Using Linux Tracing for Security

Brandon Edwards, Nick Gregory

For CSAW C2

2019-11-07

CAPSULE8

Introduction

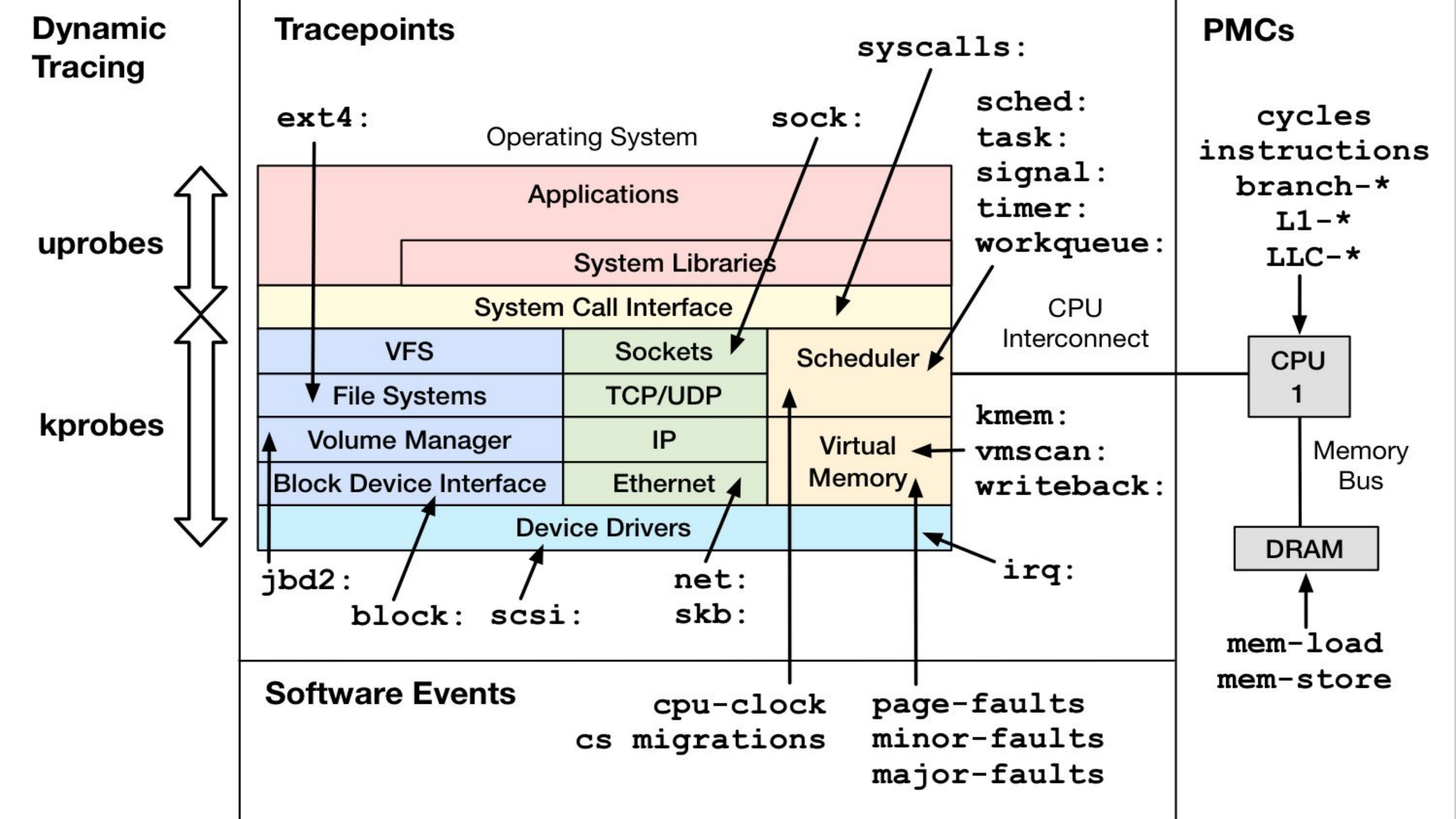
Hi

CAPSULE8

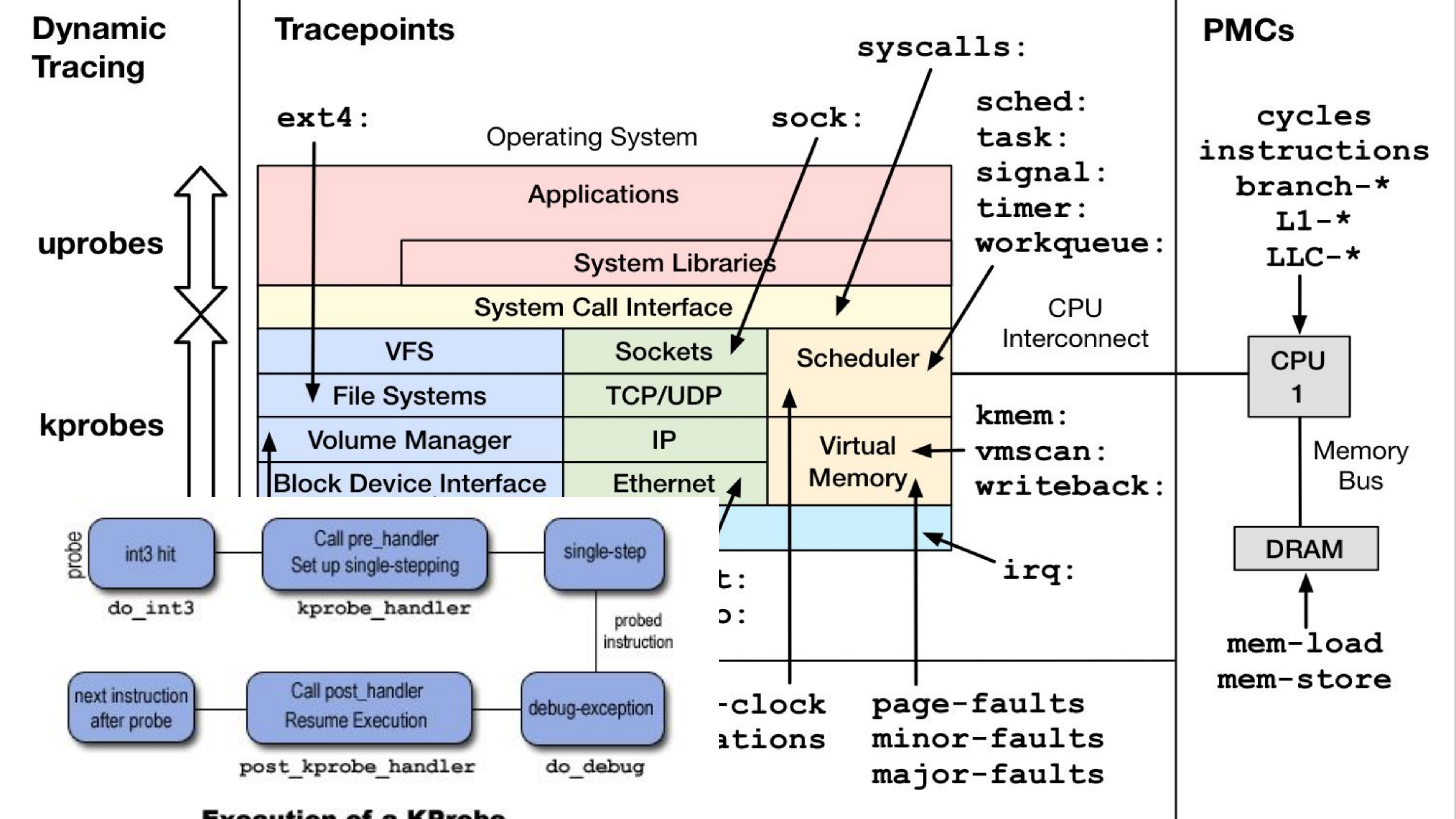
Intro

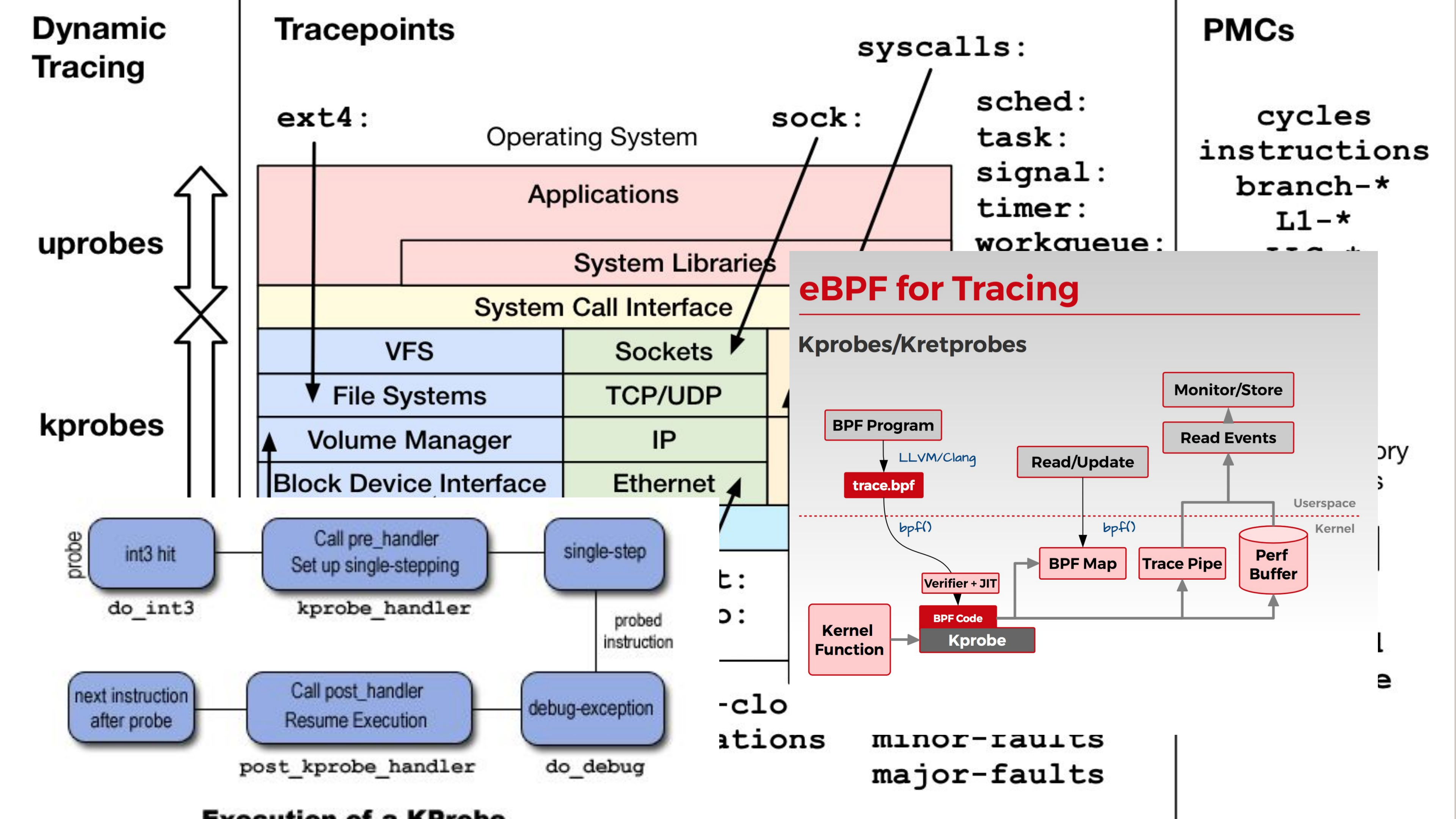
- This isn't even a crash course, but a brief demo of possibilities using Linux tracing subsystems
- Not covering the history of these systems, old kernels, or the wild changes between versions
- Very light mention of ftrace, kprobe, eBPF
- Ftrace
- For a good cry, try reading **`perf_event_open(2)`**

CAPSULE8



Dynamic Tracing
Tracepoints
PMCs
syscalls:
ext4:
sock:
sched:
task:
signal:
timer:
workqueue:
Operating System
cycles
instructions
branch-*
L1-*
LLC-*
uprobes
Applications
System Libraries
System Call Interface
CPU Interconnect
VFS
Sockets
Scheduler
CPU 1
File Systems
TCP/UDP
kprobes
Volume Manager
IP
Virtual Memory
kmem:
vmscan:
writeback:
Memory Bus
Block Device Interface
Ethernet
Device Drivers
DRAM
jbd2:
block:
scsi:
net:
skb:
irq:
mem-load
mem-store
Software Events
cpu-clock
cs migrations
page-faults
minor-faults
major-faults


Dynamic Tracing
Tracepoints
PMCs
syscalls:
ext4:
sock:
sched:
task:
signal:
timer:
workqueue:
Operating System
Applications
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device Interface
Ethernet
uprobes
kprobes
CPU Interconnect
cycles
instructions
branch-*
L1-*
LLC-*
CPU 1
Memory Bus
DRAM
mem-load
mem-store
kmem:
vmscan:
writeback:
irq:
-clock
ations
page-faults
minor-faults
major-faults
probe
int3 hit
Call pre_handler
Set up single-stepping
single-step
do_int3
kprobe_handler
probed instruction
next instruction after probe
Call post_handler
Resume Execution
debug-exception
post_kprobe_handler
do_debug


Dynamic Tracing
uprobes
kprobes
Tracepoints
ext4:
Operating System
sock:
syscalls:
sched:
task:
signal:
timer:
Applications
System Libraries
System Call Interface
VFS
Sockets
File Systems
TCP/UDP
Volume Manager
IP
Block Device Interface
Ethernet
PMCs
cycles
instructions
branch-*
L1-*
-clo
ations
minor-faults
major-faults
probe
int3 hit
do_int3
Call pre_handler
Set up single-stepping
kprobe_handler
single-step
probed instruction
next instruction after probe
Call post_handler
Resume Execution
post_kprobe_handler
debug-exception
do_debug
eBPF for Tracing
Kprobes/Kretprobes
Monitor/Store
BPF Program
Read Events
LLVM/Clang
Read/Update
trace.bpf
Userspace
bpf()
bpf()
Kernel
BPF Map
Trace Pipe
Perf Buffer
Verifier + JIT
BPF Code
Kernel Function
Kprobe

Kernel Probes

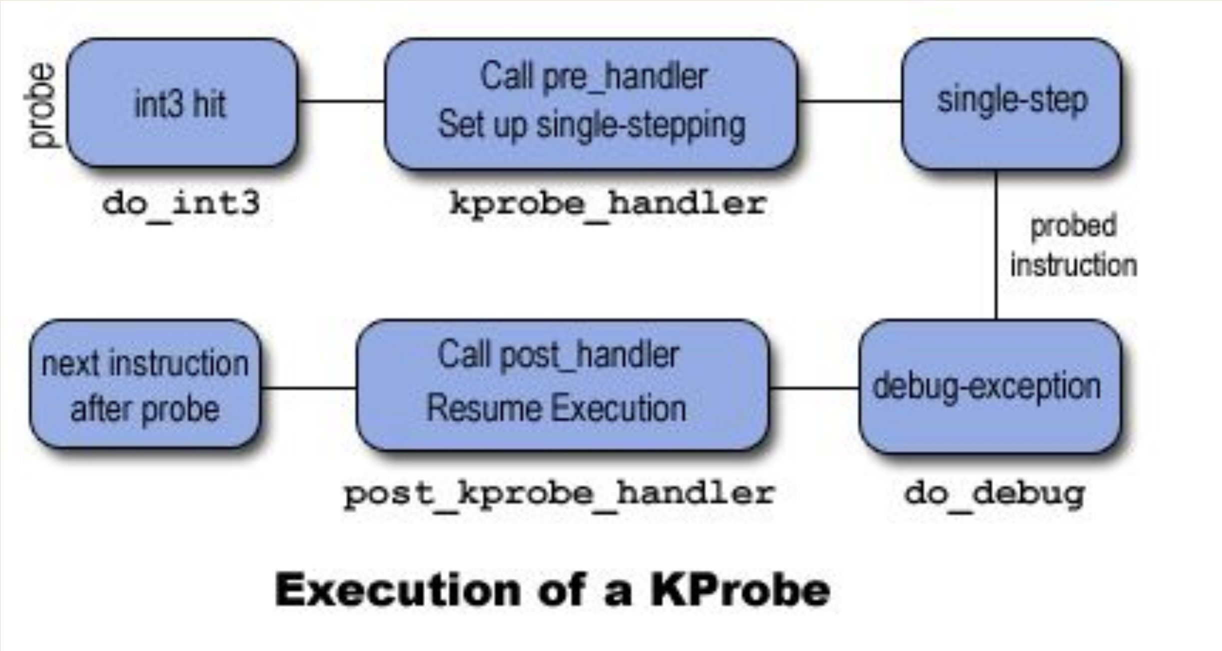
kprobes

Can be thought of as an extremely limited breakpoint/trampoline

You can set them on (most) functions in the kernel:

- Read registers
- Read kernel memory
- Filter on the data returned by them (kinda, it's super limited)
- Attach eBPF programs to do much fancier logic

CAPSULE8


probe
int3 hit
Call pre_handler
Set up single-stepping
single-step
do_int3
kprobe_handler
probed
instruction
next instruction
after probe
Call post_handler
Resume Execution
debug-exception
post_kprobe_handler
do_debug
Execution of a KProbe

CAPSULE8

kprobes

Everything is a file!

Set kprobes through sysfs by writing to:

`/sys/kernel/debug/tracing/kprobe_events`

CAPSULE8

kprobe format

```
p:listenprobe sys_listen sd=%di
```

CAPSULE8

kprobe format

User defined name of probe

```
p:listenprobe sys_listen sd=%di
```

kprobe format

Symbol of location to place the probe

```
p:listenprobe sys_listen sd=%di
```

CAPSULE8

kprobe format

User defined variable name for output

```
p:listenprobe sys_listen sd=%di
```

CAPSULE8

kprobe format

Hideous AT&T syntax for register to collect

```
p:listenprobe sys_listen sd=%di
```

CAPSULE8

kretprobe format

```
r:listenprobe sys_listen ret=%ax
```

kprobes

Enable your kprobe

```
echo 1 >
/sys/kernel/debug/tracing/events/kprobes/listenprobe/enable
```

CAPSULE8

Basic kprobe output

By default output is written to:

```
/sys/kernel/debug/tracing/trace
```

kprobes output

```
nc-5747 listenprobe: (sys_listen) sd=0x3
```

perf-tools

- execsnoop
- opensnoop
- iosnoop
- ...

zoom

```
root@host:~# ./execsnoop.py
PCOMM     PID    PPID   RET ARGS
zoom      3884   3872     0 /usr/bin/zoom
sh        3887   3884     0 zoom
zoom      3888   3887     0 /opt/zoom/zoom
…
pidof     3938   3888     0 /bin/pidof zoom
pmap      3939   3888     0 /usr/bin/pmap -x 3888
```

CAPSULE8

zoom

```
root@host:~# ./execsnoop.py
PCOMM        PID     PPID    RET ARGS
zoom         3884    3872      0 /usr/bin/zoom
sh           3887    3884      0 zoom
zoom         3888    3887      0 /opt/zoom/zoom
…
pidof        3938    3888      0 /bin/pidof zoom
pmap         3939    3888      0 /usr/bin/pmap -x 3888
```

CAPSULE8

ptrace-less strace

- No need to be attached to process; system-wide visibility
- Does not interfere with signal handling operations
    - Chrome
    - Golang
    - Malware looking for ptrace
- Easily introspect from outside container

Demo

CAPSULE8

Userland Probes

uprobes

Similar to kprobes, but instead these are set on program or library functions

```
p:probeName library:offset
```

CAPSULE8

uprobe Uses

Simple example: `readline()` to snoop on terminal activity

uprobe Uses

19	2.002089135	172.29.242.207	128.122.0.11	DNS	81	Standard query 0x415f A google.com OPT
20	2.002317227	172.29.242.207	128.122.0.11	DNS	81	Standard query 0x90f7 AAAA google.com OPT
21	2.007576752	128.122.0.11	172.29.242.207	DNS	109	Standard query response 0x90f7 AAAA google.com AAAA 2607:f8b0:4006:812::200e OPT
22	2.007633154	128.122.0.11	172.29.242.207	DNS	97	Standard query response 0x415f A google.com A 172.217.10.142 OPT
23	2.008645681	172.29.242.207	172.217.10.142	ICMP	98	Echo (ping) request id=0x52bd, seq=1/256, ttl=64 (reply in 24)
24	2.084860031	172.217.10.142	172.29.242.207	ICMP	98	Echo (ping) reply id=0x52bd, seq=1/256, ttl=48 (request in 23)
25	2.085620444	172.29.242.207	128.122.0.11	DNS	98	Standard query 0xb514 PTR 142.10.217.172.in-addr.arpa OPT
26	2.089532034	128.122.0.11	172.29.242.207	DNS	164	Standard query response 0xb514 PTR 142.10.217.172.in-addr.arpa PTR lga34s16-in-f14.1e100.net OPT
27	2.317541832	172.217.10.78	172.29.242.207	TLSv1.2	327	Application Data
28	2.317596501	172.29.242.207	172.217.10.78	TCP	66	38334 → 443 [ACK] Seq=1 Ack=3057 Win=1436 Len=0 TSval=3478092147 TSecr=4032475182
29	2.328153209	172.217.10.78	172.29.242.207	TLSv1.2	511	Application Data
30	2.328206103	172.29.242.207	172.217.10.78	TCP	66	38334 → 443 [ACK] Seq=1 Ack=3502 Win=1436 Len=0 TSval=3478092158 TSecr=4032475278
31	2.328244692	172.217.10.78	172.29.242.207	TLSv1.2	341	Application Data
32	2.328264544	172.29.242.207	172.217.10.78	TCP	66	38334 → 443 [ACK] Seq=1 Ack=3777 Win=1434 Len=0 TSval=3478092158 TSecr=4032475283

- Frame 19: 81 bytes on wire (648 bits), 81 bytes captured (648 bits) on interface 0
- Ethernet II, Src: IntelCor_ce:0f:09 (b8:8a:60:ce:0f:09), Dst: IcannIan_00:01:32 (00:00:5e:00:01:32)
- Internet Protocol Version 4, Src: 172.29.242.207, Dst: 128.122.0.11
- User Datagram Protocol, Src Port: 57568, Dst Port: 53
- Domain Name System (query)

CAPSULE8

uprobe Uses

`getaddrinfo()` (gethostbyname), to correlate `connect()` traffic to hostnames, and which process called it

CAPSULE8

Demo

Tracepoints

ftrace

- Tracers:
  - *function* – default tracer
  - *function_graph* – constructs call graph
  - *irqsoff*, *preempoff*, *preemptirqsoff*, *wakeup*, *wakeup_rt* – latency tracers
  - *nop*

- No int3! Instead mcount & nop stub tricks at each function to be traced (compile-time enabled)

- In 3.10 support for kprobes was added to ftrace: **Dynamic (f)tracing**!

Mainly debugging not profiling!

CAPSULE8

fgraph

- Having trouble figuring out what the kernel is doing?
- Let it tell you!

```
static ssize_t tty_read(struct file *file, char __user *buf, size_t count,
            loff_t *ppos)
{
    int i;
    struct inode *inode = file_inode(file);
    struct tty_struct *tty = file_tty(file);
    struct tty_ldisc *ld;

    if (tty_paranoia_check(tty, inode, "tty_read"))
        return -EIO;
    if (!tty || tty_io_error(tty))
        return -EIO;

    /* We want to wait for the line discipline to sort out in this
       situation */
    ld = tty_ldisc_ref_wait(tty);
    if (!ld)
        return hung_up_tty_read(file, buf, count, ppos);
    if (ld->ops->read)
        i = ld->ops->read(tty, file, buf, count);
```

CAPSULE8

```
struct tty_ldisc_ops {
    int magic;
    char    *name;
    int num;
    int flags;

    /*
     * The following routines are called from above.
     */
    int (*open)(struct tty_struct *);
    void    (*close)(struct tty_struct *);
    void    (*flush_buffer)(struct tty_struct *tty);
    ssize_t (*read)(struct tty_struct *tty, struct file *file,
            unsigned char __user *buf, size_t nr);
    ssize_t (*write)(struct tty_struct *tty, struct file *file,
             const unsigned char *buf, size_t nr);
    int (*ioctl)(struct tty_struct *tty, struct file *file,
```

CAPSULE8

/linux/sound/soc/codecs/			
HAD	cx20442.h	11	`extern struct tty_ldisc_ops v253_ops;`
HAD	cx20442.c	287	`struct tty_ldisc_ops v253_ops = {`
/linux/drivers/pps/clients/			
HAD	pps-ldisc.c	98	`static struct tty_ldisc_ops pps_ldisc_ops;`
/linux/drivers/input/serio/			
HAD	serport.c	272	`static struct tty_ldisc_ops serport_ldisc = {`
/linux/drivers/staging/speakup/			
HAD	spk_ttyio.c	103	`static struct tty_ldisc_ops spk_ttyio_ldisc_ops = {`
/linux/net/nfc/nci/			
HAD	uart.c	454	`static struct tty_ldisc_ops nci_uart_ldisc = {`
/linux/sound/soc/ti/			
HAD	ams-delta.c	397	`static struct tty_ldisc_ops cx81801_ops = {`
/linux/drivers/net/caif/			
HAD	caif_serial.c	382	`static struct tty_ldisc_ops caif_ldisc = {`
/linux/drivers/net/ppp/			
HAD	ppp_synctty.c	365	`static struct tty_ldisc_ops ppp_sync_ldisc = {`
HAD	ppp_async.c	372	`static struct tty_ldisc_ops ppp_ldisc = {`
/linux/drivers/net/can/			
HAD	slcan.c	688	`static struct tty_ldisc_ops slc_ldisc = {`
/linux/drivers/staging/isdn/gigaset/			
HAD	ser-gigaset.c	724	`static struct tty_ldisc_ops gigaset_ldisc = {`
/linux/drivers/misc/ti-st/			
HAD	st_core.c	828	`static struct tty_ldisc_ops st_ldisc_ops = {`
/linux/drivers/net/wan/			
HAD	x25_asy.c	752	`static struct tty_ldisc_ops x25_ldisc = {`
/linux/drivers/bluetooth/			
HAD	hci_ldisc.c	823	`static struct tty_ldisc_ops hci_uart_ldisc;` in hci_uart_init()
/linux/drivers/net/hamradio/			
HAD	mkiss.c	935	`static struct tty_ldisc_ops ax_ldisc = {`
HAD	6pack.c	750	`static struct tty_ldisc_ops sp_ldisc = {`

CAPSULE8

fgraph via trace-cmd

```
trace-cmd record -p function_graph -g tty_read
```

This will create a file called `trace.dat`

trace-cmd report

Terminal - bme@lapcloud: ~/traces/tty_read

```
2316.485872: funcgraph_entry:                    |  tty_read() {
2316.485878: funcgraph_entry:        0.056 us    |    tty_paranoia_check();
2316.485879: funcgraph_entry:                    |    tty_ldisc_ref_wait() {
2316.485898: funcgraph_exit:       + 19.268 us   |    }
2316.485899: funcgraph_entry:                    |    n_tty_read() {
2316.485933: funcgraph_exit:       + 33.870 us   |    }
2316.485933: funcgraph_entry:                    |    tty_ldisc_deref() {
2316.485933: funcgraph_exit:         0.417 us    |    }
2316.485933: funcgraph_entry:        0.104 us    |    get_seconds();
2316.485934: funcgraph_exit:       + 55.960 us   |  }
```

CAPSULE8

There is also **`kernelshark`** ...

Internals Deep Dive

https://youtu.be/93uE_kWWQjs

CAPSULE8

Conclusion

- k(ret)probes
  - Log specific internal kernel function calls
- u(ret)probes
  - Introspect userland without a debugger
- ftrace/fgraph
  - Explore kernel dynamically
  - Compare different call graphs

CAPSULE8

Recap: Tools Used

- perf-tools
  - https://github.com/brendangregg/perf-tools
  - perf-tools-unstable in Ubuntu
- trace-cmd

CAPSULE8

Questions?