERNEL_STACK_SIZE 1024
#define USER_STACK_SIZE   1024
#define USER_PRIO         3
#define KERNEL_PRIO       2

K_THREAD_STACK_DEFINE(kernel_stack, KERNEL_STACK_SIZE);
K_THREAD_STACK_DEFINE(user_stack,   USER_STACK_SIZE);

static struct k_thread kernel_thread_data;
static struct k_thread user_thread_data;

/* Forward decls */
void kernel_thread_fn(void *, void *, void *);
void user_thread_fn(void *, void *, void *);
```

```
/* ----------------------------
 * 3. Memory domain setup
 * ----------------------------
 *
 * We'll build:
 *   - kernel_domain: includes secret_partition
 *   - user_domain:   DOES NOT include secret_partition
 *
 * Both domains will still have access to their own stacks and to
 * the generic Zephyr kernel objects they're allowed to use.
 */

static struct k_mem_domain kernel_domain;
static struct k_mem_domain user_domain;

/* Helper: Build the kernel domain (privileged thread) */
static void init_kernel_domain(void)
{
    /* Give this domain access to the secret partition */
    struct k_mem_partition *parts[] = {
        &secret_partition,
    };

    k_mem_domain_init(&kernel_domain, ARRAY_SIZE(parts), parts);
```

```c
    printk("kernel_domain: initialized with secret_partition
access\n");
}

/* Helper: Build the user domain (unprivileged thread) */
static void init_user_domain(void)
{
    /* User domain intentionally gets NO secret partition */
    k_mem_domain_init(&user_domain, 0, NULL);

    printk("user_domain: initialized WITHOUT secret_partition
access\n");
}


/* ----------------------------
 * 4. Kernel thread
 * ----------------------------
 *
 * This runs in supervisor mode (privileged).
 * It is attached to the kernel_domain so it can read secret_data.
 */

void kernel_thread_fn(void *a, void *b, void *c)
```

```
{
    ARG_UNUSED(a); ARG_UNUSED(b); ARG_UNUSED(c);

    /* Attach current thread (this thread) to kernel_domain */
    k_mem_domain_add_thread(&kernel_domain, k_current_get());

    printk("[KERNEL] I am privileged.\n");
    printk("[KERNEL] I can read secret_data: \"%s\"\n",
secret_data);

    while (1) {
        printk("[KERNEL] still alive, system running.\n");
        k_sleep(K_MSEC(1000));
    }
}


/* ----------------------------
 * 5. User thread
 * ----------------------------
 *
 * This will be dropped to user mode with
k_thread_user_mode_enter().
 * It is attached to user_domain (no access to secret_data).
 *
```

```
 * When it tries to read secret_data, MPU should fault.
 */

static void user_mode_entry(void *p1, void *p2, void *p3)
{
    ARG_UNUSED(p1); ARG_UNUSED(p2); ARG_UNUSED(p3);

    printk("[USER ] I am unprivileged now.\n");
    printk("[USER ] Attempting to read secret_data...\n");

    /* Volatile read so compiler doesn't optimize it out */
    volatile uint8_t first_byte = secret_data[0];

    /* If we ever get here without fault, something is wrong */
    printk("[USER ] I managed to read secret_data[0]=0x%02x (THIS
SHOULD NOT HAPPEN!)\n",
           first_byte);

    while (1) {
        k_sleep(K_MSEC(500));
    }
}

void user_thread_fn(void *a, void *b, void *c)
{
```

```
    ARG_UNUSED(a); ARG_UNUSED(b); ARG_UNUSED(c);

    /* Attach this (still privileged-at-this-exact-moment) thread
     * to the restricted user_domain BEFORE dropping privilege.
     */
    k_mem_domain_add_thread(&user_domain, k_current_get());

    printk("[USER_SETUP] Attached to user_domain (no secret
access)\n");
    printk("[USER_SETUP] Dropping to user mode now...\n");

    /* After this call returns into user_mode_entry(), thread runs
     * unprivileged with MPU enforcing access rules.
     */
    k_thread_user_mode_enter(user_mode_entry, NULL, NULL, NULL);

    /* NOTE: We should NEVER come back here. If we do, print it. */
    printk("[USER_SETUP] ERROR: Returned from user_mode_enter?!\n");
    while (1) { k_sleep(K_MSEC(1000)); }
}


/* ----------------------------
 * 6. main()
 * ----------------------------
```

```
 *
 * main() runs as a privileged thread in Zephyr by default.
 * We:
 *  1. Create memory domains.
 *  2. Spawn kernel_thread_fn() (privileged).
 *  3. Spawn user_thread_fn()   (will drop to user mode).
 */

void main(void)
{
    printk("\n=== Zephyr MPU Fault Demo ===\n");

    init_kernel_domain();
    init_user_domain();

    /* Create KERNEL thread (privileged, higher priority) */
    k_thread_create(&kernel_thread_data,
                    kernel_stack,
                    K_THREAD_STACK_SIZEOF(kernel_stack),
                    kernel_thread_fn,
                    NULL, NULL, NULL,
                    KERNEL_PRIO, /* priority */
                    0,           /* options: 0 -> start privileged
*/
                    K_NO_WAIT);
```

```
    k_thread_name_set(&kernel_thread_data, "kernel_thread");

    /* Create USER thread (will self-demote) */
    k_thread_create(&user_thread_data,
                    user_stack,
                    K_THREAD_STACK_SIZEOF(user_stack),
                    user_thread_fn,
                    NULL, NULL, NULL,
                    USER_PRIO,   /* slightly lower priority is fine
*/
                    K_USER,      /* <-- important: create as user
thread context */
                    K_NO_WAIT);


    /*
     * NOTE:
     *  - Passing K_USER here marks the thread as a user thread
object,
     *    which means Zephyr will prepare it for user mode
constraints.
     *  - But inside user_thread_fn(), we *explicitly* drop it to
user mode
     *    with k_thread_user_mode_enter() so the MPU enforcement is
live
     *    when accessing secret_data.
     *
```

```
     * Depending on Zephyr version/arch, you can also start already
     * unprivileged and skip user_mode_enter(). We keep it explicit
     * for teaching.
     */

    k_thread_name_set(&user_thread_data, "user_thread");

    printk("main(): both threads created.\n");
}
```

---

5. What you'll see on UART / console

Happy path (kernel thread):

```
=== Zephyr MPU Fault Demo ===
kernel_domain: initialized with secret_partition access
user_domain: initialized WITHOUT secret_partition access
main(): both threads created.
[KERNEL] I am privileged.
[KERNEL] I can read secret_data: "TOP_SECRET_KEY_MUST_NOT_LEAK"
[KERNEL] still alive, system running.
...
```

User thread before fault:

```
[USER_SETUP] Attached to user_domain (no secret access)
```

```
[USER_SETUP] Dropping to user mode now...

[USER ] I am unprivileged now.

[USER ] Attempting to read secret_data...
```

Then BOOM, from Zephyr fault handler:

You'll get something like (exact wording depends on Zephyr version / arch config):

```
***** MPU FAULT *****

  Data Access Violation

  Address: 0x200000ac  (somewhere inside secret_data)

  Thread: user_thread

  Reason: Permission Denied

Fatal fault in thread user_thread! Aborting.
```

The important storytelling parts for students:

- Only `user_thread` dies.
- System doesn't reset.
- `kernel_thread` is still printing `[KERNEL] still alive...` every 1s.
  That shows isolation.

---

6. How to present this in class (talk track for you)

- "Zephyr is not 'just an RTOS'. It can enforce process-like isolation, like Linux user vs kernel."
- "Even on STM32F4, we get per-thread MPU rules, not just global MPU."
- "This is why you run untrusted sensor/comm stacks in userspace and keep crypto keys in kernel-only memory."

Then ask them:

> “Now imagine this: what if this ‘secret_data’ is your AES key or firmware decryption key? Do you really want your MQTT task to be able to read it?”

This lands the security / safety / functional safety (FuSa) message.

---

7. Common issues you should be ready to fix live

I’ll list them so you don’t get stuck mid-session:

1. **Board must support `CONFIG_USERSPACE`.**
   Some minimal Zephyr boards don’t implement it fully. STM32F4 does, nRF does, etc. If board doesn’t: you’ll get build errors about MPU/user mode symbols.

2. **Stack size too small.**
   If user thread stack is tiny, Zephyr may fault before your access test and students will think “MPU worked”. Increase `USER_STACK_SIZE` to 1024 or even 2048 for safety.

Linker section `.secret_data` not mapped.
If your build complains about `.secret_data` being unknown, add this to your board/linker or use `__attribute__((section(".app_smem")))` with Zephyr’s built-in application memory section. Some Zephyr versions require `APP_SHARED_VAR()` macros instead of manual section. If so, adaptation:

```
APP_SHARED_VAR(static uint8_t secret_data[32])

    = "TOP_SECRET_KEY_MUST_NOT_LEAK";
```

3. And then adjust partitions accordingly. Point is: keep it in a region you *don’t* grant.

4. **Optimization removes the read.**
   We used `volatile` to force actual load from memory so MPU triggers.

---

8. How you can extend this for Day 2 (Execution Modes & MPU)

If you want to make it more advanced for the next half-day slot:

- After the fault, print `k_thread_foreach()` from kernel thread to show that `user_thread` is now gone.

- Add a second user thread that *is* granted access to the secret (k_mem_domain_add_partition) and show that it succeeds. This proves fine-grained policies.

Something like:

- user_thread_A → no access → dies

- user_thread_B → added to kernel_domain → reads secret successfully even though it's still "user mode"
  This is 🔥 because it teaches: privilege level (kernel/user) and memory domain membership are orthogonal knobs.