CY-4973/7790

Kernel Security: how2rootkit

What Is an Operating System?

- A fancy resource manager.
- Abstracts away device management (CPU, memory, I/O devices, etc.).
- Exports logical interface for user applications (system calls, libraries) to interact with hardware indirectly.
- Why? **makes it easier to write programs that run on a variety of hardware without knowing all low level details **
    - we will focus on its ability to make running programs simple--i.e. without having to care about the gory details of the underlying hardware

Operating System Kernel

- The component of the OS responsible for *virtualization* of resources.
    - The kernel is a dirty dirty liar
- Coordinates CPU, memory, and I/O device interactions.
- Implements isolation and protection so userland processes don’t directly manipulate hardware.
- Comes in many flavors such as monolithic vs micro vs hybrid

The Linux Operating System

- Open-source, community-developed.
- Multi-architecture: supports 32-bit, 64-bit x86, ARM (including AArch64), RISC-V, and more.
- Broad usage in servers, desktops, embedded systems, IoT devices, HPC, mobile (Android), and beyond.
- We’ll focus on a 64-bit ARM (ARMv8-A / AArch64) Ubuntu distribution of Linux.
    - Blah blah "GNU+Linux" Blah Blah

Devices That Run Linux

- Desktop computers and laptops
- Cloud servers (e.g., AWS Graviton)
- Mobile devices and tablets (Android)
- Embedded and IoT platforms (Raspberry Pi, various ARM boards, embeded linux)
- Supercomputers (HPC clusters)
- Each environment has different constraints (power, screen size, memory, etc.).

The Linux Kernel

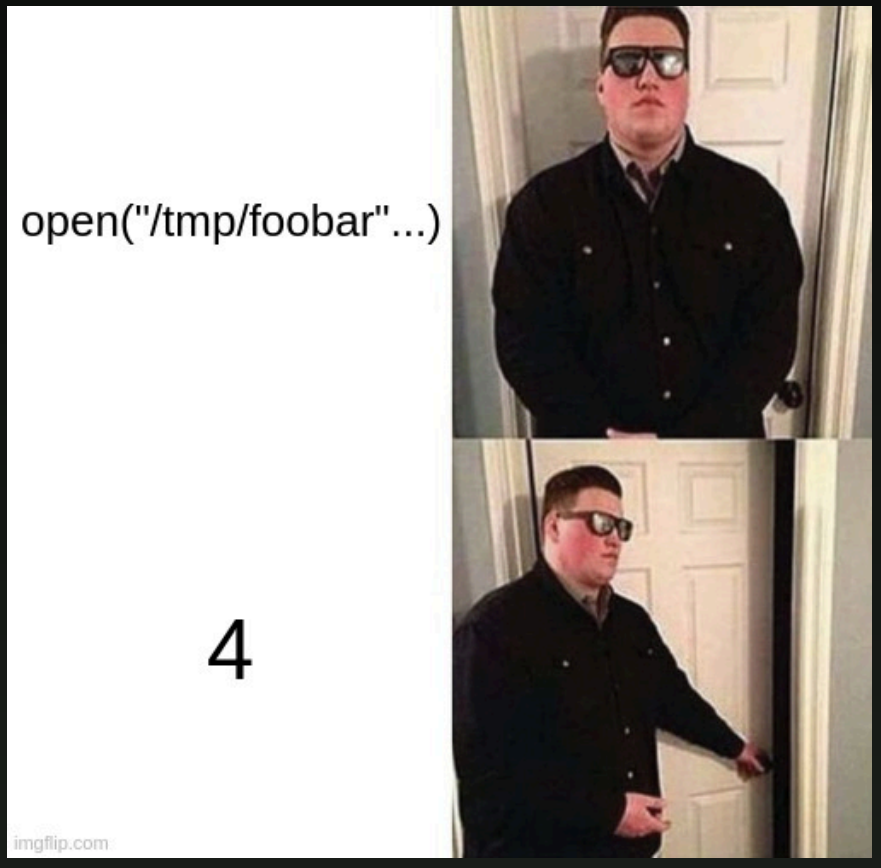
- Monolithic design (optionally) with dynamically loadable modules.
- Userland vs Kernel
- Most low level objects and their lifetimes are managed by the *kernel*
- These objects are represented as a *file descriptor* (including devices) in userland.
- Resource objects exist in-kernel (task_struct, inodes, etc.) but are not directly accessible to userland.
- The kernel brokers access to these objects and exports an interface for userland in the form of System calls (syscalls)
    - These require hardware support to implement

Kernel Objects

- The kernel organizes resources into structures, like `task_struct` (for processes), `file` and `inode` (for open files).
- These are not typically called “objects” in Linux, but are
- Access from userland is via file descriptors or handles returned by syscalls (e.g., `open()`).
- the kernel manages lifetimes of file descriptors

Objects and File Descriptors

- A file descriptor (FD) is an integer index into a per-process table referencing kernel structures.
- This allows the kernel to abstract away direct memory management of resources.
- Processes can share FDs or pass them around, enabling resource sharing with controlled security.



Interacting With the Linux OS

- "Everything is a file"
- sh/bash/zsh...etc
- System calls: The core API to talk to the kernel (e.g. `open`, `read`, `write`, `fork`, `execve`).
- Several libraries that ultimately make syscalls
- C library: Offers convenient wrappers around syscalls.
- POSIX / Extended Linux APIs.
- Direct Syscalls: Rarely done manually in C - usually you call glibc or another library. - Can be useful if for whatever reason you don't know where Libc is in memory or if it isn't loaded at all!

Example Files

- `/dev/null` – a virtual device that discards all data.
- `/dev/tty` – represents the current terminal.
- `/proc/cpuinfo` – provides CPU information.
- `/proc/self` – provides information about the current running process.
- `stdout`, `stdin`, and `stderr` are three standard streams that are pre-connected to a program when it starts.
- They are used for input, output, and error messages, respectively and are assigned file descriptors 0,1,2

procfs

- `/proc` is a pseudo-filesystem that contains runtime system information about processes and the kernel.
- i.e. a virtual file system that presents information or functionality not stored on a physical disk
- **Examples**:
    - `/proc/<PID>/maps` – memory mappings of a process.
    - `/proc/<PID>/fd/` – symbolic links to file descriptors of a process.
    - `/proc/<PID>/cmdline`: commandline (argv) of the process
    - `/proc/<PID>/status` information about the process (name, pid, ppid ..etc)
    - can reference current process via `/proc/self/`

Process Resources

- PID (unique to each process).
- Security context (UID, GID, supplementary groups).
- Table of open file descriptors.
- A private (or shared) virtual address space.
- One or more threads.

```
 $ sudo ls /proc
1      16     216    309    372    42730  470     481
11     16982  22     31     373    430      471     483
117    17     232    31239  39       431      472     4
11959  179    233    31298  4       43519  473     485
11960  18     24     31299  40       44      474     48
12     18404  25     32     40683  444      47687  49
120    19     26     32647  40689  45      477     493
12485  19054  27     32784  40737  46      47769  49328
12880  19055  278    32795  40768  462      47770  4968
13     2      29     34     41566  463      478     496
14     20     3      369    41605  46482  479     5
143    21     30     370    42       46728  47983  50
15     215    30388  371    42724  47      480     5038
$ ps aux | head
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT S
root           1  0.0  0.1 170592 11680 ?        Ss   J
root           2  0.0  0.0      0     0 ?        S    J
root           3  0.0  0.0      0     0 ?        S    J
root           4  0.0  0.0      0     0 ?        I<   J
root           5  0.0  0.0      0     0 ?        I<   J
root           6  0.0  0.0      0     0 ?        I<   J
root           7  0.0  0.0      0     0 ?        I<   J
root           9  0.0  0.0      0     0 ?        I<   J
root          11  0.0  0.0      0     0 ?        I    J
```

/proc/%d/fd/

- entries in the file descriptor table can be found in `/proc/<pid>/fd/<fd_num>`
- echo "hi there" >/proc/`echo $`/fd/0

Processes

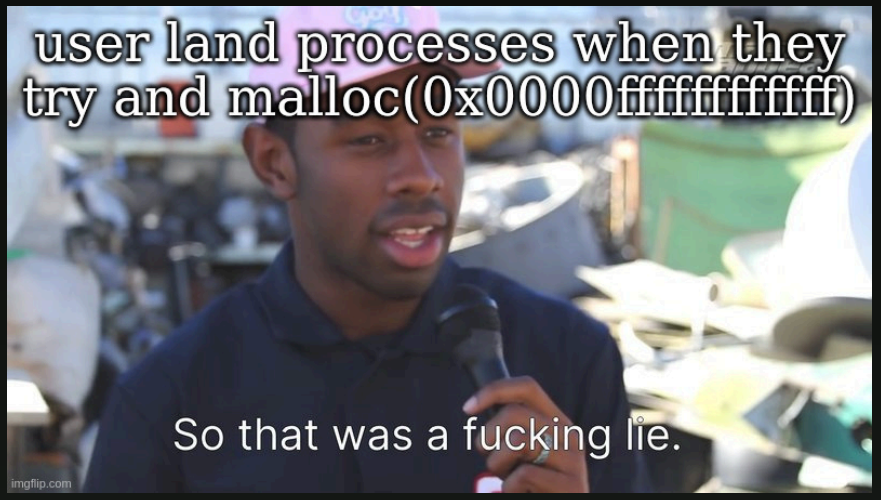
- A process is essentially a container for resources required to run a program: memory, file descriptors, scheduling parameters, etc.
- All userland code runs inside *some* process context.
- A process is identified by a PID (process ID).
- PIDs are globally unique
- PPID = parent PID
- Code actually runs via one or more threads within the process.

Threads

- The actual *unit of execution*.
- Each thread has its own CPU state (registers, stack pointer).
- Linux implements threads as processes that share many resources (e.g., memory, FD table).
- On ARM64, each thread can independently run on any available CPU core.

Virtual Memory

- User land perspective: memory is a linear block of byte-addressable memory
- Physical memory: a linear array of bytes.
- Logically organized into Physical Pages
- Usually Address is Physical Page Number (PPN) + offset
- On ARM64, addresses are 64-bit but typically not all 64 bits are used (e.g., 48-bit or 52-bit virtual addressing depending on kernel config).
- Byte-addressable means each address points to an 8-bit chunk.
- Memory is organized into segments.
  - `/proc/self/maps`
  - `/proc/self/mem`



Virtual Memory

- Each process sees its own *private virtual address space*.
- For 64-bit ARM, theoretically up to 2^{52} , though practically less.
- The kernel's memory manager maps virtual pages to physical pages.
- The process has the illusion of a large contiguous memory region.
- Memory can be shared by mapping it as shared

Process Virtual Memory

- Linux uses page tables to map virtual to physical memory.
- Kernel and user share the same address space mapping but in separate regions (on typical 64-bit distros, top addresses for kernel, lower for user).

```
$ cat /proc/self/maps
5555f9310000-5555f931c000 r-xp 00000000 103:02 1777
5555f932c000-5555f9330000 r--p 0000c000 103:02 1777
5555f9330000-5555f9334000 rw-p 00010000 103:02 1777
5555fcab8000-5555fcadc000 rw-p 00000000 00:00 0
7fff13b5c000-7fff13b84000 rw-p 00000000 00:00 0
7fff13b84000-7fff13e70000 r--p 00000000 103:02 10954
7fff13e70000-7fff13ff8000 r-xp 00000000 103:02 5543
7fff13ff8000-7fff1400c000 ---p 00188000 103:02 5543
7fff1400c000-7fff14010000 r--p 0018c000 103:02 5543
7fff14010000-7fff14014000 rw-p 00190000 103:02 5543
7fff14014000-7fff14020000 rw-p 00000000 00:00 0
7fff1403c000-7fff14064000 r-xp 00000000 103:02 5306
7fff1406c000-7fff14074000 r--p 00000000 00:00 0
7fff14074000-7fff14078000 r-xp 00000000 00:00 0
7fff14078000-7fff1407c000 r--p 0002c000 103:02 5306
7fff1407c000-7fff14080000 rw-p 00030000 103:02 5306
7fffe7464000-7fffe7488000 rw-p 00000000 00:00 0
```

Virtual Memory Concepts

- Page tables & translation walks (done by hardware MMU).
- Usually 4 KB pages by default on many ARM64 configs, sometimes 64 KB.
- Memory protections (read, write, execute bits).
- Kernel address space vs. user address space.
    - PPN/VPN

Memory Layout

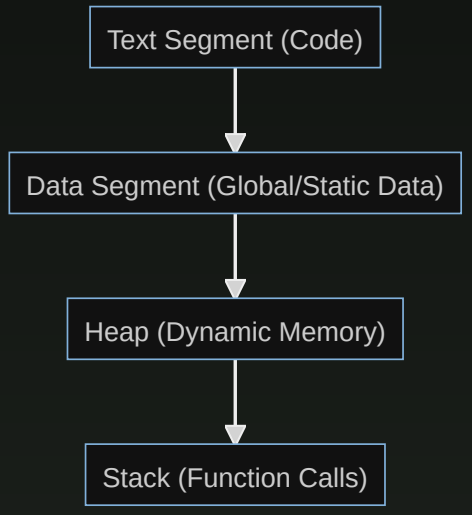
- User space typically in lower half of the virtual address space, kernel in upper half.
- The stack grows downward, the heap grows upward.
- The binary's code (.text), data (.data, .bss), and other segments are mapped into memory at runtime.

Memory

- **Virtual**: The illusion of contiguous memory for each process.
    - **Stack**: Manages local variables and function calls (grows downward).
        - generally used when the size of something is known at compile time
    - **Heap**: Dynamically allocated space (via `malloc()` / `free()` in C).
        - generally used when a size is only known at runtime

Interacting with memory

- ... Managng the heap memory of a process.
- `brk`: Sets the end of the process's data segment.
- sbkr`: change the data segment size
    - positive: increment heap
    - negatie: shrink heap
    -



sbrk example

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    void *initial = sbrk(0); // Get current break point
    printf("Initial break: %p\n", initial);

    sbrk(4096); // Grow heap by 4 KB
    void *new_break = sbrk(0);
    printf("New break: %p\n", new_break);

    return 0;
}
```

mmap/munmap

- Maps a device or file into memory
  - File-backed memory for efficient I/O.
  - Memory sharing between processes.
  - Large, contiguous memory allocation with demand-paged access.

Memory Protections

- PROT_EXEC Pages may be executed.
- PROT_READ Pages may be read.
- PROT_WRITE Pages may be written.
- PROT_NONE Pages may not be accessed
- shellcode example: `mmap` with `PROT_EXEC | PROT_READ | PROT_WRITE`

```
void *exec_mem = mmap(
  NULL,
  sizeof(shellcode),
  PROT_READ | PROT_WRITE | PROT_EXEC,
  MAP_ANONYMOUS | MAP_PRIVATE, -1, 0
  );
```

Disk

- Much slower persistent storage (eMMC, SSD, HDD).
- Cheap, large, but with high latency.
- Access typically goes through the VFS (Virtual File System).
- userland accesses disk through file system
- `/`, `/tmp`, `/etc`, `/root`, `/var`, `/bin`, `/home`
- tree structure
- Common Linux filesystems: `ext4`, `xfs`, `btrfs`.

```
 df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            3.8G     0  3.8G   0% /dev
tmpfs           806M  6.6M  799M   1% /run
/dev/nvme0n1p2  470G   22G  424G   5% /
tmpfs           4.0G     0  4.0G   0% /dev/shm
tmpfs           5.0M   48K  5.0M   1% /run/lock
/dev/nvme0n1p1  510M   64M  447M  13% /boot/firmware
tmpfs           806M   16K  806M   1% /run/user/1000
```

C Overview

- Standard way to interact with OS is to call APIs using an agreed upon ABI
    - in theory any programming language can interact with this interface
- C/C++ is statically typed, compiled, widely used for systems programming.
- Many Linux-based malware or system tools are in C/C++.
- No garbage collector—developers manually manage memory (`malloc`, `free`).
- We'll use it for low-level systems programming on ARM64 Linux.

C Concepts You Need for This Class

- Basic memory management in C (heap, stack).
- Pointer arithmetic.
- Basic C data types.
- Familiarity with Compiler toolchains

Compiler Toolchain for This Class

- **gcc / g++**: The GNU compiler collection.
- **clang / clang++**: LLVM-based toolchain.
- **zig cc**: Another compiler frontend that can cross-compile out of the box.
- We can use `make` or `CMake` to simplify building.
- Cross-compilation: e.g., `aarch64-linux-gnu-gcc` if you're on x86_64 but need ARM64 binaries.

ELF File Format: Basic Definitions

- **ELF** (Executable and Linkable Format) is used by Linux (and many other Unix-like OSes)
- Standard executable file format. When a program is run on a Linux system, it is often times an ELF file that is being executed.
- It describes how code and data are laid out, plus metadata for loading
- Tools like `readelf`, `objdump`, `file` help examine ELF files

ELF File Format on Linux/aarch64

- Used for both userland and kernel modules
  - Userland: typical `.out`, `.so`, or no extension at all
  - Kernel modules: `.ko` files
- Architecture is specified inside the ELF headers (e.g., 0xB7 for AArch64, etc.)
- Data is grouped into segments and sections. Each section has a header with layout info

Dynamically Linked Libraries (so)

Refresh: What Is a Shared Object?

- A file with the ".so" extension that contains compiled code and exported functions.
- Dynamically loaded at runtime by the linker (ld-linux, ld.so).
- Reduces memory footprint and disk usage when multiple processes share the same library code.

Shared Objects in Linux

- ELF files with a “shared” type.
- Provide exported functions that your process can call at runtime.
- Examples: `libc.so`, `libm.so`, `libpthread.so`.
    - usually there: `libcrypso.so`, `libssl.so`

The Linux C Runtime (glibc / musl)

- Most common on mainstream distros is **glibc**. Some embedded systems use **musl** or **uclibc**.
- Offers standard C library functions: I/O, string manipulation, memory allocation.
- Sits on top of the Linux syscall interface.
- Maintains compatibility across kernel versions for user apps.
- Usually dynamically linked, but can be statically linked in some cases (musl)

Example: hello world for the 10th time

```c
#include <stdio.h>

int main(){
    printf("Hello world!\n");
    return 0;
}
```

- Under the hood, this calls `write()` on file descriptor 1 (stdout) via glibc's `printf`.
- On ARM64, glibc uses the `svc #0` instruction to trigger a syscall, passing registers set up for `SYS_write`.

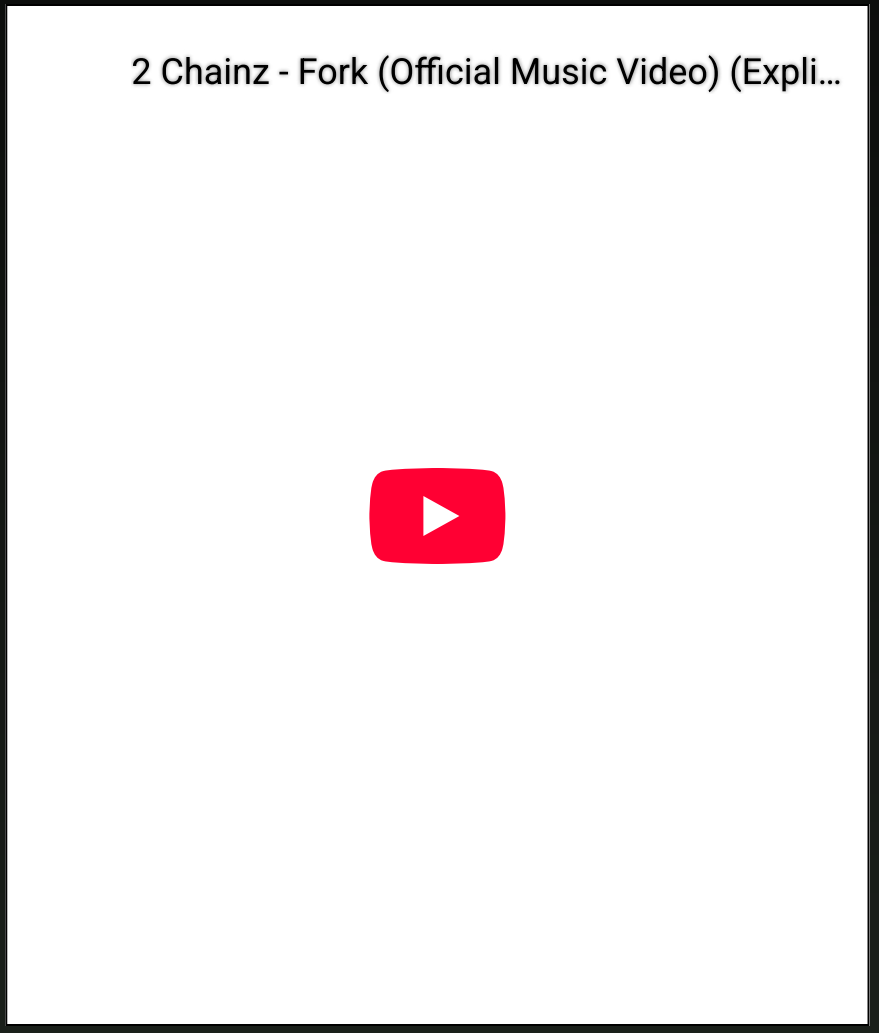
Process Creation

- ultimately, the kernel is responsible for creating/destroying processes
- Linux provides multiple ways to create new processes.
- Key system calls for process creation:
    - `fork`
    - `execve`
    - `posix_spawn()` (not covered)
    - other weird ways
- library functions
- `system()` (calls fork)
    - `popen()` (calls fork)

`fork`

Creates a duplicate of the calling process.

- **Returns:**
  - **0** for the child process.
  - **PID of the child** for the parent process.
  - **-1** if creation fails.*
  - `man fork`



execve

- Replaces the current process with a new one.
- **Does NOT return** if successful.
- **Common variants:** `execl()`, `execv()`, `execle()`, `execvp()`.
- `man execve`

Libc Process creation

- `system: system("ls -l");`
- `popen:` Runs a command and **redirects its output**

ELF File Format: Basic Definitions

- **ELF** (Executable and Linkable Format) is used by Linux (and many other Unix-like OSes)
- It describes how code and data are laid out, plus metadata for loading
- Tools like `readelf`, `objdump`, `file` help examine ELF files

ELF file format: AArch64

- Used for both userland and kernel modules
  - Userland: typical `.out`, `.so`, or no extension at all
  - Kernel modules: `.ko` files
- Architecture is specified inside the ELF headers (e.g., 0xB7 for AArch64, etc.)
- Data is grouped into segments and sections. Each section has a header with layout info

Shared Libraries

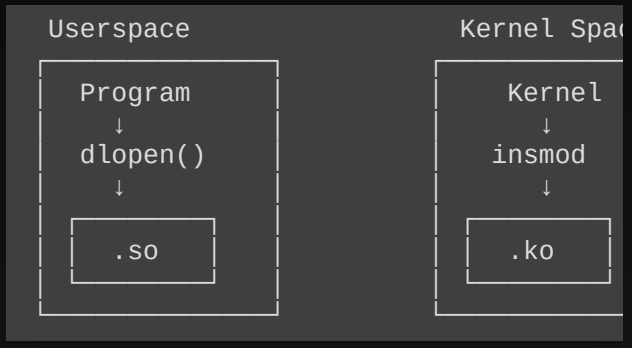
- On Linux, libraries are typically `.so` (“shared object”) files
- They export functions that other ELF executables can dynamically link against
- This is how large programs remain modular
- We’ll see more on linking in a future lecture

ELF Sections

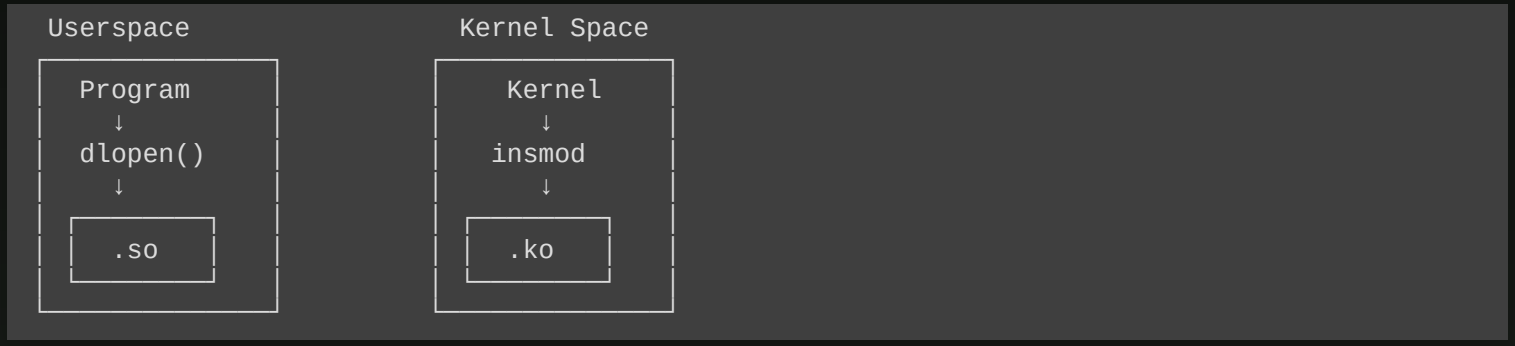
- Typical sections you'll see:
  - `.text`: executable instructions
  - `.rodata`: read-only data
  - `.data`: mutable global data
  - `.bss`: uninitialized data
  - `.got/.plt`: used for dynamic linking
  - `.symtab/.strtab`: symbol tables
- The **Program Header Table** also describes how segments map into memory

From Shared Objects to Kernel Modules

- Building and using shared libraries (`.so`)
- Dynamic linking in userspace
- Transition to kernel space: kernel modules (`.ko`)
- `.so` is to userland as `.ko` is to kernel land
- Hands-on: building, loading, and analyzing both



MOD



What is a Shared Object (.so)?

A **shared object** (`.so` file) is:

- Compiled code that can be loaded into a process at runtime
- Contains functions and data that multiple programs can share
- Reduces memory footprint (one copy in RAM, many users)
- Enables modular software design

Examples:

- `libc.so.6` - C standard library
- `libssl.so` - OpenSSL cryptography
- `libpthread.so` - POSIX threads

Shared Object: Simple Example

mylib.c - A simple shared library:

```c
// mylib.c
#include <stdio.h>

void say_hello(const char *name) {
    printf("Hello, %s!\n", name);
}

int add_numbers(int a, int b) {
    return a + b;
}
```

mylib.h - Header file:

```c
// mylib.h
#ifndef MYLIB_H
#define MYLIB_H

void say_hello(const char *name);
int add_numbers(int a, int b);

#endif
```

Building a Shared Object

Step 1: Compile to position-independent code (PIC)

```
gcc -c -fPIC mylib.c -o mylib.o
```

Step 2: Link into a shared object

```
gcc -shared mylib.o -o libmylib.so
```

Or in one step:

```
gcc -shared -fPIC mylib.c -o libmylib.so
```

flags:

- `-fPIC` : Position Independent Code (required for shared libs)
- `-shared` : Create a shared object instead of executable

Why Position Independent Code?

Problem: Shared libraries can be loaded at any address

- Process A loads `libmylib.so` at `0x7fff12340000`
- Process B loads `libmylib.so` at `0x7fff56780000`

Solution: PIC uses relative addressing

```
; Without PIC (absolute address - won't work)
adrp    x0, 0x400000        ; hardcoded address

; With PIC (relative to current position)
adrp    x0, .L_str@PAGE     ; relative to PC
add     x0, x0, .L_str@PAGEOFF
```

The Global Offset Table (GOT) and Procedure Linkage Table (PLT) handle external references.

Linking Against a Shared Object

main.c - Program that uses our library:

```c
#include "mylib.h"

int main() {
    say_hello("World");
    int result = add_numbers(5, 3);
    printf("5 + 3 = %d\n", result);
    return 0;
}
```

Compile and link:

```
# Compile main.c and link against libmylib.so
gcc main.c -L. -lmylib -o main

# -L.      : Look for libraries in current directory
# -lmylib  : Link against libmylib.so (lib prefix and .so added)
```

Running with Shared Objects

Problem: The dynamic linker can't find our library

```
$ ./main
./main: error while loading shared libraries: libmylib.so:
    cannot open shared object file: No such file or directory
```

Solutions:

```
# Option 1: Set LD_LIBRARY_PATH
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
./main

# Option 2: Install to system path
sudo cp libmylib.so /usr/local/lib/
sudo ldconfig

# Option 3: Use rpath at compile time
gcc main.c -L. -lmylib -Wl,-rpath,. -o main
```

Dynamic Loading with dlopen()

Load shared objects at runtime without linking:

```c
#include <dlfcn.h>
#include <stdio.h>

int main() {
    // Load the library
    void *handle = dlopen("./libmylib.so", RTLD_NOW);
    if (!handle) {
        fprintf(stderr, "dlopen: %s\n", dlerror());
        return 1;
    }

    // Get function pointer
    void (*say_hello)(const char*) = dlsym(handle, "say_hello");

    // Call it
    say_hello("Dynamic World");

    dlclose(handle);
    return 0;
}
```

```
gcc dynamic_main.c -ldl -o dynamic_main
```

Analyzing a Shared Object

```
# File type
$ file libmylib.so
libmylib.so: ELF 64-bit LSB shared object, ARM aarch64...

# ELF header
$ readelf -h libmylib.so
Type:                              DYN (Shared object file)
Machine:                           AArch64

# Exported symbols
$ readelf -s libmylib.so | grep FUNC
    6: 0000000000000720    32 FUNC    GLOBAL DEFAULT   12 say_hello
    7: 0000000000000740    16 FUNC    GLOBAL DEFAULT   12 add_numbers

# Dependencies
$ readelf -d libmylib.so | grep NEEDED
 0x0000000000000001 (NEEDED)    Shared library: [libc.so.6]
```

Shared Object Lifecycle

Stage	What Happens
Compile	`gcc -fPIC -c` creates relocatable object
Link	`gcc -shared` creates `.so` with PLT/GOT
Load	`ld.so` maps library into process address space
Resolve	Dynamic linker resolves symbols (lazy or eager)
Execute	Program calls library functions
Unload	`dlclose()` or process exit unmaps library

The dynamic linker (`ld-linux-aarch64.so.1`) orchestrates loading.

- Counter example: Alpine linux (MUSCL)/ busybox

The Bridge: .so → .ko

Shared objects and kernel modules are **conceptually similar**:

Aspect	Shared Object (.so)	Kernel Module (.ko)
Purpose	Extend userspace programs	Extend the kernel
Format	ELF DYN	ELF REL
Loader	`ld.so` (dynamic linker)	Kernel module loader
Load command	`dlopen()`	`insmod` / `modprobe`
Unload command	`dlclose()`	`rmmod`
Symbol table	`.dynsym`	`.symtab`
Entry point	`_init` / constructors	`module_init()`
Exit point	`_fini` / destructors	`module_exit()`

Key Differences: Userspace vs Kernel

	Userspace (.so)	Kernel (.ko)
Privilege	EL0 (user mode)	EL1 (kernel mode)
Memory	Process virtual memory	Kernel address space
Errors	Segfault, signals	Kernel panic, oops
Libraries	libc, libpthread, etc.	Kernel APIs only
Debugging	gdb, printf	kgdb, printk, dmesg
Isolation	Protected from other processes	Full kernel access

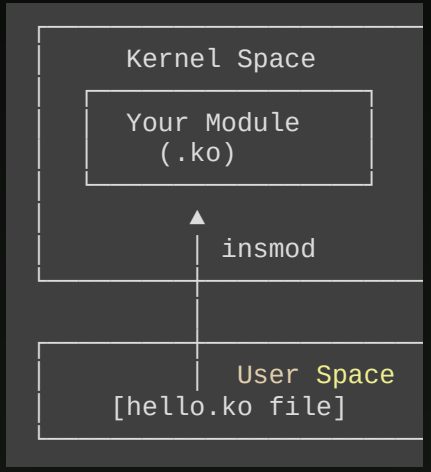
The kernel has no libc - you use kernel equivalents:

- `printf()` → `printk()` / `pr_info()`
- `malloc()` → `kmalloc()` / `vmalloc()`
- `memcpy()` → `memcpy()` (kernel version)

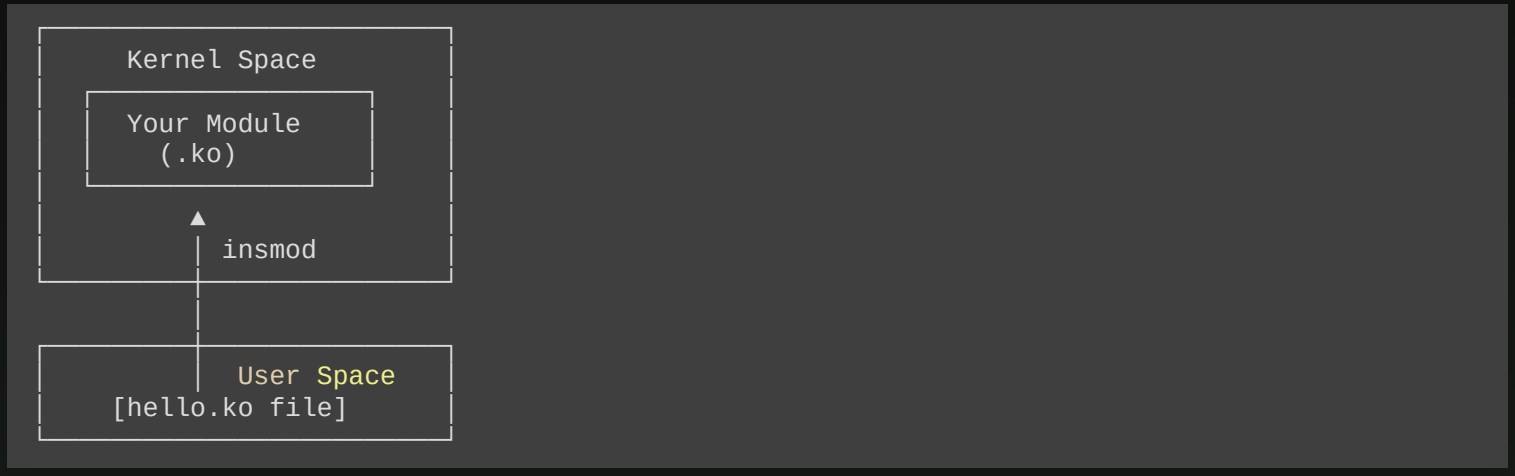
Kernel Module Development

Now that we understand shared objects, let's dive into kernel modules:

- Understanding `.ko` files and kernel space
- Privileged registers and instructions
- Building, loading, and debugging modules
- Binary analysis of kernel modules



MOD



What is a Kernel Module?

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

- Dynamically loadable code that extends the kernel at runtime
- Similar to a shared object (`.so`), but for kernel space
- Used for device drivers, filesystems, network protocols, etc.

Key difference from userspace:

- Runs in ring 0 (kernel mode) with full hardware access
- A bug = kernel panic (not just a segfault)
- No libc - uses kernel APIs only

AArch64 Exception Levels

ARM64 has **4 exception levels** (like x86 rings, but inverted numbering):

Level	Name	Purpose
EL3	Secure Monitor	Firmware, secure boot, TrustZone
EL2	Hypervisor	KVM, Xen, VM management
EL1	Kernel	Linux kernel, device drivers, **your module**
EL0	User	Applications, shell, normal programs

- Userspace (EL0) cannot execute privileged instructions - the CPU traps to EL1

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)

SCTLR_EL1 - System Control Register

The "master switches" for the CPU:

Bit	Name	Function
0	M	MMU enable - 0=flat physical, 1=virtual memory
2	C	Data cache enable
12	I	Instruction cache enable
19	WXN	Write-execute-never - writable pages cannot execute
25	EE	Endianness - 0=little, 1=big

TTBR0/TTBR1 - Translation Table Base

Page table pointers that translate virtual to physical addresses:

- **TTBR0**: User addresses (`0x0000_0000_0000_0000` - `0x0000_FFFF_FFFF_FFFF`)
- **TTBR1**: Kernel addresses (`0xFFFF_0000_0000_0000` - `0xFFFF_FFFF_FFFF_FFFF`)

On context switch:

- Kernel swaps TTBR0 to point to new process's page tables
- TTBR1 stays the same (kernel is mapped into every process)

ESR_EL1 - Exception Syndrome Register

Tells you **why** a trap occurred:

Bits [31:26] (EC): Exception Class

Value	Meaning
0x15	SVC from AArch64 (syscall)
0x20	Instruction abort from lower EL
0x21	Instruction abort from same EL
0x24	Data abort from lower EL
0x25	Data abort from same EL

Bits [24:0] (ISS): Instruction-Specific Syndrome - fault details

More Useful Registers

VBAR_EL1 - Vector Base Address Register

- Points to exception vector table (IRQ, SVC, page faults, undefined instructions)
- Exception vectors at offsets: +0x000, +0x080, +0x100, ... from VBAR

MIDR_EL1 - Main ID Register

- Bits [31:24]: Implementer (0x41=ARM, 0x51=Qualcomm)
- Bits [15:4]: Part number (0xD07=Cortex-A57, 0xD03=Cortex-A53)

MPIDR_EL1 - Multiprocessor Affinity

- Identifies which CPU core we're running on

Privileged Instructions (EL1 Only)

These instructions **trap** if executed at EL0:

```
; System register access
MRS  x0, SCTLR_EL1      ; Move from system register to general register
MSR  SCTLR_EL1, x0      ; Move from general register to system register

; Cache maintenance
DC   CIVAC, x0          ; Clean & invalidate data cache by VA
IC   IALLU              ; Invalidate all instruction caches

; TLB maintenance
TLBI VMALLE1            ; Invalidate all TLB entries at EL1
TLBI VAE1, x0           ; Invalidate TLB entry by VA

; Barriers & hints
DSB  SY                 ; Data synchronization barrier
ISB                     ; Instruction synchronization barrier
WFI                     ; Wait for interrupt (idle/sleep)
```

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
    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;
}
```

This would **SIGILL** in userspace but runs fine in a kernel module!

Inline Assembly Syntax

GCC inline assembly for AArch64:

```
// Basic read from system register
u64 value;
asm volatile("mrs %0, TTBR1_EL1" : "=r"(value));

// Write to system register (be careful!)
asm volatile("msr TTBR1_EL1, %0" : : "r"(value));

// With memory clobber (for barriers)
asm volatile("dsb sy"
```

The kernel provides macros in <asm/sysreg.h>:

```
u64 val = read_sysreg(SCTLR_EL1);
write_sysreg(val, SCTLR_EL1);
```

Kernel != "god mode" (Even at EL1)

EL1 (kernel) still has restrictions:

Action	Requires
Access *_EL2 registers	EL2 (hypervisor)
Access *_EL3 registers	EL3 (secure monitor)
Modify secure memory	EL3 / TrustZone
Bypass stage-2 translation	EL2
Access HCR_EL2	EL2

You can crash the kernel, but you can't escape the VM or access secure world.

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 Explained

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()/prinkt`	Kernel's printf - writes to dmesg

Building the Module

From the lab root directory:

```
# Build just the hello module
make module-hello

# Or build all modules
make modules

# Build and copy to shared folder
make modules-install
```

Output will be in `modules/hello/bin/hello.ko`

Loading the Module (In the Guest VM)

Start the VM and mount the shared folder:

```
# On host
make shared        # Start VM with shared folder
make debug         # In another terminal, connect GDB

# In guest VM
mount-shared       # Mounts host's ./shared to /mnt
```

Load and test:

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

lsmod | grep hello
rmmod hello

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

Module Info Commands

```
modinfo /mnt/modules/hello.ko
```

```
filename:       /mnt/modules/hello.ko
version:        1.0
description:    A simple Hello World AArch64 Driver
author:         Course Instructor
license:        GPL
srcversion:     ABC123...
depends:
name:           hello
vermagic:       6.6.0 SMP mod_unload aarch64
```

The `vermagic` must match the running kernel exactly!

Analyzing with readelf - File Header

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

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

Key fields:

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

Compare to a shared object:

```
readelf -h /lib/aarch64-linux-gnu/libc.so.6
# Type: DYN (Shared object file)
```

Analyzing with readelf - Sections

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

Important 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
`.symtab`	Symbol table
`.rela.text`	Relocations for .text

Viewing the Code with objdump

```
aarch64-linux-gnu-objdump -d modules/hello/bin/hello.ko
```

```
modules/hello/bin/hello.ko:     file format elf64-littleaarch64


Disassembly of section .init.text:

0000000000000000 <init_module>:
   0:    d503233f     paciasp
   4:    a9bf7bfd     stp    x29, x30, [sp, #-16]!
   8:    90000000     adrp    x0, 0 <init_module>
   c:    910003fd     mov    x29, sp
  10:    91000000     add    x0, x0, #0x0
  14:    94000000     bl    0 <_printk>
  18:    52800000     mov    w0, #0x0                        // #0
  1c:    a8c17bfd     ldp    x29, x30, [sp], #16
  20:    d50323bf     autiasp
  24:    d65f03c0     ret

Disassembly of section .exit.text:
...
```

Notice the zeros at offsets - these are **relocations** filled in at load time!

Examining Relocations

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

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

Relocation section '.rela.init.text' at offset 0x4880 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000008  000700000113 R_AARCH64_ADR_PRE 0000000000000000 .rodata.str1.8 + 0
000000000010  000700000115 R_AARCH64_ADD_ABS 0000000000000000 .rodata.str1.8 + 0
000000000014  00360000011b R_AARCH64_CALL26  0000000000000000 _printk + 0
```

The kernel loader:

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

Symbol Table

```
readelf -s modules/hello/bin/hello.ko | grep -E "FUNC|OBJECT"
```

Key symbols:

```
readelf -s modules/hello/bin/hello.ko | grep -E "FUNC|OBJECT"
    30: 0000000000000000    40 FUNC    LOCAL  DEFAULT    2 hello_start
    32: 0000000000000000    36 FUNC    LOCAL  DEFAULT    4 hello_end
    34: 0000000000000000     8 OBJECT  LOCAL  DEFAULT   15 __UNIQUE_ID___ad[...]
    36: 0000000000000000     8 OBJECT  LOCAL  DEFAULT   17 __UNIQUE_ID___ad[...]
    37: 0000000000000000    12 OBJECT  LOCAL  DEFAULT   10 __UNIQUE_ID_vers[...]
    38: 000000000000000c    48 OBJECT  LOCAL  DEFAULT   10 __UNIQUE_ID_desc[...]
    39: 000000000000003c    25 OBJECT  LOCAL  DEFAULT   10 __UNIQUE_ID_author335
    40: 0000000000000055    12 OBJECT  LOCAL  DEFAULT   10 __UNIQUE_ID_lice[...]
    44: 0000000000000061    35 OBJECT  LOCAL  DEFAULT   10 __UNIQUE_ID_srcv[...]
    45: 0000000000000084     9 OBJECT  LOCAL  DEFAULT   10 __UNIQUE_ID_depe[...]
    46: 000000000000008d    11 OBJECT  LOCAL  DEFAULT   10 __UNIQUE_ID_name335
    47: 0000000000000098    46 OBJECT  LOCAL  DEFAULT   10 __UNIQUE_ID_verm[...]
    49: 0000000000000000    24 OBJECT  LOCAL  DEFAULT   13 _note_15
    50: 0000000000000018    24 OBJECT  LOCAL  DEFAULT   13 _note_14
    51: 0000000000000000  1152 OBJECT  GLOBAL DEFAULT   19 __this_module
    52: 0000000000000000    36 FUNC    GLOBAL DEFAULT    4 cleanup_module
    53: 0000000000000000    40 FUNC    GLOBAL DEFAULT    2 init_module
```

The `init_module` and `cleanup_module` are aliases the kernel looks for.

.ko vs .so - Similarities

Feature	.ko (Kernel Module)	.so (Shared Object)
Format	ELF	ELF
Type	REL (relocatable)	DYN (shared)
Relocations	Yes (resolved at insmod)	Yes (resolved at runtime)
Symbol table	Yes	Yes
Dynamic loading	insmod/modprobe	dlopen()
Unloading	rmmod	dlclose()

.ko vs .so - Differences

Aspect	.ko (Kernel Module)	.so (Shared Object)
Address space	Kernel	Userspace
Symbol resolution	Kernel symbol table	Dynamic linker (ld.so)
Entry point	`init_module`	`_init` / constructor
Exit point	`cleanup_module`	`_fini` / destructor
Error handling	Kernel panic	Signal/crash
Memory allocation	kmalloc()	malloc()
Debugging	printk, kgdb	printf, gdb

The .modinfo Section

```
readelf -p .modinfo modules/hello/bin/hello.ko
```

```
String dump of section '.modinfo':
  [     0]  version=1.0
  [     c]  description=A simple Hello World AArch64 Driver
  [    3c]  author=Course Instructor
  [    55]  license=GPL
  [    61]  srcversion=2B47F2B9FDF7BF21F8B830E
  [    84]  depends=
  [    8d]  name=hello
  [    98]  vermagic=6.6.0 SMP preempt mod_unload aarch64
```

This metadata is read by `modinfo` and checked by the kernel at load time.

Hands-On Exercise

1. Build and load the hello module:

```
make module-hello
make shared   # then make debug in another terminal
# In guest:
insmod hello.ko
dmesg | tail
```

2. Examine the binary:

```
aarch64-linux-gnu-readelf -h hello.ko
aarch64-linux-gnu-readelf -S hello.ko
aarch64-linux-gnu-objdump -d hello.ko
```

3. Find the string "Hello, AArch64!" in the binary:

```
aarch64-linux-gnu-objdump -s -j .rodata.str1.1 hello.ko
```

Creating Your Own Module

```
# Create a new module from template
make new-module NAME=mydriver

# Edit the source
vim modules/mydriver/mydriver.c

# Build it
make module-mydriver

# Install to shared folder
make modules-install
```

Debugging Tips

In the guest VM:

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

Summary

AArch64 Privilege Model:

- EL0 (user) → EL1 (kernel) → EL2 (hypervisor) → EL3 (secure monitor)
- Your module runs at EL1 with access to privileged registers and instructions
- MRS/MSR to read/write system registers, cache/TLB maintenance ops, barriers

Kernel Modules:

- ELF relocatable objects (`.ko`) - similar to `.so` but for kernel space
- Use `readelf` and `objdump` to analyze structure
- Entry/exit via `module_init()` / `module_exit()` macros
- All output goes to kernel log (`dmesg`)

Quick Reference

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

# 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
modinfo /path/to/module.ko # Show module info
dmesg | tail               # View kernel log
```