
Simplicity through Transparency

How the PQC Migration is Streamlining Sigstore

Hayden Blauzvern
Google, Tech Lead





"Ensure that all artifacts and metadata used within any software supply chain have verifiable, transparent integrity and provenance, so that consumers can make explicit trust decisions." - Sigstore Mission

Sigstore: Architecture Overview

Cosign / SDKs

CLI to sign and verify blobs, containers, and attestations.

Or, use a per-ecosystem SDK, e.g. sigstore-python

Fulcio

Code-signing CA. Exchanges OIDC tokens for short-lived certificates.

Rekor

Signature transparency log. Stores artifact hashes, certificates & keys, and signatures. Built with Tesseract.

Timestamp Authority

Signed timestamp to demonstrate key ownership at signing.

How Identity-Based Signing Works

1. Auth

Client gets OIDC token (e.g. GitHub, Google)

2. Cert Request

Fulcio issues short-lived cert binding identity to ephemeral key

3. Sign Artifact

Client signs artifact with ephemeral key, timestamps signature

4. Transparency

Log signature to Rekor & get inclusion proof

5. Discard Key

Key destroyed; return signature bundle

Sigstore: Architecture Overview

Cosign / SDKs

CLI to sign and verify blobs, containers, and attestations.

Or, use a per-ecosystem SDK, e.g. sigstore-python

Rekor

Signature transparency log. Stores artifact hashes, certificates & keys, and signatures. Built with Tesseract.

How Key-Based Signing Works

1. Sign Artifact

Client signs artifact using self-managed private key (no Fulcio/OIDC)



2. Transparency

Log signature to Rekor & get inclusion proof



3. Output Bundle

Return signature bundle

Sigstore Ecosystem Adoption

- ✓ **npm:** First registry with Sigstore-backed SLSA build provenance.
- ✓ **PyPI:** PEP 740 trusted publishing attestations.
- ✓ **Maven Central:** Sigstore signature support.
- ✓ **GitHub Actions:** Build provenance for any Actions artifact and signed release attestations.
- ✓ **Homebrew:** Build provenance for all homebrew-core bottles, in beta.
- ✓ **RubyGems:** Trusted publishing attestations.



*"Ensure that **all** artifacts and metadata used within any software supply chain have verifiable, transparent integrity and provenance, so that consumers can make explicit trust decisions." - Sigstore Mission*

Growth Challenges for Public Instance

Current infrastructure & clients face several constraints, necessitating change.

High Cost

Operating the current infrastructure stack is expensive and costs (infra, storage, egress) are growing.

Growing Request & Entry Sizes

Existing requests for many entries (e.g. attestations) can be large. Current credential formats are unbounded in size, and will get larger with PQC.

Backend Bottlenecks

The current Trillian-based logging backend has hit horizontal and vertical scaling limitations, leading to throughput constraints and latency issues at high volumes.

Log Resilience

The system is at risk of log poisoning/spam, which risks severe collateral damage if bad entries trigger a log takedown that impacts ecosystems concurrently.

Scalable Verifiability

Downloading and processing entire logs with billions of entries to verify specific signatures is prohibitively resource-intensive.

Cryptographic Algorithm Agility

We want to be able to evolve algorithms, but executing changes across diverse clients is complex, and introducing too much optionality can become challenging to support.

**The transition to Post-Quantum
Cryptography presents a unique
opportunity for change.**

The transition to Post-Quantum Cryptography presents a unique opportunity for change.

01 / Cryptography

Post-quantum cryptography

(PQC): Redesign Sigstore's protocol for PQC.

02 / Architecture

A necessary breaking change: This transition provides the perfect moment to execute other architectural improvements that we wouldn't normally make as single changes. PQC presents an incentive to encourage adoption of new infrastructure.

03 / Simplify

Align with existing transparency standards and simplify the overall architecture.

**How do we make
a change to support
PQC?**

Minimize the Frequency of Breaking Changes

High-Level Requirements



Do Not Break Existing Users

New infrastructure will be opt-in. SDKs will provide new APIs. Minimize the likelihood of another breaking change by reducing complexity now.



Standardized Leaf Format

Simple and standardized format for "identity" or "signature" transparency, shared with Sigsum.



Transparency at Scale

Leveraging tile-backed logs to support high-throughput public transparency and auditability. Reuses Rekor v2 infrastructure.



Minimal Log Footprint

Small log entry size results in a manageable infrastructure footprint.



Abuse Resilience & Privacy

Inherent resilience against log poisoning while removing sensitive identifiers from entries.



Post-quantum Ready

Seamless migration path designed for ML-DSA.

Key Requirements for OSS Infrastructure

Log Entry Size Reduction

Drastically reducing log entry size is critical to solve storage scaling issues.

This is especially vital given the significantly larger size of ML-DSA signatures and public keys.

Log Resilience

We must implement comprehensive mitigation strategies against log poisoning.

Active safeguards are required to protect the integrity of the public infrastructure.

Rekor Leaf Format

Leaf & request format (roughly) for "hashedrekord"

```
{  
  bytes digest = 1;  
  bytes signature = 2;  
  oneof verifier {  
    bytes certificate = 3;  
    bytes public_key = 4;  
  }  
}
```

Rekor Leaf Format

Leaf & request format (roughly) for "hashedrekord"

```
{  
  bytes digest = 1; // What if this is meaningful?  
  bytes signature = 2; // Large with ML-DSA  
  oneof verifier {  
    bytes certificate = 3; // User-defined extensions  
    bytes public_key = 4; // Large with ML-DSA  
  }  
}
```

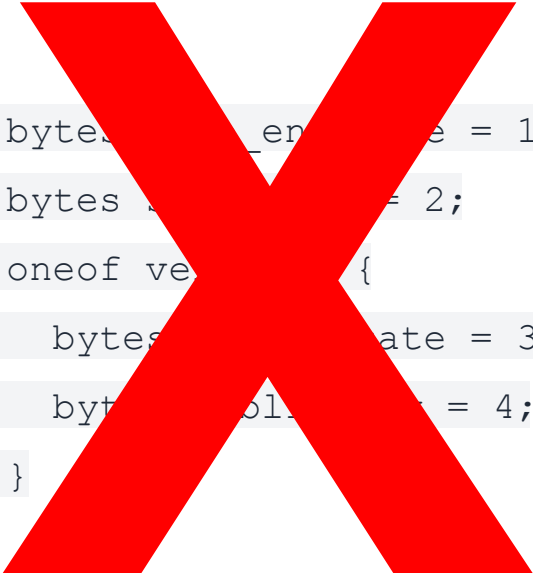
Rekor Leaf Format

Leaf & request format (roughly) for "dsse"

```
{  
  bytes env_hash = 1; // Request unbounded in size  
  bytes signature = 2;  
  oneof verifier {  
    bytes certificate = 3;  
    bytes public_key = 4;  
  }  
}
```

Rekor Leaf Format

Leaf & request format (roughly) for "dsse"



```
{  
  bytes ..._en ... = 1; // Unbounded in size  
  bytes ... = 2;  
  oneof ver {  
    bytes ..._ate = 3;  
    bytes ..._bl ... = 4;  
  }  
}
```

"Identity Transparency" Leaf Format

Leaf format

```
0x01 ||
```

```
Hash(trust anchor / leaf owner) ||
```

```
Hash(message) ||
```

```
Hash(Hash(name1)) || Hash(Hash(value1)) ||
```

```
Hash(Hash(name2)) || Hash(Hash(value2)) ||
```

```
... ||
```

```
Hash(Hash(nameN)) || Hash(Hash(valueN)) ||
```

```
Hash(receipt)
```

"Identity Transparency" Leaf Format

Leaf format

```
0x01 ||
```

```
Hash(trust anchor) || <-- OIDC provider, e.g  
                        "https://accounts.google.com"
```

```
Hash(message) ||
```

```
Hash(Hash(name1)) || Hash(Hash(value1)) ||
```

```
Hash(Hash(name2)) || Hash(Hash(value2)) ||
```

```
... ||
```

```
Hash(Hash(nameN)) || Hash(Hash(valueN)) ||
```

```
Hash(receipt)
```

"Identity Transparency" Leaf Format

Leaf format

```
0x01 ||
```

```
Hash(trust anchor) ||
```

```
Hash(message) || <-- msg = SHA256(data)
```

```
Hash(Hash(name1)) || Hash(Hash(value1)) ||
```

```
Hash(Hash(name2)) || Hash(Hash(value2)) ||
```

```
... ||
```

```
Hash(Hash(nameN)) || Hash(Hash(valueN)) ||
```

```
Hash(receipt)
```

"Identity Transparency" Leaf Format

Leaf format

```
0x01 ||  
  
Hash(trust anchor) ||  
  
Hash(message) ||  
  
Hash(Hash(name1)) || <-- name1 = "CI workflow ref"  
  
Hash(Hash(value1)) || <-- value1 =  
  
"octo-org/octo-automation/.github/workflows/oidc.yml@refs/heads/main "  
  
... ||  
  
Hash(Hash(nameN)) || Hash(Hash(valueN)) ||  
  
Hash(receipt)
```

"Identity Transparency" Leaf Format

Leaf format

```
0x01 ||
```

```
Hash(trust anchor) ||
```

```
Hash(message) ||
```

```
Hash(Hash(name1)) || Hash(Hash(value1)) ||
```

```
Hash(Hash(name2)) || Hash(Hash(value2)) ||
```

```
... ||
```

```
Hash(Hash(nameN)) || Hash(Hash(valueN)) ||
```

Hash(receipt)

receipt provides non-repudiation for the log operator, proving the log received a valid *credential* to upload to the log

"Identity Transparency" Credential

Credential

```
message Credential {  
  oneof cred {  
    CertificateChain certificate_chain = 1;  
    PubKeyCred public_key = 2;  
  }  
}
```

```
message PubKeyCred {  
  bytes pub_key = 1;  
  bytes signature = 2;  
}
```

```
message CertificateChain {  
  repeated bytes cert = 1;  
  bytes signature = 2;  
}
```

"Identity Transparency" Log API

Credential

```
message Credential {  
  oneof cred {  
    CertificateChain  
    certificate_chain = 1;  
    PubKeyCred public_key = 2;  
  }  
}
```

Request

```
message Request {  
  bytes msg = 1; // SHA256(data)  
  Credential cred = 2;  
}  
  
message Response {  
  bytes tlog_proof = 1; c2sp.org/tlog-proof  
}
```

"Identity Transparency" Leaf Format

Leaf format, for an identity token

```
0x01 ||
```

```
Hash(trust_anchor) ||
```

```
Hash(message) ||
```

```
Hash(Hash(key1)) || Hash(Hash(value1)) ||
```

```
Hash(Hash(key2)) || Hash(Hash(value2)) ||
```

```
... ||
```

```
Hash(Hash(keyN)) || Hash(Hash(valueN)) ||
```

```
Hash(receipt) <-- Signature
```

Key Requirements for OSS Infrastructure

Solved with hashing everything!

Log Entry Size Reduction

Drastically reducing log entry size is critical to solve storage scaling issues.

This is especially vital given the significantly larger size of ML-DSA signatures and public keys.

Solved with hashing everything!

Log Resilience

We must implement comprehensive mitigation strategies against log poisoning.

Active safeguards are required to protect the integrity of the public infrastructure.

Key Requirements for OSS Infrastructure

Solved with hashing everything!

Log Entry Size Reduction

Drastically reducing log entry size is critical to solve storage scaling issues.

This is especially vital given the significantly larger size of ML-DSA signatures and public keys.

Solved with hashing everything!

Log Resilience

We must implement comprehensive mitigation strategies against log poisoning.

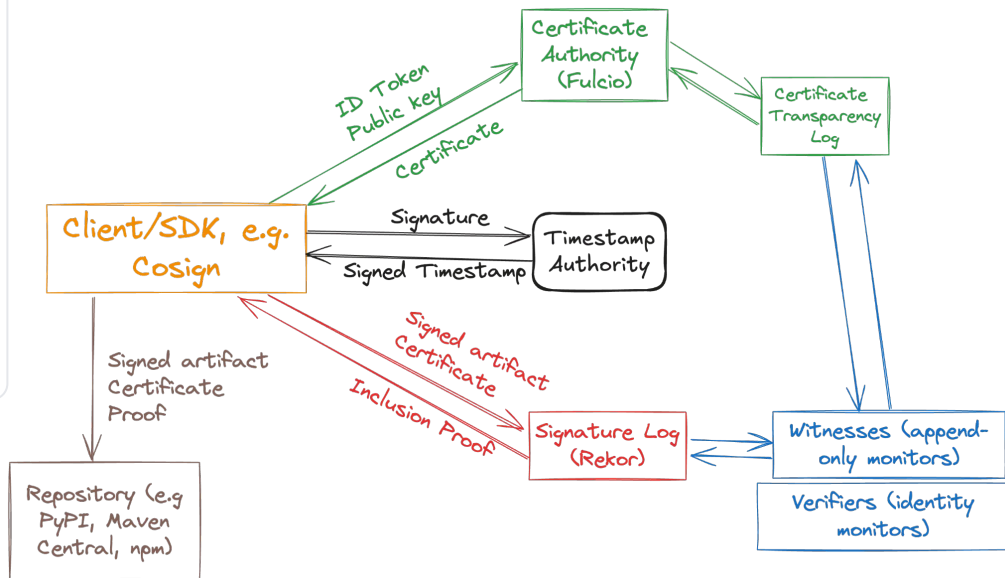
Active safeguards are required to protect the integrity of the public infrastructure.

...except not for signature bundles. Can we simplify?

Sigstore Today

Seven+ signatures per signing event:

- *artifact signature*
- *ephemeral public key*
- *CA certificate signature*
- *SCT promise*
- *signed timestamp*
- *log checkpoint signature*
- *N witness signatures*



Proposed Design: Truly "Keyless"

The insight: an OIDC token is already an identity credential. Let the log verify it directly.

SIGSTORE TODAY

Seven+ signatures per signing event:

*artifact signature · ephemeral public key · CA certificate
signature · SCT promise · signed timestamp · log checkpoint
signature · N witness signatures*



PQC SIGSTORE

Rekor verifies the OIDC token itself; identity claims go into the leaf. No certificates, no ephemeral keys, no timestamps.

The witnessed inclusion proof is the signature: 1 checkpoint signature + N witness cosignatures (+1 artifact signature only for managed keys).

"Identity Transparency" Credential

Credential

```
message Credential {  
  oneof cred {  
    string openid_connect_token = 1;  
    PubKeyCred public_key = 2;  
  }  
}
```

```
message PubKeyCred {  
  bytes pub_key = 1;  
  bytes signature = 2;  
}
```

"Identity Transparency" Log API

Credential

```
message Credential {  
  oneof cred {  
    string openid_connect_token = 1;  
    PubKeyCred public_key = 2;  
  }  
}  
  
message PubKeyCred {  
  bytes pub_key = 1;  
  bytes signature = 2;  
}
```

Request

```
message Request {  
  bytes msg = 1; // SHA256(data)  
  Credential cred = 2;  
}  
  
message Response {  
  bytes tlog_proof = 1; c2sp.org/tlog-proof  
}
```

"Identity Transparency" Leaf Format

Leaf format, for an identity token

```
0x01 ||
```

```
Hash(trust anchor) ||
```

```
Hash(message) ||
```

```
Hash(Hash(key1)) || Hash(Hash(value1)) ||
```

```
Hash(Hash(key2)) || Hash(Hash(value2)) ||
```

```
... ||
```

```
Hash(Hash(keyN)) || Hash(Hash(valueN)) ||
```

```
Hash(receipt) <-- ??? Zero-knowledge proof??
```

Proposed Design: Truly "Keyless"

The insight: an OIDC token is already an identity credential. Let the log verify it directly.

SIGSTORE TODAY

Seven+ signatures per signing event:

*artifact signature · ephemeral public key · CA certificate
signature · SCT promise · signed timestamp · log checkpoint
signature · N witness signatures*



PQC SIGSTORE

Rekor verifies the OIDC token itself; identity claims go into the leaf. No certificates, no ephemeral keys, no timestamps.

The witnessed inclusion proof is the signature: 1 checkpoint signature + N witness cosignatures (+1 artifact signature only for managed keys).



Hash-only leaves

C2SP identity-transparency leaf stores only hashes. **Mitigates log poisoning, redacts PII, keeps entries tiny** despite ML-DSA.



Built on standards

tlog-proof, tlog-checkpoint, tlog-witness,
tlog-policy, tlog-addenda



Same threat model

Trust rooted in identity. Usage is auditable.
Signed artifacts are immutable.

Hand-waving over...

- Public keys & signatures: Come to the next talk!
- End to end PQC: What about OpenID Connect?
- Optionality for private deployments: CA without log?
- Specifications: C2SP spec in progress, Sigstore spec to come!
- Sigstore public infrastructure PQC timeline: 2027 for log, 2029/2030 for turndown of current log

Thank you!