Spec: Sigstore Client

(*please join sigstore-dev@ to make changes/comments!*)

This document describes an architecture for using an automated certificate authority (specifically, Spec: Fulcio), timestamping service (RFC 3161), and transparency service (Spec: Transparency Service) for signing digital payloads.

Introduction

Having both an automated code-signing certificate authority for digital identities (CA; Spec: Fulcio) and a timestamping service (RFC 3161) enables payload signatures using short-lived, single-use certificates issued to those identities. A signer can request a certificate from the CA, sign a payload, and get the signature timestamped. Then, a verifier checks that the signature timestamp falls during the certificate's validity period. In this way, we decouple the *payload lifetime* from the *certificate lifetime*.

This approach has several advantages. First, signers no longer need to manage signing keys; they can generate them fresh for each signature. Second, the risk of a leaked signing key is lower: after the validity period expires, the key cannot be used to sign any payloads without the cooperation of the timestamping service. Finally, artifact lifetime and expiration can be managed independently of key lifetime.

In this approach, the certificate authority and timestamping services are trusted parties. To mitigate the security risks of centralization, we can introduce accountability in the form of *transparency*: public logs of all activity (certificates and signatures) that can be monitored for misbehavior. We implement this transparency property with a Certificate Transparency (CT) log (RFC 6962; see Spec: Fulcio for details on its integration with the identity service) and a transparency service (Spec: Transparency Service). The certificate authority will submit certificates to a CT log, and the signing client will submit payload metadata to the transparency service.

This document describes this flow in detail.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Terminology

Payload. The payload is the data to be signed (represented as a sequence of bytes). This might be a digital artifact, or an attestation/claim/metadata about a digital artifact.

Identity. A compact, virtual representation of an external actor, like a human or computer—for instance, a person's account on a web service, or the name of a computer system in a production environment.

Verification material. All data required to verify a signature on a payload, including the signature itself and materials from the various services.

Client. The software implementation used to sign and verify in Sigstore.

Parties

Signer. An entity who wishes to sign a payload using an identity they control.

Authentication System. a system which can authenticate the signer and in return provide an identity token. Examples include an OIDC Identity Provider. See Spec: Fulcio for requirements on the Authentication System. In particular, the identity tokens produced by the Authentication System SHOULD support a notion of “audience”—indicating the system for which the tokens are intended—and MUST support a notion of “subject” (indicating the identity). The tokens can be opaque to a signer *except* that the signer MUST be able to extract the subject.

Fulcio. A certificate authority compliant with Spec: Fulcio, configured to support the Authentication System.

Certificate Transparency Log. A service compliant with RFC 6962.

Timestamping Service. A service compliant with RFC 3161.

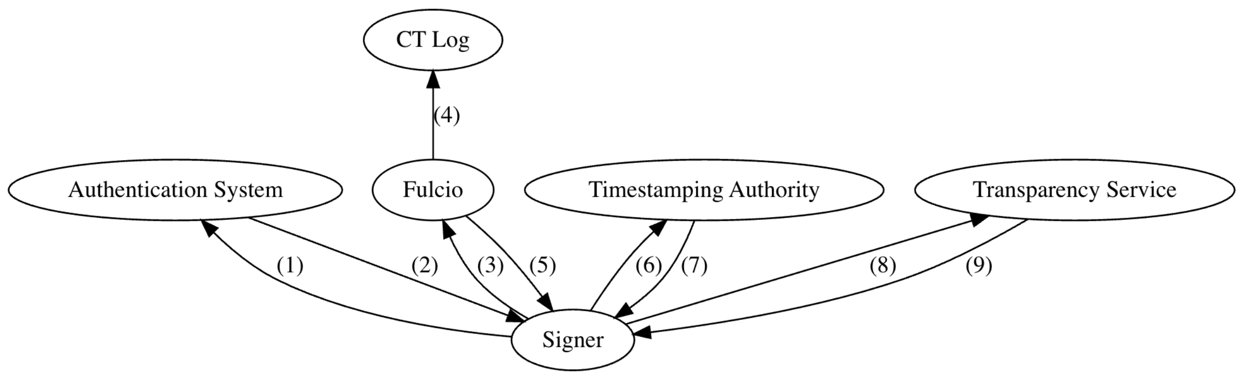
Transparency Service. A service compliant with Spec: Transparency Service.

Verifier. The entity who wishes to verify signatures on a given payload.

Overview

Configuration and root key material (henceforth, *root certificate(s)* and *optionally intermediate certificate(s)*) of each of the services are securely distributed out-of-band to any signers and verifiers; how to do this securely is outside the scope of this document.

The Signer authenticates with the Authentication System (1) and receives an identity token (2). The Signer generates a key pair, then sends the public key, along with the identity token, to Fulcio (3), which creates and signs a certificate attesting to the signer's identity. Fulcio submits the certificate to the CT Log (4), then sends it to the Signer (5). The Signer signs the payload, then sends the signature to a timestamping authority (6), receiving a timestamping response in return (7). The signer then sends the signing metadata (payload metadata, signature, and certificate) to the transparency service (8) and receives a Signed Entry Timestamp (9).



(graphviz)

Signing

The signer must make a number of choices during signing: which signature algorithm to use, which metadata format to use, whether to pre-hash the payload before signing, and whether to skip some of the signing steps. This section first describes the complete payload signing procedure, then describes considerations pertinent to those choices and how those choices affect the signing procedure.

Default Signing Procedure

This section describes the full signing workflow for a client. The client MAY omit certain of these steps (see §Signing Choices below).

Authentication

The Signer authenticates with the Authentication System. The details of how it does this are specific to the Authentication System and outside the scope of this document (see Spec: Fulcio for requirements on the authentication service). If the Authentication System supports a notion of "audience" for generated tokens, the Signer SHOULD identify the specific instance of Fulcio (based on the `identifier` in its public configuration) as the desired "audience" during authentication.

At the conclusion of the authentication protocol, the Signer will possess an authentication token; the format of this token is opaque to the Signer, except that the Signer MUST be able to extract a subject.

Key Generation

The Signer chooses an algorithm for digital signatures from the registry (Spec: Sigstore Registries); the chosen algorithm MUST be in both the Fulcio instance's and the Transparency Service instance's `supportedSigningAlgorithms`). The Signer generates a signing/verification key pair via the appropriate key generation procedure. The Signer MAY store the signing key on a secure hardware device. Regardless of the success of the signing procedure, the signer SHOULD destroy the keypair at the end.

Certificate Issuance

The Signer prepares a PKCS#10 `CertificationRequestInfo` using the key pair and authentication token from the previous steps as follows:

- The `subject` in the `CertificationRequestInfo` is an X.501 `RelativeDistinguishedName`. The `value` of the `RelativeDistinguishedName` SHOULD be the subject of the authentication token; its `type` MUST be the type identified in the Fulcio instance's public configuration.
- The `algorithm` field of the `subjectPKInfo` is the `AlgorithmIdentifier` (RFC 5280 §4.1.1.2) of the generated key pair.
- The `subjectPublicKey` field of the `subjectPKInfo` MUST be the encoding of the verification key for its algorithm.

Then, the signer prepares a `CreateSigningCertificateRequest` (definition) comprising the authentication token and the PKCS#10 certificate signing request (PEM-encoded; see RFC 7468) to the `CreateSigningCertificate` endpoint (definition) of the Fulcio instance.

In return, the Signer receives a `SigningCertificate` (definition) containing a chain of PEM-encoded X.509 certificates (RFC 5280), ordered from "leaf" to "root." See Spec: Fulcio for additional details about certificate contents. The Signer SHOULD verify the response:

1. Perform certification path validation (RFC 5280 §6) of the returned certificate chain with the pre-distributed Fulcio root certificate(s) as a trust anchor.
2. Extract a `SignedCertificateTimestamp`, which may be embedded as an X.509 extension in the leaf certificate or attached separately in the `SigningCertificate` returned from the Identity Service. Verify this `SignedCertificateTimestamp` as in RFC 9162 §8.1.3, using the root certificate from the Certificate Transparency Log.

3. Check that the leaf certificate contains the subject from the certificate signing request and encodes the appropriate `AuthenticationServiceIdentifier` in an extension with OID `1.3.6.1.4.1.57264.1.8`.

Signing

The Signer signs the payload using the signing key as in the chosen signing algorithm; the signature will be opaque binary data. The Signer MAY pre-hash the payload using a hash algorithm from the registry (Spec: Sigstore Registries) for compatibility with some signing metadata formats (see §Submission of Signing Metadata to Transparency Service).

Timestamping

The Signer sends a hash of the signature as the `messageImprint` in a `TimeStampReq` to the Timestamping Service and receives a `TimeStampResp` including a \`TimeStampToken\`. The signer MUST verify the `TimeStampToken` against the payload and Timestamping Service root certificate.

Submission of Signing Metadata to Transparency Service

The Signer chooses a format for signing metadata; this format MUST be in the `supportedMetadataFormats` in the Transparency Service configuration. The Signer prepares signing metadata containing at a minimum:

- The signature.
- The payload (possibly pre-hashed; if so, the entry also includes the identifier of the hash algorithm).
- Verification material (signing certificate or verification key).
    - If the verification material is a certificate, the client SHOULD upload only the signing certificate and SHOULD NOT upload the CA certificate chain.

The signing metadata might contain additional, application-specific metadata according to the format used. The Signer then canonically encodes the metadata (according to the chosen format).

Transparency

The Signer then sends the canonically-encoded signing metadata to the `/api/v1/log/entries` endpoint (definition) of the Transparency Service, which checks that signature is valid and responds with a `LogEntry` (definition). The signer MUST verify the log entry as in Spec: Transparency Service .

Verification

The above sections require the Signer to verify several parts of the signing process. Additionally, the Signer SHOULD perform the full verification procedure, as below (§Verification). When the expected signing identity is known before signing, the Signer MAY check that the signature matches that identity.

Distribution

The Signer conveys the following verification materials to the verifier in order to allow verification:

- Code-signing certificate. The CA root certificate MUST be provided out of band. Intermediate CA certificates SHOULD be provided out of band, but MAY be provided with the verification materials.
- Signature.
- Additional payload metadata.
- Timestamping response.
- Transparency Service `LogEntry` (definition). The log public key MUST be provided out of band.

They can do so in any manner. Signers SHOULD collate this data in the Sigstore wire format (§Serialization and Wire Format) which stores these all in one object for easy distribution. The Verifier must also obtain the artifact to verify.

Signing Choices

Authentication System. The signer MUST use an Authentication System supported by the Fulcio instance with which they can authenticate.

Digital signature algorithm. The signer must choose a digital signature algorithm for key generation and signing from the registry (see Spec: Sigstore Registries). The algorithm MUST be in the `supportedSigningAlgorithms` of both the Fulcio and Transparency Service instances.

Signature metadata format. The signature metadata format MUST be in the list of `supportedMetadataFormats` in the Transparency Service configuration. This list can include both common registry formats (see Spec: Sigstore Registries) or additional plug-in formats. Details about plug-in formats are conveyed out-of-band.

The metadata format chosen may depend on the artifact to sign (some formats encode extra metadata about specific artifact types), size (some formats require the full artifact; others allow the payload to be hashed), or compatibility with other systems.

Payload pre-hashing. Some metadata formats store a hash of the payload. In this case, the signature is over the *hashed* payload, so that the Transparency Service can validate the signature.

In such cases, the Signer must choose a hash algorithm from the registry (see Spec: Sigstore Registries); this algorithm MUST be in the `supportedHashAlgorithms` for the Transparency Service.

Long-lived signing keys. The Signer may have a pre-existing, long-lived signing key with which they would like to sign payloads. This key MUST use an algorithm in the `supportedSigningAlgorithms` of both the Fulcio configuration and Transparency Service configuration.

In such cases, the Signer can skip the key generation step; the signing procedure is otherwise unaltered.

Timestamping. Currently, the Transparency Service includes a timestamp in its response to the Signer. This timestamp comes from the Transparency Service's internal clock, which is not externally verifiable or immutable. For this reason, a Signer SHOULD get their signatures timestamped. However, a Signer MAY choose to omit the timestamping step; in this case, the Signer MUST use the Transparency Service to provide a timestamp for the signature.

Transparency. The Signer SHOULD upload signing metadata to the Transparency Service, but MAY choose to skip this step (for instance, for privacy reasons). In this case, the Signer MUST use a Timestamping Service to provide a timestamp for the signature.

Other workflows. A client may support signing workflows different from that described above. For instance, a Signer may want to use a long-lived signing key without an associated certificate; in this case, they can skip the authentication, key generation, and certificate issuance steps. A Signer may have a distinct Certificate Authority. Details for these workflows are out-of-scope for this document.

Verification

A Verifier validates a signature on a payload along with other verification material according to a *policy*. The policy specifies:

- What must be true about the identity in a certificate (whom to trust).
- Which Fulcio, Timestamping Authority, and Transparency Service instances to trust (including root key material for each).
- Whether to require signed timestamp(s) from a Timestamping Authority, and, if so, how many.

- Whether to require the signature metadata to be logged in one or more Transparency Services and, if so, how many.
- Whether to perform online or offline verification for the CT Log and the Transparency Service.
- Which Transparency Service formats the Verifier knows how to parse and validate.
- What to do with a payload, once verified.
- How to determine whether a signature has been revoked.

Knowing the verification policies of possible Verifiers may help Signers choose how to sign their payloads. Policies are application-specific and distributed out of band. A policy is an abstract procedure, not a set configuration: a client does not need to support arbitrary policies; it might instead hard-code verification for a single policy, or expose only a limited number of configuration options, like the Sigstore `ArtifactVerificationOptions` (definition) and `TrustedRoot` (definition). Below, we describe a generic verification procedure and note where policy-specific decisions or departures may occur. If any step fails, abort verification unless otherwise specified.

Inputs

The Verifier performs verification according to its policy based on the following inputs:

- The artifact.
- Verification materials (possibly in the the `Bundle` format (definition)):
  - Leaf certificate
    - When used with the Public Good Instance, only the leaf is necessary. Other Sigstore instances (such as private instances) may require one or more intermediates as well, if those intermediates are not listed in the independent root of trust.
  - Signature.
  - Additional payload metadata.
  - Timestamping response.
  - Transparency Service `LogEntry` (definition).
- Root key material for Sigstore infrastructure (from the policy).

The distribution of these inputs is out-of-scope for this document.

Recommended discovery order for verification materials

For Sigstore clients that expose a command-line interface, the following discovery order is RECOMMENDED:

1. Use whatever verification materials are supplied explicitly by the user. For example, if the client has flags and/or environment variables for configuring bundles and/or detached verification materials, these should take precedence over any implicitly discovered materials.

2. If no explicit inputs are given: for a given file `input`, attempt to discover `{input}.sigstore.json`. If `{input}.sigstore.json` is present, attempt to use it for verification.
3. If `{input}.sigstore.json` is not present, attempt to discover `{input}.sigstore` and use it for verification.

Establishing a Time for the Signature

First, establish a time for the signature. This timestamp is required to validate the certificate chain, so this step comes first.

Timestamping Service

If the verification policy uses the Timestamping Service, the Verifier MUST verify the timestamping response using the Timestamping Service root key material, as described in Spec: Timestamping Service, with the raw bytes of the signature as the timestamped data. The Verifier MUST then extract a timestamp from the timestamping response. If verification or timestamp parsing fails, the Verifier MUST abort.

Transparency Service Timestamp

If the verification policy uses timestamps from the Transparency Service, the Verifier MUST verify the signature on the Transparency Service `LogEntry` as described in Spec: Transparency Service against the pre-distributed root key material from the transparency service. The Verifier SHOULD NOT (yet) attempt to parse the `body`. The Verifier MUST then parse the `integratedTime` as a Unix timestamp (seconds since January 1, 1970 UTC). If verification or timestamp parsing fails, the Verifier MUST abort.

Certificate

For a signature with a given certificate to be considered valid, it must have a timestamp while every certificate in the chain up to the root is valid (the so-called “hybrid model” of certificate verification per Braun et al. (2013)):

None
```
Root CA:               |-----|
Intermediate:              |-----|
Leaf:                    |-----|
Valid timestamp range:     |-|
```

The Verifier MUST perform certification path validation (RFC 5280 §6) of the certificate chain with the pre-distributed Fulcio root certificate(s) as a trust anchor, but with a fake "current time." If a timestamp from the timestamping service is available, the Verifier MUST perform path validation using the timestamp from the Timestamping Service. If a timestamp from the Transparency Service is available, the Verifier MUST perform path validation using the timestamp from the Transparency Service. If both are available, the Verifier performs path validation twice. If either fails, verification fails.

Unless performing online verification (see §Alternative Workflows), the Verifier MUST extract the `SignedCertificateTimestamp` embedded in the leaf certificate, and verify it as in RFC 9162 §8.1.3, using the verification key from the Certificate Transparency Log.

The Verifier MUST then check the certificate against the verification policy. Details on how to do this depend on the verification policy, but the Verifier SHOULD check the `Issuer` X.509 extension (OID `1.3.6.1.4.1.57264.1.1`) at a minimum, and will in most cases check the `SubjectAlternativeName` as well. See Spec: Fulcio §TODO for example checks on the certificate.

Transparency Log Entry

By this point, the Verifier has already verified the signature by the Transparency Service (§Establishing a Time for the Signature). The Verifier MUST parse `body`: `body` is a base64-encoded JSON document with keys `apiVersion` and `kind`. The Verifier implementation contains a list of known Transparency Service formats (by `apiVersion` and `kind`); if no type is found, abort. The Verifier MUST parse `body` as the given type.

Then, the Verifier MUST check the following; exactly how to do this will be specified by each type in Spec: Sigstore Registries (§Signature Metadata Formats):

1. The signature from the parsed body is the same as the provided signature.
2. The key or certificate from the parsed body is the same as in the input certificate.
3. The "subject" of the parsed body matches the artifact.

The verification policy can impose additional constraints here. For instance, if a `kind` and `apiVersion` are provided in the policy (as in the `bundle` format), they must match the `kind` and `apiVersion` in `body`.

Signature Verification

The Verifier now constructs the payload to be signed from the artifact and the additional payload metadata according to the verification policy and Transparency. Methods for doing so include:

- Using the raw bytes of the artifact as the payload.
- Hashing the artifact, then using the resultant digest as the payload.
- Using DSSE as an envelope for the payload with a known DSSE payload type.

The Verifier MUST verify the provided signature for the constructed payload against the key in the leaf of the certificate chain.

Alternative Workflows

Verification according to some verification policies may deviate from the above procedure as follows:

No Timestamping Service. The Verifier MAY choose to rely on the Transparency Service for timestamps. In this case, the Verifier MUST use a timestamp from the Transparency Service during certificate verification. The Verifier can skip verification of the timestamping response, as well as certificate verification using the timestamp from the Timestamping Service.

No Transparency Service. The Verifier MAY choose not to require that signatures are in the Transparency Service. In this case, the Verifier MUST use a timestamp from the Timestamping Service during certificate verification. The Verifier can skip verification of the Transparency Service `LogEntry` for timestamping, certificate verification using the timestamp from the Transparency Service, and Transparency Log Entry validation.

Online Certificate Transparency Log verification. The above procedure describes using `SignedCertificateTimestamp`s to verify inclusion in the certificate transparency log. Instead, Verifiers MAY perform online verification by fetching and validating inclusion proofs (RFC 9162 §8.1.4) against a signed tree head. The Verifier SHOULD fetch the signed tree head in a manner that prevents equivocation by the Certificate Transparency log (e.g., by requiring signatures from independent "witnesses").

Online Transparency Service verification. The above procedure describes using signed inclusion promises from the Transparency Service for verifying membership in a transparency log ("offline verification.") Instead, a Verifier MAY perform online verification. In this case, the Verifier checks an inclusion proof for the `LogEntry` against a `SignedTreeHead`. See Spec: Transparency Service for details.

Threshold verification. The Verifier MAY require that the leaf certificate be included in multiple Certificate Transparency Logs or that the formatted metadata be included in multiple Transparency Service logs. This is equivalent to verifying multiple times with different logs. Verifiers MUST ensure that multiple entries in the same log do not both count towards the threshold.

Omitted Transparency Service `body`. In some cases, the Transparency Service `body` may be simple to compute from the artifact. In this case, the `LogEntry` may omit the body, and the Verifier can reconstruct the body.

Out-of-band intermediate CAs. The Verifier SHOULD require that intermediate certificates are distributed out-of-band as well. In this case, the Verifier MUST remove certificates from the certificate chain that are not also found in the out-of-band list of intermediate certificates.

While other signing and verification workflows are possible using the Transparency Service (third-party certificate authorities, using hardcoded keys), this document focuses on verification using the Sigstore Certificate Authority.

Serialization and Wire Format

This section describes the “Sigstore wire format” for verification materials.

To produce verification materials in this format, a client MUST use the Protocol Buffers Bundle format to collate these materials, serialized to JSON using the canonical proto3 JSON serialization, *except* that:

1. The bundle MUST use `lowerCamelCase` rather than `snake_case` for keys.
2. The bundle MUST use the string representation for enum values.

This is the same as the JSON Schema schema in the protobuf-specs repository for clients which prefer JSON Schema. Clients SHOULD NOT accept other variants of the canonical JSON proto3 serialization.

If clients serialize the bundle to a file, the file SHOULD have the extension `.sigstore.json`. To write multiple bundles in one file, clients SHOULD use the JSON Lines format (format each bundle without newlines, and write one bundle per line) and the extension `.sigstore.jsonl`.

Security Considerations

This document describes a security system, and security considerations are present throughout. The Sigstore threat model details a threat model, including which parties are trusted to be honest, and the consequences if various subsets of those parties are in fact malicious.

Revocation and expiration. This document *does not* describe how to perform revocation and rotation for the key material for the Sigstore infrastructure. Revocation, rotation, and expiration should be handled where the key material is distributed. Specifically, when fetching root key material, a Verifier should fetch metadata indicating the validity period for that key material (which may be a subset of the validity period indicated in an X.509 certificate, for instance).

Then, *that* validity period should be used during certificate verification. This is described in more detail ….