eBPF Runtime Security Agent Design

2026-02-05

Contents

AegisBPF Architecture

This document describes the internal architecture of AegisBPF, an eBPF-based runtime security agent.

Overview

AegisBPF uses eBPF (extended Berkeley Packet Filter) to monitor and optionally block process executions at the kernel level. It leverages the BPF LSM (Linux Security Module) hooks for enforcement and tracepoints for audit-only monitoring.

```
                  User Space

                 aegisbpf daemon

Policy      BPF       Event         Metrics
Manager     Ops       Handler       (Prometheus)




                 libbpf



                         bpf() syscall

                  Kernel Space


                BPF Subsystem

LSM Hook        Tracepoint           BPF Maps
file_open       sched_exec
                                  deny_inode
 (enforce)       (audit)          deny_path
                                  allow_cgroup
                      events (ring)
                                   block_stats
```

```
                    exec() syscall
```

Components

Kernel Space (BPF Programs)

bpf/aegis.bpf.c The BPF program runs in kernel context and implements:

1. **LSM Hook (file_open)**
   - Called on file open attempts
   - Can return -EPERM to block access
   - Checks deny_inode and allow_cgroup maps
   - Only active when BPF LSM is enabled
2. **Tracepoints**
   - `sched_process_exec`: emits EXEC events for process executions
   - `sys_enter_openat`: emits audit-only BLOCK events for deny_path matches
   - Works without BPF LSM (audit-only paths)
3. **BPF Maps**
   - `deny_inode`: Hash map of blocked (dev, inode) pairs
   - `deny_path`: Hash map of blocked path hashes
   - `allow_cgroup`: Hash map of allowed cgroup IDs
   - `events`: Ring buffer for sending events to userspace
   - `block_stats`: Global counters for blocks and drops
   - `deny_cgroup_stats`: Per-cgroup block counters
   - `deny_inode_stats`: Per-inode block counters
   - `deny_path_stats`: Per-path block counters
   - `agent_meta`: Agent metadata (layout version)

User Space

src/main.cpp Entry point and CLI interface: - Parses command-line arguments - Initializes logging - Dispatches to appropriate command handler - Manages signal handling for graceful shutdown

src/bpf_ops.cpp BPF operations layer: - Loads BPF object file - Attaches programs to hooks - Manages map operations - Handles BPF filesystem pins

Key functions: - `load_bpf()`: Load and optionally pin BPF objects - `attach_all()`: Attach programs to LSM/tracepoints - `add_deny_inode()`: Add entry to deny list - `read_block_stats_map()`: Read statistics

src/policy.cpp Policy file management: - Parses policy files (INI-style format) - Validates policy syntax and semantics - Applies policies to BPF maps - Handles policy rollback

src/events.cpp Event handling: - Receives events from BPF ring buffer - Formats events for output (JSON/text) - Sends events to journald or stdout

src/utils.cpp Utility functions: - Path validation and canonicalization - Cgroup path resolution - Inode-to-path mapping - String manipulation

src/sha256.cpp SHA256 implementation: - Pure C++ implementation (no external deps) - Used for policy file verification - Used for path-based hash lookups

src/seccomp.cpp Seccomp filter: - Applies syscall allowlist after initialization - Reduces attack surface if agent is compromised

src/logging.hpp Structured logging: - Chainable API for field addition - Text and JSON output formats - Log level filtering - Thread-safe singleton

src/result.hpp Error handling: - `Result<T>` type for success/failure - `Error` class with code, message, context - `TRY()` macro for early return

Data Flow

File Access Blocking (Enforce Mode)

```
1. Process calls open("/etc/shadow")


2. Kernel invokes file_open LSM hook


3. BPF program handle_file_open runs

       Check allow_cgroup map
         If cgroup allowed → ALLOW

       Check deny_inode map
         If inode blocked → DENY + emit event


4. Return 0 (allow) or -EPERM (deny)
```

5. If denied, ring buffer event sent to userspace

6. aegisbpf daemon receives event

7. Event logged to journald/stdout

Audit Mode

1. Process executes a binary (execve)

2. execve() completes successfully

3. Kernel fires sched_process_exec tracepoint

4. BPF program handle_execve runs

5. EXEC event emitted to ring buffer

6. aegisbpf daemon receives event

7. Event logged to journald/stdout

1. Process opens a file (open/openat)

2. Kernel fires sys_enter_openat tracepoint

3. BPF program handle_openat runs

4. If path is in deny_path, emit audit-only BLOCK event

5. aegisbpf daemon receives event

6. Event logged to journald/stdout

BPF Map Pinning

Maps are pinned to /sys/fs/bpf/aegis/ for persistence:

```
/sys/fs/bpf/aegis/
  deny_inode          # Blocked inodes
  deny_path           # Blocked paths
  allow_cgroup        # Allowed cgroups
  events              # Ring buffer (not pinned)
  block_stats         # Global counters
  deny_cgroup_stats   # Per-cgroup stats
  deny_inode_stats    # Per-inode stats
  deny_path_stats     # Per-path stats
  agent_meta          # Layout version
```

Pinning allows: - Persistent deny lists across agent restarts - Multiple agent instances sharing state - External tools to inspect/modify maps

Error Handling Strategy

AegisBPF uses a Result monad pattern:

```
Result<InodeId> path_to_inode(const std::string& path) {
    struct stat st;
    if (stat(path.c_str(), &st) != 0) {
        return Error::system(errno, "stat failed");
    }
    return InodeId{static_cast<uint32_t>(st.st_dev), st.st_ino};
}

// Usage with TRY macro
Result<void> block_file(const std::string& path) {
    auto inode = TRY(path_to_inode(path));
    TRY(add_to_deny_map(inode));
    return {};
}
```

Benefits: - Explicit error handling at every call site - Error context preservation through the call stack - No exceptions (suitable for signal handlers)

Concurrency Model

- **Single-threaded main loop**: Ring buffer polling
- **Thread-safe caches**: CgroupPathCache and CwdCache use mutex
- **Atomic counters**: std::atomic for journal error state
- **No shared mutable state**: BPF operations are inherently safe

Security Considerations

1. **Principle of Least Privilege**
   - Minimal capability set (SYS_ADMIN, BPF, PERFMON)
   - Seccomp filter restricts syscalls
   - AppArmor/SELinux confine file access
2. **Input Validation**
   - All CLI paths validated before use
   - Policy files parsed with strict error handling
   - SHA256 verification for policy integrity
3. **Defense in Depth**
   - Multiple security layers (BPF, seccomp, MAC)
   - RAII for resource cleanup
   - Crash-safe persistent state

Performance Characteristics

- **BPF overhead**: ~100-500ns per file open
- **Map lookups**: O(1) hash table operations
- **Ring buffer**: Lock-free producer-consumer
- **Memory usage**: ~10MB base + map sizes
- **CPU usage**: Minimal when idle, proportional to exec rate

Kernel Requirements

Feature	Minimum Version	Notes
BPF CO-RE	5.5	Compile-once, run-everywhere
BPF LSM	5.7	Required for enforce mode
Ring buffer	5.8	More efficient than perf buffer
CAP_BPF	5.8	Dedicated capability
BTF	5.2	Type information

AegisBPF Threat Model

Version: 1.0 (2026-02-05) Status: Canonical threat model for the v1 contract.

This document defines what AegisBPF is designed to defend, what is explicitly out of scope, and where bypass risk remains.

For a dispositioned list of known bypass surfaces, see `docs/BYPASS_CATALOG.md`.

Security objective

Prevent unauthorized file and network operations from untrusted workloads on a Linux host, while producing usable audit evidence for incident response.

Assets and trust boundaries

Protected assets

- File access policy state (`deny_*`, `allow_*` maps and applied policy files)
- BPF object integrity (`aegis.bpf.o` + expected SHA256)
- Policy trust chain (trusted public keys + signed bundles)
- Audit/event stream (ring buffer output + metrics)

Trust boundaries

- Kernel-space BPF programs enforce decisions; user-space consumes telemetry.
- The agent runs with elevated privilege and is part of the trusted computing base.
- Consumers of logs/metrics are not trusted to make enforcement decisions.

Attacker model

In scope

- Unprivileged host processes attempting blocked file opens.
- Container workloads attempting disallowed file/network operations.
- Policy tampering attempts without access to trusted signing keys.
- Supply-chain tampering of the BPF object on disk.

Out of scope

- Host root compromise.
- Kernel compromise, malicious kernel modules, or verifier bugs.
- Physical/firmware attacks.
- Privileged container escape where attacker gains host-level capabilities equivalent to root.

Coverage boundaries

Surface	Covered	Notes
File enforce path	Yes (LSM `file_open` + `inode_permission`)	Returns `-EPERM` in enforce mode
File audit fallback	Partial (`openat` tracepoint)	Audit only; cannot block
Network egress/connect	Yes (`socket_connect`)	IPv4 + IPv6 exact/CIDR/port
Network bind	Yes (`socket_bind`)	Port-oriented deny logic
Inbound accept/listen/sendmsg	No	Currently out of scope
Non-LSM kernel paths	Partial	Depends on available hooks

Syscall path coverage boundaries

This release treats file enforcement coverage as limited to LSM-backed open permission paths:

Path	Contract status	Notes
`open` / `openat` / `openat2`	In scope	Covered via LSM permission hooks, not by tracepoints alone
`execve`	Partial	File access policy still enforced via inode checks; process-exec telemetry is audit signal
`mmap` executable mapping	Partial	Depends on kernel hook behavior and file-open path leading to mapping
`socket_connect` / `socket_bind`	In scope for network deny policy	Rule types documented in `docs/POLICY_SEMANTICS.md`
`accept` / `listen` / `sendmsg`	Out of scope	No block guarantees in this release

Filesystem caveats

- ext4/xfs are the primary validated filesystems for enforcement semantics.
- OverlayFS is supported with caveats: upper/lower inode behavior can differ from single-layer filesystems.
- Bind mounts may present the same inode under multiple paths; inode deny still applies, path telemetry can differ.

- Network/distributed filesystems (for example NFS/FUSE variants) are not a primary guarantee surface in this release.

Container and orchestration caveats

- Mount namespaces can cause path-view differences between the agent and target workloads; inode enforcement remains the primary invariant.
- User namespaces are treated as a bypass-risk amplifier when combined with privileged runtime configuration.
- Privileged Kubernetes pods (`privileged: true`, broad host namespace/device access, or equivalent host-level capability) are outside the default trust boundary and must be treated as high-risk exceptions.

Known blind spots and bypass surface

Namespaces and mounts

- Path rules are canonicalized in the agent's mount namespace at policy-apply time; namespace-specific path views can differ.
- Bind mounts can expose the same inode under multiple paths. Inode deny rules still hold, but path-only observability can appear inconsistent.
- OverlayFS can produce different upper/lower inode behavior than plain ext4/xfs.

Filesystem/object lifecycle

- Inode-based enforcement is robust to rename/hardlink but can be affected by inode reuse after delete/recreate cycles.
- Path entries in `deny_path_map` mainly support tracepoint audit fallback; enforce decisions are inode-driven.

Privilege and policy bypass vectors

- `allow_cgroup` and break-glass are intentional bypass controls and must be tightly governed.
- Privileged workloads (`CAP_SYS_ADMIN`/equivalent) can undermine host controls; treat as trust boundary breach.
- If BPF LSM is unavailable, enforcement degrades to audit-only.

Accepted vs non-accepted bypasses

Case	Status	Required handling
`allow_cgroup` exemption	Accepted (explicit)	Change-controlled, audited, short-lived where possible

Case	Status	Required handling
Break-glass mode	Accepted (explicit emergency control)	Incident ticket + postmortem + mandatory rollback
Host root or kernel compromise	Not accepted (out of scope)	Treat as full trust-boundary failure
Privileged orchestrator workloads bypassing host controls	Not accepted for core guarantee	Scope as exception, not as protected workload

Security guarantees (when prerequisites hold)

If BPF LSM is enabled and the host is not root-compromised: - Denied inodes are blocked with `-EPERM`. - Audit events include stable process correlation fields (`exec_id`, `trace_id`). - Signed bundle anti-rollback is enforced via monotonic `policy_version`. - BPF object hash verification prevents silent binary swap at load time.

Residual risk management

- Run with signature enforcement: `policy apply --require-signature`.
- Keep trusted keys in root-owned path, rotate at defined cadence.
- Treat audit-only mode as reduced-security operation and alert on it.
- Use runbooks for ring-buffer drops, false positives, and break-glass events.

Related docs: - `docs/POLICY_SEMANTICS.md` - `docs/COMPATIBILITY.md` - `docs/KEY_MANAGEMENT.md` - `docs/runbooks/`

Policy Semantics

Version: 1.0 (2026-02-05) Status: Canonical semantics reference for the v1 contract.

This document defines how policy rules are interpreted at runtime, including edge cases that matter for production correctness.

Rule model

Policy sections: - `[deny_path]` -> canonicalized path + inode-derived deny entries - `[deny_inode]` -> explicit `dev:ino` deny entries - `[allow_cgroup]` -> cgroup exemptions (`/sys/fs/cgroup/...` or `cgid:<id>`) - `[deny_ip]`, `[deny_cidr]`, `[deny_port]` -> network deny rules

Supported versions: - `version=1` and `version=2` are accepted by parser. - Use `version=2` for network-aware policies.

File decision semantics

In enforce-capable mode (BPF LSM enabled), file decisions are inode-driven: 1. If inode is not in `deny_inode_map` -> allow 2. If inode is in survival allowlist -> allow 3. If cgroup is in `allow_cgroup_map` -> allow 4. Otherwise -> audit or deny (`-EPERM`) based on mode

In fallback tracepoint mode: - `deny_path_map` is checked on `openat` events for audit only. - No kernel block is possible in this path.

Deterministic precedence and conflict resolution

File-path precedence (highest to lowest): 1. Survival allowlist entry for the current inode -> allow 2. `allow_cgroup` match -> allow 3. `deny_inode` / inode-derived deny from `deny_path` -> deny in enforce mode 4. No deny match -> allow

Network-path precedence: 1. `allow_cgroup` match -> allow (intentional bypass control) 2. Exact IP deny -> deny 3. CIDR deny -> deny 4. Port deny -> deny 5. No deny match -> allow

Conflict handling: - Duplicate deny entries are de-duplicated by map key identity. - `deny_path` and `deny_inode` converge to the same inode deny key when they refer to the same object. - `allow_cgroup` always has explicit precedence over deny rules by design; this is security-sensitive and must be tightly controlled.

Path rule normalization

When applying `deny_path` rules: - Input path MUST be non-empty and contain no NUL byte. - Path is canonicalized (resolves symlinks, `./..`). - Canonical path length must be `< kDenyPathMax`. - Canonical inode is inserted into `deny_inode_map`. - Canonical path is inserted into `deny_path_map`. - If raw input differs from canonical path, raw path is also added to `deny_path_map` for observability.

Implication: enforcement follows inode identity; path entries mostly support audit fallback and operator readability.

Inode rule semantics

`deny_inode` rules match exact `{dev, ino}` pairs: - Survive rename and hardlink changes. - Are independent of textual path. - Can be affected by inode reuse after file deletion/recreation. - May appear under multiple path views (bind mounts, container mount namespaces) while still enforcing on the same inode identity.

Cgroup allow semantics

`allow_cgroup` is an explicit bypass control: - If process cgroup ID matches allowed entry, deny rules are skipped. - This applies to both file and network hooks. - Use sparingly and treat changes as security-sensitive.

Network rule semantics

`socket_connect` match order: 1. Exact IP deny (`deny_ipv4` / `deny_ipv6`) 2. CIDR deny (`deny_cidr_v4` / `deny_cidr_v6` LPM trie) 3. Port deny (`deny_port`) with protocol+direction matching

`socket_bind` currently applies port deny logic only.

IPv6: - IPv6 exact and CIDR matching are enforced in connect hooks. - AF_INET6 traffic is not default-allowed; it is evaluated by rule maps.

Enforcement action semantics

Enforce mode always denies with `-EPERM`. Optional process signaling is separate from the deny decision: - `none`: deny only - `term` (default): send `SIGTERM` + deny - `kill`: escalate `TERM -> KILL` based on strike threshold/window - `int`: send `SIGINT` + deny

Signed bundle semantics

For signed policies: - Signature must verify against trusted key set. - `policy_version` must be monotonic (anti-rollback counter). - Bundle expiration is enforced when set. - On successful apply, version counter is updated.

Edge-case matrix

Scenario	Behavior
Symlink in `deny_path`	Canonical target inode is enforced
File rename	Inode deny continues to apply
Hard link path change	Inode deny continues to apply
Bind mount alias	Same inode still denied; path telemetry may differ
Mount namespace path drift	Policy apply canonicalization uses agent namespace view
Inode reused after delete/recreate	Old deny entry may no longer map to intended object
No BPF LSM	Audit-only fallback (no block)

Namespace and mount consistency contract

- Policy canonicalization is resolved from the agent's mount namespace at apply time.
- Enforcement guarantee is inode-based, not path-string-based.
- Container/user namespace path differences do not change inode deny decisions, but they can change operator-facing telemetry paths.
- For Kubernetes/containers using bind mounts or overlay layers, treat inode rules as authoritative and path rules as operator convenience + fallback audit signal.

Authoring guidance

- Prefer `deny_inode` for strict enforcement invariants.
- Use `deny_path` for operator-friendly inputs and fallback observability.
- Keep `allow_cgroup` minimal and review via security workflow.
- Treat policy changes as deployable artifacts: lint, sign, canary, then rollout.

AegisBPF Network Layer Design

Executive Summary

This document specifies the architecture for adding network monitoring and enforcement capabilities to AegisBPF. The design extends the existing file access control framework to provide egress/ingress network policy enforcement using LSM socket hooks, maintaining consistency with existing patterns while adding network-specific functionality.

---

1. Architecture Overview

1.1 High-Level Design

```
                        USER SPACE (aegisbpf daemon)

  CLI Parser    Policy Mgr    Event Handler    Metrics    Network
                (extended)    (extended)       Exporter   Manager

                                libbpf




  BPF Maps (existing+new)          BPF Programs (existing+new)
  (pinned in bpffs)                (kernel space)

 [EXISTING]                       [EXISTING]
 deny_inode                       LSM: file_open
 deny_path                        LSM: inode_permission
 allow_cgroup                     TP: sys_enter_openat
 events (ringbuf)                 TP: sched_*
 block_stats
 ...                              [NEW - NETWORK]
                                  LSM: socket_connect
 [NEW - NETWORK]                  LSM: socket_bind
 deny_ipv4                        LSM: socket_listen
 deny_ipv6                        LSM: socket_sendmsg (optional)
 deny_port
 deny_cidr_v4
 deny_cidr_v6
 net_block_stats
 net_conn_stats
```

1.2 Design Principles

1. **Consistency**: Mirror existing file access patterns (deny maps, cgroup allowlist, audit/enforce modes)
2. **Minimal Overhead**: Use efficient map structures (LPM trie for CIDR, bloom filter for fast-path rejection)
3. **Fail-Safe**: Network enforcement respects break-glass mode and deadman switch
4. **Incremental**: Network layer is optional - file-only deployments remain supported
5. **Observable**: Full event visibility with Prometheus metrics integration

---

2. BPF Program Design

2.1 New LSM Hooks

2.1.1 `socket_connect` - Egress Control

```
SEC("lsm/socket_connect")
int BPF_PROG(aegis_socket_connect, struct socket *sock,
             struct sockaddr *address, int addrlen)
```

Purpose: Control outbound connections (TCP connect, UDP sendto with destination)

Decision Flow:

```
1. Extract address family (AF_INET/AF_INET6)
2. Extract destination IP and port
3. Check cgroup allowlist → ALLOW if matched
4. Check deny_port map → DENY if matched
5. Check deny_ipv4/ipv6 exact match → DENY if matched
6. Check deny_cidr_v4/v6 LPM trie → DENY if matched
7. DEFAULT: ALLOW
```

2.1.2 `socket_bind` - Service Exposure Control

```
SEC("lsm/socket_bind")
int BPF_PROG(aegis_socket_bind, struct socket *sock,
             struct sockaddr *address, int addrlen)
```

Purpose: Control which ports/addresses processes can bind to

Use Cases: - Prevent unauthorized services from starting - Restrict bind addresses (e.g., block 0.0.0.0 binds)

2.1.3 `socket_listen` - Server Control (Optional)

```
SEC("lsm/socket_listen")
int BPF_PROG(aegis_socket_listen, struct socket *sock, int backlog)
```

Purpose: Additional control point for server sockets

2.2 Event Types

Extend `enum event_type`:

```
enum event_type {
    EVENT_EXEC = 1,
    EVENT_BLOCK = 2,
    // New network events
    EVENT_NET_CONNECT_BLOCK = 10,
    EVENT_NET_BIND_BLOCK = 11,
    EVENT_NET_LISTEN_BLOCK = 12,
};
```

2.3 Network Event Structure

```
struct net_block_event {
    // Process context (same as file events)
    __u32 pid;
    __u32 ppid;
    __u64 start_time;
    __u64 parent_start_time;
    __u64 cgid;
    char comm[16];

    // Network specific
    __u8 family;        // AF_INET or AF_INET6
    __u8 protocol;      // IPPROTO_TCP, IPPROTO_UDP
    __u16 local_port;
    __u16 remote_port;
    __u8 direction;     // 0=egress, 1=ingress/bind
    __u8 _pad;

    union {
        __be32 ipv4;
        __u8 ipv6[16];
    } local_addr;

    union {
        __be32 ipv4;
        __u8 ipv6[16];
    } remote_addr;
    char action[8];     // "AUDIT" or "KILL"
    char rule_type[16]; // "ip", "port", "cidr"
};
```

---

3. BPF Map Design

3.1 New Maps

3.1.1 IPv4 Deny Map (Exact Match)

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 65536);
    __type(key, __be32);           // IPv4 address
    __type(value, __u8);           // flags (reserved)
} deny_ipv4 SEC(".maps");
```

3.1.2 IPv6 Deny Map (Exact Match)

```
struct ipv6_key {
    __u8 addr[16];
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 65536);
    __type(key, struct ipv6_key);
    __type(value, __u8);
} deny_ipv6 SEC(".maps");
```

3.1.3 IPv4 CIDR Deny Map (LPM Trie)

```
struct ipv4_lpm_key {
    __u32 prefixlen;
    __be32 addr;
};

struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(max_entries, 16384);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __type(key, struct ipv4_lpm_key);
    __type(value, __u8);
} deny_cidr_v4 SEC(".maps");
```

3.1.4 IPv6 CIDR Deny Map (LPM Trie)

```
struct ipv6_lpm_key {
    __u32 prefixlen;
    __u8 addr[16];
};

struct {
    __uint(type, BPF_MAP_TYPE_LPM_TRIE);
    __uint(max_entries, 16384);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __type(key, struct ipv6_lpm_key);
    __type(value, __u8);
} deny_cidr_v6 SEC(".maps");
```

3.1.5 Port Deny Map

```
struct port_key {
    __u16 port;
    __u8 protocol;  // IPPROTO_TCP=6, IPPROTO_UDP=17, 0=any
    __u8 direction; // 0=egress, 1=bind, 2=both
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 4096);
    __type(key, struct port_key);
    __type(value, __u8);
} deny_port SEC(".maps");
```

3.1.6 IP:Port Combo Deny Map

```
struct ip_port_key_v4 {
    __be32 addr;
    __u16 port;
    __u8 protocol;
    __u8 _pad;
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 32768);
    __type(key, struct ip_port_key_v4);
    __type(value, __u8);
} deny_ip_port_v4 SEC(".maps");
```

3.1.7 Network Statistics Maps

```
// Per-IP block counts
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_HASH);
    __uint(max_entries, 16384);
    __type(key, __be32);  // IPv4
    __type(value, __u64);
} net_ip_stats SEC(".maps");

// Per-port block counts
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_HASH);
    __uint(max_entries, 4096);
    __type(key, __u16);
    __type(value, __u64);
} net_port_stats SEC(".maps");

// Global network block stats
struct net_stats_entry {
    __u64 connect_blocks;
    __u64 bind_blocks;
    __u64 listen_blocks;
    __u64 ringbuf_drops;
};

struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __uint(max_entries, 1);
    __type(key, __u32);
    __type(value, struct net_stats_entry);
} net_block_stats SEC(".maps");
```

3.2 Map Pinning Paths

```
// New pin paths in types.hpp
inline constexpr const char* kDenyIpv4Pin = "/sys/fs/bpf/aegisbpf/deny_ipv4";
inline constexpr const char* kDenyIpv6Pin = "/sys/fs/bpf/aegisbpf/deny_ipv6";
inline constexpr const char* kDenyCidrV4Pin = "/sys/fs/bpf/aegisbpf/deny_cidr_v4";
inline constexpr const char* kDenyCidrV6Pin = "/sys/fs/bpf/aegisbpf/deny_cidr_v6";
inline constexpr const char* kDenyPortPin = "/sys/fs/bpf/aegisbpf/deny_port";
inline constexpr const char* kDenyIpPortV4Pin = "/sys/fs/bpf/aegisbpf/deny_ip_port_v4";
inline constexpr const char* kNetBlockStatsPin = "/sys/fs/bpf/aegisbpf/net_block_stats";
inline constexpr const char* kNetIpStatsPin = "/sys/fs/bpf/aegisbpf/net_ip_stats";
inline constexpr const char* kNetPortStatsPin = "/sys/fs/bpf/aegisbpf/net_port_stats";
```

4. Policy Schema Extension

4.1 Extended Policy Format

```
version=2

[deny_path]
/etc/shadow
/etc/passwd

[deny_inode]
8388609:131073

[allow_cgroup]
/sys/fs/cgroup/system.slice/docker.service

# ===== NEW NETWORK SECTIONS =====

[deny_ip]
# Single IPs
192.168.1.100
10.0.0.1
2001:db8::1

[deny_cidr]
# CIDR ranges
10.0.0.0/8
192.168.0.0/16
2001:db8::/32

[deny_port]
# port[:protocol[:direction]]
# protocol: tcp, udp, any (default: any)
# direction: egress, bind, both (default: both)
22
3389:tcp:egress
53:udp:egress

[deny_ip_port]
# ip:port[:protocol]
192.168.1.1:443
10.0.0.1:22:tcp

[allow_egress]
# Explicit egress allowlist (if deny-by-default mode)
# Only consulted if network_default=deny
```

```
8.8.8.8:53:udp
1.1.1.1:53:udp
```

4.2 Policy Struct Extension

```
// In types.hpp
struct NetworkPolicy {
    std::vector<std::string> deny_ips;          // Exact IPs
    std::vector<std::string> deny_cidrs;        // CIDR ranges
    std::vector<PortRule> deny_ports;           // Port rules
    std::vector<IpPortRule> deny_ip_ports;      // IP:port combos
    std::vector<std::string> allow_egress;      // Allowlist (optional)
    bool network_enabled = false;
    bool default_deny_egress = false;           // Future: deny-by-default mode
};

struct PortRule {
    uint16_t port;
    uint8_t protocol;    // 0=any, 6=tcp, 17=udp
    uint8_t direction;   // 0=egress, 1=bind, 2=both
};

struct IpPortRule {
    std::string ip;
    uint16_t port;
    uint8_t protocol;
};

struct Policy {
    int version = 0;
    // Existing file rules
    std::vector<std::string> deny_paths;
    std::vector<InodeId> deny_inodes;
    std::vector<std::string> allow_cgroup_paths;
    std::vector<uint64_t> allow_cgroup_ids;
    // New network rules
    NetworkPolicy network;
};
```

---

5. User Space Components

5.1 CLI Extensions

```
# Network deny management
aegisbpf network deny add --ip 192.168.1.100
aegisbpf network deny add --cidr 10.0.0.0/8
aegisbpf network deny add --port 22 --protocol tcp --direction egress
aegisbpf network deny add --ip-port 192.168.1.1:443
aegisbpf network deny del --ip 192.168.1.100
aegisbpf network deny list
aegisbpf network deny clear

# Network statistics
aegisbpf network stats
aegisbpf network stats --by-ip
aegisbpf network stats --by-port

# Combined status
aegisbpf stats --all  # File + network stats
```

5.2 New Source Files

```
src/
  network_ops.hpp      # Network BPF map operations
  network_ops.cpp
  network_policy.hpp   # Network policy parsing
  network_policy.cpp
  network_events.hpp   # Network event handling
  network_events.cpp
  network_types.hpp    # Network-specific types
```

5.3 BpfState Extension

```
class BpfState {
public:
    // Existing...

    // New network maps
    bpf_map* deny_ipv4 = nullptr;
    bpf_map* deny_ipv6 = nullptr;
    bpf_map* deny_cidr_v4 = nullptr;
    bpf_map* deny_cidr_v6 = nullptr;
    bpf_map* deny_port = nullptr;
    bpf_map* deny_ip_port_v4 = nullptr;
    bpf_map* net_block_stats = nullptr;
    bpf_map* net_ip_stats = nullptr;
    bpf_map* net_port_stats = nullptr;

    // Reuse flags
    bool deny_ipv4_reused = false;
    bool deny_ipv6_reused = false;
    // ... etc
};
```

5.4 Event Handler Extension

```
// In events.cpp
void print_net_block_event(const NetBlockEvent& ev) {
    std::ostringstream oss;
    oss << "{\"type\":\"net_block\""
        << ",\"pid\":" << ev.pid
        << ",\"ppid\":" << ev.ppid
        << ",\"cgid\":" << ev.cgid
        << ",\"family\":\"" << (ev.family == AF_INET ? "ipv4" : "ipv6") << "\""
        << ",\"protocol\":\"" << protocol_name(ev.protocol) << "\""
        << ",\"remote_ip\":\"" << format_ip(ev) << "\""
        << ",\"remote_port\":" << ntohs(ev.remote_port)
        << ",\"direction\":\"" << (ev.direction ? "bind" : "egress") << "\""
        << ",\"action\":\"" << ev.action << "\""
        << ",\"rule_type\":\"" << ev.rule_type << "\""
        << ",\"comm\":\"" << json_escape(ev.comm) << "\"}";
    // ... output to stdout/journald
}
```

---

6. Prometheus Metrics Extension

```
// New metrics
aegisbpf_net_blocks_total{type="connect"}
aegisbpf_net_blocks_total{type="bind"}
aegisbpf_net_blocks_by_ip_total{ip="192.168.1.100"}
aegisbpf_net_blocks_by_port_total{port="22"}
aegisbpf_net_ringbuf_drops_total
aegisbpf_net_rules_total{type="ip"}
aegisbpf_net_rules_total{type="cidr"}
aegisbpf_net_rules_total{type="port"}
```

---

7. Implementation Phases

Phase 1: Core Infrastructure (Week 1-2)

- ☐ Add network event types and structures to BPF code
- ☐ Implement `deny_ipv4`, `deny_port` maps
- ☐ Implement `socket_connect` LSM hook (IPv4 only)
- ☐ Add basic userspace event handling
- ☐ Unit tests for map operations

Phase 2: Full IPv4 Support (Week 2-3)

- □ Implement CIDR matching with LPM trie
- □ Implement `socket_bind` hook
- □ Add policy parser extensions
- □ CLI commands for network rules
- □ Integration tests

Phase 3: IPv6 Support (Week 3-4)

- □ Add IPv6 maps and hook logic
- □ Test dual-stack scenarios
- □ Performance benchmarking

Phase 4: Production Hardening (Week 4-5)

- □ Prometheus metrics integration
- □ Journald event logging
- □ Documentation
- □ Helm chart updates
- □ Load testing

---

8. Performance Considerations

8.1 Hot Path Optimization

```
// Fast-path check order in socket_connect:
// 1. Cgroup allowlist (HASH lookup) - skip if trusted
// 2. Bloom filter for IPs (optional, reduces false lookups)
// 3. Exact IP match (HASH)
// 4. Port match (HASH)
// 5. CIDR match (LPM) - most expensive, do last
```

8.2 Expected Performance

Operation	Latency
Cgroup allowlist check	~50-100ns
Exact IP lookup	~50-150ns
Port lookup	~50-100ns
CIDR LPM lookup	~200-500ns
Total (worst case)	**~500-900ns**

8.3 Map Sizing Guidelines

Map	Max Entries	Memory
deny_ipv4	65,536	~512KB
deny_ipv6	65,536	~1.5MB
deny_cidr_v4	16,384	~256KB
deny_cidr_v6	16,384	~512KB
deny_port	4,096	~32KB

---

9. Testing Strategy

9.1 Unit Tests

- Map operation correctness
- Policy parsing (valid/invalid inputs)
- IP/CIDR parsing utilities

9.2 Integration Tests

```
# Test egress blocking
aegisbpf network deny add --ip 1.2.3.4
curl -m 1 http://1.2.3.4  # Should fail/timeout

# Test port blocking
aegisbpf network deny add --port 8080 --direction egress
curl -m 1 http://localhost:8080  # Should fail

# Test cgroup bypass
aegisbpf allow add /sys/fs/cgroup/trusted.slice
# Process in trusted.slice should connect despite rules
```

9.3 Performance Tests

- Connection rate with rules loaded
- Latency impact measurement
- Memory usage under load

---

10. Security Considerations

10.1 Bypass Prevention

- DNS bypass: Consider optional DNS query monitoring
- Localhost: Decide on loopback policy (default: allow)
- IPv4-mapped IPv6: Handle `::ffff:` addresses

10.2 Fail-Safe Behavior

- Break-glass mode disables network enforcement
- Deadman switch reverts to audit mode
- Survival allowlist doesn't apply to network (no "critical IPs")

---

11. Future Extensions

1. **DNS Monitoring**: Track DNS queries for domain-based rules
2. **Connection Tracking**: Stateful connection monitoring
3. **Bandwidth Limits**: Rate limiting per process/cgroup
4. **Network Namespaces**: Container-aware policies
5. **Default-Deny Mode**: Explicit allowlist for egress

Appendix A: Complete BPF Hook Implementation

```
// bpf/aegis_net.bpf.c (new file or merged into aegis.bpf.c)

SEC("lsm/socket_connect")
int BPF_PROG(aegis_socket_connect, struct socket *sock,
             struct sockaddr *address, int addrlen)
{
    if (!address)
        return 0;

    __u16 family = address->sa_family;
    if (family != AF_INET && family != AF_INET6)
        return 0;

    __u64 cgid = bpf_get_current_cgroup_id();
    if (is_cgroup_allowed(cgid))
        return 0;

    __u8 audit = get_effective_audit_mode();
    __be32 ipv4 = 0;
    __u16 port = 0;

    if (family == AF_INET) {
        struct sockaddr_in *sin = (struct sockaddr_in *)address;
        ipv4 = BPF_CORE_READ(sin, sin_addr.s_addr);
        port = BPF_CORE_READ(sin, sin_port);

        // Check exact IP
        if (bpf_map_lookup_elem(&deny_ipv4, &ipv4))
            goto deny;

        // Check CIDR
        struct ipv4_lpm_key lpm_key = {
            .prefixlen = 32,
            .addr = ipv4
        };
        if (bpf_map_lookup_elem(&deny_cidr_v4, &lpm_key))
            goto deny;
    }

    // Check port
    struct port_key pk = {
        .port = bpf_ntohs(port),
        .protocol = 0,  // any
        .direction = 0  // egress
    };
    if (bpf_map_lookup_elem(&deny_port, &pk))
        goto deny;

    return 0;

deny:
    increment_net_connect_stats();
    emit_net_block_event(cgid, family, ipv4, port, audit);

    if (!audit)
        bpf_send_signal(SIGKILL);

    return audit ? 0 : -EPERM;
}
```

Appendix B: Migration Path

For existing deployments:

1. **Policy version bump**: `version=2` indicates network support
2. **Backward compatible**: `version=1` policies work unchanged
3. **Optional activation**: Network hooks only attach if policy has network rules
4. **Gradual rollout**: Start with `--network-audit-only` flag

AegisBPF Production Roadmap (Market-Leadership Plan)

This roadmap defines the execution plan to reach a trusted, market-leading security product. It is intentionally strict: no new surface area is added until the current phase gates are met.

Scope Freeze (v1 Golden Contract)

Contract (v1): “Aegis enforces deny decisions for file open/exec using inode-first rules on selected cgroups, with audited policy provenance and safe rollback.”

Non-goals (v1): - Network deny enforcement - Kill escalation modes - Broad SIEM integrations - Multi-surface enforcement beyond file open/exec

These items are planned only after v1 gates are met.

Phase A — Defensibility Core

Goal: Semantically defensible enforcement.

A1. Versioned Security Contract

- Threat model and trust boundaries
- Explicit non-goals
- TOCTOU stance

Gate A1: Contract published and referenced from README.

A2. Enforcement Semantics Whitepaper (v1)

- Path vs inode authority
- Namespace semantics
- Bind/overlay/rename/link behavior
- Bypass catalog (accepted/mitigated/roadmap)

Gate A2: `docs/POLICY_SEMANTICS.md` published and versioned. **Gate A2:** `docs/BYPASS_CATALOG.md` published and versioned.

A3. Edge-Case Compliance Suite

- = 60 enforced e2e scenarios (baseline)
- Coverage basis set: symlink swap, hardlink, rename, overlayfs, mount ns
- Golden policy vectors + deterministic replay

Gate A3: Suite passes in CI with 0 failed checks. **Gate A3:** Scenario count >= 60. **Gate A3:** `docs/EDGE_CASE_COMPLIANCE_SUITE.md` published.

A4. Verifier / CO-RE Robustness

- Verifier budget regression tests
- Attach-failure matrix
- Explicit partial-attach states
- BTF detection + clear errors

Gate A4: 0 silent partial attaches in tests.

Phase B — Production Survivability

Goal: Will not break production.

B1. Safe Defaults + Rollback Guarantees

- Audit-first default
- Atomic policy swap + deterministic rollback
- Break-glass tests

Gate B1: 100% rollback success in 1,000-iteration stress test. **Gate B1:** p99 rollback <= 5s.

B2. Degraded-Mode Contract

- Explicit fail-open/fail-closed per hook
- Map full / ringbuf overflow behavior
- Backpressure + sampling policy

Gate B2: 0 undocumented failure modes in tests.

B3. Performance Credibility

- Pinned methodology (kernel, FS, workload)
- Metrics: p50/p95/p99 overhead, CPU, drop rate
- Soak tests >= 6h

Gate B3: p99 syscall overhead < 5% (initial gate). **Gate B3:** unexplained drop rate < 0.1%.

B4. Observability + Diagnostics

- Versioned log schema
- Metrics contract
- Enforcement-active signal
- `aegis doctor` CLI

Gate B4: doctor passes on 2 kernels. **Gate B4:** one pilot deployment completed (real environment). **Gate B4:** external review kickoff (scope + timeline defined).

Phase C — Trust & Adoption

Goal: Externally credible and preferred.

C1. Meta-Security & Supply Chain

- Mandatory policy signatures in enforce
- Key rotation + revocation drills
- Signed releases + SBOM + provenance

Gate C1: signature lifecycle tests pass; SBOM + signatures present.

C2. UX & Explainability

- Policy linter with fixes
- Dry-run traces
- `aegis explain <event>`

Gate C2: time-to-correct-policy <= baseline from 3 pilot users.

C3. Deployment Hardening

- Hardened systemd / Helm defaults
- Air-gapped install path
- Secrets handling guide

Gate C3: production deployment blueprint published. Blueprint: `docs/PRODUCTION_DEPLOYMENT_BLUEPRINT.md`.

C4. Governance & Evidence Pack

- Strict DoD for enforcement changes
- CVE workflow + changelog discipline
- Independent security review (mid-cycle)
- Pilot case study (mid-cycle)
- Evidence pack per release (links to CI + artifacts)

Gate C4: Evidence pack published for each release. **Gate C4:** external compliance suite run by third party.

Execution Strategy (Critical Path)

1) Decide CI realism now:

- Self-hosted runners with fixed kernels (preferred), or
- VM-kernel CI for determinism

2) Pull external validation earlier:

- One pilot and one external review during Phase B

KPI Summary (Non-negotiable Gates)

- = 60 enforced e2e scenarios (Phase A baseline)
- = 100 enforced e2e scenarios (Phase B authority baseline)
- = 120 enforced e2e scenarios + external run (Phase C)
- = 4 kernels, >= 2 distro families
- Rollback success 100% (1,000 iterations)
- p99 rollback <= 5s
- p99 syscall overhead < 5% (Phase B gate)
- p99 syscall overhead < 3% (Phase C stretch)
- Unexplained drop rate < 0.1%
- 0 silent partial attaches

Differentiation KPIs (Adoption Drivers)

At least one of the following must be tracked in pilots: - Time-to-correct-policy (median minutes) - Time-to-diagnose-deny (median minutes) - Operator cognitive load (steps required to resolve incident)

Reference Enforcement Slice (Decision-Grade)

One inode-first deny path is designated as the decision-grade reference. It must pass all Phase A/B gates and an independent review before any expansion of enforcement scope.

Canonical reference: `docs/REFERENCE_ENFORCEMENT_SLICE.md`.

Differentiation Commitment

At least one adoption KPI must improve by >= Z% vs baseline before any "market leadership" claim is made.

Kill Criteria (Hard Stop Conditions)

- If no pilot expansion or renewal within X months -> scope freeze + reassessment.
- If differentiation KPI does not beat baseline by Y% -> reposition or narrow use-case.

Market Leadership Plan

This plan turns AegisBPF market-leadership goals into measurable, release-blocking gates.

Flagship category and differentiation

Flagship category to win first:

> Inode-first Linux file enforcement for production workloads, with explicit semantics, safe rollback, and auditable policy provenance.

Differentiation claims (must be artifact-backed):

1. **Defensible semantics:** policy decisions are deterministic and documented.
2. **Operational survivability:** rollout and rollback are safe under failure.
3. **Low overhead with explainability:** overhead stays bounded and every deny decision can be explained.

Execution model (3 super-phases)

Super-Phase A - Defensibility Core

Goal: Semantically defensible enforcement.

Workstreams:

- Final threat model and trust-boundary contract (`docs/THREAT_MODEL.md`).
- Final policy semantics contract (`docs/POLICY_SEMANTICS.md`).
- Bypass catalog with disposition (accepted/mitigated/planned).
- Edge-case kernel e2e basis set and replay harness.
- Verifier/CO-RE robustness: attach matrix, feature detection, map-pressure behavior.

Numeric gate (all required):

- `>=100` kernel e2e enforcement cases.
- `>=4` kernel targets across `>=2` distro families.
- `0` ambiguous policy decisions in golden decision vectors.
- `0` silent partial attaches (explicit hard-fail or explicit degraded mode).

Super-Phase B - Production Survivability

Goal: The system does not break production when stressed or misconfigured.

Workstreams:

- Audit-first safe defaults and explicit enforce promotion.
- Canary -> ramp workflows and emergency break-glass controls.
- Atomic policy swap and last-known-good rollback guarantees.

- Degraded-mode contracts for map pressure, ring buffer drops, and logging failures.
- Pinned benchmark + soak/reliability tracks.
- Operational observability for attach/enforcement/drop/rollback states.

Numeric gate (all required):

- Rollback success `100%` over `1,000` stress iterations.
- Rollback completion `p99 <= 5s`.
- Unexplained event drops `<0.1%` at declared sustained load.
- Syscall overhead `p95 <= 5%` (stretch target `<=3%`).
- `0` false-green health states (process alive while enforcement inactive).

Super-Phase C - Trust and Adoption

Goal: External trust signals and real production adoption.

Workstreams:

- Signed policy lifecycle (rotate/revoke/anti-rollback) with drills.
- Signed releases + SBOM + provenance + reproducibility checks.
- Explainability tooling (`why denied` chain) and policy-lint ergonomics.
- Hardened deployment guides (systemd + Kubernetes + air-gapped path).
- Governance and disclosure operations with SLA-backed response.
- Independent review and design-partner pilots.

Numeric gate (all required):

- `0` unresolved critical external-review findings.
- `100%` release artifacts signed with SBOM + provenance attached.
- Explainability coverage for every deny decision class.
- `>=2` pilot environments with weekly evidence reports.

External validation timing

External validation is not deferred to the end of the program:

- Independent review scoping starts in Super-Phase A.
- Pilot deployments start in Super-Phase B.
- Pilot and reviewer feedback can reprioritize backlog before Super-Phase C closes.
- External review prep checklist: `docs/EXTERNAL_REVIEW_PREP.md`.
- External review closure tracker: `docs/EXTERNAL_REVIEW_STATUS.md`.
- Weekly pilot evidence template: `docs/PILOT_EVIDENCE_TEMPLATE.md`.
- Weekly pilot evidence reports: `docs/pilots/`.

Claim discipline policy

Every externally-visible claim must link to:

1. Spec section
2. Automated test
3. CI artifact
4. Runbook/operator guidance

Claims must be tagged as `ENFORCED`, `AUDITED`, or `PLANNED`. Unmapped claims are release blockers.

Program cadence

- Weekly: KPI/status review, risk register update, top-3 blocker burn-down.
- Monthly: claim audit (remove or downgrade unsupported claims).
- Per release: go/no-go uses `docs/MARKET_SCORECARD.md` plus `docs/GO_LIVE_CHECKLIST.md`.

Immediate implementation backlog

1. Lock KPI thresholds in contract tests and release templates.
2. Expand edge-case e2e matrix to satisfy Super-Phase A floor.
3. Harden degraded-mode and rollback tests for Super-Phase B gates.
4. Run pilot onboarding with `docs/PILOT_EVIDENCE_TEMPLATE.md`, publish weekly reports in `docs/pilots/`, and track reviewer closure in `docs/EXTERNAL_REVIEW_STATUS.md`.