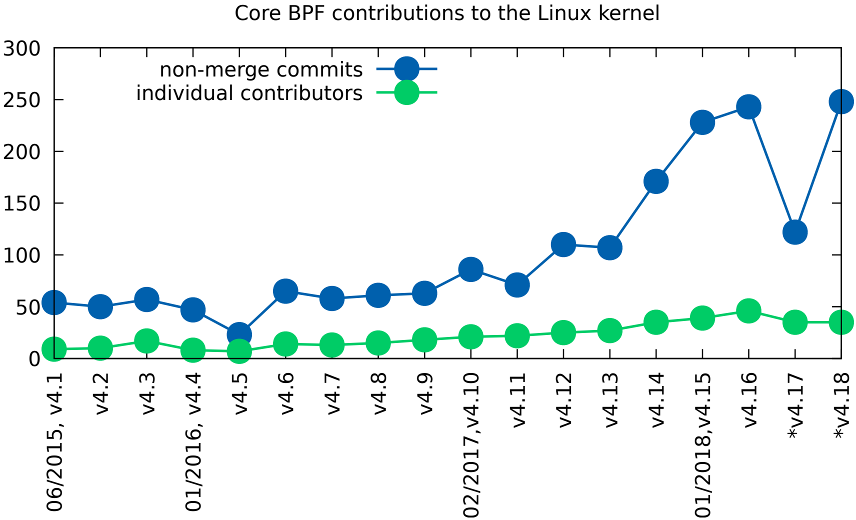
netconf: BPF, Cilium, bpfilter items.

Daniel Borkmann
<daniel@covalent.io>
Covalent IO

netconf, May 31, 2018

Part 1: BPF Maintenance.

BPF Maintenance: Updated stats



BPF Maintenance: Current state

- 4.18 kernel stats will be a new record for BPF
  - 248 patches (excluding XDP driver changes)
  - 35 different contributors
- Since approx last netconf, end of Nov 2017:
  - 2,065 patches in patchwork's bpf delegate
    - avg. 17 patches per work day
  - 794 ended up in state 'accepted'
    - avg. 7 patches per work day
  - 18 pull-requests for bpf-next
  - 26 pull-requests for bpf

BPF Maintenance: Scalability

- Bottleneck: reviews of incoming patches
  - Current solution: two stage review process
  - Weekly review oncall rotation for spreading load:
    - Yonghong, Martin, Song, Alexei, Daniel
  - Goal: promtly review of *all* BPF related patches coming to netdev
  - Basic rules of thumb:
    - Started review also includes subsequent patches, even beyond rotation
    - Changes requested from reviewer → purged from patchwork queue
    - Okay to make mistakes in reviews, okay to ask questions of course
    - Once series is acked by reviewer → second, final stage
  - Final vetting: Alexei, Daniel
  - Benefit of this model:
    - Avoidance of us being bottleneck → better scalability
    - Reviewers gain more insights into various BPF parts
    - Improved quality of reviews and submitted code

BPF Maintenance: Scalability

- stable kernel backports
  - Regularly batched depending on load
  - Whenever conflict free and tests okay → punting cherry-pick to Greg
  - Otherwise manual timely backport and testing
    - Includes Cilium test suite and BPF selftests
    - Extensive BPF kernel selftests crucial (!)
  - Supported (current) stable branches: 4.9, 4.14, 4.16
  - No / little handling of 4.4 due to lack of test hardware
- bpf / bpf-next interdependencies dependencies
  - 1-3 days worst case stalls on dependency chains like *bpf* → *net* → *net-next* → *bpf-next* to get bpf into bpf-next
  - Trying to keep this to a minimum iff possible
  - But upon request we'll flush either tree out immediately to avoid annoying waiting time for regular syncs

BPF Maintenance: Testing, debugging, documentation

- BPF kernel selftests
  - test_verifier + test_kmod.sh alone run 2,018 test progs
  - Biggest subsystem under kselftests along with RCU-torture
  - Often mandatory and developed in sync with features or fixes
  - Very happy with state of syzkaller as well (!)
- Debugging, introspection: bpftool
  - Goal: bpftool ≡ *the* go-to tool around all things (e)BPF
  - Similar co-development model for new features as with selftests
  - Still lacking behind (also libbpf): general availability as distro packages
- Documentation for lowering entrance barrier
  - Proper BPF (uapi) helper documentation made mandatory as part of bpf.h, integration into man-pages project ongoing. Alternative: bpftool
  - BPF/XDP refguide currently in Cilium (`http://docs.cilium.io`)
  - Design and devel FAQ added. Feature matrix for JITs / XDP drivers?

BPF Maintenance: Misc random items

- BPF tooling include infrastructure - better options?
- BPF boot options vs sysctl discussion
- Removal of `attr->kern_version` from kprobes side
- AF_XDP temporary UAPI removal till full zero-copy lands
- Widespread JIT changes and testing challenging
  - Good test coverage: x86_64, arm64, s390x, nfp, ppc64*, sparc64*
  - State unclear: mips64, arm32, x86_32

Part 2: Cilium and BPF.

Cilium + BPF: Today from kernel PoV

- Orchestration layer pluggable (e.g.) into Kubernetes for providing security and connectivity for microservices at L3/L4 and L7
- Fully distributed, service mesh data path with BPF
- Uses three flavors of BPF programs: XDP, cls_bpf, sk progs*
    - XDP use case mainly for early drop DDoS mitigation today (CIDR + Cilium endpoints), potentially DSR LB in future
    - cls_bpf in direct-action mode for all heavy duty data path work in BPF (ingress + egress policy enforcement (label-, CIDR-based on L3, L4, redirect to L7 proxy; individual + combined policies), DSR LB, connection tracking, NAT, NAT64, vxlan / geneve-based collect-meta encap / decap, routing / forwarding, host delivery, metrics collection, perf RB events (debugging, pkt tracing), arp handling, ...)
        - Progs attached on phys device, overlays, veths (enforcement on host-facing veth side, ingress)
    - sk progs for L7 in-kernel policy enforcement and redirect to accelerate L7 proxies like Envoy* (e.g. HTTP, gRPC, Kafka, others)

Cilium + BPF: sockmap programs, ULP

- Microservices and shift to 'API economy'
- Typical k8s deployment: Istio service mesh with Envoy side-car proxy
- Control plane API → Pilot (deploys configs to Envoy), Mixer (policy, routing, telemetry, quota), Auth (TLS certs for Envoy)
- Data plane: Envoy L7 proxy with protos like HTTP, gRPC, Kafka, with or without TLS
  - Deployed in front of every service across fleet, talk only via proxy
  - Combinable with other services e.g. Prometheus, Zipkin for monitoring
- E.g. allows for rolling updates of services via Mixer routing
- All transparent to app developer, no extra code for functionality
- Today: Cilium deployed along with Istio to provide L3 - L7 policy
  - BPF data path handles all forwarding logic to / from Envoy
  - Envoy has BPF specific extensions to exchange information with Cilium's BPF data path

Cilium + BPF: sockmap programs, ULP

- Issue with side-cars: from serivce to service min 6 stack traversals
- Cilium's BPF sockmap transparently accelerate proxies like Envoy
  - Think of sockmap progs like cls_bpf for socket layer
  - Parser/verdict BPF program pair for sk_msg / sk_skb
  - msg_apply_cork(), msg_apply_bytes(), msg_pull_data()
  - Managed from Cilium side via cgroups v2
  - Next up: native loops and kTLS integration
  - Complex protos via slow-path 'umh module' or AF_XDP?
- ULP interplay with sockmap and kTLS tricky, today: pick one
- Potential options
  - Stacked ULP (egress: sockmap → kTLS, ingress: kTLS → sockmap)
    - Fixed order so BPF can run on unencrypted data
  - Single ULP only but with optional sockmap / kTLS extension

Cilium + BPF: BPF related observations from deployments

- Minimal kernel of 4.9.17+ required, 4.9 seems reasonable base
- Various verifier proglets to test feature availability, works okay
- Main headache on verifier complexity side, but with code workarounds (e.g. avoidance of dynamic map access) under control
  - Causes for complexity increase often like finding needle in haystack
  - Iff we keep complexity limit in future → bpftool for debugging
- From time to time other LLVM / verifier quirks but largely decreased
- BPF side stable in general, user reported issues mainly higher in stack
  - Test matrix of supported 'kernel x LLVM' versions rather big
- Cilium has heavy use of tail call tricks → more retpoline cases
- Distro support with recent kernels and BPF enabled mostly okay

Cilium + BPF: Misc random items

- ipvlan + BPF integration bit of a hack compared to veths
  - Policy enforcement on host facing side only via VEPA mode
  - Enforces all traffic back to host side, BPF on tc egress on phys device
  - Redirects skb back to ingress side of phys dev to push to another netns
  - One resched point less than with veths though
- Upgrade / downgrade of BPF maps without full data loss iff possible (analysis via BTF)
  - Current loader detects map property changes and creates new map

Part 3: bpfilter.

bpfilter: Rationale and long term plan

- Transparently convert iptables / nftables requests into eBPF bytecode
  - Keeping existing UAPI working as is
  - Full reuse of efficient BPF infrastructure in data path
  - For example, JITs, XDP, even offloads for SmartNICs
    - XDP hints for !SmartNICs also beneficial for transparent reuse
  - Possibility to reduce kernel attack surface through BPF
    - BPF insns pushed through verifier from special 'umh module'
    - Code generated out of user space behind syscall boundary
    - Hooks would eventually become call to BPF_PROG_RUN() only
    - Enables removal of old xtables kernel code, etc
- Advantages of 'umh module' concept
  - Delegate potentially complex transformation in user space
  - Crash of bpfilter 'umh' module doesn't take down kernel
  - Built and shipped as part of kernel, no difference to kernel modules
  - Debugging, test suite, sanitizers, fuzzers, etc out of user space

bpfilter: Current state

- 'umh' module code and basic bpfilter skeleton merged
  - New fork_usermode_blob() helper
  - Kernel allocates unique file in tmpfs, populates it with data blob
  - UMH helper will exec that file, kernel creates 2 pipes on start (pipe to UMH, pipe from UMH)
  - Allows for bidirectional communication between kernel, UMH module
  - bpfilter.ko → Contains two pieces
    - bpfilter kernel module code for UMH setup / teardown
    - mbox proto for hooking UMH to bpfilter sockopt handling
    - rmmod of bpfilter kernel mod will remove UMH as well
  - bpfilter user space bit has main mbox handling loop
    - Currently just dummy, bailing out with error
  - RFC dissected iptables request blob, assembled generated BPF insns, and called into bpf(2) from there

bpfilter: Next steps

- Discussion on 'make BPFILTER_UMH depend on X86'
  - bpfilter_umh build issues due to hostprogs getting built with 'gcc' rather than '$(CROSS_COMPILE)gcc'
  - Test on HOSTCC's arch == kernel arch
- Remainder of bpfilter RFC cleaned up and matches further extended
  - Initial setting via XDP hook so far
  - Planned for next net-next cycle to push out
  - Basic L3, L4 and CIDRs, non-linear codegen optimizations via maps
- Initial framework for bpfilter selftest suite, potentially for kselftests
- Missing helpers designed also for potential reuse
- Up next: connection tracker, NAT engine, tproxy, ...
- Discussion via Alexei later: removing glibc dependency