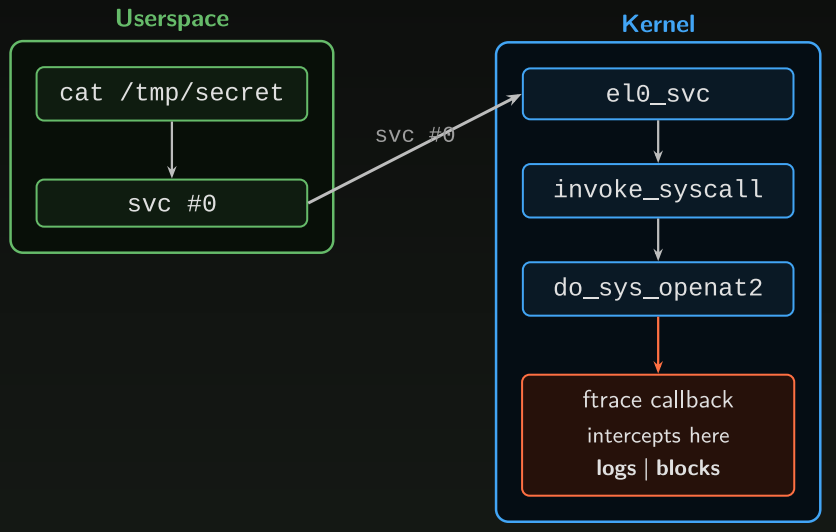
CY-4973/7790

Kernel Security: how2rootkit

Ftrace: Kernel Function Hooking

Todays Agenda:

- The syscall dispatch path on AArch64 (review)
- Dynamic function tracing via ftrace (NOP → BL patching)
- The ftrace API: resolve, filter, register
- AArch64 specifics (`WITH_ARGS` vs `WITH_REGS`)
- Code walkthrough: `trace_openat_ftrace.c` and `bouncer_ftrace.c`
- Detection and kernel defenses



Where We Are:

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.

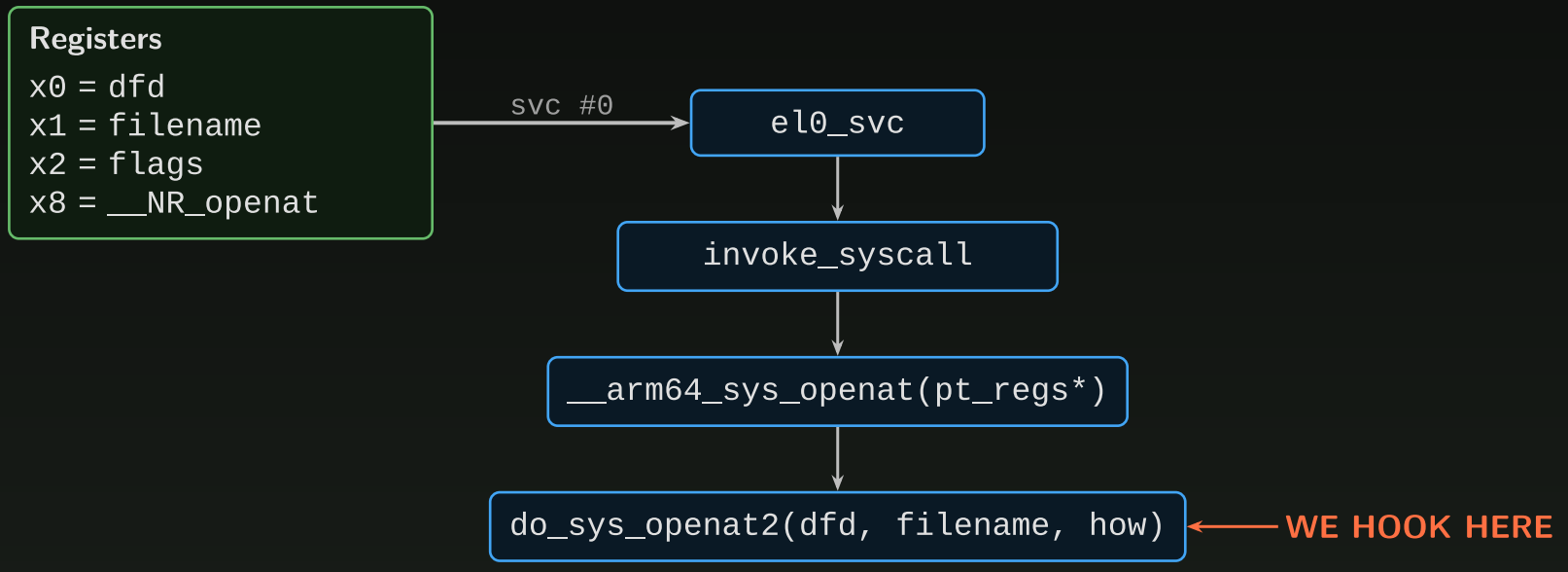
Today: ftrace: the mechanism live patching, (eBPF relies on this!).

`ftrace` **Basics**

- **Built into the kernel**: `CONFIG_DYNAMIC_FTRACE=y` (enabled by default on most distros)
- **(almost) Zero overhead when off**: functions start with NOPs, patched only when a tracer registers a callback
- **API**: `register_ftrace_function()` / `unregister_ftrace_function()`
- **Used by**: live patching (kLP), BPF trampolines, perf, trace-cmd ...and rootkits
- **Compared to kprobes**: BL (function call) vs BRK (exception). The BL path is **much** faster.
- **Cleaner arg access on AArch64**: hook the inner function, skip double `pt_regs`

Ftrace is the most practical,portable and performant intended hooking mechanism for modern kernels

The openat Syscall Path



Sanity Check

- Why don't we hook at `invoke_syscall`? we could instead write one hook handler for all of our rootkit logic.

Sanity Check

- Why don't we hook at `invoke_syscall`? we could instead write one hook handler for all of our rootkit logic.
- invoke_syscall is a hot path. If we did, our hook would be triggered significantly more often than it would need to be. This would incur a significant performance penalty! It would also greatly complicate our hook handler code.

openat syscall path

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

We hook `do_sys_openat2` because it receives args directly and catches both syscalls (openat/openat2). Plus it receives args directly which is nice.

pt_regs

Kprobe path (trace_openat.c, lines 49–55)

Hook `__arm64_sys_openat` — the wrapper:

```
/* Double indirection to reach user args */
user_regs = (struct pt_regs *)regs->regs[0];
filename_ptr = (char __user *)user_regs->regs[1];
```

The wrapper receives one `struct pt_regs *`. The real syscall arguments are buried inside: `regs->regs[0]` points to the **user** `pt_regs`.

Two dereferences to get the filename.

Ftrace path (trace_openat_ftrace.c, line 95)

Hook `do_sys_openat2` — the implementation:

```
/* Direct argument access — no indirection */
filename = (const char __user *)
    ftrace_regs_get_argument(fregs, 1);
```

`do_sys_openat2` receives arguments directly:

- `x0` = dfd
- `x1` = filename
- `x2` = how

Kprobe vs Ftrace Target Comparison

	Kprobe (trace_openat.c)	Ftrace (trace_openat_ftrace.c)
Target	`__arm64_sys_openat`	`do_sys_openat2`
Arg access	Double `pt_regs` deref (lines 54–55)	`ftrace_regs_get_argument(fregs, 1)` (line 95)
Block method	`user_regs->regs[1] = 0`	`fregs->regs[1] = 0`
Mechanism	BRK → exception handler	NOP → `BL ftrace_caller`
Overhead	Higher (exception)	Lower (function call)
Install	`register_kprobe()`	resolve + filter + register
Scope	Any instruction	Only intended locations

Both achieve the same result. Ftrace is cleaner for function-entry hooking; kprobes are more flexible (can probe any instruction).

Other Hookable Syscall Targets

Pattern: find the **inner function** via `/proc/kallsyms`, verify args via source.

Syscall	Inner function	x0	x1	x2
`openat`	`do_sys_openat2`	dfd	filename	how
`read`	`ksys_read`	fd	buf	count
`write`	`ksys_write`	fd	buf	count
`getdents64`	`iterate_dir`	file	ctx	—

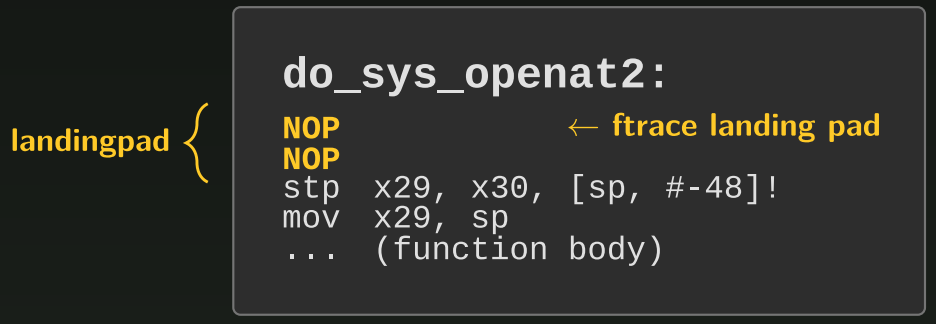
```
# Find hookable targets
cat /proc/kallsyms | grep ' T do_sys_'
cat /proc/kallsyms | grep ' T ksys_'
```

The `__arm64_sys_*` wrappers always have the double `pt_regs` . The `do_*` / `ksys_*` inner functions receive arguments directly.

Patchable Function Entry

When Ftrace is enabled, (and on some versions regardless!) the kernel is compiled with GCC `-fpatchable-function-entry=N`:

- Inserts N NOP instructions at the **beginning of every function**
- These NOPs are the landing pad for ftrace
- Almost zero overhead when disabled! the CPU just slides through NOPs
    - "What if we just made functions easy to hook?"



Every function in the kernel has this preamble. Thousands of NOPs, doing nothing — until a tracer registers.

NOP → branch: Runtime Code Patching

Before `register_ftrace_function()`

```
do_sys_openat2:
NOP     ← no cost
NOP
stp  x29, x30, [sp, #-48]!
mov  x29, sp
...
```

After `register_ftrace_function()`

```
do_sys_openat2:
BL  ftrace_caller     ← patched!
NOP
stp  x29, x30, [sp, #-48]!
mov  x29, sp
...
```

Nop --> branch

NOP is patched to BL `ftrace_caller`. Every call now detours through the ftrace trampoline.

AArch64 detail: `aarch64_insn_patch_text_nosync()` atomically replaces one instruction.

The Ftrace Trampoline

What happens when `BL ftrace_caller` executes:

vs kbprobes

`BRK` → exception handler → `do_debug_exception` → kprobe dispatch.

The ftrace path is a **function call**. The kprobe path is an **exception**. Function calls are faster.

Fun fact, kprobes can sometimes be implemented **with** ftrace if it is enabled.

Dynamic Ftrace: Zero Overhead When Off

Two ftrace configurations:

	`CONFIG_DYNAMIC_FTRACE` (what we use)	`CONFIG_FTRACE` only (rare)
Functions at boot	Start with NOPs	Always call `ftrace_caller`
No tracer active	Zero overhead	Every function pays call cost
Tracer registers	NOP patched to BL	Callback enabled in trampoline
Tracer unregisters	BL patched back to NOP	Callback disabled

Dynamic ftrace is the default. Without it, the kernel would be ~5-10% slower at all times. But the speedup comes at a security cost...

Other Ways to hook functions

- Inline hooks
- hook syscall table (Messy, racy, and generally easy to detect)
- Your own ftrace style hooks

Three Hook Mechanisms Compared

	Kprobes	Ftrace	Syscall Tab
Install	`register_kprobe()`	resolve + filter + register	PTE manipu + pointer w
Dispatch	BRK exception	BL function call	Normal disp (pointer alr replaced)
Overhead	Medium (exception)	Low (function call)	None (direc
Scope	Any instruction	Function entry	Syscall entr
Kernel support	Supported API	Supported API	Unsupporte hack
Restore	`unregister_kprobe()`	unregister + clear filter	Restore poi PTE
Detection	`/sys/.../kprobes/list`	`.../enabled_functions`	Compare `sys_call_` vs kallsyms

What About Function Returns?

- The comparison table says ftrace hooks **function entry**.
- How do we hides files by filtering the `getdents64` buffer **after** the syscall completes?
    - that requires a **return hook**. Kretprobes handle this with entry + return handlers.

If you only had ftrace (no kprobes), how would you intercept getdents64's output?

Hints:

- You can read *and write* registers at function entry (`FL_IPMODIFY`)
- The blocking strategies table mentions "Redirect IP — modify `fregs->pc`"
- What if your replacement function called the original, then post-processed?

Replacement via FL_IPMODIFY

The pattern:

1. Save the original function address
2. In your ftrace callback, redirect `fregs->pc` to your **wrapper**
3. Wrapper calls the original, then filters the dirent buffer
    1. Careful: where do we call it?
4. Return the modified result

pseudocode

```
static asmlinkage long
my_getdents64(struct pt_regs *regs)
{
    long ret = orig_getdents64(regs);
    if (ret > 0)
        filter_dirents(regs, ret); // our code
    return ret;
}

static void notrace
my_callback(unsigned long ip,
            unsigned long parent_ip,
            struct ftrace_ops *op,
            struct ftrace_regs *fregs)
{
    fregs->pc = (unsigned long)my_getdents64;
}
```

FL_IPMODIFY

Downsides:

- Must save/manage the original function pointer
- Wrapper must be `notrace` or excluded from ftrace
- Only **one** `FL_IPMODIFY` hook per function
- Must match the target's calling convention exactly

For a true "run the original, then post-process" pattern, we need **return hooking**.

How Ftrace Hooks Returns: The Trampoline Trick

The same mechanism kretprobes use works for ftrace. At function entry:

1. Save the return address (LR / `x30`)
2. Replace LR with a **return trampoline** address
3. Function runs normally and returns...
4. ...to the trampoline, which runs your **exit handler**
5. Trampoline jumps to the original caller

This is identical to `kretprobe_trampoline` from the kprobes lecture. The only difference: kretprobes trigger via BRK at entry; here, ftrace triggers via BL at entry. Same return trick.

Since 5.5+, the ftrace infrastructure supports building these return trampolines internally. BPF `fexit` uses this mechanism but you don't need BPF to use it.

fprobe: Ftrace + Return Hooks (5.18+)

```c
#include <linux/fprobe.h>

static int my_entry(struct fprobe *fp,
                    unsigned long ip,
                    unsigned long ret_ip,
                    struct ftrace_regs *fregs,
                    void *data)
{
    /* save args for exit handler */
    return 0;
}

static void my_exit(struct fprobe *fp,
                    unsigned long ip,
                    unsigned long ret_ip,
                    struct ftrace_regs *fregs,
                    void *data)
{
    /* post-process: filter dirents here */
}

static struct fprobe fp = {
    .entry_handler = my_entry,
    .exit_handler  = my_exit,
};
```

rethooks compared

	kretprobe	fprobe
Entry trigger	BRK (exception)	BL (ftrace, low overhead)
Return trigger	LR trampoline	LR trampoline (same)
Per-instance data	`ri->data`	`data` param
Symbol resolution	Direct	Direct
Available since	2.6 (not sure for arm)	5.18

`fprobe` shows where the kernel is heading: **ftrace as the intended tracing infra**,

struct ftrace_ops

The configuration structure for an ftrace hook.

Field	Purpose
`.func`	Your callback function
`.flags`	Behavior control flags
`.private`	Opaque user data (optional)

Compared to kprobe

Kprobes	Ftrace
`struct kprobe`	`struct ftrace_ops`
`.pre_handler`	`.func`
`.symbol_name`	(set via `ftrace_set_filter_ip`)

ftrace_ops cont

`trace_openat_ftrace.c`, lines 109–117:

```
static struct ftrace_ops trace_ops = {
    .func  = trace_openat_ftrace_callback,
    /*
     * Do NOT set FTRACE_OPS_FL_SAVE_REGS
     * on arm64 — it requires
     * HAVE_DYNAMIC_FTRACE_WITH_REGS which
     * arm64 doesn't have.
     * arm64 uses DYNAMIC_FTRACE_WITH_ARGS:
     * ftrace_regs always contains argument
     * registers, no flag needed.
     */
    .flags = FTRACE_OPS_FL_RECURSION,
};
```

Read-only hook: only `FL_RECURSION` needed. No register modification.

ftrace_ops Flags

Flag	When to use	Notes
`FTRACE_OPS_FL_RECURSION`	**Always**	Prevents callback re-entry if it calls an ftraced function
`FTRACE_OPS_FL_IPMODIFY`	When modifying `fregs->regs[N]`	Tells ftrace to save/restore regs carefully. Only one IPMODIFY hook per function.
`FTRACE_OPS_FL_SAVE_REGS`	**Never on arm64**	Requires `HAVE_DYNAMIC_FTRACE_WITH_REGS` (x86 only)

Details

arm64 uses `DYNAMIC_FTRACE_WITH_ARGS`, not `HAVE_DYNAMIC_FTRACE_WITH_REGS`. The argument registers (x0–x7) are **always saved** .

Module	Flags	Why
`trace_openat_ftrace.c`	`FL_RECURSION`	Read-only (logging)
`bouncer_ftrace.c`	`FL_IPMODIFY | FL_RECURSION`	Read-write (blocking)

Install

Step 1: Resolve the target address

```
kprobe_lookup("do_sys_openat2")
```

Step 2: Set the ftrace filter

```
ftrace_set_filter_ip(&ops, addr, 0, 0)
```

Without this, your callback fires on **every ftraced function in the kernel** — thousands of them.

Step 3: Register the function

```
register_ftrace_function(&ops)
```

This patches the NOP to BL. Your callback is now live.

If step 3 fails, you must **undo step 2** (clear the filter).

Example code

`trace_openat_ftrace.c`, lines 119–149:

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

    /* Step 1: Resolve target address */
    target_func_addr =
        kprobe_lookup("do_sys_openat2");
    if (!target_func_addr) {
        pr_err("failed to find do_sys_openat2\n");
        return -ENOENT;
    }

    /* Step 2: Set ftrace filter */
    ret = ftrace_set_filter_ip(
        &trace_ops, target_func_addr, 0, 0);
    if (ret) {
        pr_err("failed to set filter: %d\n", ret);
        return ret;
    }

    /* Step 3: Register ftrace function */
    ret = register_ftrace_function(&trace_ops);
    if (ret) {
        pr_err("failed to register: %d\n", ret);
        /* Undo step 2 */
        ftrace_set_filter_ip(
```

Uninstall

Step 1: Unregister the function

```
unregister_ftrace_function(&ops)
```

Stops callbacks. Patches `BL` back to `NOP`.

Step 2: Clear the filter

```
ftrace_set_filter_ip(&ops, addr, 1, 0)
```

The `1` in the third argument means "remove" (vs `0` for "add").

Caution

Order matters: clearing the filter first could cause the callback to fire on unintended functions during the window before unregister.

Compare to kprobes: `unregister_kprobe()` does both steps in one call.

Example code

`trace_openat_ftrace.c`, lines 151–156:

```
static void __exit trace_openat_ftrace_exit(void)
{
    unregister_ftrace_function(&trace_ops);
    ftrace_set_filter_ip(
        &trace_ops, target_func_addr, 1, 0);
    pr_info("trace_openat_ftrace: hook removed\n");
}
```

Clean, deterministic teardown. After `unregister_ftrace_function()` returns, the callback is guaranteed not to be running on any CPU.

DYNAMIC_FTRACE_WITH_ARGS vs WITH_REGS

x86_64: HAVE_DYNAMIC_FTRACE_WITH_REGS

```
/* x86 path — works */
struct pt_regs *regs = ftrace_get_regs(fregs);
filename = (char *)regs->si;  /* 2nd arg */
```

- `ftrace_get_regs(fregs)` returns valid `pt_regs *`
- `FL_SAVE_REGS` works
- Access args via `regs->di`, `regs->si`, `regs->dx`

arm64: DYNAMIC_FTRACE_WITH_ARGS

```
/* arm64 path — use this */
filename = (char *)
    ftrace_regs_get_argument(fregs, 1);
```

- `ftrace_get_regs(fregs)` returns **NULL**
- `FL_SAVE_REGS` causes **-EINVAL**
- Only x0–x7 (argument regs) are saved
- Use `ftrace_regs_get_argument(fregs, N)`

The Callback Signature

```
static void notrace callback(unsigned long ip,
                             unsigned long parent_ip,
                             struct ftrace_ops *op,
                             struct ftrace_regs *fregs)
```

Callback

Parameter	Meaning	Example
`ip`	Address of the hooked function	Address of `do_sys_openat2`
`parent_ip`	Return address (who called the function)	Address inside `invoke_syscall`
`op`	Your `ftrace_ops` struct	`&trace_ops`
`fregs`	Register state at function entry	Contains x0–x7 on arm64

Reading args (arm64): `ftrace_regs_get_argument(fregs, N)` — returns arg N as `unsigned long`

Writing args (arm64): `fregs->regs[N] = value` — modifies register N (requires `FL_IPMODIFY`)

See more

- https://www.kernel.org/doc/html/v5.3/trace/ftrace-uses.html

notrace and Recursion Prevention

Two layers of protection:

Layer	Mechanism	What it prevents
`notrace` (compile-time)	Tells GCC: don't insert patchable-entry NOPs in this function	Ftrace **cannot instrument** the callback itself — prevents direct self-tracing
`FL_RECURSION` (runtime)	Ftrace framework checks a per-CPU flag before calling your callback	Prevents re-entry if your callback calls a function that is also ftraced

Recursion prevention pt2

Both are recommended. `notrace` prevents the obvious case (callback tracing itself). `FL_RECURSION` prevents the subtle case (callback → `printk` → ftraced function → callback again).

```
static void notrace trace_openat_ftrace_callback(...)
/*            ^^^^^^^
 * This function will NOT have NOP preamble.
 * Ftrace cannot hook it even if it wanted to.
 */
```

The Callback: Reading Arguments

`trace_openat_ftrace.c`, lines 81–107:

```
static void notrace trace_openat_ftrace_callback(
    unsigned long ip, unsigned long parent_ip,
    struct ftrace_ops *op,
    struct ftrace_regs *fregs)
{
    const char __user *filename;
    char kbuf[MAX_PATH_LEN];  /* stack buffer */
    int dfd;
    unsigned long how;
    long len;

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

    /* Arg 0 = dfd, Arg 1 = filename, Arg 2 = how */
    dfd = (int)ftrace_regs_get_argument(fregs, 0);
    filename = (const char __user *)
        ftrace_regs_get_argument(fregs, 1);
    how = ftrace_regs_get_argument(fregs, 2);
    if (!filename)
        return;

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

tracer

1. `ftrace_regs_get_argument(fregs, 1)` — reads x1, the filename user pointer
2. `kbuf[MAX_PATH_LEN]` — **stack buffer** avoids `kmalloc` in atomic context
3. `strncpy_from_user(kbuf, filename, ...)` — copies string from userspace to kernel
4. `kbuf[len] = '\0'` — null-terminate (strncpy_from_user doesn't always)
5. `current->pid` / `current->comm` — the calling process's PID and name

Note the lack of `kmalloc` here: stack allocation is fine for small buffers (256 bytes). The HW4 version uses `kmalloc(GFP_ATOMIC)` for larger buffers.

strncpy_from_user in Ftrace Context

`strncpy_from_user(dst, src, maxlen)` copies a null-terminated string from userspace.

	Details
Returns	String length on success, negative errno on fault
Null termination	Terminates at null byte or `maxlen`, whichever comes first
Safety	Always null-terminate manually: `kbuf[len] = '\0'`
In ftrace context	Safe for small copies — doesn't sleep for pinned pages
Failure	Returns `-EFAULT` if page not present (no page fault servicing)

Init: Resolve → Filter → Register

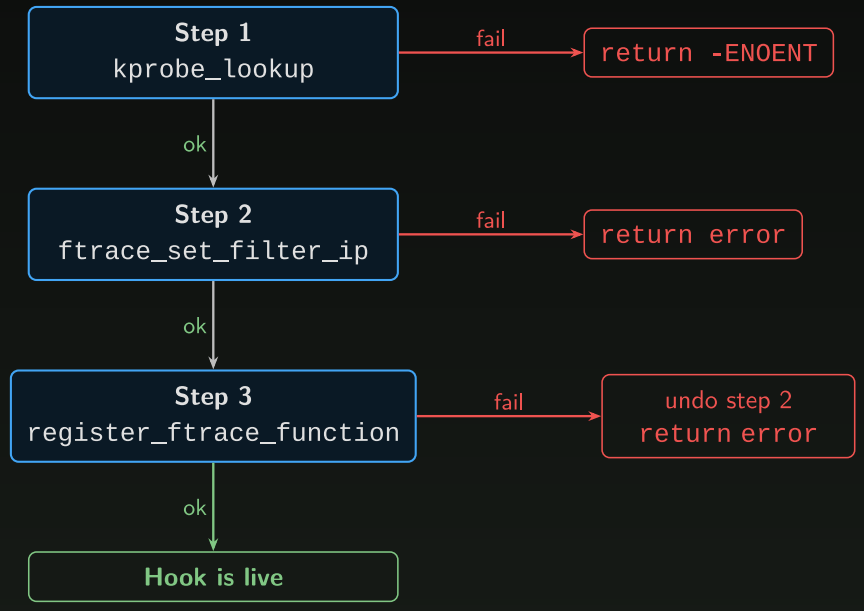
`trace_openat_ftrace.c`, lines 119–149:

```c
/* Step 1: Resolve target */
target_func_addr =
    kprobe_lookup("do_sys_openat2");
if (!target_func_addr) {
    pr_err("failed to find do_sys_openat2\n");
    pr_err("ensure CONFIG_KALLSYMS=y\n");
    return -ENOENT;
}

/* Step 2: Set filter */
ret = ftrace_set_filter_ip(
    &trace_ops, target_func_addr, 0, 0);
if (ret) {
    pr_err("failed to set filter: %d\n", ret);
    return ret;
}

/* Step 3: Register */
ret = register_ftrace_function(&trace_ops);
if (ret) {
    pr_err("failed to register: %d\n", ret);
    /* Undo step 2 on failure */
    ftrace_set_filter_ip(
        &trace_ops, target_func_addr, 1, 0);
    return ret;
}
```

Error handling pattern:



If step 3 fails, we must clear the filter (undo step 2). Otherwise the filter entry leaks and the next `register_ftrace_function` could behave unexpectedly.

Demo: trace_openat_ftrace

```
# Load the module
insmod trace_openat_ftrace.ko

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

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

```
[  42.123] trace_openat_ftrace: found do_sys_openat2 at 0xffff800080abcdef
[  42.124] trace_openat_ftrace: hook registered
[  45.200] trace_openat_ftrace: PID 156 (cat) openat(dfd=-100, "/etc/hostname", how=0x...)
[  46.300] trace_openat_ftrace: PID 157 (ls) openat(dfd=-100, "/tmp", how=0x...)
```

```
# Verify the hook is registered in ftrace
cat /sys/kernel/debug/tracing/enabled_functions

# Unload
rmmod trace_openat_ftrace
```

Every `open()` in the system passes through our callback. This is what EDR agents do — at scale.

From Logging to Blocking: FL_IPMODIFY

`trace_openat_ftrace.c` only **reads** registers (logging). To **block** access, we modify register state before `do_sys_openat2` runs.

`FL_IPMODIFY` tells ftrace to save/restore registers more carefully. Without it, modifications may be silently dropped.

```
/* Read-only (trace_openat_ftrace.c) */
.flags = FTRACE_OPS_FL_RECURSION,

/* Read-write (bouncer_ftrace.c) */
.flags = FTRACE_OPS_FL_IPMODIFY |
         FTRACE_OPS_FL_RECURSION,
```

Constraint: only **one** `FL_IPMODIFY` hook per function at a time. If two modules both try to IPMODIFY `do_sys_openat2`, the second `register_ftrace_function()` returns `-EEXIST`.

HW bouncer_ftrace: Blocking Callback

`bouncer_ftrace.c`, lines 181–213:

```
static void notrace bouncer_callback(
    unsigned long ip, unsigned long parent_ip,
    struct ftrace_ops *op,
    struct ftrace_regs *fregs)
{
    const char __user *filename_ptr;
    char kbuf[MAX_PATH_LEN];
    long len;

...

}
```

Hints for Homework

1. Same argument extraction
2. `strcmp(kbuf, PROTECTED_PATH)`: check if blocked
3. If blocked, we need to actually block. How?
    1. Hint: When `do_sys_openat2` runs with a NULL filename, it returns `-EFAULT` to userspace.

The user sees:

```
$ cat /tmp/secret
cat: /tmp/secret: Bad address
```

Why this works: `fregs->regs[1]` is x1 — the filename argument. By zeroing it before the function body runs, the kernel tries to read path from NULL. The fault handler returns `-EFAULT`.

Blocking Strategies

Strategy	Code	Result	Limitation
Null filename	`fregs->regs[1] = 0`	`-EFAULT`	User sees "Bad address"
Corrupt dfd	`fregs->regs[0] = -1`	`-EBADF`	Doesn't work for absolute paths (dfd ignored)
Redirect IP	Modify `fregs->pc`	Complex	Fragile, needs a return stub

`bouncer_ftrace.c` uses **strategy 1** (null filename):

```
/* bouncer_ftrace.c line 212 */
fregs->regs[1] = 0;
```

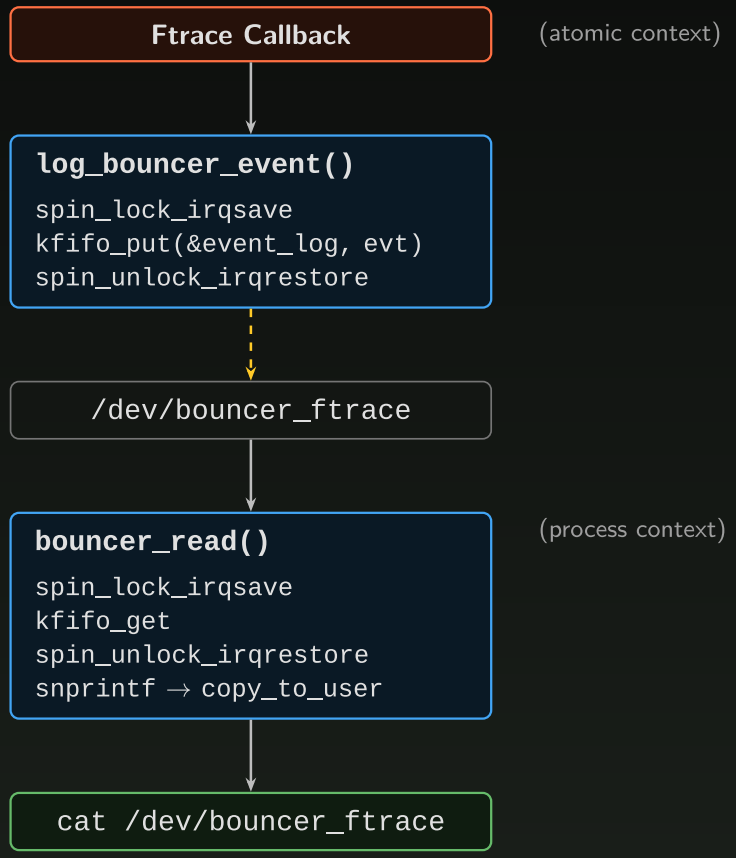
Context Restrictions in Ftrace Callbacks

Ftrace callbacks run with **preemption disabled**. You cannot sleep.

Forbidden	Use instead
`kmalloc(GFP_KERNEL)`	`kmalloc(GFP_ATOMIC)`
`mutex_lock()`	`spin_lock_irqsave()`
`schedule()`	(don't — return quickly)
`vmalloc()`	Stack buffers or pre-allocated memory
`copy_to_user()` (can sleep)	Log to kfifo, read from userspace later

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

bouncer_ftrace:



This is the same pattern students use in HW4 via `hooklab_log_event()`.

Hooking Multiple Functions

Pattern: array of `struct ftrace_ops`, one per target function.

```
struct hook_entry {
    const char *name;
    ftrace_func_t callback;
    struct ftrace_ops ops;
    unsigned long addr;
};

static struct hook_entry hooks[] = {
    { "do_sys_openat2",  openat_cb,  { .flags = FL }, 0 },
    { "ksys_read",       read_cb,    { .flags = FL }, 0 },
    { "ksys_write",      write_cb,   { .flags = FL }, 0 },
};
```

Init: for each entry → `kprobe_lookup(name)` → `ftrace_set_filter_ip()` → `register_ftrace_function()`

Exit: reverse order → `unregister_ftrace_function()` → `ftrace_set_filter_ip(remove)`

Ftrace in the wild

User	How they use ftrace	Flags
Live patching (kLP)	`FL_IPMODIFY` to redirect function entry to patched version	`FL_IPMODIFY`
BPF trampolines	Attach BPF programs to kernel function entry/exit	Internal flags
perf	Function entry/exit tracing for profiling	`FL_RECURSION`
trace-cmd	User-friendly frontend to `/sys/kernel/debug/tracing`	Via tracefs
Rootkits	Hook syscall inner functions, modify args/returns	`FL_IPMODIFY`
EDR agents	Monitor file/network/process activity	`FL_RECURSION`

`register_ftrace_function()` doesn't know or care whether you're patching a security vulnerability or hiding a backdoor.

Detecting Ftrace Hooks

```
# List all ftrace-hooked functions
cat /sys/kernel/debug/tracing/enabled_functions

# Check which modules are loaded
lsmod

# Compare: is there a module hooking
# a syscall-related function?
cat /sys/kernel/debug/tracing/enabled_functions \
    | grep do_sys_
```

Why detection is possible:

- `enabled_functions` is maintained by the ftrace core , not by the hook module. A module cannot easily hide from this file.

Technique	Detection method	Difficulty
Kprobes	`/sys/.../kprobes/list`	Easy
Ftrace	`.../enabled_functions`	Easy
Syscall table	Compare table vs kallsyms	Easy

Kernel Mitigations vs Hook Techniques

Defense	Kprobes	Ftrace	Syscall Table
`CONFIG_STRICT_KERNEL_RWX`	No effect	No effect	**Blocks** (table is read-only)
PAC (Pointer Auth)	No effect	No effect	**Blocks** (signed pointers)
BTI (Branch Target ID)	No effect	No effect	**Blocks** (indirect branch checks)
`CONFIG_STATIC_CALL`	No effect	No effect	**Blocks** (compile-time dispatch)
Module signing	**Blocks** (can't load module)	**Blocks**	**Blocks**

Defense	Kprobes	Ftrace	Syscall Table
`CONFIG_KPROBES=n`	**Blocks**	No effect	No effect
`CONFIG_FTRACE=n`	No effect	**Blocks**	No effect

Summary

Topic	Memorize
Syscall path	Hook the inner function (`do_sys_openat2`), avoid double `pt_regs`
How ftrace works	NOP → BL patching, zero overhead when off
arm64 specifics	`WITH_ARGS` not `WITH_REGS`; use `ftrace_regs_get_argument()`
The API	`kprobe_lookup` → `ftrace_set_filter_ip` → `register_ftrace_function`
Read-only vs read-write	`FL_RECURSION` for logging; add `FL_IPMODIFY` for blocking
Atomic context	No sleeping — `GFP_ATOMIC`, stack buffers, spinlocks
Blocking	`fregs->regs[1] = 0` → `-EFAULT`
Detection	`cat /sys/kernel/debug/tracing/enabled_functions`
vs kprobes	BL (fast) vs BRK (exception); function entry vs any instruction
vs table patching	Supported API vs kernel hack; survives PAC/BTI

Hands-On Exercises

- **Trace file opens**: Load `trace_openat_ftrace.ko`, open some files, check dmesg. Then check `/sys/kernel/debug/tracing/enabled_functions` — find your hook in the list.
- **Hook a second syscall**: Modify `trace_openat_ftrace.c` to also hook `ksys_read`. Log the fd (arg 0) and byte count (arg 2) for every read call. You'll need a second `struct ftrace_ops` and a second callback.
- **PID filtering**: Add `module_param(target_pid, int, 0644)`. Only log/block when `current->pid == target_pid` (or log all if `target_pid == 0`). Test with `echo $ > /sys/module/.../parameters/target_pid`.
- **(Challenge)** Hook `iterate_dir` (used by `getdents64`). Filter out directory entries for a specific filename — implement basic file hiding. This requires understanding `struct dir_context` and the `filldir` callback.

API Quick Reference

Function / Macro	Header	Purpose
`ftrace_set_filter_ip()`	`<linux/ftrace.h>`	Restrict hook to one function
`register_ftrace_function()`	`<linux/ftrace.h>`	Activate the hook (NOP → BL)
`unregister_ftrace_function()`	`<linux/ftrace.h>`	Deactivate (BL → NOP)
`ftrace_regs_get_argument()`	`<linux/ftrace.h>`	Read arg N on arm64
`struct ftrace_ops`	`<linux/ftrace.h>`	Hook configuration
`FL_RECURSION`, `FL_IPMODIFY`	`<linux/ftrace.h>`	Behavior flags
`register_kprobe()` / `unregister_kprobe()`	`<linux/kprobes.h>`	Symbol resolution trick
`strncpy_from_user()`	`<linux/uaccess.h>`	Copy string from userspace
`current->pid`, `current->comm`	`<linux/sched.h>`	Current process info
`kmalloc(GFP_ATOMIC)` / `kfree()`	`<linux/slab.h>`	Atomic allocation
`spin_lock_irqsave()` / `spin_unlock_irqrestore()`	`<linux/spinlock.h>`	Interrupt-safe locking
`kfifo_put()` / `kfifo_get()`	`<linux/kfifo.h>`	Lock-free ring buffer
`ktime_get_ns()`	`<linux/ktime.h>`	Monotonic nanosecond timestamp
`strcmp()` / `strscpy()`	`<linux/string.h>`	String operations
`notrace`	(compiler)	Exclude function from ftrace