BpfJailer: eBPF Mandatory Access Control

Liam Wisehart
liamwisehart@meta.com

Meta

Agenda

01 Jailing Cases

02 Architecture and Deployment

03 Process Tracking and Enrollment

04 Path Matching with eBPF

05 Challenges of Root User Jailing

06 Signed Binaries with eBPF

Meta

Overview

- Created to sandbox/jail untrusted workloads
  - Jailing: Reducing privilege for low-trust processes
- Has become a popular tool for security orgs
- Now covers system-wide MAC including both restricted and privileged roles
- Tracks processes using a task_storage map, copied to children on clone()
  - Original process is enrolled in various ways
- Deployed across most on infra in socket activated mode, key surfaces in an always-on mode and/or baked into init
  - Working to eliminate socket activation

Plan to open source key components (eBPF and eBPF loading code) in 2026

Meta

Why Create a New MAC Solution

- SELinux: slow
  - Completely eliminated at Meta
- AppArmor: didn’t have all required features, path based exclusively
- Landlock: didn’t have all required features, processes always need to enroll themselves with API

- BpfJailer can operate in both a path based (involuntary), and non path based mode (voluntary)
- Covers full range of features, including features not present in other MAC solutions
  - Signed binaries
  - Proxying/Protocol interception
- eBPF implementation allows development to move quickly and safely without changing upstream
- No measurable performance regressions across any workload/host type

Meta

Some Use Cases

- **Original:** Sandboxing VMs running untrusted (AI generated) code
  - Extremely low privilege
- Restricting access to Trusted Execution Environment (AMD SEV-SNP) devices
  - Needs to defend against potentially root level insiders
- Restricting access to database files
  - Needs to defend against root level insiders (less stringent)
- Jailing AI agents
  - Needs to allow most development actions
  - Needs to support legacy code that doesn’t expect to be restricted
- Restricting access to certificates on development servers
  - Needs to allow a permissive environment overall (including root)

Meta

Complete

Planned

Capabilities

Networking

- TCP bind()
- TCP connect()
- UDP sendmsg()
- UDP recvmsg()
- Unix domain bind()
- Unix domain connect()
- Unix domain sendmsg()
- Vsock bind()
- Vsock connect()
- accept()
- DNS
- DBus
- TCP and UDP Proxy
- CA injection and local TLS interception

Execution

- Exec binary and arguments
- setuid/gid flags

Filesystem

- File path matching
- mount()
- Restricted devices

IPC

- SysV
- Posix

Kernel

- Kernel modules
- eBpf programs and maps

Permissions

- setuid()/setgid()
- Capability changes

Tampering

- ptrace()
- process_vm_readv()/writev()
- /proc

Other

- Signed binaries
- ioctl()

Meta

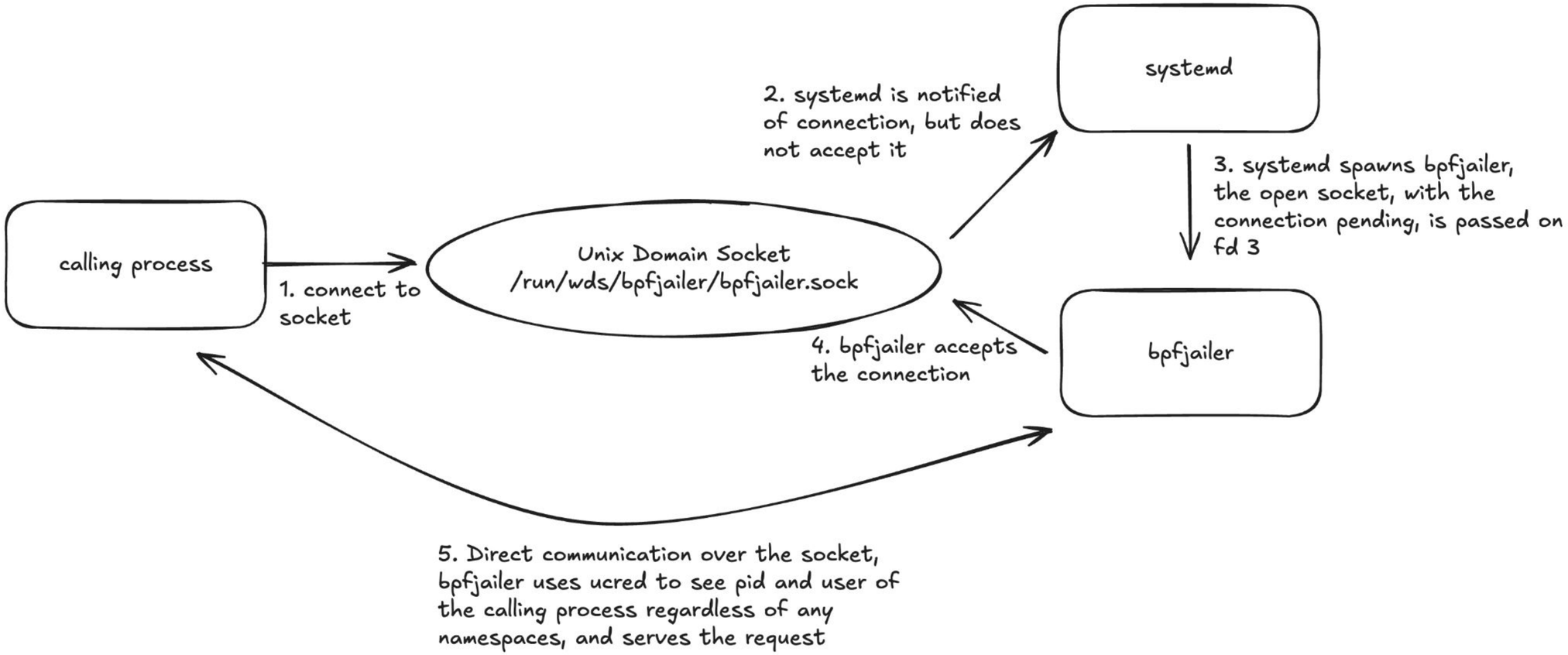
In This Talk

- Only going to go in depth on the deployments and the more complex features
  - Path matching
    - Used for filesystem, unix socket, and exec restrictions
    - Used for binary path and cgroup enrollment
  - Signature validation

- Networking and other restrictions are simpler and have more mundane solutions
  - Don’t need any support from the broader community here

Meta

Socket Activation



Meta

More Enrollment Options

- Unix socket enrollment is directly included in Meta's container scheduler (tupperware)
- Can inject mount namespaces and symlinks as part of jailing
- We found this method to be too brittle and introduce friction/latency for some applications
  - Sometimes difficult/impossible to force open-source/legacy code to utilize BpfJailer API
- Added several other options:
  - Executable Path
  - Cgroup Path
  - Xattr on executable
- BpfJailer has more or less outgrown socket activation as it has been deployed more widely, and now is usually deployed as a normal systemd service or as part of init (covered later)

Meta

Roles

- Each pod (unit of tracked processes) is mapped to a role (set of policy)
- BpfJailer supports stacked pods, pods within pods
- Currently allows up to 4 layers of stacking
- Roles are evaluated cumulatively allowing different levels of isolation or exemptions for certain processes
- Examples:
  - Container receives one policy, workload inside container receives more restrictive policy
  - Container receives one policy, specific process in init receives exception to policy
  - Devserver users receive one policy and agents receive a second, stacked policy

Meta

Init (Daemonless) Installation

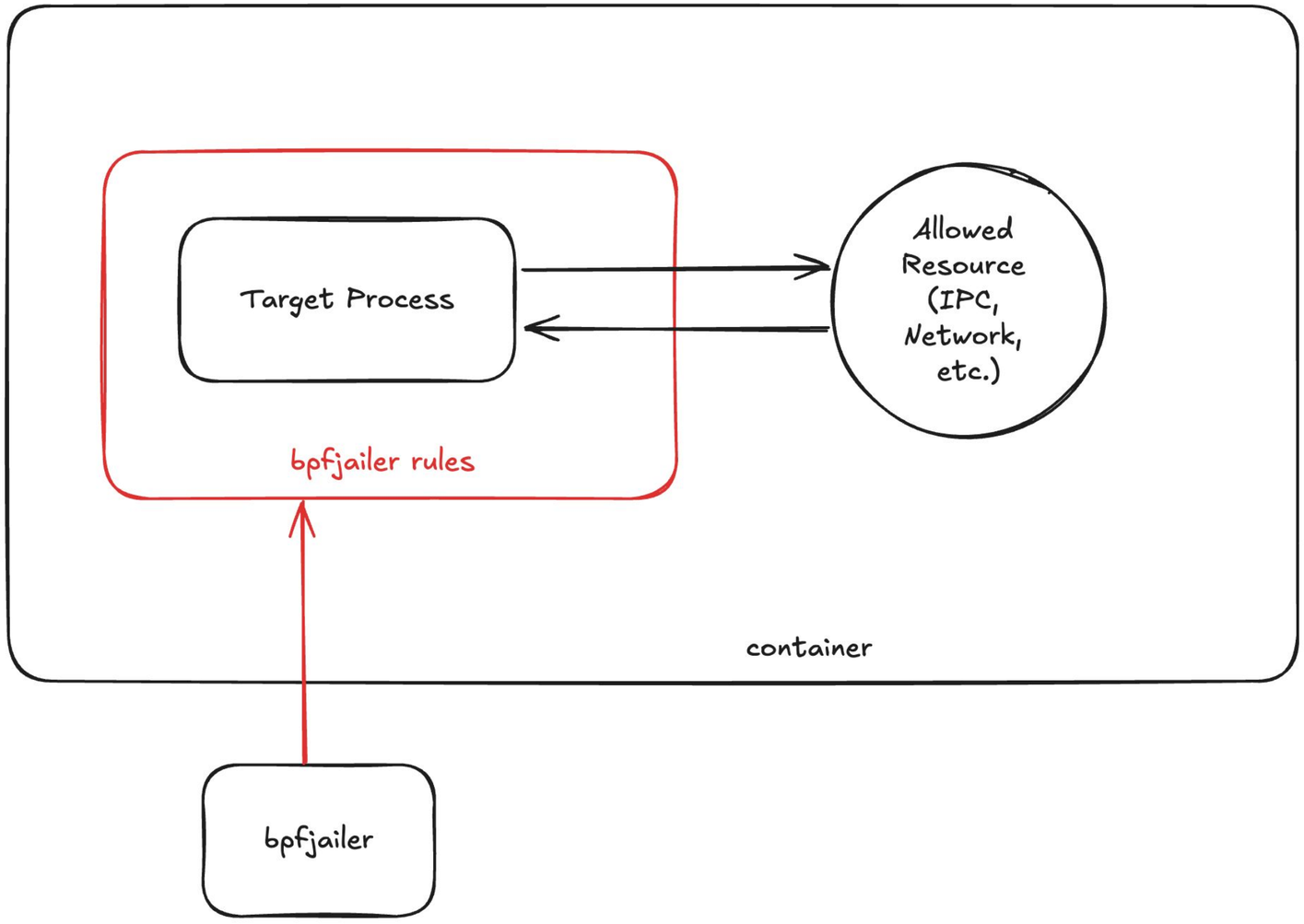
- Using a daemon to manage eBPF programs opens a gateway to attack
- For some applications (eventually all) BpfJailer is baked into the init system
  - Bootstrap binary pins eBPF programs and maps then dies before switch to rootfs
  - Programs cannot be removed or upgraded without reboot
- Logging daemon (installed later) has access to ringbuffers for logging access events and errors
  - Would be nice to eliminate this in favor of eBPF networking helpers that have been proposed
    - Would solve cross binary versioning issues
- All other access to BpfJailer eBPF programs and maps is blocked
- Can’t enroll processes via socket, needs to use other methods

We will work to open source this stack in 2026

Meta

Jailing

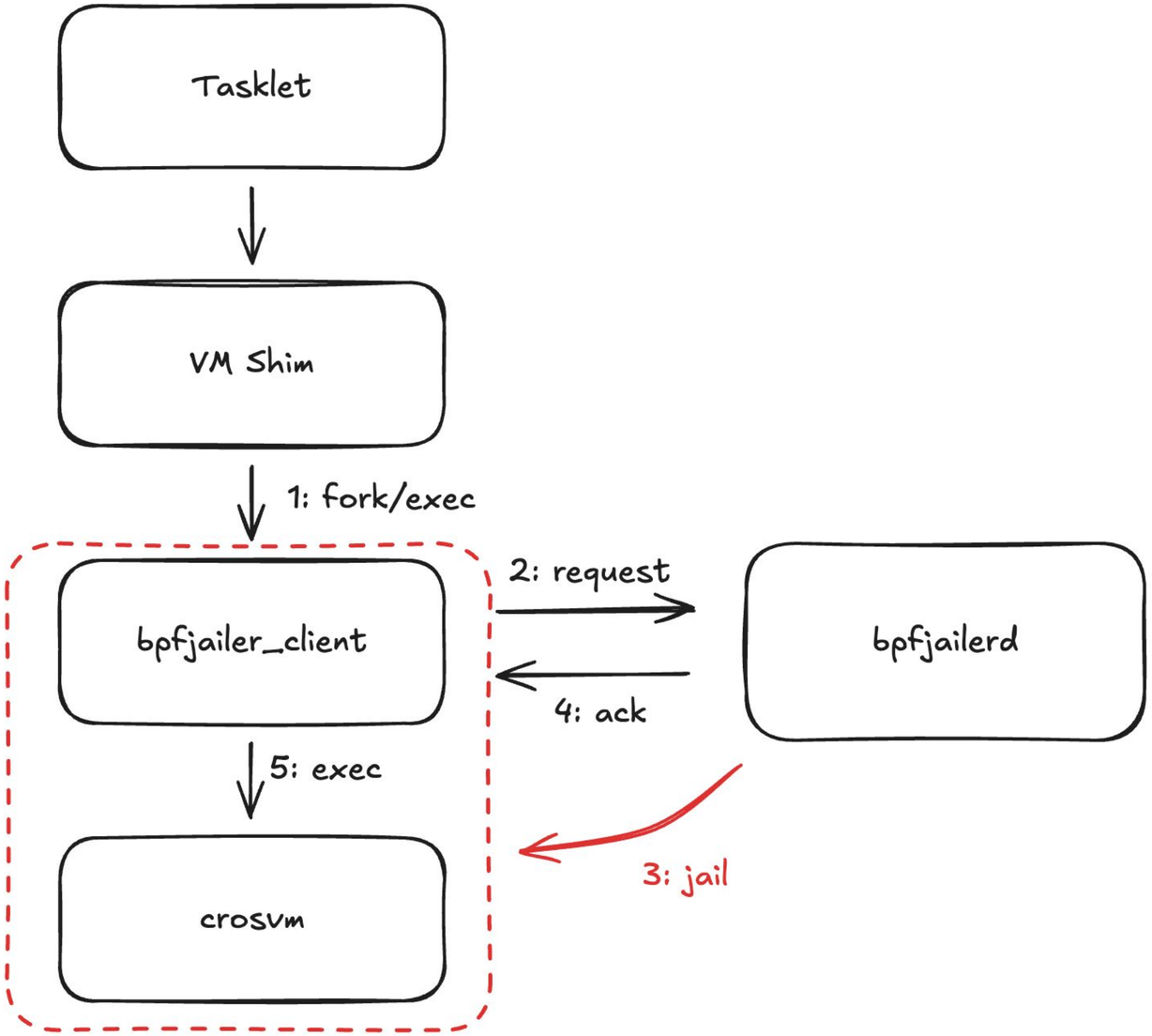
- Specific processes are jailed
- Allows extremely locked down policy
  - Individual IPC and network resources are allowlisted
- Jailing entire containers is also supported
- BpfJailer service can support 10,000 simultaneous pods (jails)
  - Number of tracked processes unbounded



Meta

Example: VM Jailing

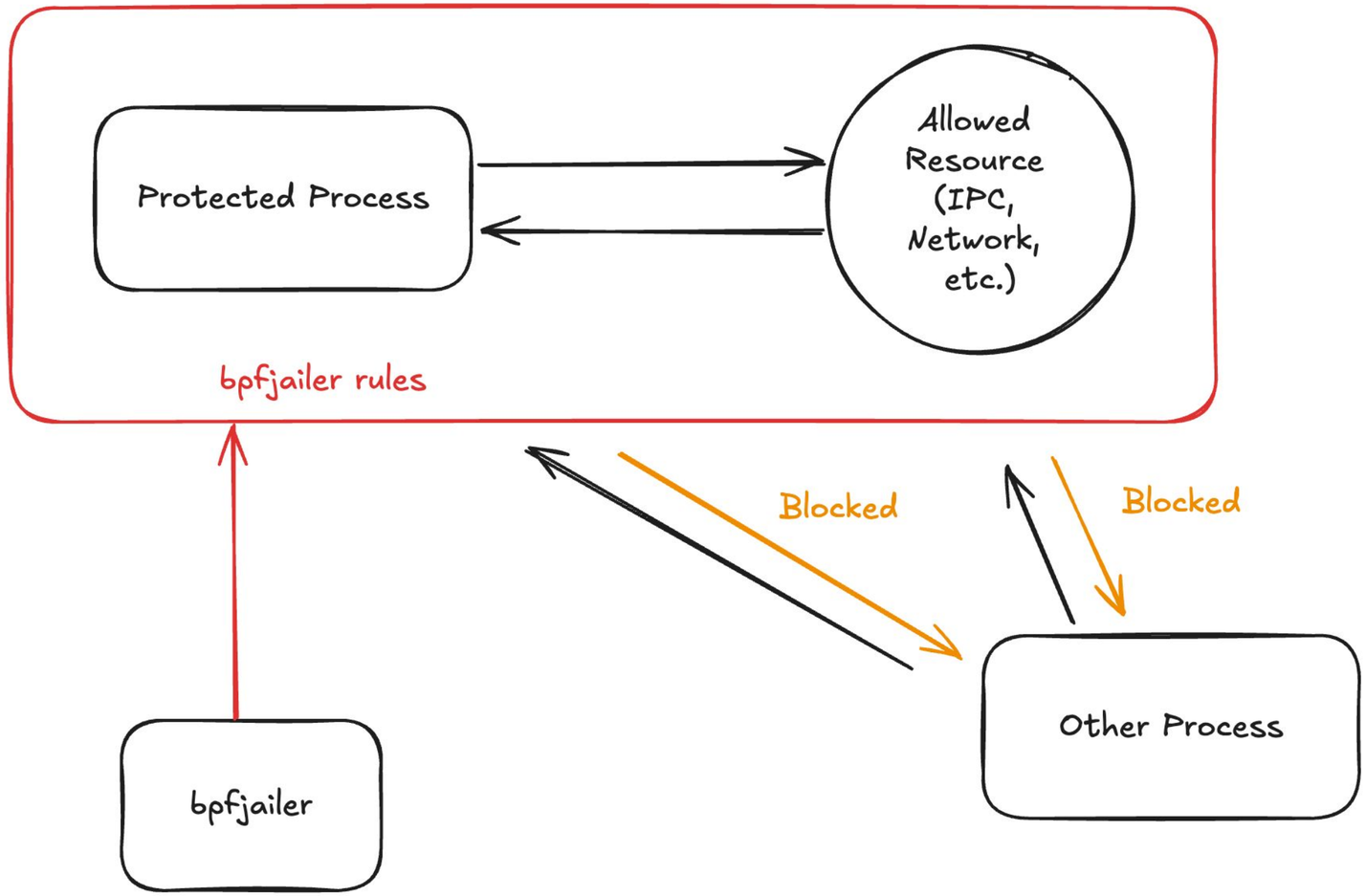
- VM shim controls crosvm lifetime
- Forks into bpfjailer_client which jails itself then enters crosvm
- Jailing is in the path of execution which enforces jailing of crosvm and children processes
- Drawback is that BpfJailer downtime can block crosvm from spawning
  - Systemd orchestration and container orchestration not synchronized



Meta

Access Control

- Can also function in reverse, protecting certain processes, blocking external access
  - kill()
  - ptrace()
  - /proc
  - Binary tampering
- The protected process is the only one allowed to access the protected resource

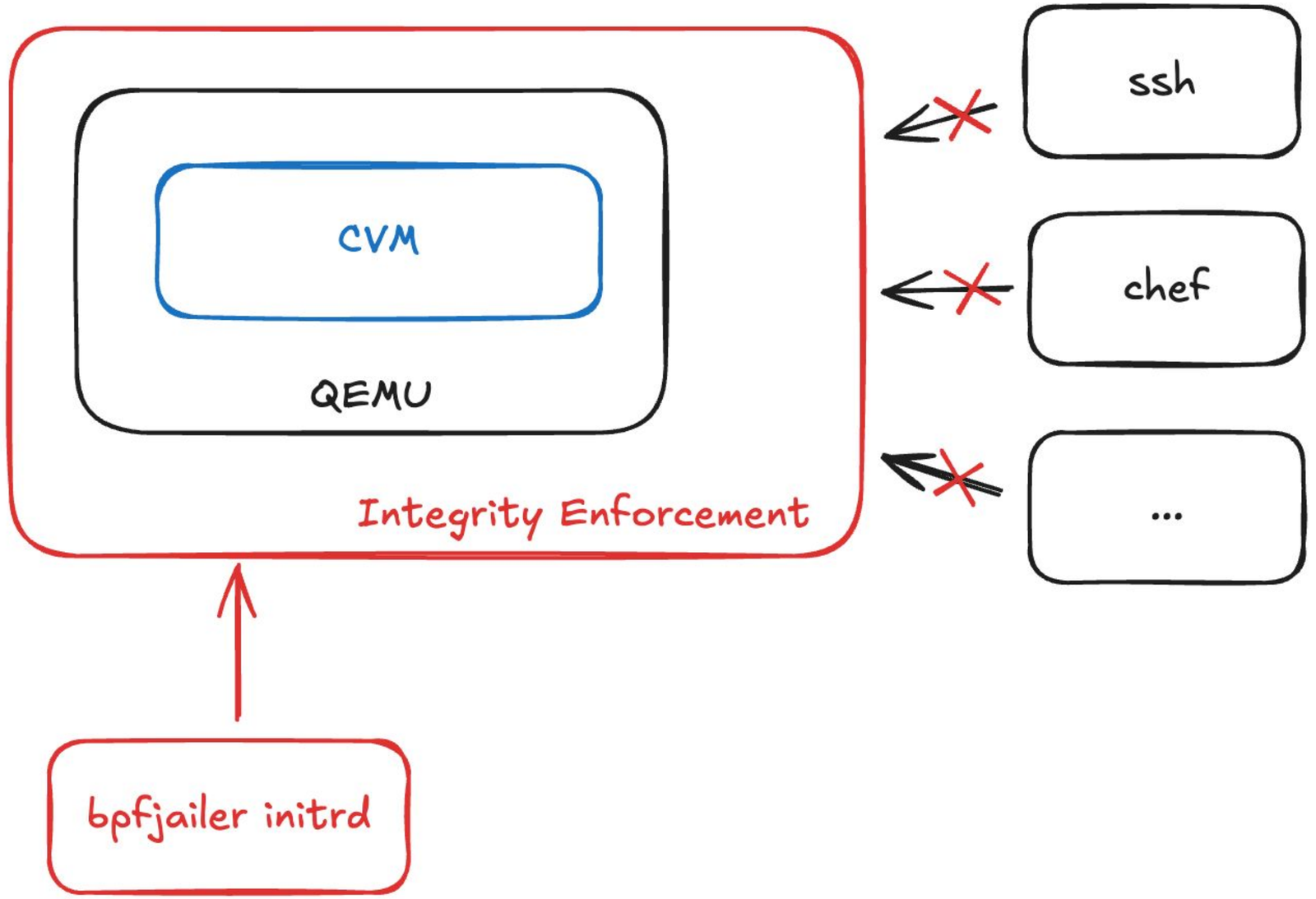


Meta

Example: WhatsApp TEE

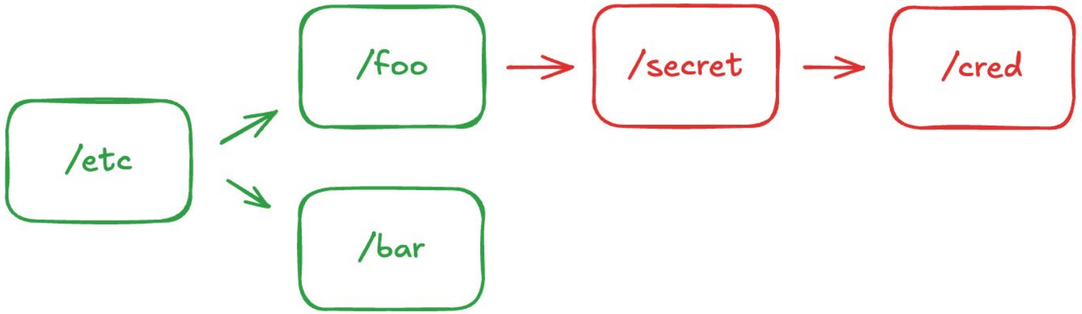
- Need to protect CVM and QEMU from tampering
- Signed binaries
- Ptrace and /proc blocked to protected processes
- Kernel modules restricted

Rules installed by initrd and immutable + measured as part of attestation

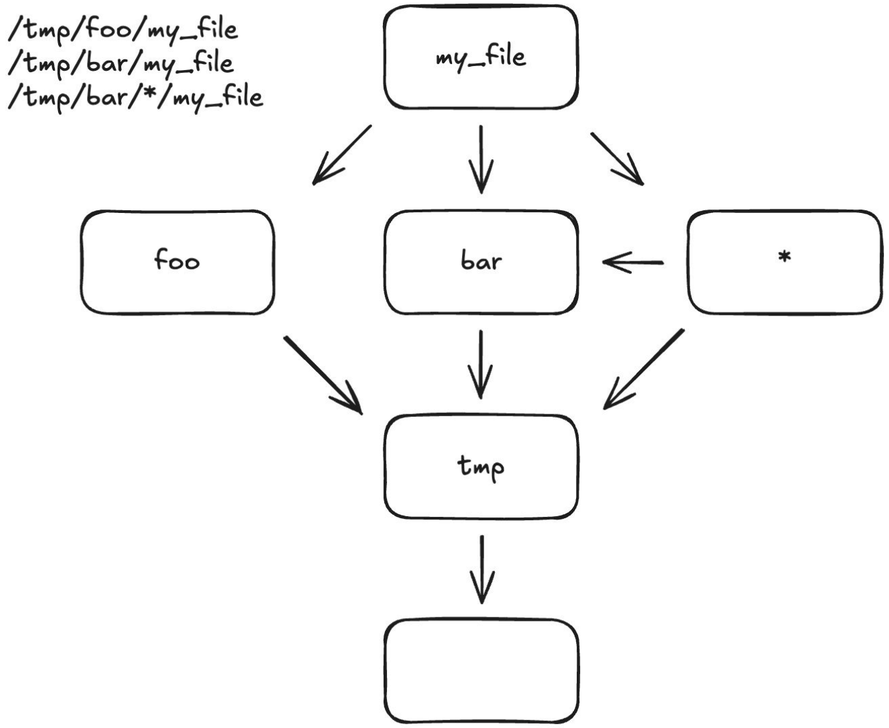


Meta

Path Matching



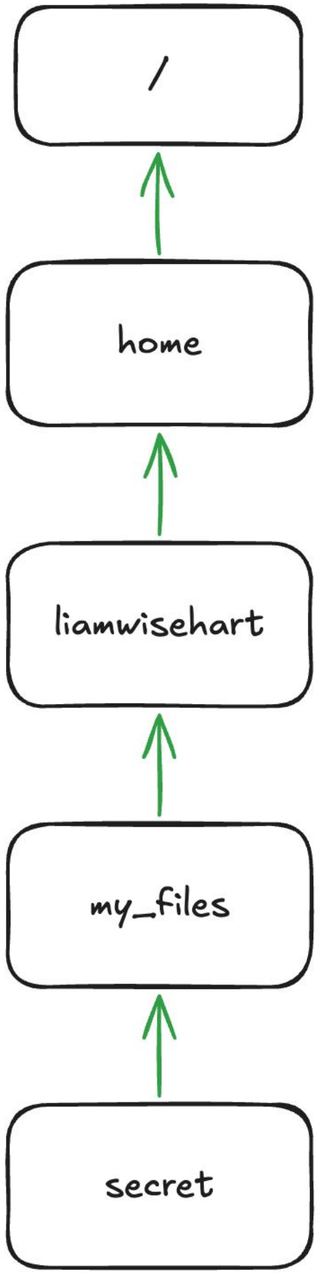
- Path based (what users expect)
  - Supports hierarchical structures of directory based permissions
- Walks dentry/mount tree in order to match each open()/rename()/etc.
- Constructs a graph/state machine built out of hash maps and array maps (originally kernel 6.4+, now 6.9+). Supports:
  - Wildcarding (for example, need to ignore ids in container paths)
  - More efficient use of CPU/RAM then hashing whole path at once
  - Corpus of thousands of patterns
- Cache inodes and invalidate on rename and mount lock changes



Meta

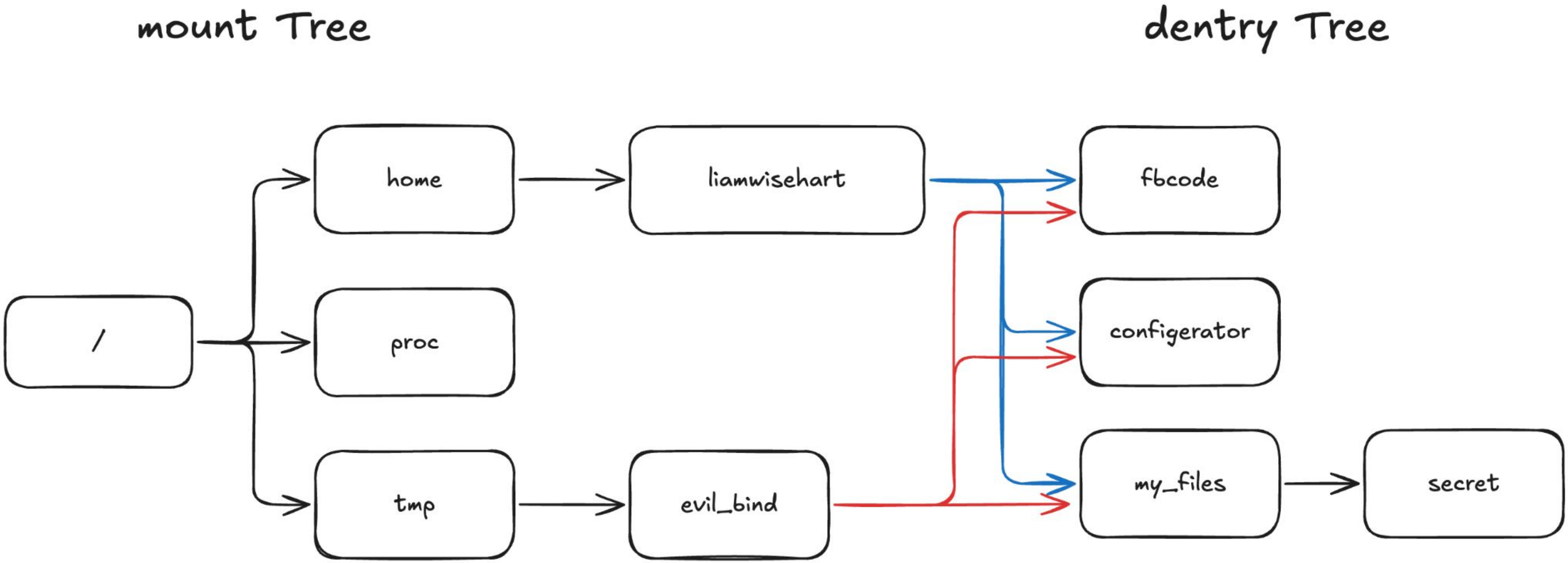
Extract Vector of String Views

- Two step process, construct vector of string views, then try to match
  - Takes 2,000 lines of eBPF code and runs up against stack limit, instruction limit, etc.
  - Required significant work against the verifier
- Constructs a vector of string views (d_name) up to 255 directories deep of up to 255 byte names
- Can’t filter bind mounts due to the limitations of securty_sb_mount
  - Doesn’t expose bind mount path in dentry form (vulnerable to symlink based attack)
  - Hard links and renames can just be blocked with existing LSM hooks



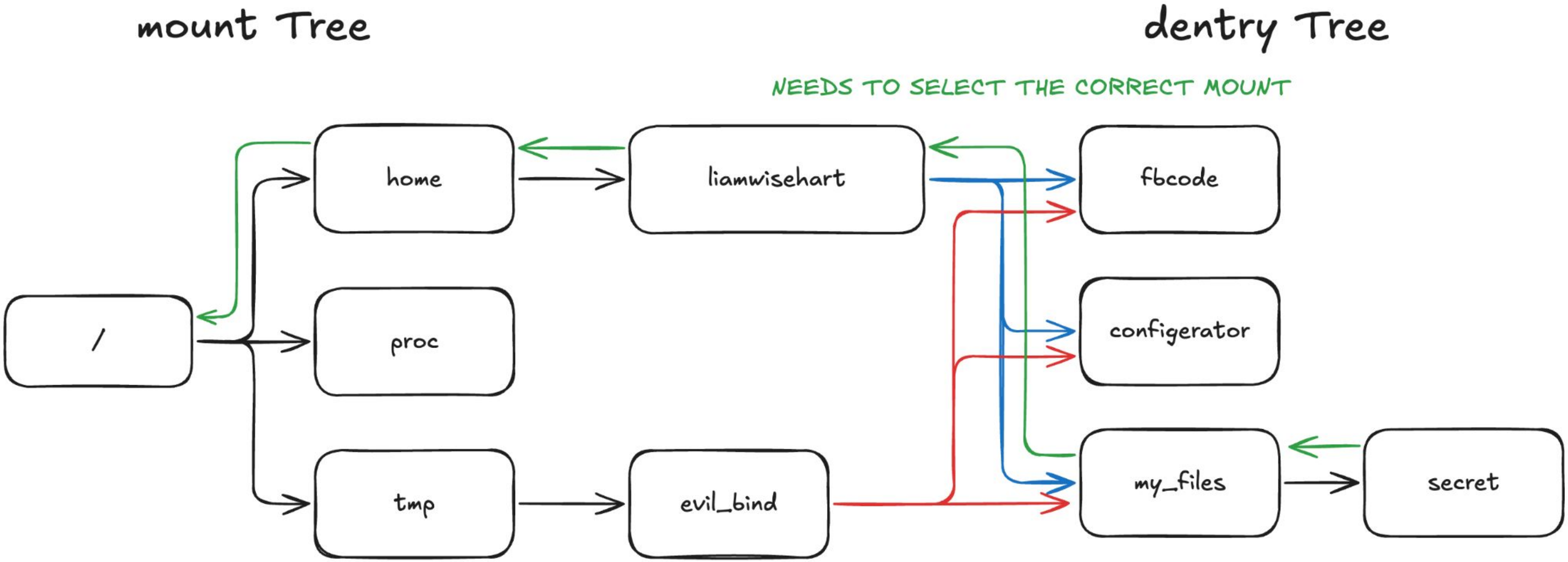
Meta

Path Matching: Bind Mounts Create Alternate Paths



Meta

Path Matching: Need To Walk Mount Tree Correctly



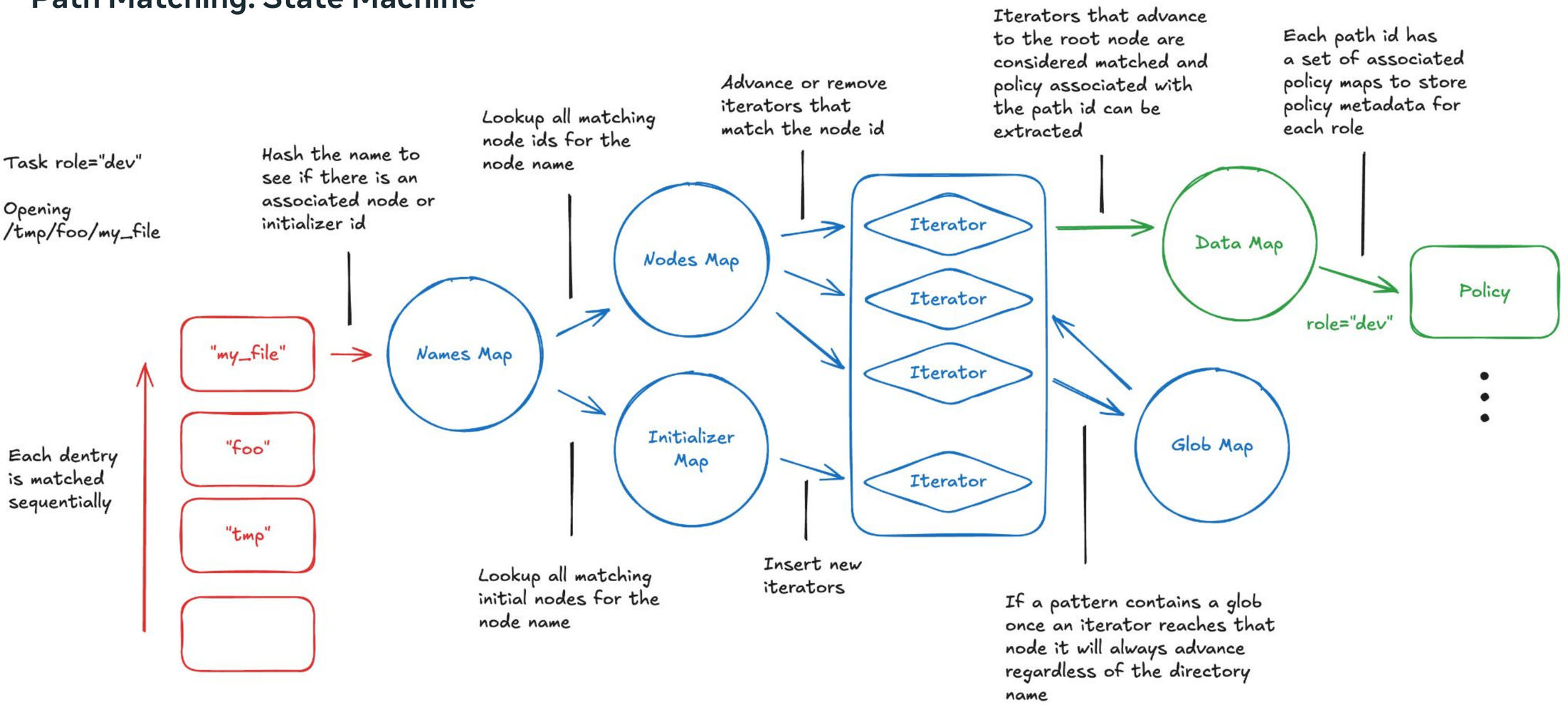
Meta

Mount Traversal

- Traversal algorithm:
  - Enumerate all mounts in the **root (pid 1)** mount namespace (cached based on mount lock)
  - Build a map of mount root dentry -> struct mount
  - For each dentry select the **oldest** mount (lowest mnt_id_unique)
  - Attempt to resolve a path, by walking dentries/mounts, up to the root dentry of the root mount namespace
  - If we can't find the root this is not a tracked file/directory (likely isolated to a different mount namespace), otherwise try to match
- Use LSM to protect against mount/unmount and renaming directories on the paths as well as hard links

Meta

Path Matching: State Machine



Meta

Key Issues

- Filesystem matching is very difficult, and requires hacks to get around verifier limits
  - Matching is significantly slower than other LSM operations – though no measurable regressions have been observed, even in IO heavy workloads (compiling, database tiers)
- Need to match capabilities of LKM/user space security products which do not have eBPF limitations
- Policy can become very complex to cover all possible attacks (rename, remounting, etc.)

- Need to simulate the environment large amounts of legacy code expects
  - Need to inject certificates and other files into jailed (agentic) processes
  - Currently uses a mount namespace and bind mounts

Meta

Possible New Kernel Feature: open() Redirection

- Can we redirect the open syscall (and friends) similar to how connect and sendmsg can be redirected
- Would effectively allow eBPF to force a process into a mount namespace
  - Putting a shim into a process's startup/orchestrator can be significantly more difficult then just identifying it with eBPF based on some criteria
  - The issue here is that enrolling legacy code into mount namespaces can be difficult
  - In addition, jailed agents still have broad privileges and can learn (or be taught) to escape namespaces
- Would give security daemons more options then a binary block/allow
- Would require several new helpers and new hooks

Meta

What Guarantees can we Make About A Host with Malicious Root Users

- Had some lofty goals that had to be stripped down to essentials due to the difficulties of trying to restrict root users
- In the end for certain (eventually all) workloads
  - BpfJailer is placed in initrd, clients will not connect to hosts unless init and lower level boot artifacts pass hash test
    - **Guarantee: BpfJailer was started on the host**
  - BpfJailer defends its pinned programs and maps against tampering, blocks LKM loading
    - **Guarantee: BpfJailer is still running**
  - Key resources are restricted only to a single role
    - **Guarantee: Unauthorized access to key resource only is blocked**
  - Service accessing key resource have signed binaries, vetted command-line arguments, signed config files, and is protected from tampering (ptrace and friends)
    - **Guarantee: Key resource is only accessed by expected service**

Meta

Signed Binaries

- x509 certs are loaded into child of root keyring at startup (packaged into init if BpfJailer is put there)
  - BpfJailer protects access to its keyring
- Binary validation:
  - Fsverity hash signed, signature included in xattr on binary and all shared objects needed
  - When a binary executes into a role that requires signing BpfJailer validates the signatures
    - Validation occurs on mmap_file if the file is in exec mode
- Currently used for QEMU when used in TEE to block unauthorized access to kvm and AMD SEV-SNP interfaces
  - VM image is verified by client
- Command line arguments will be vetted in 2026
- Config files and other artifacts will be signed in 2026

Meta

Future Work

- Continuing to iterate on filesystem path matching (regex)
- Easy to configure profiles for users who don’t understand eBPF
- Enrolling more of the Meta fleet in these features
- Migrating into a daemonless/initrd model fleetwide
- Open source

Meta

BpfJailer: eBPF Mandatory Access Control

Liam Wisehart
liamwisehart@meta.com

Meta