Kprobes internals

Thomas Bitzberger
bitz@lse.epita.fr

LSE
Security System
Laboratory of Epita

What are kprobes

“*Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address, specifying a handler routine to be invoked when the breakpoint is hit.*”

From Documentation/kprobes.txt

They were introduced in 2.6.9 (Oct. 2004).

LSE Security System
Laboratory of Epita

What are kprobes

Activated by default on many distributions (ArchLinux, Debian, ...).

Can be (de)activated in sysfs:

`echo 1 > /sys/kernel/debug/kprobes/enabled`

Requires ‘CONFIG_KPROBES’ during kernel build.

Why this

- It’s interesting (at least for me…)
- Projects from the lab using kprobes
- Kprobes implements stuff I needed for other purposes
- Nicely engineered mechanisms to present

LSE
Security System
Laboratory of Epita

Different probes

1) Kprobes
2) Jprobes → Function entry
3) Kretprobes → Function entry (optional) + Function return

Jprobes and Kretprobes are implemented using kprobes.

There are instructions/functions that cannot be probed.

LSE Security System
Laboratory of Epita

Kprobes

How it works:

1) Save probed instruction
2) Replace instruction by a breakpoint
3) When BP is hit, kprobe pre_handler is executed.
4) The original instruction is single-stepped
5) Kprobe post_handler is executed if any
6) Function execution resume

Kprobes

```
typedef int  (*kprobe_pre_handler_t)(struct kprobe *, struct pt_regs *);
struct kprobe {
        kprobe_opcode_t *addr;
        const char *symbol_name;
        kprobe_pre_handler_t pre_handler;
        kprobe_post_handler_t post_handler;
        kprobe_fault_handler_t fault_handler;
        kprobe_break_handler_t break_handler;
        struct arch_specific_insn insn;

        /* …. */
};
```

LSE Security System
Laboratory of Epita

Kprobes registration

You must give at least an address and an offset, or a symbol in kallsyms.

When you call ‘register_kprobe’, it basically does:

1) check_addr_safe() // check if addr can be probed
2) prepare_kprobe() // copy probed instruction
3) arm_kprobe() // insert the breakpoint

LSE Security System
Laboratory of Epita

prepare_kprobe

- Need to save the original instruction !
- Kernel uses executable page(s) to store probed instructions.
- The max instruction size is always copied.
- Adjusts rip-relative instructions if needed.

If you’re interested, the kernel uses custom cache allocation to get executable slots for probed instructions.

Look in ‘kernel/kprobes.c’, ‘include/linux/kprobes.h’, or simply `grep -Hnr ‘struct kprobe_insn_cache’`

LSE Security System
Laboratory of Epita

Fixmaps

From the header (arch/x86/include/asm/fixmap.h):

"*The point is to have a constant address at compile-time, but to set the physical address only in the boot process*."

- Represented as an enum
- Fixed size 4k pages
- Not flushed from TLB during task switch
- set_fixmap(idx, phys_addr)
- set_fixmap_nocache(...)

LSE Security System
Laboratory of Epita

Fixmaps

```
static __always_inline unsigned long fix_to_virt(const unsigned int idx)
{
	BUILD_BUG_ON(idx >= __end_of_fixed_addresses);
	return __fix_to_virt(idx);
}
```

Returns the virtual address for a given fixmap.

Completely done at compilation time thanks to optimization.

LSE Security System
Laboratory of Epita

arm_kprobe

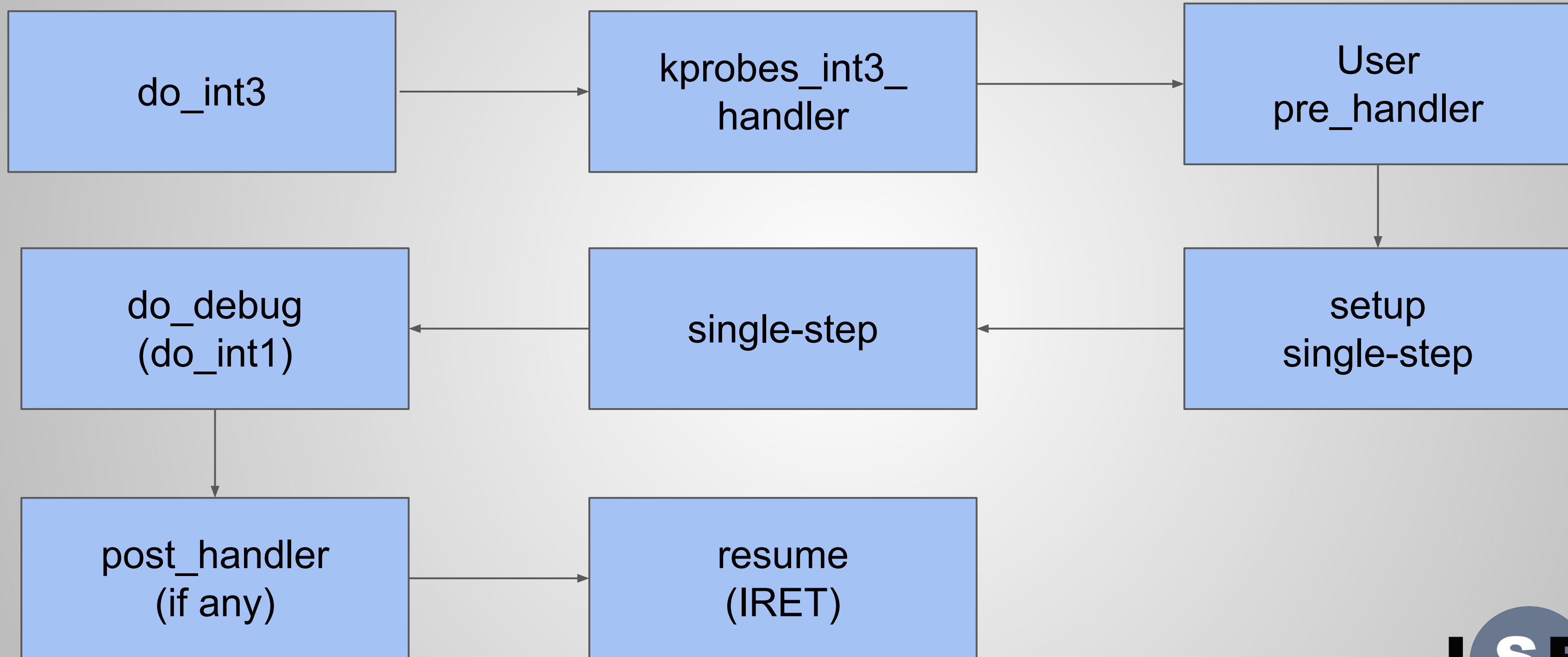
Last thing to do is to insert the breakpoint (int3 → 0xcc on x86).

It is done by text_poke() (arch/x86/kernel/alternative.c)

1) Disable local interrupts
2) Get a RW shadow mapping using TEXT_POKE{0,1} fixmaps.
3) Insert breakpoint atomically (writing a char there)
4) Clear the fixmap and flush TLB
5) Invalidate icache and prefetched instructions (IRET-to-Self)
6) Invalidate data cache
7) Re-enable local interrupts
8) kprobe is armed !

LSE
Security System
Laboratory of Epita

What happens now



LSE Security System
Laboratory of Epita

Single-stepping on x86

To single step the instruction:

1) Clear Branch Tracing in DEBUGCTL MSR
2) Enable Trap Flag in RFLAGS
3) Let’s go !

Jprobes

- Kprobe on function entry point
- The given handler has same signature as the probed function
- Handler must always end by ‘jprobe_return()’
- Uses kind of a setjmp/longjmp trick
- Jprobes uses it’s own pre_handler and break_handler.

Jprobes

```
struct jprobe {
        struct kprobe kp;
        void *entry;      /* probe handling code to jump to (handler) */
};
```

Init example:

```
static struct jprobe my_jprobe = {
        .entry                  = j_do_fork_handler,       // our handler
        .kp = {
                .symbol_name = "_do_fork",
        },
};
```

LSE Security System Laboratory of Epita

How it works

When the jprobe is hit, it first prepares to execute the user handler:

1) Breakpoint is hit
2) kprobe_int3_handler calls ‘setjmp_pre_handler()’
3) Registers and part of the stack are copied
4) IP is set to the given handler
5) setjmp_pre_handler() returns 1 → No single-stepping now
6) IRET on handler

LSE Security System
Laboratory of Epita

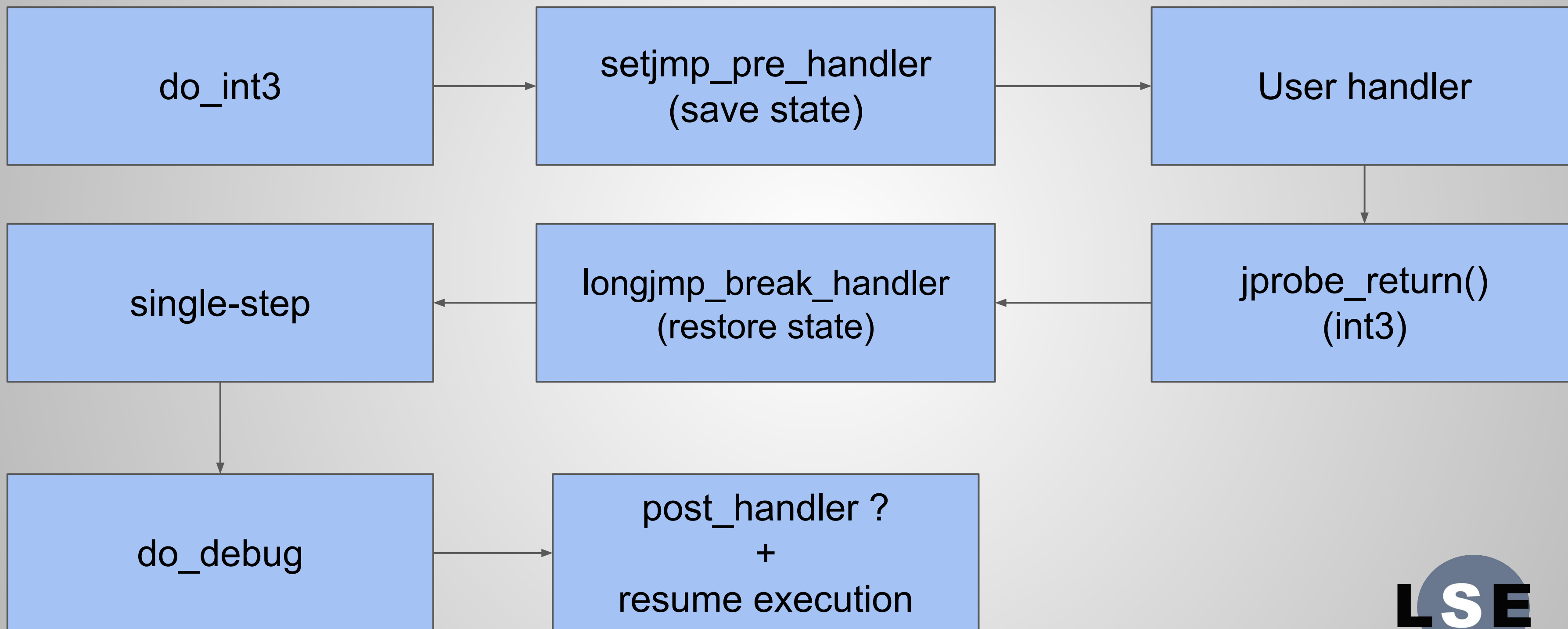
How it works

How the function is resumed:

1) The handler ends by jprobe_return() // restore stack pointer + int3
2) There’s no kprobe at this address !
3) Kprobe manager looks in a Per-CPU saved state
4) Calls the ‘longjmp_break_handler()’ // restore stack + regs
5) Single-step probed instruction
6) do_debug → optional post_kprobe_handler
7) Resume → IRET

LSE Security System
Laboratory of Epita

Execution of a JProbe



LSE Security System Laboratory of Epita

Kretprobes

- Kprobe on function entry
- User can provide two handlers
- One is called at entry, the other just before returing
- You can keep state between entry and exit handler
- Works with a trampoline system

LSE Security System
Laboratory of Epita

Kretprobes

```
typedef int (*kretprobe_handler_t) (struct kretprobe_instance *, struct pt_regs *);

struct kretprobe {
        struct kprobe kp;
        kretprobe_handler_t handler;
        kretprobe_handler_t entry_handler;
        int maxactive;
        size_t data_size;

        /* … */

};
```

LSE Security System
Laboratory of Epita

Kretprobe instance

```
struct kretprobe_instance {
        struct hlist_node hlist;          // instance hash table
        struct kretprobe *rp;             // kretprobe
        kprobe_opcode_t *ret_addr;        // saved return address
        struct task_struct *task;         // probed task
        char data[0];                     // pointer to user data
};
```

LSE Security System
Laboratory of Epita

How it works

1) Breakpoint is hit
2) Kretprobe pre_handler is called
3) It saves the function return address
4) It modifies the return address on the stack
5) The function now returns to kretprobe trampoline

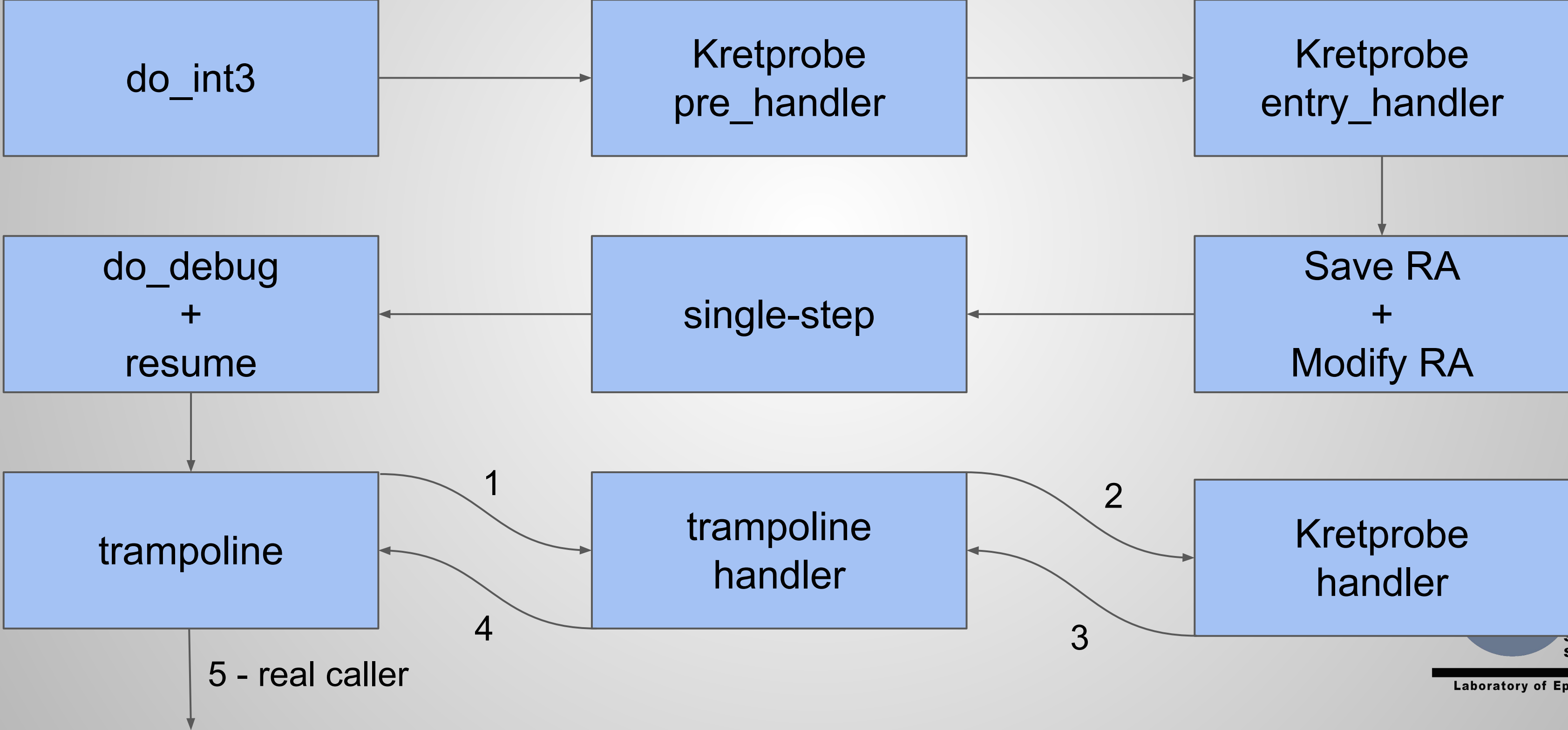
LSE
Security System
Laboratory of Epita

Kretprobe trampoline

```
asm (
        ".global kretprobe_trampoline\n"
        ".type kretprobe_trampoline, @function\n"
        "kretprobe_trampoline:\n"
        /* We don't bother saving the ss register */
        "       pushq %rsp\n"
        "       pushfq\n"
        SAVE_REGS_STRING
        "       movq %rsp, %rdi\n"
        "       call trampoline_handler\n"
        /* Replace saved sp with true return address. */
        "       movq %rax, 152(%rsp)\n"
        RESTORE_REGS_STRING
        "       popfq\n"
        "       ret\n"
        ".size kretprobe_trampoline, .-kretprobe_trampoline\n"
);
```

LSE Security System
Laboratory of Epita

Kretprobes in action



Security System
Laboratory of Epita

Optimization time

The presented implementation is perfectly working.

However, for every probe, you do at least an int3 and an int1.

In some cases, kprobes can be optimized to avoid this.

The breakpoint is then replaced by a relative jump.

It requires ‘CONFIG_OPTPROBES’ during kernel build.

LSE Security System Laboratory of Epita

Optimization time

Optimization is done after kprobe registration (BP insertion).

Primary conditions to optimize:

- Probed region lies in one function
- The entire function is scanned to verify that there’s no jump to the probed region
- Verify that each instruction in the probed region can be executed out-of-line

LSE
Security System
Laboratory of Epita

Detour buffer

Kprobe manager prepares a trampoline containing:

- Code to push CPU’s registers (emulates BP trap)
- Calls a trampoline handler which calls the user handler
- Code to restore CPU’s registers
- The instructions from optimized region
- A jump back to the original execution path

LSE Security System
Laboratory of Epita

Detour buffer

It is an assembly generic template that each optprobe copies.

This template will be patched with the right instructions.

Each optprobe has finally its own trampoline.

That’s because it uses rip-relative instructions (call and jmp).

LSE Security System
Laboratory of Epita

Pre-optimization

After preparing the trampoline, kprobe manager verifies:

- It’s not a jprobe // setjmp/longjmp will not work
- It has no post_handler // no more single-stepping
- Optimized instruction are not probed

If it’s ok, the probe is placed in a list.

Kprobe-optimizer workqueue is woken up.

```
asm (   "optprobe_template_entry:\n"
        /* We don't bother saving the ss register */
        "       pushq %rsp\n"
        "       pushfq\n"
        SAVE_REGS_STRING
        "       movq %rsp, %rsi\n"
        "optprobe_template_val:\n"
        ASM_NOP5// mov $optprobe, %rdi
        ASM_NOP5
        "optprobe_template_call:\n"
        ASM_NOP5// call optimized_callback(optprobe, regs)
        /* Move flags to rsp */
        "       movq 144(%rsp), %rdx\n"
        "       movq %rdx, 152(%rsp)\n"
        RESTORE_REGS_STRING
        /* Skip flags entry */
        "       addq $8, %rsp\n"
        "       popfq\n"
        "optprobe_template_end:\n");   // patched insn + jmp
```

LSE Security System

Laboratory of Epita

Optimization

Once the trampoline is ready, the BP is replaced by a reljmp.

It’s a five bytes length instruction.

The jump is inserted using text_poke_bp() function.

(arch/x86/kernel/alternative.c)

LSE Security System
Laboratory of Epita

Optimization

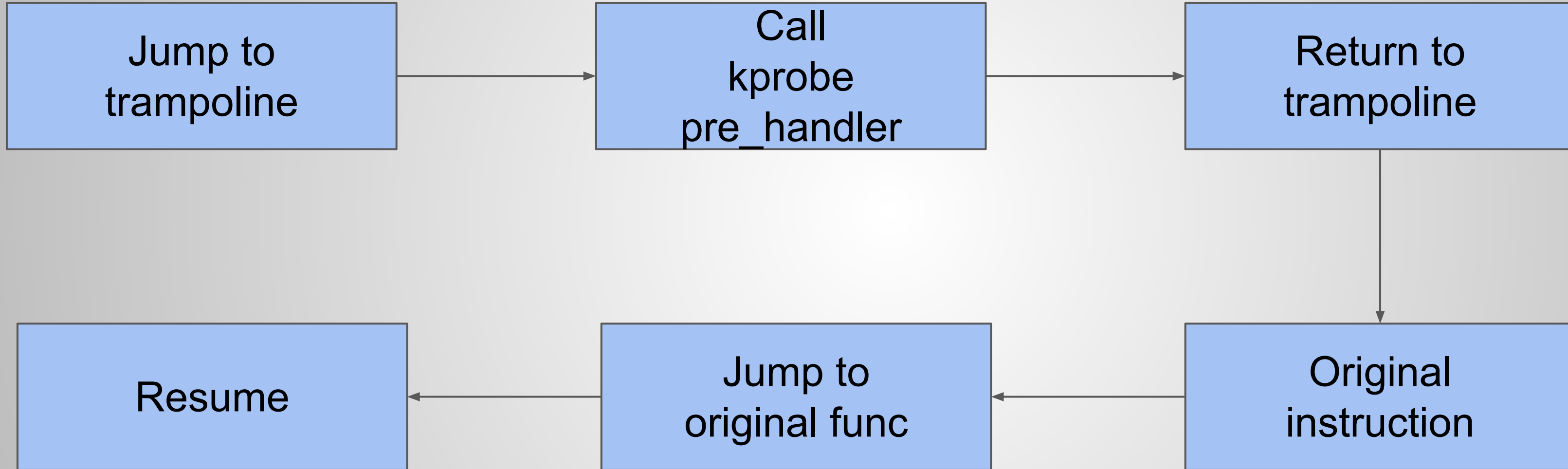
```
void *text_poke_bp(void *addr, const void *opcode, size_t len, void *handler);
```

How it works:

1) Add an int3 trap to the patched address
2) Sync cores
3) Update all but the first byte of the patched range
4) Sync cores
5) Replace the first byte (int3)
6) Sync cores
7) Done !

LSE Security System
Laboratory of Epita

Optimized probe



LSE Security System
Laboratory of Epita

The end

If you are interested:

- Documentation/kprobes.txt
- arch/x86/kernel/kprobes/*
- kernel/kprobes.c
- include/linux/kprobes.h
- samples/kprobes/*
- https://lwn.net/Articles/132196/ // small intro to kprobes
- http://phrack.org/issues/67/6.html // doing shit with kprobes
- Man intel

LSE Security System
Laboratory of Epita

Thank you !

LSE
Security
System
Laboratory of Epita