Networking resource control with per-cgroup LSM

Presenter: Stanislav Fomichev, Google

Contributors: Mahesh Bandewar, YiFei Zhu, Wat Lim

Linux Plumbers Conference, 2022

What is "networking resource control"?

- Multiple workloads (containers) on the same machine
- Might have different policies and priorities
- Orthogonal to networking namespaces, the tasks might or might not run in a netns depending on the environment settings
  - networking namespacing is about isolating networking environment
  - cgroup is about controlling what the task can do in this environment
- For each container, we'd like to have:
  - cgroup_id or cgroup_id-like unique identifier - something to get to container policy from skb
  - enforce different socket options and set defaults
  - enforce some other networking syscalls (socket(), bind())

Where is networking policy coming from?

- Upon task startup, container management system populates the policy for the task
    - this policy is stored in BPF cgroup local storage
- Some policy should be applied by default
    - unconditionally set socket's priority upon socket creation
- Some policy can be optionally exercised by the task itself
    - accomplished via setsockopt() calls from the task

What exactly do we want to control?

- SO_PRIORITY - carry metadata to uniquely identify the container, which means:
    - SO_PRIORITY prohibited to be set directly by the tasks
    - SO_PRIORITY has to be set by the kernel (bpf)
    - long-term: converge on cgroup_id, still depend on it due to legacy HTB assumptions everywhere
- IP_TOS - per-container list for which TOS values it might use
- List of ports which containers might bind to / listen on (mostly from historic borg requirements)
- Prohibit IPv4
- Control permission for Google-only socket options

Note, sandboxing (doing netns unshare in this cgroup) should ignore most of the above

How it has been done historically?

- Custom networking cgroup, similar to upstream net_cls / net_prio in the kernel
- Tried to upstream long time ago, but at that point net_cls / net_prio were already in place and were largely doing the same things

What's wrong with custom cgroup?

- Constant source of pain:
  - Rebases breaking it
  - Upstream breaking it (we, somewhat unconventional, also run BPF on top of it)
- Want to be closer to upstream:
  - There is really no secret sauce in here
  - Doing similar resource control might be useful for others
  - Still v1 based which is deprecated and doesn't get any new BPF features

So what are we trying to do?

- Get rid of custom kernel patches
- Redo existing functionality with BPF
- Not widely deployed, but the experimental data is promising
- Next slides show some examples of the functionality

IP_TOS/IPV6_TCLASS

- Have a fixed set of supported TOS values in cgroup local storage
- When task bind()'s or calls setsockopt(..., IP_TOS, ...) - compare the value against the list

IP_TOS

```
__section("cgroup/setsockopt") int _setsockopt(struct bpf_sockopt *ctx)
{
    struct *cg = bpf_get_local_storage(...);
    if (ctx->level == IPPROTO_IP && ctx->optname == IP_TOS)
        return valid_tos_range(ctx, cg); // simple range checks
}
```

Limit bind ports

- Essentially the same idea as in IP_TOS, but applied at bind hooks
- Only about lower 16k ports, can't affect the ones selected by autobind

IPv4 "hiding"

- We used to do real hiding where cgroup knob would completely hide IPv4 addresses on the interfaces (via proc/netlink/etc)
  - A lot of things prever v4 address as soon at something v4-related shows up in the environment
- socket(AF_INET) would return -EAFNOSUPPORT
- Can't do all of that with BPF, doing only socket(AF_INET) part
- Originally in BPF were returning -EPERM, but some runtimes aren't happy, from JRE:

```
if ((sock = socket(proto, SOCK_DGRAM, 0)) < 0) {
    // If we lack support for this address family or protocol,
    // don't throw an exception.
    if (errno != EPROTONOSUPPORT && errno != EAFNOSUPPORT) {
```

IPv4 "hiding"

```
__section("cgroup/sock") int _sock(struct bpf_sockopt *ctx)
{
    struct *cg = bpf_get_local_storage(...);
    if (ctx->family == AF_INET &&  !(cg->permissions & PERMITTED_AF_INET)) {
        bpf_set_retval(-EAFNOSUPPORT);
        return -1;
    }
}
```

SO_PRIORITY (naive)

- Set default socket priority upon creation
- Seems to be super easy with BPF_PROG_TYPE_CGROUP_SOCK which triggers upon socket creation
- The devil is in the details
  - Doesn't trigger for passive open
  - Doesn't trigger for non-INET families (AF_PACKET)

SO_PRIORITY (naive)

```
__section("cgroup/sock") int _sock(struct bpf_sock *ctx)
{
    struct *cg = bpf_get_local_storage(...);
    // Not enough to catch every socket :-(
    ctx->priority = cg->priority;
}
```

How does per-cgroup LSM fit into the picture?

- So far we were able to leverage existing networking hooks
- However, SO_PRIORITY program doesn't work 100%
- BPF_PROG_TYPE_CGROUP_SOCK triggers only for AF_INET/AF_INET6
- BPF_PROG_TYPE_CGROUP_SOCK triggers only for "active" sockets
- What do we do?
    - Add more hooks? :-(

Per-cgroup LSM

- Same as regular BPF LSM, but can be attached to a particular cgroup
- Behind the scenes creates fentry-like trampoline that demuxes into cgroup
  - Need to provide extra attach_btf_id to indicate LSM hook
- bpf_getsockopt helper to mutate socket state
- See tools/testing/selftests/bpf/prog_tests/lsm_cgroup.c for more examples
- Addresses our issue with existing BPF_CGROUP_INET_SOCK_CREATE not triggering where we want it to trigger

SO_PRIORITY

- Can leverage several existing LSM hooks to initialize default socket state:
- lsm_cgroup/socket_post_create - after active socket has been allocated
- lsm_cgroup/inet_csk_clone - after passive socket has been allocated

SO_PRIORITY

```
SEC("lsm_cgroup/inet_csk_clone")
int BPF_PROG(socket_clone, struct sock *newsk, const struct request_sock *req)
{
    bpf_setsockopt(newsk, SOL_SOCKET, SO_PRIORITY, &prio, sizeof(prio));
}
// same for "lsm_cgroup/socket_post_create"
```

NET_RAW_XMIT

- TX-only version of NET_RAW capability
- For prober containers, we'd like to be able to send out raw packets only
  - both PF_INET6/SOCK_RAW and PF_PACKET
- Want to protect (in init-netns) other containers from sniffing NET_RAW tenant
- lsm_cgroup/socket_bind
  - prohibit rebinding
- lsm_cgroup/socket_post_create
  - probit with protocol == 0 (aka ETH_P_ALL)

NET_RAW_XMIT

```
SEC("lsm_cgroup/socket_post_create")
int BPF_PROG(...)
{
    if (family == AF_PACKET && protocol != 0)
        return 0; /* EPERM */
}
```

NET_RAW_XMIT

```
SEC("lsm_cgroup/socket_bind")
int BPF_PROG(..., struct sockaddr *address, ...)
{
    struct sockaddr_ll sa = {};
    if (sock->sk->__sk_common.skc_family != AF_PACKET) return 1;
    bpf_probe_read_kernel(&sa, sizeof(sa), address);
    if (sa.sll_protocol) return 0; /* EPERM */
}
```

Challenges

- Unprivileged users/readers (up until recently, everything requires CAP_BPF)
- CAP_BPF doesn't work with user namespaces
- No way to create unprivileged containers
- Hierarchical properties have to be handled manually (programs need some way to communicate who's been called and what has been handled)
- Userspace expecting specific errno
- sendmsg cmsg options are not enforced

Summary

- We were able to cover 95% of existing custom cgroup with BPF
- Still in the experimental phase with promising results running this on some % of the fleet
- Some of the kernel BPF features we had to add to support our use-cases:
    - getsockopt & setsockopt hooks
        - 0d01da6afc54 - bpf: implement getsockopt and setsockopt hooks
    - global mode for cgroup storage map
        - 7d9c3427894f - bpf: Make cgroup storages shared between programs on the same cgroup
    - bpf_get_retval / bpf_set_retval
        - b44123b4a3dc - bpf: Add cgroup helpers bpf_{get,set}_retval to get/set syscall return value
    - lsm_cgroup
        - 69fd337a975c - bpf: per-cgroup lsm flavor
    - rebinding to privileged ports
        - 772412176fb9 - bpf: Allow rewriting to ports under ip_unprivileged_port_start

Questions? Suggestions?