Zero-Trust Validator Operations and Service Identity for Blockchain Infrastructure

Zach Kelling

Lux Industries

2025

Abstract

Traditional blockchain validator deployments grant implicit trust based on network position: if a process can reach the signing key and the P2P port, it is assumed to be the validator. This assumption fails in shared infrastructure, containerized environments, and multi-tenant Kubernetes clusters. We present Lux Network's zero-trust validator architecture, where every service interaction—signing requests, key access, peer connections, RPC calls—is authenticated via cryptographic workload identity, authorized by policy, and continuously verified. No component trusts another based on network adjacency. We describe the identity model (SPIFFE-compatible workload identifiers bound to Q-Chain NodeIDs), the authorization framework (capability-based access to signer services), and the continuous authorization protocol that re-validates identity on every signing request. The architecture is deployed on Lux's Kubernetes-based validator infrastructure across mainnet, testnet, and devnet.

1 Introduction

A Lux validator consists of multiple cooperating components: the `luxd` node process, EVM plugin processes, signer services (for BLS and Ringtail keys), metrics exporters, and operational tooling. In a traditional deployment, these components trust each other implicitly—they share a filesystem, a network namespace, or a pod boundary. This implicit trust creates lateral movement opportunities: a compromised metrics exporter can access the signing key, a bug in an RPC handler can reach the consensus engine.

Zero-trust architecture eliminates implicit trust. Every interaction between components is:

1. **Authenticated**: The caller proves its identity cryptographically.
2. **Authorized**: The caller's identity is checked against a policy that specifies what actions it may perform.
3. **Continuously verified**: Authorization is re-evaluated on every request, not cached from a prior session.

This paper describes how these principles are applied to Lux validator operations, with particular focus on the signer service—the component that holds the validator's BLS and Ringtail private keys and must be protected above all else.

2 Workload Identity Model

2.1 Identity Structure

Each validator component receives a SPIFFE-compatible workload identity:

`spiffe://lux.network/ns/<namespace>/sa/<service-account>`

Examples:

- `spiffe://lux.network/ns/lux-mainnet/sa/luxd-0` — Node process
- `spiffe://lux.network/ns/lux-mainnet/sa/signer-0` — Signer service
- `spiffe://lux.network/ns/lux-mainnet/sa/evm-plugin` — EVM plugin

These identities are issued by the cluster's SPIRE server and bound to Kubernetes service accounts. The SVID (SPIFFE Verifiable Identity Document) is a short-lived X.509 certificate rotated every 60 seconds.

2.2 Binding to Q-Chain NodeID

The workload identity is bound to the validator's Q-Chain NodeID through an attestation:

Definition 1 (NodeID Binding)**.** *A NodeID binding* $\mathcal{B} = (SPIFFE\_ID, NodeID, \sigma_{BLS}, \sigma_{MLDSA})$ *proves that the workload identified by SPIFFE_ID is authorized to act on behalf of the validator with the given NodeID. The signatures are produced by the validator's registered keys.*

This binding is registered on Q-Chain as part of validator enrollment and verified by any component that receives a request claiming to act on behalf of a specific NodeID.

3 Signer Service Architecture

3.1 Isolation

The signer service is the sole component with access to private key material. It runs as a separate process (or container) with:

- No network access except a Unix domain socket (or mTLS-authenticated gRPC).
- No filesystem access except the key material directory (read-only after initialization).
- No capability to make outbound connections.
- Memory encryption via SEV-SNP or equivalent (where available).

3.2 Request Authentication

Every signing request must present a valid SVID and a request-specific nonce:

1: **function** HandleSignRequest(req)
2: $svid \leftarrow$ ExtractSVID(req.tlsContext)
3: **if** ¬ValidateSVID($svid$) **then**
4: **return** UNAUTHENTICATED
5: **end if**
6: **if** ¬AuthorizeSignRequest($svid$.spiffeID, req.operation) **then**
7: **return** UNAUTHORIZED
8: **end if**
9: AuditLog($svid$.spiffeID, req.operation, req.digest)
10: **return** Sign(req.digest)
11: **end function**

3.3 Authorization Policy

Signing authorization is defined by a capability matrix:

The policy is declarative and stored as a Kubernetes ConfigMap. Changes require a signed commit and a rolling restart of the signer service.

Caller Identity	Operation	Allowed	Rate Limit
luxd-0	SignBlock	Yes	1000/s
luxd-0	SignWarp	Yes	100/s
evm-plugin	SignBlock	No	—
evm-plugin	SignTx	No	—
signer-0	KeyExport	No	—
admin-tool	RotateKey	Yes	1/day

Table 1: Signer service authorization policy excerpt.

4 Continuous Authorization

Unlike session-based authentication, zero-trust demands that every request is independently authorized. This is implemented through:

1. **Short-lived SVIDs**: 60-second lifetime. A compromised credential is usable for at most one minute.
2. **Per-request policy evaluation**: The signer service evaluates the authorization policy on every signing request, not once per connection.
3. **Anomaly detection**: Signing rate, message entropy, and request timing are monitored. Deviations trigger automatic signer lockout pending manual review.

function ContinuousAuth(req, history)
  $rate \leftarrow$ RequestRate(req.callerID, $history$)
  if $rate >$ RateLimit(req.callerID, req.operation) **then**
    LockSigner()
    Alert("Rate limit exceeded")
    return DENIED
  end if
  $entropy \leftarrow$ MessageEntropy(req.digest)
  if $entropy <$ MinEntropy **then**
    Alert("Low entropy signing request")
  end if
  return ALLOWED
end function

5 Peer Connection Authentication

Validator-to-validator connections use mTLS with hybrid certificates [1]. The zero-trust layer adds:

1. **NodeID verification**: The connecting peer's TLS certificate must contain a NodeID that is currently registered and staked on Q-Chain.
2. **Epoch binding**: The certificate must be valid for the current epoch. Certificates from future or expired epochs are rejected.
3. **IP binding** (optional): For validators that opt in, the certificate is bound to a specific IP range registered on Q-Chain. Connections from unexpected IPs are rejected.

6 Kubernetes Deployment Model

Lux validators run on Kubernetes (DOKS clusters: lux-k8s for mainnet, lux-test-k8s for testnet, lux-dev-k8s for devnet). The zero-trust architecture maps to Kubernetes primitives:

Zero-Trust Concept	**K8s Implementation**
Workload identity	SPIRE + K8s service accounts
Network isolation	NetworkPolicy (deny-all default)
Signer isolation	Separate pod, no network egress
Secret management	KMS (kms.hanzo.ai) $\rightarrow$ KMSSecret CRDs
Policy enforcement	OPA Gatekeeper admission controller
Audit logging	Structured logs $\rightarrow$ Loki

Table 2: Zero-trust to Kubernetes mapping.

All secrets are managed through KMS (kms.hanzo.ai), never stored in Kubernetes Secrets directly. The KMSSecret CRD syncs encrypted secrets from KMS and mounts them as volumes.

7 Security Analysis

Theorem 1 (Lateral Movement Containment)**.** *Compromise of any single component (other than the signer) does not provide access to signing key material.*

Proof. The signer service accepts requests only from authenticated callers with valid SVIDs and matching authorization policy. A compromised luxd process cannot sign arbitrary messages beyond its authorized operations. A compromised EVM plugin has no signing authorization at all. The signer's network policy allows inbound connections only from the luxd pod's service account. No component can reach the key material filesystem. □

8 Conclusion

Zero-trust validator operations eliminate the largest class of infrastructure attacks: those that exploit implicit trust between co-located components. By requiring cryptographic authentication and continuous authorization for every interaction—especially signing requests—Lux's architecture ensures that compromising one component does not cascade into a full validator compromise. The signer service, as the holder of key material, is protected by multiple independent controls: workload identity, authorization policy, rate limiting, anomaly detection, and network isolation.

References

[1] Kelling, Z. (2024). *Hybrid Certificate Chains and Post-Quantum Trust Anchors for Validator Networks*. Lux Industries Research.

[2] Kelling, Z. (2025). *Quasar: Quantum-Secure Multi-Engine Consensus with Triple-Proof Quantum Finality*. Lux Industries Research.

[3] Kelling, Z. (2025). *Secure Enclave, HSM, and Threshold Boundary Design*. Lux Industries Research.

[4] SPIFFE Authors (2024). *Secure Production Identity Framework for Everyone (SPIFFE)*. CNCF.

[5] Rose, S. et al. (2020). *Zero Trust Architecture*. NIST SP 800-207.

[6] Kelling, Z. (2024). *Cryptographic Agility Architecture for Long-Lived Blockchain Systems*. Lux Industries Research.