Improving BPF Socket Destruction

LSF/MM/BPF 2026

Jordan Rife

About Me

- I work on networking for Google’s GKE (Google Kubernetes Engine) platform whose dataplane is built on top of Cilium.
- As part of my work, I contribute to upstream Cilium with a focus on datapath (BPF stuff).
    - Cilium’s socket LB feature both in Cilium and in the kernel.
    - And other things…

Background

- Cilium’s socketLB mode performs service load balancing at the socket level using BPF_PROG_TYPE_CGROUP_SOCK_ADDR hooks.
  - E.g. connect() to a service VIP rewrites destination to point to a real service backend address.
  - Avoids the need for per-packet NAT to implement Kubernetes Services.
- With connected UDP sockets, if the selected backend goes away the socket will continue to try talking to the now non-existent backend.
  - Non-connected or per-packet LB would redirect to a new backend for the service
- To remedy this, Cilium forcefully terminates such sockets to force a (well-behaved) application to create a new socket.

Background (continued)

- Socket termination used to be done with netlink sock_diag.
- Today, Cilium uses socket iterators + bpf_sock_destroy on systems that support it.
- Initial work was done by Aditi Ghag at Isovalent
    - Talk from LPC 2022 https://lpc.events/event/16/contributions/1358

How It Works Today

- Must be driven using a TCP/UDP BPF socket iterator program.
- Need to iterate through each socket in each network namespace to find the one(s) we’re interested in destroying.

Iteration Example

Three sockets in different network namespaces were connected to a backend. We want to find them and destroy them using bpf_sock_destroy.

UDP/TCP socket hash (e.g. UDP portaddr hash)

Key

connected to old backend

Sockets (each color is a network namespace)

Iteration Example

Current Namespace:
Sockets To Destroy: 0

UDP/TCP socket hash (e.g. UDP portaddr hash)

Key

connected to old backend

Sockets (each color is a network namespace)

Iteration Example

Current Namespace:
Sockets To Destroy: 0

UDP/TCP socket hash (e.g. UDP portaddr hash)

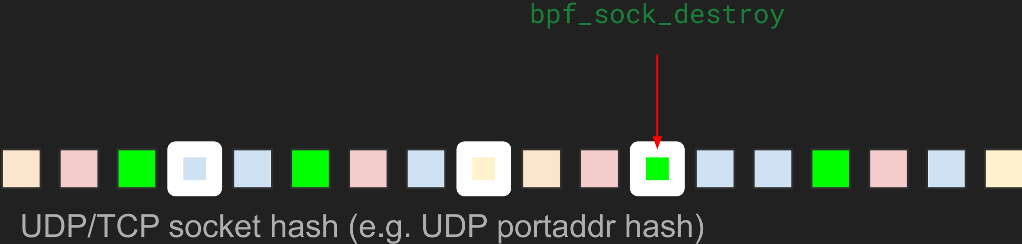
Key

connected to old backend

Sockets (each color is a network namespace)

Iteration Example

Current Namespace:
Sockets To Destroy: 1



UDP/TCP socket hash (e.g. UDP portaddr hash)

Key

connected to old backend

Sockets (each color is a network namespace)

Iteration Example

Current Namespace:
Sockets To Destroy: 1

`bpf_sock_destroy`

UDP/TCP socket hash (e.g. UDP portaddr hash)

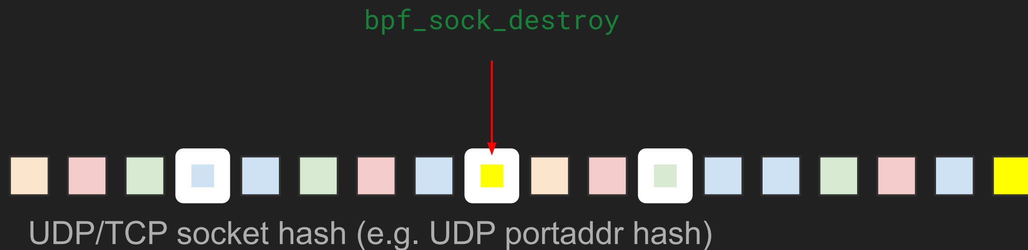
Key

connected to old backend

Sockets (each color is a network namespace)

Iteration Example

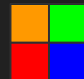
Current Namespace:
Sockets To Destroy: 1



UDP/TCP socket hash (e.g. UDP portaddr hash)

Key

connected to old backend

 Sockets (each color is a network namespace)

Not Ideal

- Lots of scanning and filtering under the hood.
    - Repeat for each network namespace (up to ~256x).
    - Look through every hash bucket and check each socket to see if it’s in the current namespace (256 - 65536 x).
    - Run BPF program on each one and check if socket is connected to our backend.
    - All to find a few sockets.
- Userspace orchestration needs to juggle network namespaces which is inconvenient.

Can This Be Better?

- See all sockets in one pass (no per-namespace iteration).
- Only visit the set of sockets we care about or at least a more constrained set of sockets (less filtering).
- Simplify control plane logic as well…

Initial Idea

- https://lore.kernel.org/netdev/20250909170011.239356-1-jordan@jrife.io/
    - Make `bpf_sock_destroy` work for `BPF_MAP_TYPE_SOCKHASH` map iterator programs
    - Bucketing sockets based on key prefixes (e.g. (daddr+dport)).
    - Iterate over just the bucket containing sockets we care about and destroy those sockets.
    - Basically a way to create a custom socket index.
- Too complicated
- Martin’s suggestion
    - A sleepable variant of `bpf_sock_destroy` that would acquire a socket lock itself.
    - Driven from a sleepable BPF program using test run.
    - Iterate over a generic map or data structure.

Some Other Ideas

- **Multi-Namespace Socket Iterators**
  - ***Summary:*** Add a mode that lets TCP and UDP socket iterators see sockets from every namespace as they iterate
  - ***Pros:***
    - No need to modify other iterator types.
  - ***Cons:***
    - Complicated by `net.ipv4.udp_child_hash_entries`, `net.ipv4.tcp_child_ehash_entries`, etc. since each net namespace may contain its own standalone socket hash. Not sufficient to iterate over the global hash.
- **Support `bpf_sock_destroy` from `BPF_MAP_TYPE_SK_STORAGE` map iterator.**
  - ***Summary:*** Modify iterator so that it locks the socket, making it safe to use `bpf_sock_destroy` or other kfuncs which require a locked socket.
  - ***Pros:***
    - Less iteration?
    - Cilium will soon use `BPF_MAP_TYPE_SK_STORAGE` anyway to store metadata related to connected sockets, so it would be convenient.
  - ***Cons:***
    - Tricky to implement. Not sure if possible without having a reference count in bpf_local_storage_elem or making a copy of sdata inside RCU.

Implementation Challenges

- `BPF_MAP_TYPE_SK_STORAGE` iteration and lock_sock.
  - start(), show(), stop() takes place inside an RCU read-side critical section, so can't just lock_sock directly on each sk.
- Add refcount_t to bpf_local_storage_elem?
  - Use reference count to prevent elem from being freed outside of RCU critical section.
  - Iterator takes a reference inside RCU critical section, and lock_sock outside.
  - **Con:** Adds 8 byte refcount_t to map elements and more complexity in local storage logic for small use case.
- Another way: Make a copy of sdata?
  - Each time we resume iteration after seq_show we need to seek to the last position
  - Seeking to last position based on offset can be unreliable if the bucket state changes under our feet.

Things To Explore

Multi-namespace iteration

ICMP thing - ICMP unreachable for UDP socks

Sock pointer hash