

BPF-RBACD: DELEGATING BPF PERMISSIONS WITH HIGH GRANULARITY

Toke Høiland-Jørgensen - Red Hat
Daniel Mellado - Red Hat

AGENDA

- Problem statement
- Background: kernel BPF access control and delegation
- bpf-rbacd and policy design
- Demonstration
- Pain points in the kernel APIs

PROBLEM STATEMENT

On regular distros, BPF is an all-or-nothing root-only feature.

We want to enable:

- Delegating BPF permissions to non-root entities
- **Granular** control over BPF features:
 - For each delegation
 - System-wide

MOTIVATING EXAMPLES

None of the following is possible today on Fedora/RHEL out of the box:

MOTIVATING EXAMPLES

None of the following is possible today on Fedora/RHEL out of the box:

1. Run an application without root, allowing it to use a subset of BPF features.

MOTIVATING EXAMPLES

None of the following is possible today on Fedora/RHEL out of the box:

1. Run an application without root, allowing it to use a subset of BPF features.
2. Administratively disable certain BPF features (e.g., XDP) on a system.

MOTIVATING EXAMPLES

None of the following is possible today on Fedora/RHEL out of the box:

1. Run an application without root, allowing it to use a subset of BPF features.
2. Administratively disable certain BPF features (e.g., XDP) on a system.
3. Provide access to (e.g.) libbpf-tools to a user account without sudo access.

BACKGROUND: KERNEL BPF ACCESS CONTROLS

PRIVILEGES REQUIRED FOR THE USE OF BPF FEATURES

Operation	System-wide (no users)	Inside a users
Inspecting objects etc.	CAP_SYS_ADMIN	Not possible
Loading programs	CAP_BPF ¹	NS CAP_BPF ¹ + bpf_token
Creating maps	CAP_BPF ²	NS CAP_BPF ² + bpf_token
Attaching programs	Valid file descriptor	Valid file descriptor
Updating maps	Valid file descriptor	Valid file descriptor

¹ (+ CAP_PERFMON/CAP_NET_ADMIN for some program types) ² (+ CAP_NET_ADMIN for some map types)

Implication: Delegating outside a users requires giving **system-wide** capabilities.
CAP_NET_ADMIN in particular has wide implications.

PRIVILEGES REQUIRED: PER BPF() COMMAND

Valid fd only

BPF_ITER_CREATE	BPF_MAP_LOOKUP_*	BPF_PROG_ATTACH
BPF_LINK_CREATE	BPF_MAP_UPDATE_*	BPF_PROG_DETACH
BPF_LINK_DETACH	BPF_MAP_DELETE_*	BPF_PROG_TEST_RUN
BPF_LINK_UPDATE	BPF_MAP_GET_NEXT_KEY	BPF_PROG_BIND_MAP
	BPF_MAP_FREEZE	BPF_PROG_STREAM_READ_BY_FD
BPF_OBJ_PIN		BPF_PROG_ASSOC_STRUCT_OPS
BPF_OBJ_GET		
BPF_OBJ_GET_INFO_BY_FD		BPF_RAW_TRACEPOINT_OPEN

Allowed with a suitable bpf_token

BPF_MAP_CREATE	BPF_BTF_LOAD	BPF_TOKEN_CREATE
BPF_PROG_LOAD	BPF_BTF_GET_FD_BY_ID	

Requires system CAP_SYS_ADMIN

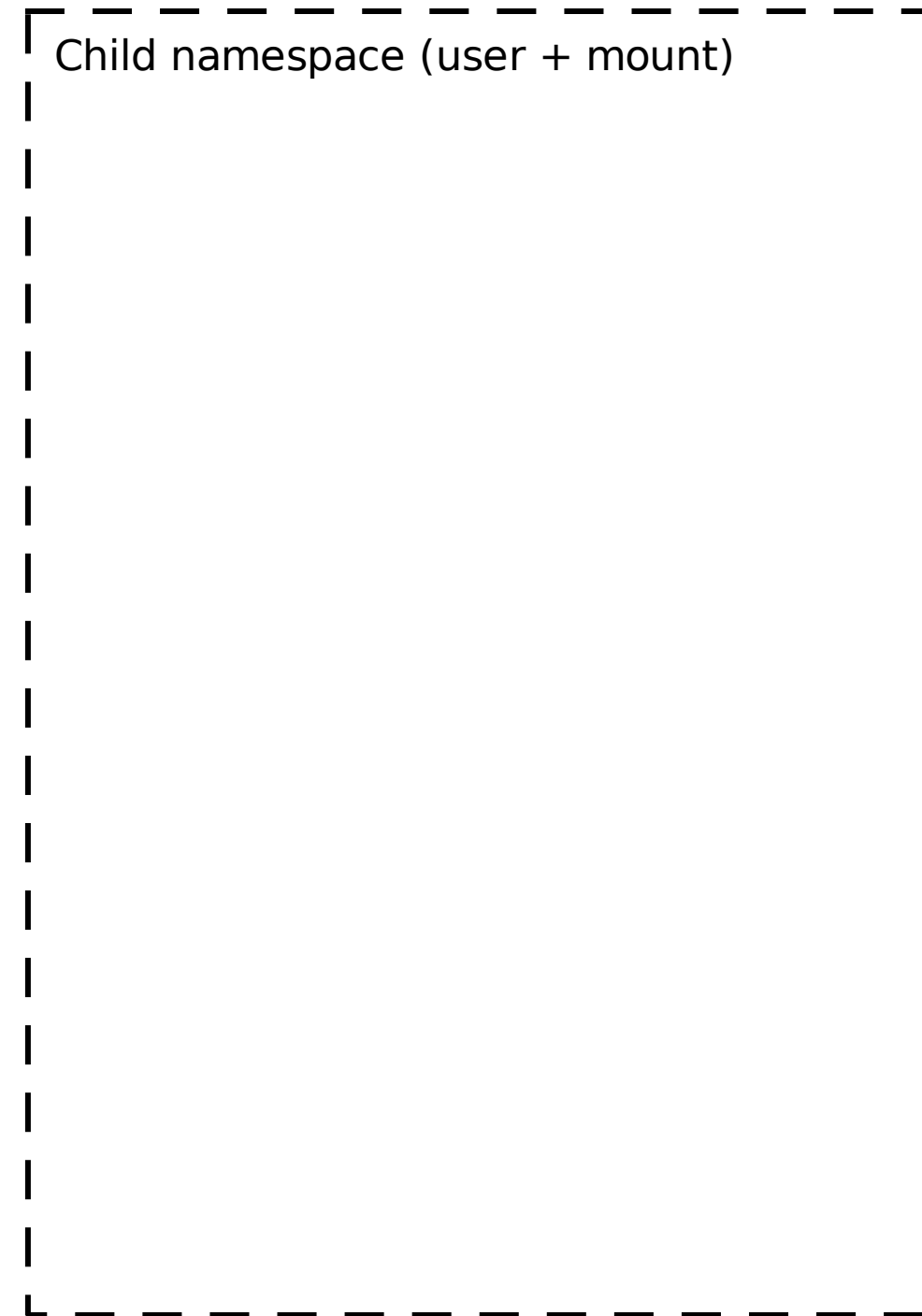
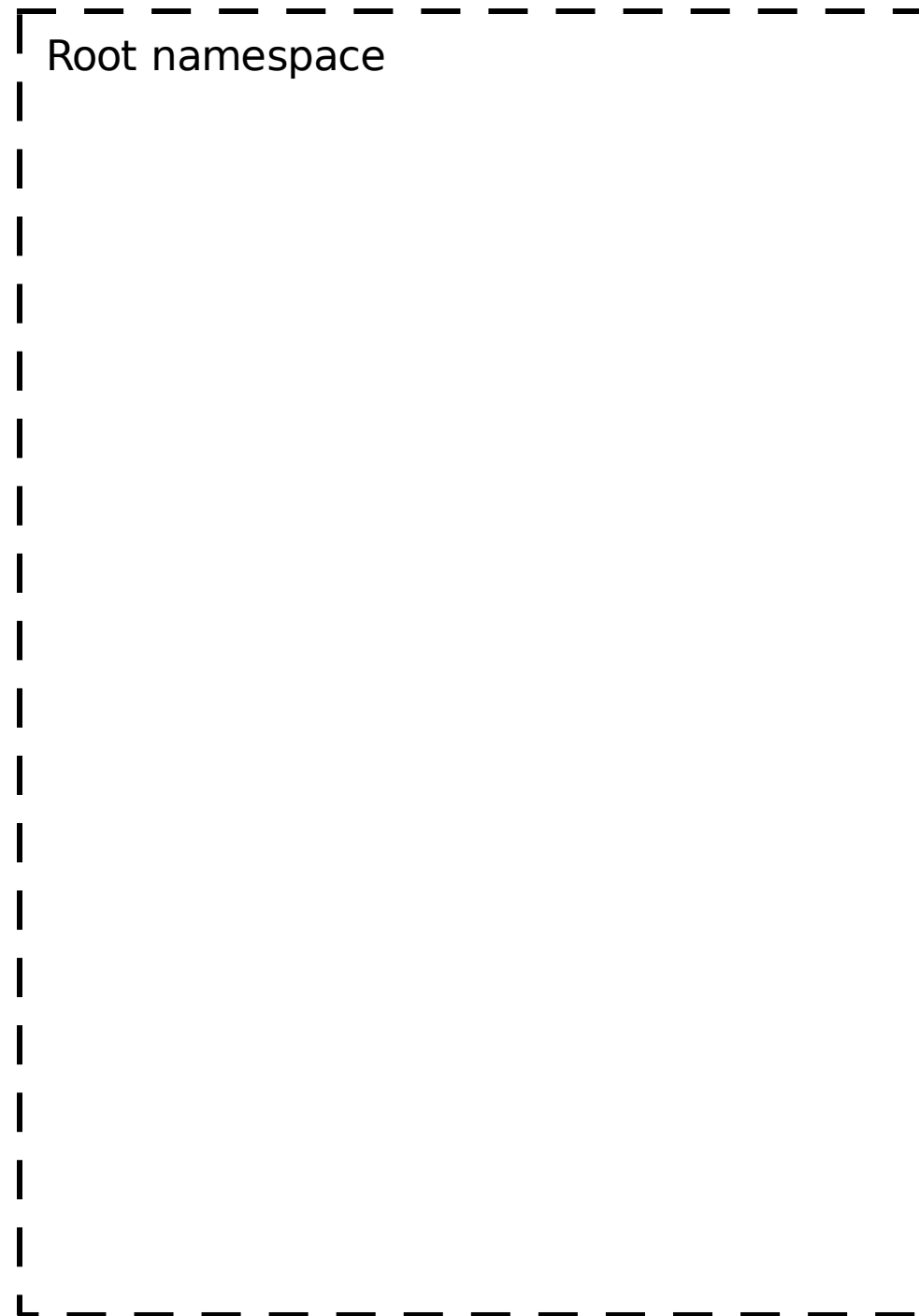
BPF_PROG_QUERY	BPF_PROG_GET_NEXT_ID	BPF_PROG_GET_FD_BY_ID
BPF_TASK_FD_QUERY	BPF_MAP_GET_NEXT_ID	BPF_MAP_GET_FD_BY_ID
BPF_ENABLE_STATS	BPF_LINK__GET_NEXT_ID	BPF_LINK_GET_FD_BY_ID
	BPF_BTF_GET_NEXT_ID	

THE BPF_TOKEN KERNEL FEATURE

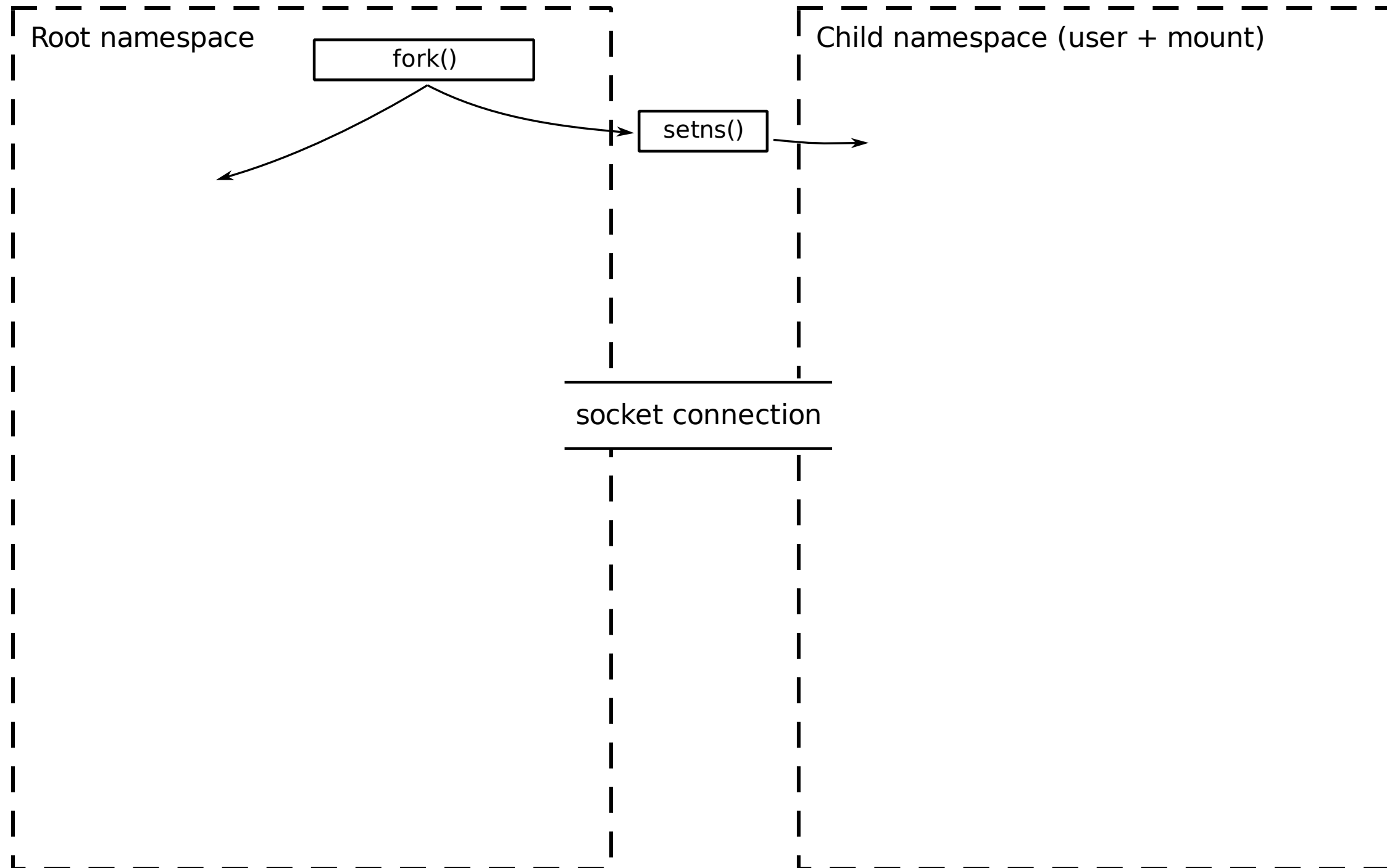
- A file descriptor obtained via `bpf ( BPF_TOKEN_CREATE )` from a bpffs
- Passed to subsequent `bpf ( )` calls
- Grants permissions inside its originating users
- Automatically used by libbpf if available

The bpffs must have the permission grants at create time.

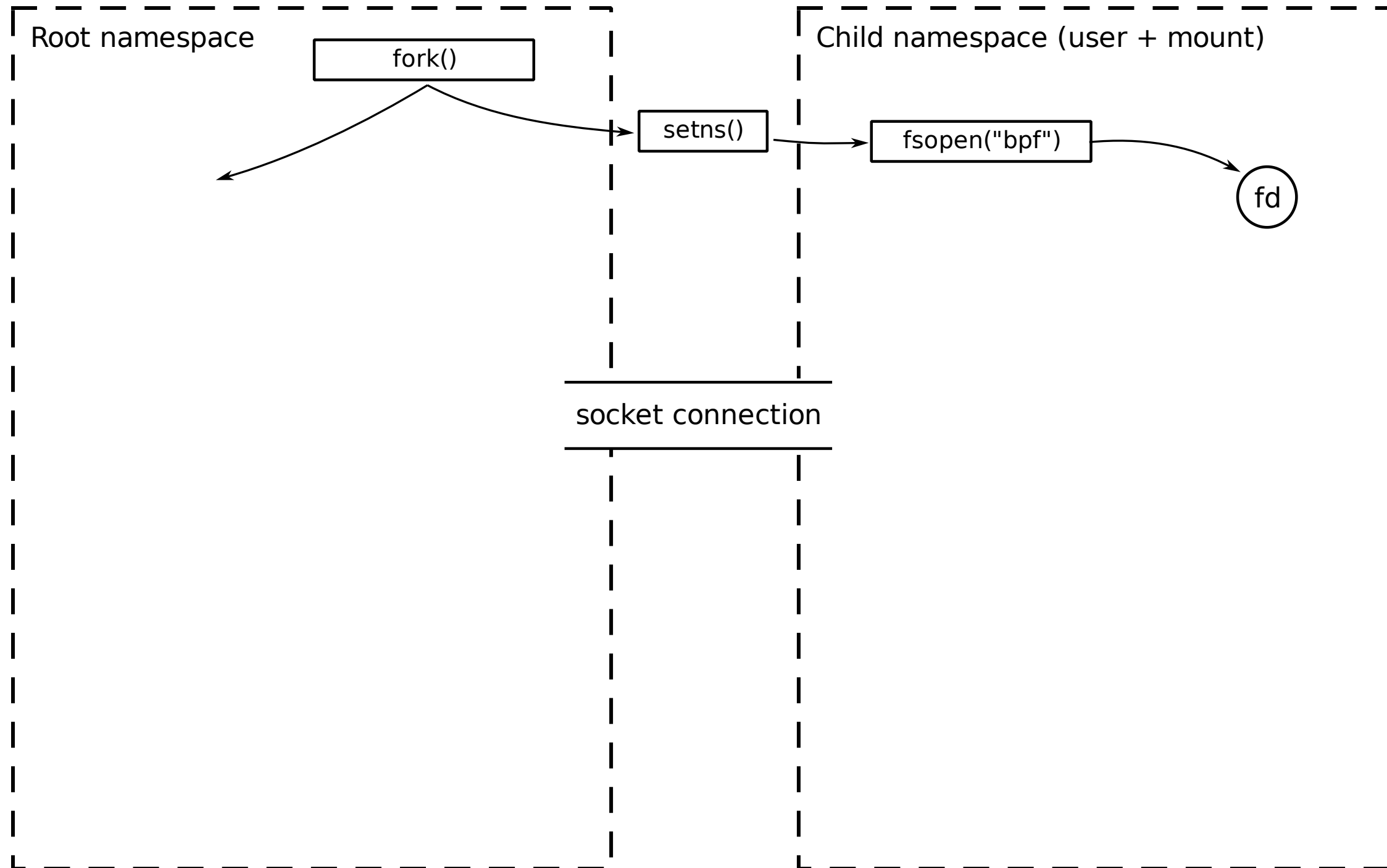
THE BPFFS PERMISSION DELEGATION DANCE



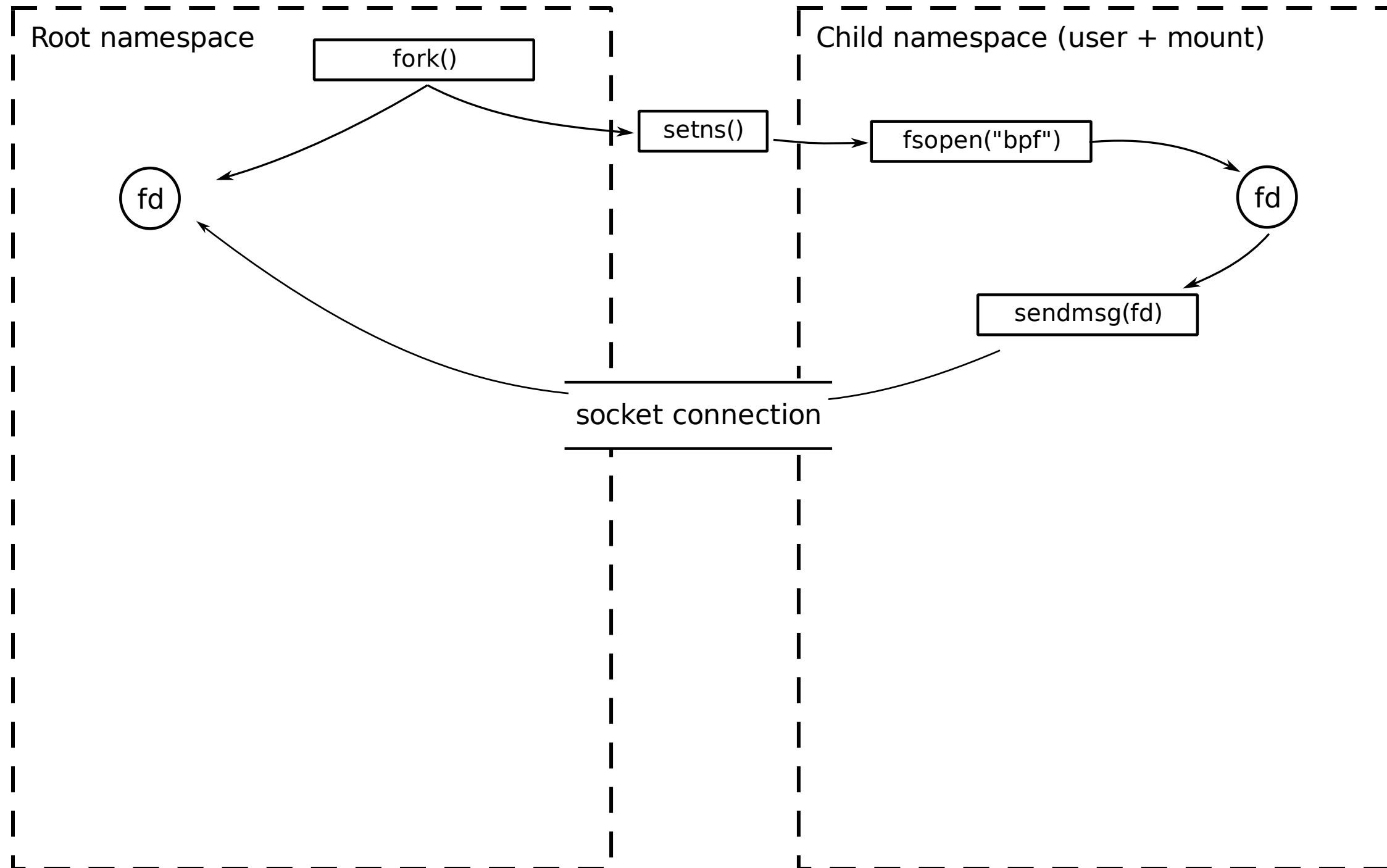
THE BPF FS PERMISSION DELEGATION DANCE



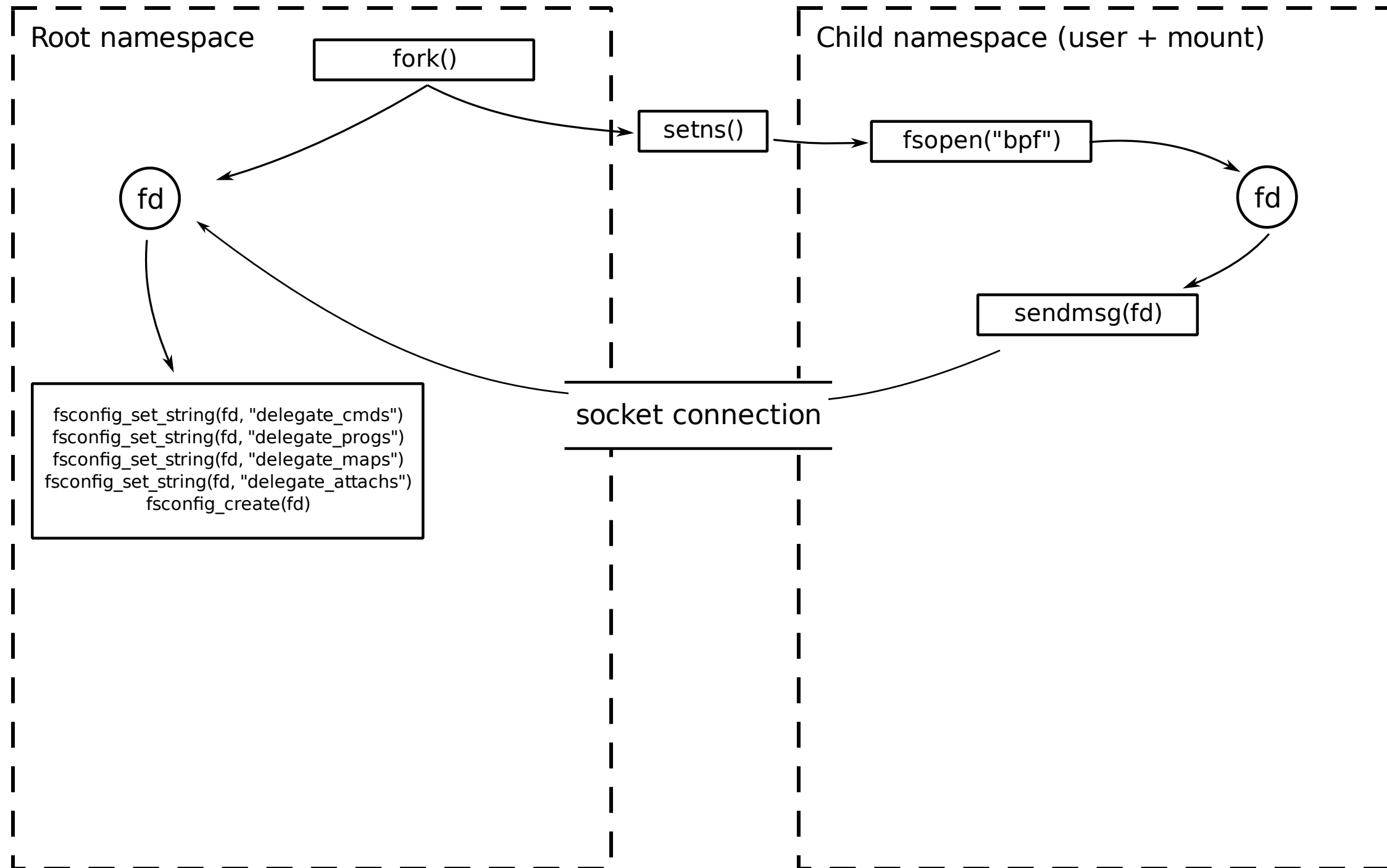
THE BPF FS PERMISSION DELEGATION DANCE



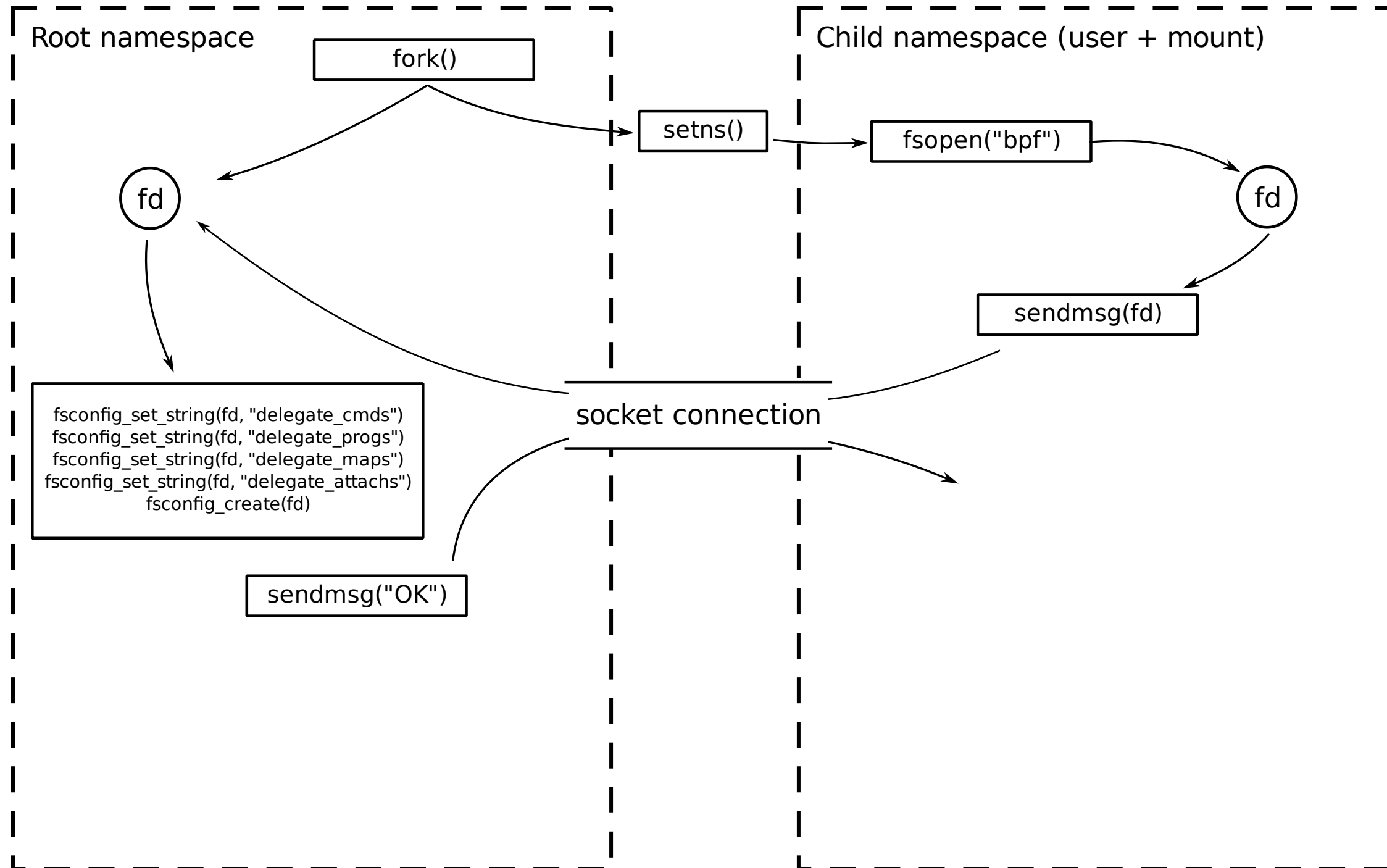
THE BPFES PERMISSION DELEGATION DANCE



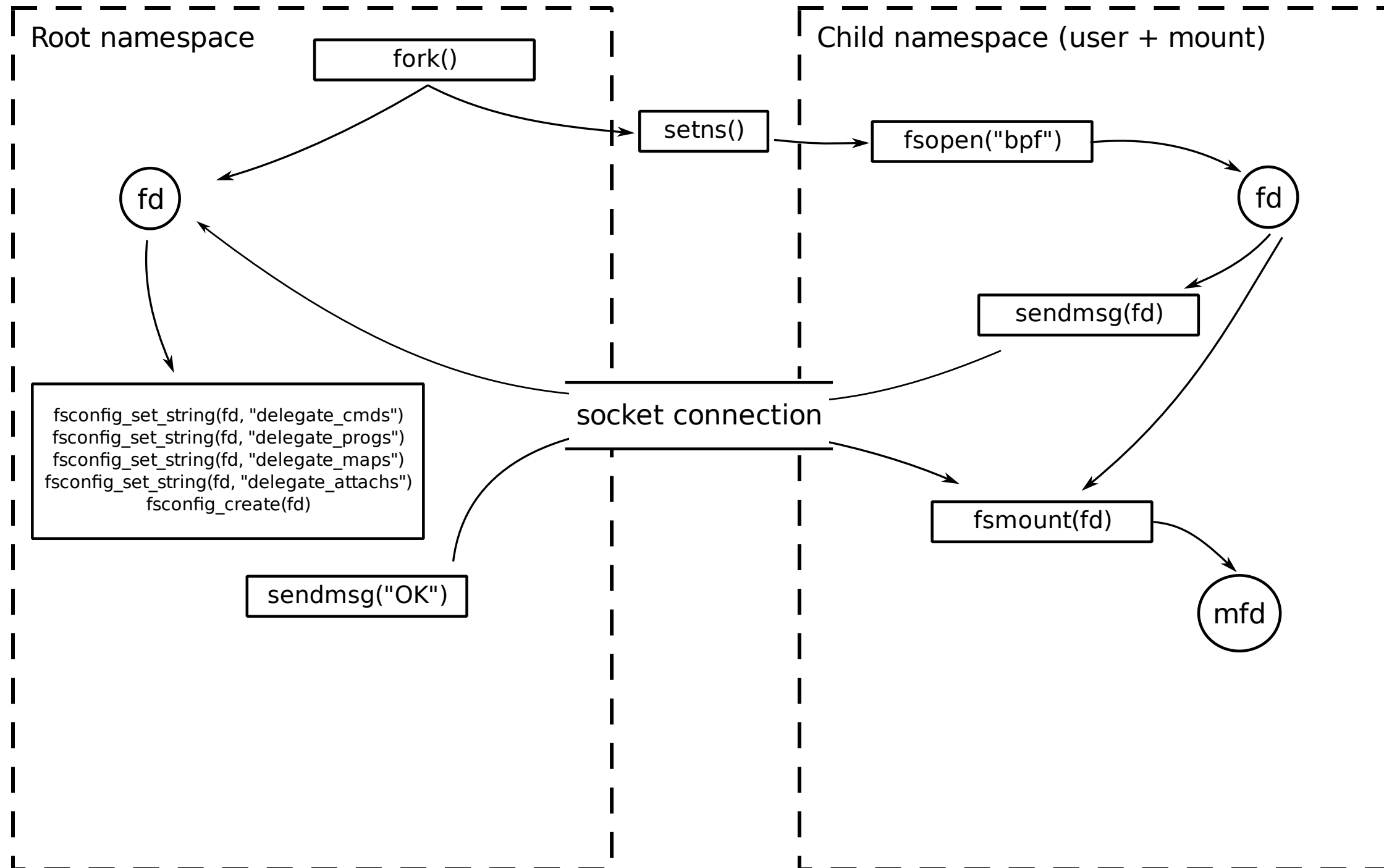
THE BPF FS PERMISSION DELEGATION DANCE



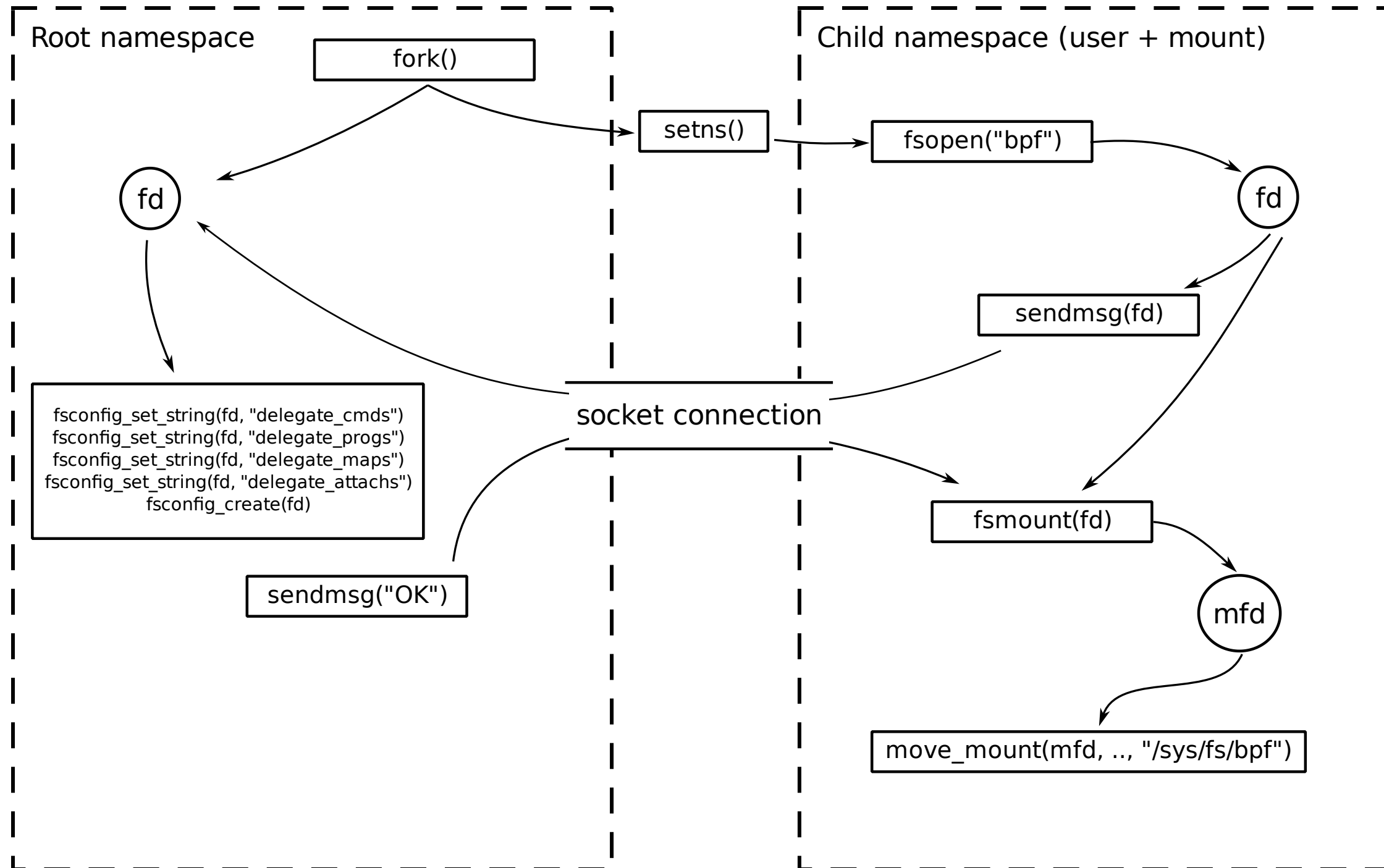
THE BPF FS PERMISSION DELEGATION DANCE



THE BPF FS PERMISSION DELEGATION DANCE



THE BPF FS PERMISSION DELEGATION DANCE



THE BPF-RBACD DESIGN

DESIGN COMPONENTS

The bpf-rbacd design consists of three components, in various stages of prototyping:

DESIGN COMPONENTS

The bpf-rbacd design consists of three components, in various stages of prototyping:

- The **main permission enforcement daemon**
 - Implements a Linux Security Module (LSM) in BPF to enforce policies
 - Exposes a varlink service to query and install new policies
 - Performs the bpffs delegation dance when installing a policy

DESIGN COMPONENTS

The bpf-rbacd design consists of three components, in various stages of prototyping:

- The **main permission enforcement daemon**
 - Implements a Linux Security Module (LSM) in BPF to enforce policies
 - Exposes a varlink service to query and install new policies
 - Performs the bpffs delegation dance when installing a policy
- A separate privilege granting daemon
 - Exposes an interface for unprivileged users to request privileges

DESIGN COMPONENTS

The bpf-rbacd design consists of three components, in various stages of prototyping:

- The **main permission enforcement daemon**
 - Implements a Linux Security Module (LSM) in BPF to enforce policies
 - Exposes a varlink service to query and install new policies
 - Performs the bpffs delegation dance when installing a policy
- A separate privilege granting daemon
 - Exposes an interface for unprivileged users to request privileges
- A Kubernetes operator
 - Grants permissions to containers in Kubernetes

LSM POLICY ENFORCEMENT HOOKS

We attach to the following LSM hook points:

<code>security_bpf()</code>	general BPF command hook
<code>security_bpf_map()</code>	map file descriptor creation
<code>security_bpf_map_create()</code>	map creation
<code>security_bpf_prog()</code>	prog file descriptor creation
<code>security_bpf_prog_load()</code>	prog load
<code>security_mmap_file()</code>	map fd access

LSM POLICY CHECKS

The policy lookup is done on **user namespace ID**. The policy controls:

- Allowed bpf () commands
- Allowed map types (with create/read/write sub-permissions)
- Allowed program types
- Allowed attachment types
- Allowed helpers and kfuncs
- Allowed attachment *points*
- Allow/deny cBPF usage

If the policy lookup fails, access is denied.

IDENTIFYING HELPER AND KFUNC CALLS

```
// called from security_bpf_prog_load()
static int walk_bpf_instructions(struct bpf_prog *prog)
{
    int insn_cnt = prog->len, i;
    bpf_for(i, 0, insn_cnt) {
        u16 call_type, btf_id;
        struct bpf_insn insn;
        u32 func_id;

        if (bpf_probe_read_kernel(&insn, sizeof(insn), &prog->insnsi[i]))
            continue;

        if (insn.code != (BPF_JMP | BPF_CALL))
            continue;

        if (insn.src_reg == 0) {
            call_type = FUNC_HELPER;
            btf_id = 0;
        } else {
            call_type = FUNC_KFUNC;
            btf_id = insn.off;
        }
        func_id = insn.imm;
        // save call_type, btf_id and func_id
    }
}
```

```
12:03:26.757639 delay-kfunc(235433) in 4026531837 (verdict -8): Prog load (name: delay_function type: BPF_PROG_TYPE_TRACING
attach: BPF_TRACE_FENTRY funcalls: 6 (Helper(BPF_FUNC_get_current_pid_tgid) Helper(BPF_FUNC_ktime_get_boot_ns)
Helper(BPF_FUNC_loop) Helper(BPF_FUNC_ringbuf_reserve) Helper(BPF_FUNC_get_current_comm) Helper(BPF_FUNC_ringbuf_submit) ))
```

SYSTEM-WIDE POLICY

We define a **system policy**:

- This is the *bounding set* of all allowed BPF operations on the system.
- Installed as the active policy for the root namespace at LSM startup
- Policies for other namespaces are constrained to the contents of the system policy:
 - `effective_policy = namespace_policy & system_policy`

EVENT LOGGING

- The LSM emits an event to userspace on every hook operation.
- This allows for a global view of all BPF-related activity on a system

DEMO TIME!

STARTING THE LSM DAEMON

```
$ sudo ../target/debug/bpf-rbacd-lsm
Server: Listening on unix:/run/bpf-policy/org.example.bpf.policy
18:37:48.051274 bpf-rbacd-lsm(205273) in 4026531837 (verdict 0): Syscall (BPF_LINK_CREATE)
18:37:48.051389 bpf-rbacd-lsm(205273) in 4026531837 (verdict 0): Syscall (BPF_LINK_CREATE)
18:37:48.051403 bpf-rbacd-lsm(205273) in 4026531837 (verdict 0): Syscall (BPF_LINK_CREATE)
18:37:48.051413 bpf-rbacd-lsm(205273) in 4026531837 (verdict 0): Syscall (BPF_LINK_CREATE)
18:37:48.051423 bpf-rbacd-lsm(205273) in 4026531837 (verdict 0): Syscall (BPF_LINK_CREATE)
18:37:48.051434 bpf-rbacd-lsm(205273) in 4026531837 (verdict 0): Syscall (BPF_OBJ_GET_INFO_BY_FD)
18:37:48.051446 bpf-rbacd-lsm(205273) in 4026531837 (verdict 0): Map MMAP (id: 54 name: events type: BPF_MAP_TYPE_RINGBUF mode: rw)
18:37:48.051462 bpf-rbacd-lsm(205273) in 4026531837 (verdict 0): Map MMAP (id: 54 name: events type: BPF_MAP_TYPE_RINGBUF mode: r)
```

OUTPUT FROM RUNNING BPFTOOL

```
# bpftool prog list (in different shell)
18:37:59.719712 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_PROG_LOAD)
18:37:59.719792 bpftool(205367) in 4026531837 (verdict -1): Prog load (name: type: BPF_PROG_TYPE_SOCKET_FILTER attach: BPF_CGROUP_INET_INGRE
18:37:59.719825 bpftool(205367) in 4026531837 (verdict -1): Prog FD access (id: 173 name: type: BPF_PROG_TYPE_SOCKET_FILTER)
18:37:59.719841 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_TOKEN_CREATE)
18:37:59.719855 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_PROG_LOAD)
18:37:59.719867 bpftool(205367) in 4026531837 (verdict -8): Prog load (name: type: BPF_PROG_TYPE_SOCKET_FILTER attach: BPF_CGROUP_INET_INGRE
18:37:59.719895 bpftool(205367) in 4026531837 (verdict -1): Prog FD access (id: 174 name: type: BPF_PROG_TYPE_SOCKET_FILTER)
18:37:59.719911 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_PROG_LOAD)
18:37:59.719925 bpftool(205367) in 4026531837 (verdict -1): Prog load (name: type: BPF_PROG_TYPE_SOCKET_FILTER attach: BPF_CGROUP_INET_INGRE
18:37:59.719942 bpftool(205367) in 4026531837 (verdict -1): Prog FD access (id: 175 name: type: BPF_PROG_TYPE_SOCKET_FILTER)
18:38:00.097578 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_PROG_LOAD)
18:38:00.097756 bpftool(205367) in 4026531837 (verdict -1): Prog load (name: libbpf_nametest type: BPF_PROG_TYPE_SOCKET_FILTER attach: BPF_CG
18:38:00.097785 bpftool(205367) in 4026531837 (verdict -1): Prog FD access (id: 176 name: libbpf_nametest type: BPF_PROG_TYPE_SOCKET_FILTER)
18:38:00.097791 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_MAP_CREATE)
18:38:00.097796 bpftool(205367) in 4026531837 (verdict -1): Map create (name: libbpf_mmap type: BPF_MAP_TYPE_ARRAY)
18:38:00.097803 bpftool(205367) in 4026531837 (verdict -1): Map FD access (id: 61 name: libbpf_mmap type: BPF_MAP_TYPE_ARRAY mode: rw)
18:38:00.115478 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_BTF_LOAD)
18:38:00.115529 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_BTF_LOAD)
18:38:00.115535 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_BTF_LOAD)
18:38:00.115540 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_PROG_LOAD)
18:38:00.115545 bpftool(205367) in 4026531837 (verdict -8): Prog load (name: det_arg_ctx type: BPF_PROG_TYPE_KPROBE attach: BPF_CGROUP_INET_I
18:38:00.115568 bpftool(205367) in 4026531837 (verdict -1): Prog FD access (id: 177 name: det_arg_ctx type: BPF_PROG_TYPE_KPROBE)
18:38:00.115574 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_BTF_LOAD)
18:38:00.115579 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_BTF_LOAD)
18:38:00.115584 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_BTF_LOAD)
18:38:00.115588 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_BTF_LOAD)
18:38:00.115593 bpftool(205367) in 4026531837 (verdict -1): Syscall (BPF_BTF_LOAD)
... etc
```

POLICY INSPECTION

```
$ sudo varlinkctl call unix:/run/bpf-policy/org.example.bpf.policy org.example.bpf.policy.GetPolicy '{"namespace_id": 4026531837}'
{
  "policy" : {
    "allowed_attach_types" : [],
    "allowed_commands" : [
      "BPF_LINK_CREATE",
      "BPF_OBJ_GET_INFO_BY_FD"
    ],
    "allowed_helpers" : [],
    "allowed_map_types" : {
      "BPF_MAP_TYPE_RINGBUF" : {
        "create" : true,
        "read" : true,
        "write" : true
      }
    },
    "allowed_prog_types" : []
  }
}
```

RUNNING IN A NEW USER NAMESPACE

```
$ cd ~/git/bpf-examples/delay-kfunc
$ unshare -Um -r

# ls -l /proc/self/ns/{user,mnt}
0 lrwxrwxrwx 1 root root 0 Oct  4 18:45 /proc/self/ns/mnt -> mnt:[4026533083]
0 lrwxrwxrwx 1 root root 0 Oct  4 18:45 /proc/self/ns/user -> user:[4026533068]

# mount | grep bpf
bpf on /sys/fs/bpf type bpf (rw,nosuid,nodev,noexec,relatime,mode=700)

# ./delay-kfunc xsk_mmap 1000
libbpf: Failed to bump RLIMIT_MEMLOCK (err = -EPERM), you might need to do it explicitly!
libbpf: Error in bpf_object__probe_loading(): -EPERM. Couldn't load trivial BPF program. Make sure your kernel supports BPF (CONFIG_BPF_SYSCALL)
libbpf: failed to load BPF skeleton 'delay_kfunc_bpf': -EPERM
```

A POLICY FOR THE DELAY-KFUNC APPLICATION

```
$ cat policy.json
{
  "allowed_attach_types" : ["BPF_CGROUP_INET_INGRESS", "BPF_TRACE_FENTRY"],
  "allowed_commands" : [
    "BPF_BTFF_GET_FD_BY_ID",
    "BPF_BTFF_GET_NEXT_ID",
    "BPF_BTFF_LOAD",
    "BPF_LINK_CREATE",
    "BPF_MAP_CREATE",
    "BPF_MAP_GET_NEXT_ID",
    "BPF_PROG_LOAD",
    "BPF_PROG_TEST_RUN",
    "BPF_OBJ_GET_INFO_BY_FD",
    "BPF_TOKEN_CREATE"
  ],
  "allowed_helpers" : [
    "get_current_comm",
    "get_current_pid_tgid",
    "ktime_get_boot_ns",
    "loop",
    "ringbuf_reserve",
    "ringbuf_submit"
  ],
  "allowed_map_types" : {
    "BPF_MAP_TYPE_ARRAY" : { "create" : true, "read" : true, "write" : true },
    "BPF_MAP_TYPE_RINGBUF" : { "create" : true, "read" : true, "write" : true }
  },
  "allowed_prog_types" : [
    "BPF_PROG_TYPE_TRACEPOINT",
    "BPF_PROG_TYPE_TRACING",
    "BPF_PROG_TYPE_SOCKET_FILTER",
    "BPF_PROG_TYPE_SYSCALL"
  ]
}
```

INSTALLING THE NEW POLICY

```
( in root namespace )
$ sudo varlinkctl call unix:/run/bpf-policy/org.example.bpf.policy org.example.bpf.policy.AddPolicy \
    "{\"namespace_id\": 4026533068, \"policy\": $(< policy.json)}"

( inside namespace )
# mount | grep bpf
bpf on /sys/fs/bpf type bpf (rw,nosuid,nodev,noexec,relatime,mode=700)
none on /sys/fs/bpf type bpf (rw,relatime,delegate_cmds=map_create:prog_load:prog_test_run:map_get_next_id:obj_get_info_by_fd:btfl_load:btfl_ge

( works now! )
# ./delay-kfunc xsk_mmap 1000
libbpf: Failed to bump RLIMIT_MEMLOCK (err = -EPERM), you might need to do it explicitly!
Loop calibration: average loop duration 22835 ns / 10000 iterations - looping 44 * 10000 times to hit target
Delay function attached to xsk_mmap(). Press Ctrl-C to interrupt...
^C

( but not on modules )
# ./delay-kfunc veth_dellink 1000
Could not attach to function 'veth_dellink': No such process
```

OUTPUT FROM LSM DAEMON FOR POLICY INSTALL

Got add_policy call for namespace 4026533068

```
19:03:10.122513 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_TOKEN_CREATE)
19:03:10.122591 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_PROG_LOAD)
19:03:10.122609 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_PROG_LOAD)
19:03:10.122624 delay-kfunc(211185) in 4026533068 (verdict 0): Prog load (name: type: BPF_PROG_TYPE_SOCKET_FILTER attach: BPF_CGROUP_INET_IN
19:03:10.122670 delay-kfunc(211185) in 4026533068 (verdict 0): Prog FD access (id: 253 name: type: BPF_PROG_TYPE_SOCKET_FILTER)
19:03:10.122687 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_PROG_LOAD)
19:03:10.122702 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_MAP_CREATE)
19:03:10.122717 delay-kfunc(211185) in 4026533068 (verdict 0): Map create (name: type: BPF_MAP_TYPE_ARRAY)
19:03:10.122734 delay-kfunc(211185) in 4026533068 (verdict 0): Map FD access (id: 109 name: type: BPF_MAP_TYPE_ARRAY mode: rw)
19:03:10.122752 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122768 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122782 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122796 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_PROG_LOAD)
19:03:10.122809 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122822 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122838 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122854 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122868 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122882 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122896 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122909 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_BTFF_LOAD)
19:03:10.122923 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_PROG_LOAD)
19:03:10.122937 delay-kfunc(211185) in 4026533068 (verdict 0): Prog load (name: libbpf_nametest type: BPF_PROG_TYPE_SOCKET_FILTER attach: BPF
19:03:10.122956 delay-kfunc(211185) in 4026533068 (verdict 0): Prog FD access (id: 254 name: libbpf_nametest type: BPF_PROG_TYPE_SOCKET_FILTE
19:03:10.122974 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_MAP_CREATE)
19:03:10.122990 delay-kfunc(211185) in 4026533068 (verdict 0): Map create (name: type: BPF_MAP_TYPE_RINGBUF)
19:03:10.123007 delay-kfunc(211185) in 4026533068 (verdict 0): Map FD access (id: 110 name: type: BPF_MAP_TYPE_RINGBUF mode: rw)
19:03:10.123024 delay-kfunc(211185) in 4026533068 (verdict 0): Syscall (BPF_MAP_CREATE)
19:03:10.123041 delay-kfunc(211185) in 4026533068 (verdict 0): Map create (name: type: BPF_MAP_TYPE_ARRAY)
(.. etc)
```

PAIN POINTS FOR LSM AND BPF_TOKEN APIS

KERNEL BPF_TOKEN ISSUES

The `bpf_obj_get_next_id()` function is system `CAP_SYS_ADMIN` only.

KERNEL BPF_TOKEN ISSUES

The `bpf_obj_get_next_id()` function is system `CAP_SYS_ADMIN` only.

- This blocks `BPF_BTFF_GET_NEXT_ID`:
 - Prevents libbpf from accessing module BTF
 - Which in turn fails tracing attachment to module functions
 - Presumably also affects CO-RE(?)

KERNEL BPF_TOKEN ISSUES

The `bpf_obj_get_next_id()` function is system `CAP_SYS_ADMIN` only.

- This blocks `BPF_BTFF_GET_NEXT_ID`:
 - Prevents libbpf from accessing module BTF
 - Which in turn fails tracing attachment to module functions
 - Presumably also affects CO-RE(?)

Can we relax this? For BTF only? For all enumerations?

WIDER KERNEL PERMISSION AND LSM ISSUES

WIDER KERNEL PERMISSION AND LSM ISSUES

- The CAP_NET_ADMIN capability is very broad
 - Makes it hard to grant targeted BPF-only permissions
 - This is the blocker for doing something more granular outside usersns

WIDER KERNEL PERMISSION AND LSM ISSUES

- The CAP_NET_ADMIN capability is very broad
 - Makes it hard to grant targeted BPF-only permissions
 - This is the blocker for doing something more granular outside usersns
- The security_bpf() hook only has access to the uattr struct
 - File descriptors have not been resolved, so can't check prog/map types etc
 - Expose bpf_map_get() / bpf_prog_get() as kfuncs?

NON-IDEAL LIBBPF BEHAVIOUR

During init, libbpf will probe the kernel:

- Refuses to proceed if it can't load a socket filter and tracing program
- Complains about memlimit failures in user namespace

FUTURE PLANS

IMPLEMENT MISSING FEATURES

- **Learning mode** for generating policies during development
- Kubernetes operator (possibly in [bpfman](#))
- Privilege granting utility for unprivileged users
- Tracking of kfuncs and attach points in LSM policy
- Additional LSM hooks to control cBPF and netlink attachment
- An option to require signed programs in the policy
- Tracking user namespace teardown
- Transition the eBPF code from C to Rust (when CO-RE support lands)
- Interface stabilisation, test coverage, etc.

DEPLOYMENT PLANS

- Fedora proposal (for Fedora 46, not yet submitted):
 - <https://fedoraproject.org/wiki/Changes/UnprivilegedEBPFRBACDaemon>
- RHEL and OpenShift: TBD after that
- Other distros: we hope so! Come talk to me :)

REFERENCES

- Code repository:
 - <https://forge.fedoraproject.org/ebpf/bpf-rbacd/src/branch/lsm-dev>
- Fedora eBPF SIG:
 - <https://fedoraproject.org/wiki/SIGs/eBPF>
 - <https://matrix.to/#/%23ebpf:fedoraproject.org>
- Research paper exploring these ideas (by Bolaji Gbadamosi, my PhD student):
 - <https://dl.acm.org/doi/10.1145/3744200.3744781>
- The delay-kfunc application
 - <https://github.com/xdp-project/bpf-examples/tree/main/delay-kfunc>

BONUS SLIDES

ROLE-BASED PERMISSIONS FOR UNPRIVILEGED USERS

```
roles:
  # Tracing role
  ebpf:
    groups: [ebpf]
    policy:
      commands: [PROG_LOAD, MAP_CREATE, MAP_LOOKUP_ELEM, MAP_UPDATE_ELEM, LINK_CREATE, BTF_LOAD]
      prog_types: [kprobe, tracepoint]
      map_types:
        hash: [create, read, write]
        array: [create, read, write]
        ringbuf: [create, read]

  # Networking role
  ebpf-net:
    groups: [ebpf-net]
    policy:
      commands: [PROG_LOAD, MAP_CREATE, MAP_LOOKUP_ELEM, MAP_UPDATE_ELEM, LINK_CREATE, PROG_ATTACH, BTF_LOAD]
      prog_types: [xdp, sched_cls]
      map_types:
        hash: [create, read, write]
        devmap: [create, read, write]

  # Admin – unrestricted
  ebpf-admin:
    groups: [ebpf-admin, wheel]
    policy:
      commands: [any]
      prog_types: [any]
      map_types: {any: [any]}
```

SYSTEMD BPF_TOKEN SUPPORT

```
[Unit]
Description=A bpf_token test unit

[Service]
Type=oneshot
ExecStart=/usr/bin/bpftool token

# Enable private user and mount namespaces, and make sure the user
# namespace owns the mount namespace
PrivateUsers=yes
PrivateMounts=yes
DelegateNamespaces=mnt

# Create a private bpffs for the service
PrivateBPF=yes

# Delegate BPF token capabilities for the service bpffs
BPFDelegateCommands=BPFObjPin
BPFDelegateMaps=BPFMapTypeArray
BPFDelegatePrograms=BPFProgTypeTracepoint
BPFDelegateAttachments=BPFFlowDissector

# Make sure the service only has CAP_BPF and CAP_PTRACE (within the user namespace)
CapabilityBoundingSet=CAP_BPF CAP_PTRACE
```

Could straight-forwardly be extended to install policies to the LSM.