Trusted unprivileged BPF

Andrii Nakryiko
Software Engineer

Meta

Current state

- Root or root-like capabilities are required:
  - CAP_BPF + {CAP_PERFMON, CAP_NET_ADMIN}
  - CAP_SYS_ADMIN
- Coarse-grained, broad, and permissive
  - You can do **way more** with those CAPs than just BPF
- Vanilla unprivileged BPF is *dangerous* and *impractical*

Problem: capable(CAP_BPF)

- bpf() expects CAP_{BPF, PERFMON, NET_ADMIN} in **init namespace**
- **Incompatible with user namespaces**
- FAQ: Can we *just* **namespace** CAP_BPF?
  - It’s just capable(...) -> ns_capable(...), right?
  - A: **No.**
    - BPF programs can’t be prevented from peeking at everything in the kernel
    - bpf_probe_read_kernel() + bpf_probe_read_user() = **no sandboxing is possible**
    - System-wide hooks and observability

Solution: **1st attempt**

- /dev/bpf proposal by Song ([0])
  - Get fd by opening /dev/bpf
  - ioctl(fd, BPF_DEV_IOCTL_ENABLE_SYS_BPF)
  - Set persistent current->bpf_permitted bit
  - bpf() syscall takes current->bpf_permitted into account
- Rejected by upstream
- *Eventually we ended up with current CAP_{BPF, PERFMON, NET_ADMIN}*

[0] https://lore.kernel.org/bpf/20190627201923.2589391-2-songliubraving@fb.com/

Solution: **2nd attempt**

- Authoritative LSM approach ([1])
  - New LSM hooks for map, prog, BTF creation
  - Reject, grant, pass through operations
  - Would pair nicely with BPF LSM:
    - BPF LSM policy determines application **trustworthiness**
    - BPF subsystem access is **granted**, **rejected**, or **delegated** to kernel
- Rejected by upstream

[1] https://lore.kernel.org/bpf/20230412043300.360803-1-andrii@kernel.org/

Solution: **3rd time’s a charm?**

- Take good ideas from /dev/bpf and fix bad parts
    - FD as a proof of access grant is good
    - ioctl() and struct task_struct global bits are bad
    - device file is suboptimal and error-prone
- Augment with restrictive LSM for dynamic and fine-granular policy

Solution: 3rd time's a charm?

BPF token

- New bpf() syscall command: **BPF_TOKEN_CREATE**
  - Returns FD representing access token
  - (?) Needs capable(CAP_SYS_ADMIN)
- BPF_PROG_LOAD accepts optional **token_fd** attribute
  - If valid, allows to proceed
  - If missing, usual capable(...) checks
  - Same for others: BPF_*_GET_FD_BY_ID, BPF_MAP_CREATE, etc

Solution: 3rd time's a charm?

BPF token: transfer

- BPF token has to **originate from privileged process**
- … and then is transferred to **trusted** unprivileged one(s):
  - Unix domain sockets and SCM_RIGHTS
  - Or use **BPF FS pinning**, like any other BPF kernel object!
    - **Privileged:** BPF_OBJ_PIN -> /sys/fs/bpf/<token-path>
      - chmod, chown, etc
    - **Unprivileged:** BPF_OBJ_GET /sys/fs/bpf/<token-path> → **token_fd**
- BPF LSM for dynamic and fine-grained control, if necessary

Solution: 3rd time's a charm?

BPF token: practical aspects

- Extensible with `union bpf_attr` approach
  - Initially all-or-nothing (and thus CAP_SYS_ADMIN to create)
  - Adjust BPF verifier limits (e.g., max insns limit)
  - Limit types of progs, maps, helpers, etc?
- Custom user context to identify use cases
  - (?) Up to opaque 64KB per BPF token (BPF cookie on steroids)
  - To be accessed by BPF LSM hooks (per use-case policy config)
- Not a singleton: BPF **token per use case**

Ecosystem support for BPF token

- Standard location of BPF token within container
    - (?) **/sys/fs/bpf/.token**
    - libbpf/BPF loaders, bpftrace, bpftool, etc. to do BPF_OBJ_GET(/sys/fs/bpf/.token)
    - BPF apps **automatically** will work with BPF as unprivileged
- Systemd (and container managers) support
    - Create BPF token from init namespace
    - Mount BPF FS inside container
    - Pin /sys/fs/bpf/.token

Thank you!