Linux tc and eBPF.

Daniel Borkmann
<daniel@iogearbox.net>
Noiro Networks / Cisco Systems

fosdem16, January 31, 2016

Background, history.

- BPF origins as a generic, fast and 'safe' solution to packet parsing
- tcpdump → libpcap → compiler → bytecode → kernel interpreter
- Intended as early drop point in `AF_PACKET` kernel receive path
- JIT'able for x86_64 since 2011, ppc, sparc, arm, arm64, s390, mips

```
# tcpdump -i any -d ip
(000) ldh      [14]
(001) jeq      #0x800           jt 2 jf 3
(002) ret      #65535
(003) ret      #0
```

2012, No longer networking only!

- BPF engine used for seccomp (syscall filtering)
- Used inside Chrome as a sandbox, minimal example in `bpf_asm`:

```
ld [4]                  /* offsetof(struct seccomp_data, arch) */
jne #0xc000003e, bad    /* AUDIT_ARCH_X86_64 */
ld [0]                  /* offsetof(struct seccomp_data, nr) */
jeq  #15, good          /* __NR_rt_sigreturn */
jeq #231, good          /* __NR_exit_group */
jeq  #60, good          /* __NR_exit */
jeq   #0, good          /* __NR_read */
jeq   #1, good          /* __NR_write */
jeq   #5, good          /* __NR_fstat */
jeq   #9, good          /* __NR_mmap */
[...]
bad:  ret #0            /* SECCOMP_RET_KILL */
good: ret #0x7fff0000   /* SECCOMP_RET_ALLOW */
```

BPF (any flavour) used in the kernel today.

- Networking
  - Socket filtering for most protocols
  - `AF_PACKET` fanout demuxing
  - `SO_REUSEPORT` socket demuxing
  - tc classifier (`cls_bpf`) and actions (`act_bpf`)
  - team driver load balancing
  - netfilter xtables (`xt_bpf`)
  - Some misc ones: PTP classifier, PPP and ISDN
- Tracing
  - BPF as kprobes-based extensions
- Sandboxing
  - syscall filtering with seccomp

Classic BPF (cBPF) in a nutshell.

- 32 bit, available register: A, X, M[0-15], (pc)
- A used for almost everything, X temporary register, M[] stack
- Insn: 64 bit (`u16:code, u8:jt, u8:jf, u32:k`)
- Insn classes: ld, ldx, st, stx, alu, jmp, ret, misc
- Forward jumps, max 4096 instructions, statically verified in kernel
- Linux-specific extensions overload `ldb/ldh/ldw` with `k`← `off+x`
- `bpf_asm`: 33 instructions, 11 addressing modes, 16 extensions
- Input data/"context" (`ctx`), e.g. `skb`, `seccomp_data`
- Semantics of exit code defined by application

Extended BPF (eBPF) as next step.

- 64 bit, 32 bit sub-registers, available register: R0-R10, stack, (pc)
- Insn: 64 bit (`u8:code`, `u8:dst_reg`, `u8:src_reg`, `s16:off`, `s32:imm`)
- New insns: dw ld/st, mov, alu64 + signed shift, endian, calls, xadd
- Forward (& backward*) jumps, max 4096 instructions
- Generic helper function concept, several kernel-provided helpers
- Maps with arbitrary sharing (user space apps, between eBPF progs)
- Tail call concept for eBPF programs, eBPF object pinning
- LLVM eBPF backend: `clang -O2 -target bpf -o foo.o foo.c`
  - C → LLVM → ELF → tc → kernel (verification/JIT) → cls_bpf (exec)

eBPF, General remarks.

- Stable ABI for user space, like the case with cBPF
- Management via `bpf(2)` syscall through file descriptors
- Points to kernel resource $\rightarrow$ eBPF map / program
- No cBPF interpreter in kernel anymore, all eBPF!
- Kernel performs internal cBPF to eBPF migration for cBPF users
- JITs for eBPF: x86_64, s390, arm64 (remaining ones are still cBPF)
- Various stages for in-kernel cBPF loader
- Security (verifier, JIT spraying mitigations, RO images, unpriv restr.)

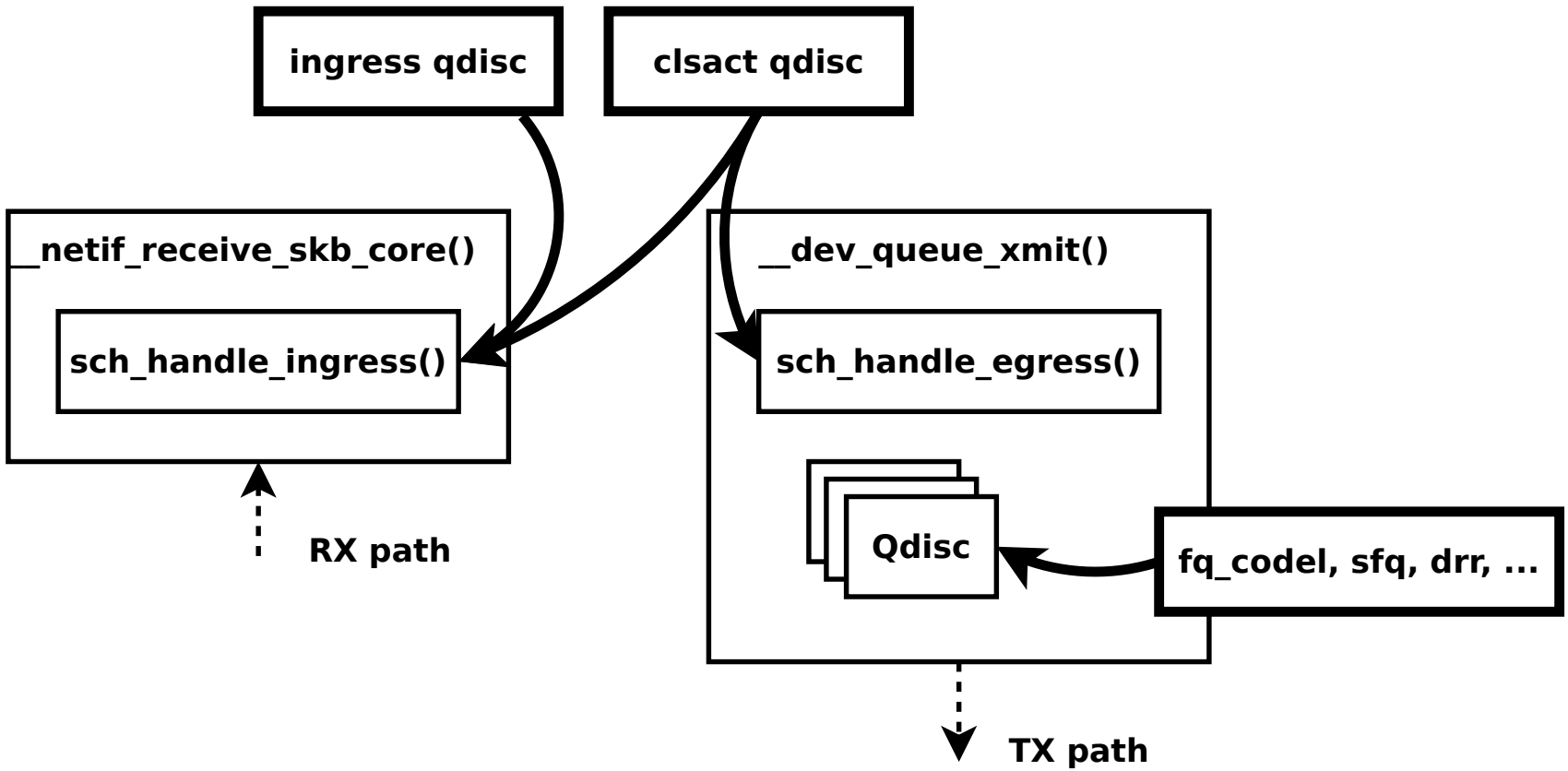
eBPF and cls_bpf.

- cls_bpf as cBPF-based classifier in 2013, eBPF support since 2015
- Minimal fast-path, just calls into `BPF_PROG_RUN()`
- Instance holds one or more BPF programs, 2 operation modes:
  - Calls into full tc action engine `tcf_exts_exec()` for e.g. `act_bpf`
  - Direct-action (DA) fast-path for immediate return after BPF run
- In DA, eBPF prog sets `skb->tc_classid`, returns action code
  - Possible codes: ok, shot, stolen, redirect, unspec
- tc frontend does all the setup work, just sends fd via netlink

eBPF and cls_bpf.

- skb metadata:
  - Read/write: mark, priority, tc_index, cb[5], tc_classid
  - Read: len, pkt_type, queue_mapping, protocol, vlan_*, ifindex, hash
- Tunnel metadata:
  - Read/write: tunnel key for IPv4/IPv6 (dst-meta by vxlan, geneve, gre)
- Helpers:
  - eBPF map access (lookup/update/delete)
  - Tail call support
  - Store/load payload (multi-)bytes
  - L3/L4 csum fixups
  - skb redirection (ingress/egress)
  - Vlan push/pop and tunnel key
  - `trace_printk` debugging
  - `net_cls` cgroup classid
  - Routing realms (`dst->tclassid`)
  - Get random number/cpu/ktime

cls_bpf, Invocation points.



cls_bpf, Example setup in 1 slide.

```
$ clang -O2 -target bpf -o foo.o foo.c

# tc qdisc  add dev em1 clsact
# tc qdisc show dev em1
[...]
qdisc clsact ffff: parent ffff:fff1

# tc filter add dev em1 ingress bpf da obj foo.o sec p1
# tc filter add dev em1 egress  bpf da obj foo.o sec p2

# tc filter show dev em1 ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 foo.o:[p1] direct-action

# tc filter show dev em1 egress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 foo.o:[p2] direct-action

# tc filter del dev em1 ingress pref 49152
# tc filter del dev em1 egress  pref 49152
```

tc frontend.

- Common loader backend for `f_bpf`, `m_bpf`, `e_bpf`
- Walks ELF file to generate program fd, or fetches fd from pinned
- Setup via ELF object file in multiple steps:
  - Mounts bpf fs, fetches all ancillary sections
  - Sets up maps (fd from pinned or new with pinning)
  - Relocations for injecting map fds into program
  - Loading of actual eBPF program code into kernel
  - Setup and injection of tail called sections
- Grafting of existing prog arrays, dumping trace pipe
- Also supports passing map fds via UDS to agent

tc eBPF examples, minimal module.

```
$ cat >foo.c <<EOF
   #include "bpf_api.h"

   __section_cls_entry
   int cls_entry(struct __sk_buff *skb)
   {
           /* char fmt[] = "hello prio%u world!\n"; */
           skb->priority = get_cgroup_classid(skb);
           /* trace_printk(fmt, sizeof(fmt), skb->priority); */
           return TC_ACT_OK;
   }

   BPF_LICENSE("GPL");
EOF

$ clang -O2 -target bpf -o foo.o foo.c
# tc filter add dev em1 egress bpf da obj foo.o
# tc exec bpf dbg              # -> dumps trace_printk()

# cgcreate -g net_cls:/foo
# echo 6 > foo/net_cls.classid
# cgexec -g net_cls:foo ./bar  # -> app ./bar xmits with priority of 6
```

tc eBPF examples, map sharing.

```
#include "bpf_api.h"

BPF_ARRAY4(map_sh, 0, PIN_OBJECT_NS, 1);
BPF_LICENSE("GPL");

__section("egress") int egr_main(struct __sk_buff *skb)
{
        int key = 0, *val;
        val = map_lookup_elem(&map_sh, &key);
        if (val)
                lock_xadd(val, 1);
        return BPF_H_DEFAULT;
}

__section("ingress") int ing_main(struct __sk_buff *skb)
{
        char fmt[] = "map val: %d\n";
        int key = 0, *val;
        val = map_lookup_elem(&map_sh, &key);
        if (val)
                trace_printk(fmt, sizeof(fmt), *val);
        return BPF_H_DEFAULT;
}
```

tc eBPF examples, tail calls.

```
#include "bpf_api.h"

BPF_PROG_ARRAY(jmp_tc, JMP_MAP, PIN_GLOBAL_NS, 1);
BPF_LICENSE("GPL");

__section_tail(JMP_MAP, 0) int cls_foo(struct __sk_buff *skb)
{
        char fmt[] = "in cls_foo\n";
        trace_printk(fmt, sizeof(fmt));
        return TC_H_MAKE(1, 42);
}

__section_cls_entry int cls_entry(struct __sk_buff *skb)
{
        char fmt[] = "fallthrough\n";
        tail_call(skb, &jmp_tc, 0);
        trace_printk(fmt, sizeof(fmt));
        return BPF_H_DEFAULT;
}

$ clang -O2 -DJMP_MAP=0 -target bpf -o graft.o graft.c
# tc filter add dev em1 ingress bpf obj graft.o
```

Code and further information.

- Take-aways:
  - No, development on `tc` is not in deep hibernation mode ;)
  - eBPF implementation details may be complex, BUT workflow and writing eBPF programs is really easy (perhaps easiest in `tc`?)
  - Low overhead, fully programmable for your specific use-case
  - Native performance when JITed!
- Code:
  - Everything upstream in kernel, iproute2 and llvm!
  - Available from usual places, e.g. `https://git.kernel.org/`
- Some further information:
  - Man pages `bpf(2)`, `tc-bpf(8)`
  - Examples in iproute2's `examples/bpf/`
  - `Documentation/networking/filter.txt`