TLS Working Group
Internet-Draft
Intended status: Informational
Expires: 31 December 2025

B. McMillion

D. O'Brien
Google LLC
D. Jackson
Mozilla

29 June 2025

Reliable Transparency and Revocation Mechanisms
draft-mcmillion-tls-transparency-revocation-00

Abstract

This document describes reliable mechanisms for the publication and revocation of Transport Layer Security (TLS) certificates. This reliability takes several forms. First, it provides browsers a strong guarantee that all certificates they accept are truly published and unrevoked at the time they're accepted. Second, it allows operators to monitor for mis-issuances related to their websites in a highly efficient way without relying on third-party services. Third, it provides a high degree of operational redundancy to minimize the risk of cascading outages.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://Bren2010.github.io/draft-transparency-revocation/draft-mcmillion-tls-transparency-revocation.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-mcmillion-tls-transparency-revocation/.

Source for this draft and an issue tracker can be found at https://github.com/Bren2010/draft-transparency-revocation.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Introduction

The Certificate Transparency (CT) ecosystem created by [RFC6962] has been immensely valuable to security on the internet. However, shortcomings in the design have become apparent over time:

The security that CT provides to verifiers is based on an assumption of non-collusion between multiple parties. Historically, this assumption has been challenging to maintain, as it degrades quickly without active management. The compromise of a single Transparency Log or the unexpected acquisition of a single business is often sufficient to allow the possibility of undetectable mis-issued certificates. This is compounded by the fact that multiple parties in the CT ecosystem play multiple roles (such as Certificate Authorities that are also Transparency Log operators), which makes reasoning about the possibility of collusion even more tricky.

It is also becoming far too expensive to both operate a CT log and to monitor CT logs. Logs are required to serve their entire contents to anyone on the internet, which consumes a significant amount of outbound bandwidth. Similarly, monitoring certificate issuance in CT requires downloading the entire contents of all logs, which is several terabytes of data at minimum. The total bandwidth costs of the CT ecosystem scale linearly in the number of certificates issued, the number of logs active, and the number of interested monitors.

One of the primary motivations for publishing TLS certificates is to allow site operators to identify and revoke those certificates which are mis-issued. However, revocation systems have historically been plagued by a requirement to "fail open". That is, revocation checks would stop being enforced in certain (often, easily engineered) scenarios. For example, clients using OCSP must typically fail open in the event the OCSP server is unreachable. Alternatives like OCSP Must-Staple [RFC7633] were designed to close this loophole, but have been stymied by lack of support in popular web servers.

More recent alternatives like CRLSets [CRLSets], CRLite [CRLite], and Clubcards [Clubcards] provide fail-closed revocation checks to clients, but are unstandardized, rely on trusting the server operator (who is typically a client software vendor, rather than a CA) and offer limited transparency properties.

This motivates a need for a new system of publishing certificates that’s resistant to collusion and dramatically more efficient to operate, and a need for a new system of revoking certificates that can be consistently enforced.

Since the initial deployment of Certificate Transparency in 2013, there has been a considerable body of research published on transparency systems. In recent years, Key Transparency systems have been deployed by Apple [AppleKT], Meta [MetaKT], and ProtonMail [ProtonKT]. These systems not only provide stronger security properties, but also support transparent revocation, and also scale with less bandwidth costs.

Key Transparency is being standardized in the KeyTrans IETF WG [KeyTransWG]. This document describes how a similar design as the one being considered in the KeyTrans WG could be applied to TLS in order to provide stronger security properties whilst also reducing the TLS handshake size.

2. Conventions and Definitions

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.

3. Architecture

The system has several roles, which we describe in more detail below. Parties are allowed to assume multiple roles.

Certificate Authority: A service that performs domain-control validation and authenticates certificates and revocations.

Transparency Log: A service that provides an append-only, publicly-auditable log of certificates and revocations issued by a wide range of Certificate Authorities.

Site Operator: The individual or organization responsible for operating and maintaining a website, as identified by a set of domain names or IP addresses.

User Agent: A software application, typically but not necessarily a web browser, that acts on behalf of a user to access and interact with websites.

3.1. Requirements

The following baseline requirements for the system are as follows:

1. For a certificate to be mis-issued and not eventually published: two trusted parties must collude to misbehave and all User Agents that observed the certificate must be stateless.

2. For a User Agent to accept a revoked certificate and this not be eventually detected: one trusted party must misbehave and all User Agents that accepted the certificate must be stateless.

3. It must be reasonably efficient for Site Operators to audit all trusted Certificate Authorities for mis-issuances affecting their domain names and IP addresses.

4. It must not be possible for a third-party service’s outage to cause an outage for a Site Operator, other than how it could in the web PKI as it exists today.

5. User Agents must be able to connect to a server without having immediate connectivity to third-party services.

6. The domain names and/or IP addresses of websites visited by a User Agent must not be leaked, other than how they are in the web PKI as it exists today.

7. The system must be reasonable for non-browser User Agents to deploy.

8. The system must have a reasonable path to scale to an indefinite number of Site Operators and User Agents.

These requirements and their main consequences are discussed in more detail in the following subsection.

3.2. Discussion

Transparency must support stateful verification. The fundamental goal of any transparency system is to ensure that data shown to one participant is equally visible to any other participant. Transparency systems that achieve this by relying solely on stateless verification are more accurately referred to as "co-signing" schemes. This is because the security of these systems reduces solely to successful verification of the co-signers' signatures on a claim that some data is properly published, and ultimately to an assumption that the co-signers are not colluding to suppress the data.

In the web PKI, the co-signers in such a system would be a Certificate Authority and one or more Transparency Logs. The diverse and rapidly changing nature of the web makes collusion between participants easy to achieve and difficult to detect. Without a technical mechanism to detect or prevent collusion, it can be covertly achieved and maintained for very long periods of time.

The typical solution to this problem is to construct the system such that a larger number of co-signers need to collude to succeed in launching an attack. However, requiring that a large number of signatures from different co-signers be presented in every TLS handshake would easily bloat the handshake to an unacceptable degree. This is especially true when the signatures are required to come from post-quantum signature schemes, given that they're much larger than their classical alternatives.

While the system described in this document supports stateless verification, which is secure as long as there's no collusion between trusted parties, it also provides stateful verification. Stateful verification allows verifiers to detect violations of the system's core security guarantees, that all certificates they accept are published and unrevoked, regardless of any amount of collusion by trusted parties. This obviates the need for having a large number of co-signers, allowing TLS handshakes to remain small and providing the broader ecosystem (including stateless verifiers) with a reliable early warning system for misbehavior by trusted parties.

Servers must provide proof directly to clients that their certificate satisfies transparency requirements and isn't revoked. If proof of transparency and non-revocation isn't provided by the server, it must be fetched from one or more third-party services. The primary issue with this is that it ties the server's global availability to the global availability of these third-party

services, for which the server operator has no way to preempt or resolve deficiencies. Proposals for transparency and revocation that rely on connectivity to third-party services have historically been required to "fail open", meaning that if the third-party service is inaccessible for too long, then the client stops enforcing these security properties altogether. Failing open is incompatible with requirements 1 and 2 because it introduces the possibility that a client might accept an unpublished or revoked certificate without explicit misbehavior by a trusted party.

A second issue is that, since we're primarily interested in describing a system that works equally well regardless of the client's software vendor (requirement 7), such a service would reasonably be prohibited from restricting access to itself. It would receive regular requests from all internet-connected devices, creating significant scaling and centralization concerns.

Servers must refresh their certificates regularly and automatically. This is a direct consequence of the decision that servers must be responsible for providing clients proof that their certificates are not revoked. If a server is not required to refresh its certificate, it can arbitrarily delay the client from learning about changes in the certificate's revocation status. This would be incompatible with requirement 2, since servers are not considered trusted parties and should not be capable of undermining revocation on their own.

Revocation must be provided by the transparency system. A CA can initiate revocation either by declining to sign new statements related to a certificate (for example, by not renewing a short-lived certificate or not creating an OCSP staple), or by signing a new "statement of revocation" for the certificate.

In the case where CAs initiate revocation by signing a new "statement of revocation", proving that a certificate is not revoked consists of proving that such a statement does not exist. Proving that a statement does not exist requires exhaustive knowledge of all such statements. Any method of conveying this exhaustive knowledge, if it is not a transparency system, admits the possibility of split-view attacks which are not eventually detected. Split-view attacks in this context would allow a CA to mis-issue a certificate, claim to revoke it, and then maintain the certificate's utility by presenting different views of its revocation status to attack victims than to other participants. The possibility of such a split-view attack would violate requirement 2, as it would not be after-the-fact detectable. More practically though, it would render revocation fundamentally insufficient for correcting mis-issuance and would create a need for a second (more effective) revocation mechanism.

In the case where CAs initiate revocation by declining to sign new statements, this makes the CA a single point-of-failure for websites relying on it. A prolonged CA outage would have the effect of revoking all certificates and causing a cascading outage, violating requirement 4. Proposals for revocation that fall into this category have historically mitigated this risk by providing very slow revocation, bounded by the longest conceivable outage that a CA may have (typically at least one week). Additionally, it's clear that the same potential for split-view attacks would still exist, as discussed above.

When specifically considering short-lived certificates as an approach to revocation, effectiveness depends on whether or not *all* certificates in the PKI are required to be short-lived. If clients enforce that all certificates are short-lived, and issuance is transparent, then revocation is provided by the transparency system as claimed. If certificates may be issued with longer lifespans, then a second revocation mechanism for these certificates is necessary.

All certificate lifetimes are already planned to fall from 398 days today to 47 days in 2029 [CertLifetimes], bringing a number of security and agility benefits to the ecosystem. However, 47 days is still much longer than is tolerable without effective revocation. Although a further reduction to 7 days or 24 hours is possible in theory, each halving in lifetime results in doubling the issuance load for CAs, CT logs and monitors. The net effect would be a ~56x increase in issuance rate in order to maintain the same size PKI as we have today.

Transparency Logs must implement a protocol similar to Key Transparency [KEYTRANS]. As stated at the beginning of this section, the goal of any transparency system is to ensure that data shown to one participant is equally visible to any other participant. In the context of the web PKI, this means that a Site Operator that contacts all trusted Transparency Logs should come away with an exhaustive knowledge of all certificates that a client might be presented with when connecting to their servers.

Most transparency systems require downloading the entirety of the log's contents to ensure that all potentially relevant entries are found. This quickly becomes prohibitively expensive for all parties, violating requirements 3 and 8. Currently roughly 7.5 million certificates are issued per day, with an average size of 3 kilobytes. This means that a Site Operator would need to download almost 700 gigabytes of certificates to cover a single month of issuance. Outbound bandwidth typically costs between 1 to 9 cents per gigabyte, which means that providing this data to a single Site Operator would

cost the Transparency Log between $6 to $60. If even a small percentage of all Site Operators on the internet were to do this, it would create an exceptional financial and operational burden for the Transparency Log.

In the existing Certificate Transparency ecosystem, because of this exceptional cost, Site Operators have overwhelmingly elected not to do this work themselves and have instead outsourced it to third-party monitors. Third-party monitors represent a problematic break in the security guarantees of Certificate Transparency as there are no enforceable requirements on their behavior. They are not audited for correct behavior like Certificate Authorities are, and there are no technical mechanisms to prevent misbehavior like a Transparency Log would have.

Key Transparency systems are generally distinguished from other transparency systems by the fact that they augment a Transparency Log with additional structure to allow efficient and verifiable searches for specific data. In the context of providing transparency to the web PKI, this would allow Site Operators to download only a small subset of the data in a Transparency Log and still be assured to have received all certificates that are relevant to them. Due to the significantly reduced need for outbound bandwidth, operating such a Transparency Log would cost roughly one million times less than it would if Site Operators were required to download the log’s entire contents.

Transparency Logs may still choose to allow parties to download their entire contents. However, this is not necessary for the protocol to be secure and this document doesn’t prescribe a specific mechanism for it. Additionally, while the protocol described in [KEYTRANS] could be applied directly, some optimizations and simplifications specific to the web PKI are provided in Section 4.

3.3. Summary

In summary, the system described in this document works as follows:

* Site Operators obtain a certificate from a Certificate Authority and submit it to one of many trusted Transparency Logs to obtain an inclusion proof.

* User Agents that contact the Site Operator’s server over TLS include compact transparency-related state in their ClientHello. The server provides its certificate and inclusion proof (potentially modified based on the client’s advertised state) in the Certificate message. The User Agent verifies that the inclusion proof aligns with its state, is sufficiently recent, and indicates the certificate is unrevoked.

* As time goes on, the current inclusion proof will become stale. Site Operators refresh their inclusion proof by requesting a new one from the Transparency Log.

  - If the first Transparency Log is offline, the Site Operator may failover to any of several other trusted Transparency Logs.

* At any time in the background, the Site Operator may query any of the trusted Transparency Logs and verifiably learn about all new certificates that have been issued affecting their domain names or IP addresses. Since the Key Transparency protocol specifies a "correct" location for a certificate to be stored, and since User Agents enforce this when verifying inclusion proofs, requesting all certificates for a single identifier always remains highly efficient.

The remainder of this document describes these steps in more detail.

4. Transparency Log

The data structure maintained by a Transparency Log is identical to the Combined Tree described in [KEYTRANS], with two exceptions: the search keys that are used to navigate the Prefix Tree, and the data committed to by each Prefix Tree leaf node, are different.

The search key used to navigate the Prefix Tree is a function of the certificate’s *reference identifier* in the TLS handshake. The reference identifier is the domain name or IP address that the TLS client will verify the certificate against. When the reference identifier is a domain name, the corresponding search key is the domain name with the components in reverse order and a trailing dot, meaning that "com.example.sub." would be the search key for the domain name "sub.example.com". If the final component of a domain name is a wildcard then the wildcard is stripped, such that "com.example." would be the search key for "*.example.com". The search key corresponding to an IPv4 addresses is the address rendered in dotted-decimal notation. The search key corresponding to an IPv6 address is the address rendered as described in option 1 of Section 2.2 of [RFC4291]. No VRF is used, or version counter is included, as they would be in [KEYTRANS].

Rather than a privacy-preserving commitment, each Prefix Tree leaf contains the hash of a DomainCertificates structure:

```
struct {
  opaque root[Hash.Nh];
  uint32 first_valid;
  uint32 invalid_entries<0..2^8-1>;
} DomainCertificates;
```

The root field contains the root hash of a Log Tree, which will be referred to as the *Certificate Subtree* to avoid confusion with the top-level Log Tree. The Certificate Subtree contains all certificate chains that may be presented for a particular domain name or IP address, in the order they were logged. The leaves of the Certificate Subtree are represented as SubtreeLogLeaf structures, used in place of LogLeaf:

```
opaque Certificate<0..2^16-1>;

struct {
  Certificate chain<0..2^8-1>;
} SubtreeLogLeaf;
```

The first_valid field contains the index of the first entry in the Certificate Subtree where no certificate in the chain is revoked or expired. The invalid_entries field contains the list of indices of all entries to the right of first_valid where one or more of the certificates in the chain have been revoked.

Transparency Logs SHOULD determine whether a certificate chain is expired by comparing the Not After field of each certificate in the chain to the timestamp of the Log Tree’s rightmost leaf. However, Transparency Logs do not have a proactive responsibility to keep the first_valid field updated; it is simply provided as a mechanism to drain invalid_entries.

When computing PrefixLeaf, the hash of the leaf certificate’s reference identifier is stored in the vrf_output field and the hash of DomainCertificates is stored in the commitment field.

4.1. Subtree Inclusion Proofs

It is often necessary in later parts of this document to provide proofs of inclusion for entries in the Certificate Subtree. Such proofs are provided as follows:

```
struct {
  uint32 position;
  uint32 size;
  InclusionProof inclusion;
  uint32 first_valid;
  uint32 invalid_entries<0..2^8-1>;
} SubtreeInclusionProof;
```

The position field contains the index of the leaf for which inclusion is being proven. The size field contains the total number of leaves in the Certificate Subtree. The proof in inclusion allows a recipient to recompute the root hash of the Certificate Subtree, given the correct value for the leaf at position.

The first_valid and invalid_entries fields duplicate the contents of the DomainCertificates structure. This allows recipients to verify that the leaf at position is not revoked, and also allows them to recompute the hash of the DomainCertificates structure stored at a given leaf of the Prefix Tree.

Note that this document follows the pattern established in [KEYTRANS] of requiring each element of an InclusionProof to be a balanced subtree. An InclusionProof may also function as a "consistency proof" if the recipient is known to have observed a previous version of the tree.

4.2. Tree Heads

Transparency Logs are generally expected to add only a small number of new entries to their Log Tree per day. This keeps proof sizes small and also ensures that, when User Agents advertise having observed a particular tree head, there are a large number of potential hosts that could’ve conveyed the tree head. To support providing inclusion proofs for new submissions quickly, Transparency Logs issue *provisional* tree heads. A provisional head is, in essence, a work-in-progress log entry that will be added to the rightmost edge of the Log Tree within a bounded amount of time.

```
struct {
  uint64 tree_size;
  uint32 counter;
  opaque signature<0..2^16-1>;
} ProvisionalTreeHead;
```

The tree_size field is equal to the tree_size field of the last (non-provisional) TreeHead that was published. The counter field starts at zero and is incremented by 1 for each subsequent ProvisionalTreeHead that's published without the creation of a new TreeHead.

The signature field of both TreeHead and ProvisionalTreeHead structures is computed over a serialized TreeHeadTBS structure:

```
struct {
  CipherSuite ciphersuite;
  opaque signature_public_key<0..2^16-1>;

  uint64 max_ahead;
  uint64 max_behind;
  optional<uint64> maximum_lifetime;
} Configuration;

enum {
  reserved(0),
  standard(1),
  provisional(2),
  (255)
} TreeHeadType;

struct {
  Configuration config;

  TreeHeadType tree_head_type;
  select (TreeHeadTBS.tree_head_type) {
    case standard:
      uint64 tree_size;
    case provisional:
      uint64 tree_size;
      uint32 counter;
      opaque prefix_root[Hash.Nh];
  };
  opaque root[Hash.Nh];
} TreeHeadTBS;
```

The max_ahead and max_behind fields contain the maximum amount of time in milliseconds that a tree head may be ahead of or behind the user's local clock without being rejected. If the Transparency Log has chosen to define a maximum lifetime for log entries, per Section 5.2 of [KEYTRANS], this duration in milliseconds is stored in the maximum_lifetime field.

When the TreeHeadTBS structure is for a provisional tree head type, prefix_root contains the work-in-progress root hash of the Prefix Tree. This value may change further before it is added as a new rightmost log entry. However, stateful clients will enforce that none of the certificates they observe are removed or un-revoked.

5. TLS Extension

The following three subsections define the ClientHello, ServerHello, and Certificate message portions of a TLS 1.3 extension. This extension allows the host server to provide an inclusion proof for its certificate chain from a Transparency Log that the client supports.

5.1. ClientHello

Clients include the extension in their ClientHello to communicate which Transparency Logs they support and whether or not they have previously observed a provisional inclusion proof from the server.

```
opaque BearerToken<0..2^8-1>;

struct {
  uint16 transparency_log_id;
  TreeHeadType tree_head_type;
  select (SupportedTransparencyLog.tree_head_type) {
    case standard:
      uint64 tree_size;
  };
} SupportedTransparencyLog;

struct {
  SupportedTransparencyLog supported<0..2^8-1>;
  select (any supported[i].tree_head_type is provisional) {
    case true:
      BearerToken bearer_token;
  }
} TransparencyRequest;
```

The extension has type "transparency_revocation" and consists of a serialized TransparencyRequest structure in the extension_data field.

Clients include an entry in the supported array for each Transparency Log that they support receiving inclusion proofs from, containing the Transparency Log’s assigned unique identifier in transparency_log_id. The supported array MUST be sorted in ascending order by transparency_log_id, and each transparency_log_id MUST only be advertised once.

If a client was shown a provisional inclusion proof from a Transparency Log in a previous connection to the server, the client sets tree_head_type to be provisional for that Transparency Log's entry in supported until one of the following conditions is met:

1. The client receives proof that the provisional proof was integrated into the subsequent log entry, or:

2. The timestamp of the rightmost log entry that existed in the provisional inclusion proof is more than 10*max_behind milliseconds in the past.

For Transparency Logs where the client does not need to resolve a provisional inclusion proof, tree_head_type is set to standard. The tree_size field is set as follows:

* If the client is aware of two consecutive log entries where the timestamp of the left log entry is greater than max_behind milliseconds in the past and the timestamp of the right log entry is less than or equal to max_behind milliseconds in the past, then the tree_size field is set so as to make the right log entry the rightmost log entry.

* If the client isn't aware of two log entries that meet the above criteria, but is aware of a log entry whose timestamp is greater than max_behind milliseconds in the past and less than 10*max_behind milliseconds in the past, then the tree_size field is set so as to make the rightmost such log entry the rightmost log entry.

* Otherwise, the tree_size field is set to 0.

With the first criteria, the client is aiming to advertise the oldest tree_size that a server could provide an inclusion proof against without it being expired. If the client is unable to do this, the second criteria aims to advertise a tree_size that's old enough that it would not be de-anonymizing, but not so old that servers are unaware of it.

If the tree_head_type of any SupportedTransparencyLog structure is set to provisional, then a bearer token is provided in bearer_token. The bearer token to use is provided by the server in a previous connection. The same bearer token MAY be advertised in multiple ClientHello messages but clients SHOULD take care to minimize the likelihood of this.

5.2. ServerHello

Servers that receive a TLS 1.3 ClientHello with an extension of type "transparency_revocation", where the extension_data field is properly formed, have the option of providing an inclusion proof for their certificate chain. The proof provided by the server MUST come from one of the Transparency Logs advertised in the client's ClientHello. If the server is not able to provide a proof from one of the client's supported Transparency Logs, it MUST respond as if the ClientHello extension was not advertised at all. Servers SHOULD preferentially respond with an inclusion proof from one of the Transparency Logs that the client advertised a provisional tree head type for, provided that an acceptable proof is readily available.

If and only if the server intends to provide an inclusion proof (provisional or not) from a Transparency Log where the client has advertised a provisional tree head type, it includes an extension of type "transparency_revocation" in its ServerHello. The extension_data is the unique identifier of the Transparency Log:

```
uint16 transparency_log_id;
```

5.3. Certificate

The server provides an inclusion proof for its certificate chain by adding an extension of type "transparency_revocation" to the first CertificateEntry structure in certificate_list. The extension_data field is a serialized TransparencyProof structure:

```
enum {
  reserved(0),
  standard(1),
  provisional(2),
  same_head(3),
  (255)
} TransparencyProofType;

struct {
  uint32 size
  uint32 first_valid;
  uint32 invalid_entries<0..2^8-1>;
} SequencingProof;

struct {
  TreeHead tree_head;
  CombinedTreeProof combined;
  select (condition) {
    case true:
```

```
        SequencingProof sequencing<1..2^8-1>;
    }
    SubtreeInclusionProof subtree<1..2^8-1>;
  } StandardProof;

  struct {
    ProvisionalTreeHead tree_head;
    CombinedTreeProof combined;
    select (condition) {
      case true:
        SequencingProof sequencing<1..2^8-1>;
    }

    PrefixProof prefix;
    SubtreeInclusionProof subtree<1..2^8-1>;
  } ProvisionalProof;

  struct {
    PrefixProof prefix;
    SubtreeInclusionProof subtree<1..2^8-1>;
  } SameHeadProof;

  struct {
    uint16 transparency_log_id;
    uint8 reference_ids<1..2^8-1>;

    TransparencyProofType proof_type;
    select(TransparencyProof.proof_type) {
      case standard:
        StandardProof proof;
      case provisional:
        ProvisionalProof proof;
      case same_head:
        SameHeadProof proof;
    };

    select (condition) {
      case true:
        BearerToken bearer_token;
        opaque pre_shared_key<Npsk>;
    }
  } TransparencyProof;
```

The transparency_log_id field specifies the Transparency Log that the proof comes from. The reference_ids field specifies the set of reference identifiers that the proof will cover, specified as a zero-indexed offset into the leaf certificate's subjectAltName extension. The proof_type field specifies the type of proof that follows.

If any of the following conditions are true, then the bearer_token and pre_shared_key fields are present:

1. proof_type is set to provisional,

2. A provisional tree head type was advertised in the ClientHello for any Transparency Log other than that identified by transparency_log_id,

3. A provisional tree head type was advertised in the ClientHello for the Transparency Log identified by transparency_log_id and proof_type is set to same_head.

Clients verify that the reference_ids field is sorted in ascending order, contains at least one entry, contains no duplicates, and that each index corresponds to a domain name or IP address in the leaf certificate's subjectAltName extension. Clients MUST NOT trust a connection for any name other than those in reference_ids.

Clients prepare a list, referred to as the *Current Authenticated Search Keys*, that contains the corresponding Prefix Tree search key for each reference identifier in reference_ids. This list is de-duplicated and stored in ascending lexicographic order. If the client advertised a provisional tree head type for the chosen Transparency Log, the client additionally prepares a list referred to as the *Past Authenticated Search Keys*. This list contains the union of every Current Authenticated Search Keys list from past connections where a provisional proof from the same Transparency Log was presented and the provisional proof has not yet been proven consistent with the *subsequent log entry* (defined as the first log entry sequenced after the provisional tree head was created). This list is also de-duplicated and stored in ascending lexicographic order.

Verification proceeds as follows, based on proof_type:

When proof_type is set to standard, this indicates that the inclusion proof is against a log entry that is currently published by the Transparency Log but more recent than the client may be aware of. These proofs are verified as follows:

1. If the client advertised a standard tree head type, verify that tree_head.size is greater than the size advertised by the client. If the client advertised a provisional tree head type, verify that tree_head.size is greater than the size of the associated provisional tree head.

2. Verify combined as executing the following proofs in this order:

1. Updating the view of the tree (Section 10.3.1 of [KEYTRANS]). This also verifies that the rightmost log entry's timestamp is within the bounds set by max_ahead and max_behind.

2. The client requires a PrefixProof from the rightmost log entry for the Current Authenticated Search Keys. If the client advertised a provisional tree head type for the chosen Transparency Log, the client additionally requires a PrefixProof from the subsequent log entry for the Past Authenticated Search Keys.

   If the rightmost log entry and the subsequent log entry are the same, then only one PrefixProof for the union of the two lists is provided in combined. Otherwise, if needed, the subsequent log entry's timestamp and a PrefixProof from the subsequent log entry for the Past Authenticated Search Keys is provided. This is followed by a second PrefixProof from the rightmost log entry for the Current Authenticated Search keys.

3. If the client advertised a provisional tree head type for the chosen Transparency Log, and the subsequent log entry is not the rightmost log entry, the client prepares a list referred to as the *Sequenced Search Keys*. This list is the intersection the Past Authenticated Search Keys and the Current Authenticated Search Keys lists, stored in lexicographic order.

   If this list is not empty, the sequencing field is expected to be present. Each entry in the sequencing field corresponds to each entry in the Sequenced Search Keys list. Each SequencingProof structure contains size of the Certificate Subtree for the given search key, and the first_valid and invalid_entries fields of its DomainCertificates structure, as it exists in the subsequent log entry. Clients: 1. Verify that SequencingProof.size is greater than or equal to the retained size of the Certificate Subtree. 2. Verify that SequencingProof.{first_valid, invalid_entries} would not unrevoke any previously revoked certificates.

4. Each entry in the subtree field corresponds to the union of the Current Authenticated Search Keys and Past Authenticated Search Keys lists, in lexicographic order. Entries corresponding to search keys that are in the Current Authenticated Search Keys list contain the search key's state in the rightmost log entry. Entries corresponding to search keys that are *only* in the Past Authenticated Search Keys list contain the search key's state in the subsequent log entry. For each entry in the subtree field:

1. If the corresponding search key is in the Sequenced Search Keys list, this means that there is an entry corresponding to the search key in sequencing. Compared to the fields of that SequencingProof structure:

    1. Verify that SubtreeInclusionProof.size is greater than or equal.

    2. Verify that SubtreeInclusionProof.{first_valid, invalid_entries} would not unrevoke any previously revoked certificates.

2. If the corresponding search key is in the Past Authenticated Search Keys list but not in the Current Authenticated Search Keys list:

    1. Verify that SubtreeInclusionProof.size is greater than or equal to the retained size of the Certificate Subtree.

    2. Verify that SubtreeInclusionProof.{first_valid, invalid_entries} would not unrevoke any previously revoked certificates.

3. If the corresponding search key is in the Current Authenticated Search Keys list, verify that SubtreeInclusionProof.{first_valid, invalid_entries} do not exclude SubtreeInclusionProof.position.

5. Compute the root hash of the Log Tree and verify the signature in tree_head.signature:

    1. For each entry in subtree, compute the root hash of the Certificate Subtree. This is done by interpreting SubtreeInclusionProof.inclusion as an inclusion proof in the Certificate Subtree for SubtreeInclusionProof.position. If the client advertised a provisional tree head and the corresponding search key is in the Past Authenticated Search Keys list, the proof will also function as a consistency proof as described in Section 3.2 of [KEYTRANS].

    2. For each entry in sequencing, compute the root hash of the Certificate Subtree. The inclusion proof in the subtree entry for the same search key will additionally contain the minimum set of node hashes necessary to compute the root of the Certificate Subtree at size SequencingProof.size.

3. With the root hashes of the Certificate Subtrees and the other fields in subtree and sequencing, compute the Prefix Tree leaf hash for each lookup that was done. As mentioned earlier, each search key in the Current Authenticated Search Keys list is looked up in the rightmost log entry and each search key in the Past Authenticated Search Keys list is looked up in the subsequent log entry, deduplicating if these log entries are the same.

4. With the Prefix Tree leaf hashes, compute the root hash of the Log Tree with combined. If the client advertised a provisional tree head, the inclusion proof in combined will also function as a consistency proof as described in Section 3.2 of [KEYTRANS].

When proof_type is set to provisional, this indicates that the inclusion proof is against a log entry that is not yet published by the Transparency Log. These proofs are verified as follows:

1. If the client advertised a standard tree head type, verify that tree_head.size is greater than or equal to the size advertised by the client. If the client advertised a provisional tree head type, verify that tree_head.size is greater than that of the advertised tree head, or that tree_head.size is equal and tree_head.counter is greater than that of the advertised tree head.

2. Verify combined as executing the following proofs in this order:

    1. Updating the view of the tree (Section 10.3.1 of [KEYTRANS]). This also verifies that the rightmost log entry’s timestamp is within the bounds set by max_ahead and max_behind.

    2. If the client advertised a provisional tree head type for the chosen Transparency Log and the subsequent log entry exists, then the subsequent log entry’s timestamp and a PrefixProof from it for the Past Authenticated Search Keys is added to combined.

3. If the client advertised a provisional tree head type for the chosen Transparency Log, and the subsequent log entry exists, the client prepares a list referred to as the *Sequenced Search Keys*. This list is the intersection the Past Authenticated Search Keys and the Current Authenticated Search Keys lists, stored in lexicographic order.

If this list is not empty, the sequencing field is expected to be present. Each entry in the sequencing field corresponds to each entry in the Sequenced Search Keys list. Each SequencingProof structure contains size of the Certificate Subtree for the given search key, and the first_valid and invalid_entries fields of its DomainCertificates structure, as it exists in the subsequent log entry. Clients: 1. Verify that SequencingProof.size is greater than or equal to the retained size of the Certificate Subtree. 2. Verify that SequencingProof.{first_valid, invalid_entries} would not unrevoke any previously revoked certificates.

4. Each entry in the subtree field corresponds to the union of the Current Authenticated Search Keys list and, if the client advertised a provisional tree head and the subsequent log entry exists, the Past Authenticated Search Keys list. Entries corresponding to search keys that are in the Current Authenticated Search Keys list contain the search key’s state in the provisional Prefix Tree. Entries corresponding to search keys that are *only* in the Past Authenticated Search Keys list contain the search key’s state in the subsequent log entry. For each entry in the subtree field:

    1. If the corresponding search key is in the Sequenced Search Keys list, this means that there is an entry corresponding to the search key in sequencing. Compared to the fields of that SequencingProof structure:

        1. Verify that SubtreeInclusionProof.size is greater than or equal.

        2. Verify that SubtreeInclusionProof.{first_valid, invalid_entries} would not unrevoke any previously revoked certificates.

    2. If the corresponding search key is in the Past Authenticated Search Keys list but not in the Current Authenticated Search Keys list:

        1. Verify that SubtreeInclusionProof.size is greater than or equal to the retained size of the Certificate Subtree.

        2. Verify that SubtreeInclusionProof.{first_valid, invalid_entries} would not unrevoke any previously revoked certificates.

3. If the corresponding search key is in the Current Authenticated Search Keys list, verify that SubtreeInclusionProof.{first_valid, invalid_entries} do not exclude SubtreeInclusionProof.position.

5. Compute the root hash of the Log Tree, the root hash of the provisional Prefix Tree, and verify the signature in tree_head.signature:

    1. For each entry in subtree, compute the root hash of the Certificate Subtree. This is done by interpreting SubtreeInclusionProof.inclusion as an inclusion proof in the Certificate Subtree for SubtreeInclusionProof.position. If the client advertised a provisional tree head and the corresponding search key is in the Past Authenticated Search Keys list, the proof will also function as a consistency proof as described in Section 3.2 of [KEYTRANS].

    2. For each entry in sequencing, compute the root hash of the Certificate Subtree. The inclusion proof in the subtree entry for the same search key will additionally contain the minimum set of node hashes necessary to compute the root of the Certificate Subtree at size SequencingProof.size.

    3. With the root hashes of the Certificate Subtrees and the other fields in subtree and sequencing, compute the Prefix Tree leaf hash for each lookup that was done. As mentioned earlier, each search key in the Current Authenticated Search Keys list is looked up in the provisional Prefix Tree and each search key in the Past Authenticated Search Keys list is looked up in the subsequent log entry if and only if it exists.

    4. With the Prefix Tree leaf hashes, compute the root hash of the Log Tree with combined, and compute the root hash of the provisional Prefix Tree with the proof in prefix. If the client advertised a provisional tree head, the inclusion proof in combined will also function as a consistency proof as described in Section 3.2 of [KEYTRANS].

When proof_type is set to same_head, this indicates that the inclusion proof is against the same tree head that was specified in the SupportedTransparencyLog structure for the chosen Transparency Log. These proofs are verified as follows:

1. For the advertised tree head, verify that the rightmost log entry’s timestamp is within the bounds set by max_ahead and max_behind.

2. Each entry in the subtree field corresponds to each entry of the Current Authenticated Search Keys list. If the client advertised a standard tree head type, each entry contains the corresponding search key's state as it exists in the rightmost log entry. If the client advertised a provisional tree head type, each entry contains the corresponding search key's state as it exists in the provisional Prefix Tree.

   For each entry in the subtree field, verify that SubtreeInclusionProof.{first_valid, invalid_entries} do not exclude SubtreeInclusionProof.position.

3. Compute the root hash of the Prefix Tree and verify that it matches the retained value:

   1. For each entry in subtree, compute the root hash of the Certificate Subtree. This is done by interpreting SubtreeInclusionProof.inclusion as an inclusion proof in the Certificate Subtree for SubtreeInclusionProof.position. If the client advertised a provisional tree head and the corresponding search key is in the Past Authenticated Search Keys list, the proof will also function as a consistency proof as described in Section 3.2 of [KEYTRANS].

   2. With the root hashes of the Certificate Subtrees and the other fields in subtree, compute the Prefix Tree leaf hash for each lookup that was done.

   3. With the Prefix Tree leaf hashes, compute the root hash of the Prefix Tree with the proof in prefix.

As was mentioned in Section 4, the entire certificate chain that will be presented by the server is stored in the leaf of the Certificate Subtree. If the server presents a different certificate chain to the client than was logged in the Transparency Log (for example, by including or omitting intermediates), the client will be unable to compute the correct Certificate Subtree root hash and proof verification will fail. Authenticating the entire certificate chain, instead of just the leaf, prevents the possibility of unlogged intermediate certificates. That is, it prevents the possibility of a leaf certificate being logged with a chain to a testing (or otherwise untrusted) trust anchor, and then being presented in TLS connections with additional intermediates that connect it to a different trust anchor.

6. Transparency Log Endpoints

Transparency Logs are online services that maintain a tree data structure and provide access to it through the endpoints described below. Generally, only Site Operators contact Transparency Logs. Site Operators regularly issue requests to the Transparency Log's endpoints to either obtain fresh inclusion proofs for their certificates or to monitor for mis-issuances affecting their properties.

Endpoints are accessible over HTTP or HTTPS. Which endpoint is being accessed is determined by the request's method and path. Request and response bodies are specific structures, which are encoded according to TLS format and then base64 encoded.

6.1. Get Tree

Site Operators access this endpoint to initialize or update their view of the tree.

GET /get-tree{?tree_size=X}

```
struct {
  TreeHead tree_head;
  CombinedTreeProof combined;
} GetTreeResponse;
```

The request optionally contains a tree_size query parameter containing the size of the most recent tree head that has been observed by the client. If present, tree_size MUST be the size of a tree head that was published less than or equal to max_behind milliseconds in the past.

If the tree_size query parameter was provided but a more recent tree head than tree_size has not been issued, the Transparency Log MAY respond with a 204 No Content status code and an empty response body. Otherwise, the response body is GetTreeResponse. The tree_head field contains the most recent tree head published by the Transparency Log. The combined field contains the following:

* The output of updating the view of the tree (from tree_size, if provided) to tree_head.tree_size (Section 10.3.1 of [KEYTRANS]).

* The timestamps of all log entries that have timestamps less than or equal to 10*max_behind, excluding those contained in tree_size. These timestamps are provided in left-to-right order. Note that some of them may be omitted if they are duplicates with the previous bullet, as explained in Section 10.3 of [KEYTRANS].

If tree_size is provided, the proof in combined.inclusion is additionally a consistency proof.

6.2. Add Chain

Site Operators access this endpoint to produce an inclusion proof for their certificate chain. Transparency Logs SHOULD accept certificates issued by a broad set of the current widely-trusted Certificate Authorities. This ensures that, if one Transparency Log has an outage, there are several other Transparency Logs that Site Operators can failover to and fetch fresh inclusion proofs from.

POST /add-chain

```
struct {
  uint64 tree_size;
  uint32 subtree_sizes<1..2^8-1>;

  uint8 reference_ids<1..2^8-1>;
  Certificate chain<1..2^8-1>;
  opaque signature<0..2^16-1>;
} AddChainRequest;

struct {
  TreeHeadType tree_head_type;
  select (AddChainResponse.tree_head_type) {
    case provisional:
      ProvisionalTreeHead tree_head;
      CombinedTreeProof combined;
      PrefixProof prefix;
      SubtreeInclusionProof subtree<1..2^8-1>;
  }
  BearerToken bearer_token;
} AddChainResponse;
```

The request body is AddChainRequest. The tree_size field contains the size of the most recent tree head observed by the client. The subtree_sizes field contains the greatest size of the Certificate Subtree observed by the client for each reference identifier, in the order the reference identifiers are provided in reference_ids. The reference_ids field contains the reference identifiers that the certificate chain will be validated against, identified by their zero-indexed offset into the leaf certificate's subjectAltName extension. The chain field contains the certificate chain itself. The first entry in chain is assumed to be the leaf certificate and each subsequent certificate is assumed to authenticate the one prior, ending at a certificate which is authenticated by a trust anchor. The signature field contains a signature from the public key in the leaf certificate over the following:

TODO Specify signature challenge

The Transparency Log verifies that:

1. The tree_size field corresponds to a tree head that was issued less than or equal to max_behind milliseconds in the past.

2. The subtree_sizes field has the same length as the reference_ids field and each entry is less than or equal to the most recent size of the corresponding Certificate Subtree.

3. The reference_ids field is in ascending order, each entry corresponds to a domain name or IP address in the leaf certificate's subjectAltName extension, and each entry corresponds to a unique Prefix Tree search key.

4. The certificate chain in chain is valid.

5. The signature in signature is valid.

The response body is AddChainResponse. If tree_head_type is set to standard, this indicates that the exact certificate chain is already present in the Certificate Subtrees for all of the reference identifiers and published in the rightmost log entry. If tree_head_type is set to provisional, this indicates that a provisional tree head has been issued for the chain. The provisional tree head is given in tree_head and an inclusion proof in the provisional Prefix Tree for all requested reference identifiers is given in prefix. The combined field contains the same contents as the Get Tree endpoint would. The subtree field contains an inclusion proof in the Certificate Subtree for each reference identifier, in the order the reference identifiers are provided in reference_ids. If an entry of AddChainRequest.subtree_sizes is greater than zero, then the corresponding AddChainResponse.subtree[i].inclusion is additionally a consistency proof.

The bearer_token field of an AddChainResponse is arbitrarily set by the Transparency Log and used to authenticate requests to the Refresh Proof endpoint. The bearer token will stop working once the associated certificate chain has expired, regardless of revocation status.

6.3. Refresh Proof

Site Operators access this endpoint to refresh an inclusion proof for their certificate chain, making it acceptable to clients for a longer period of time.

```
GET /refresh-proof?bearer_token=X&position=Y
```

```
struct {
  PrefixProof prefix;
  SubtreeInclusionProof subtree<1..2^8-1>;
} RefreshProofResponse;
```

The request contains a bearer token obtained from the Add Chain endpoint in the bearer_token query parameter, and the position of a log entry in the position query parameter. The bearer token is encoded with URL-safe base64, described in Section 5 of [RFC4648]. The position query parameter MUST either be the subsequent log entry issued after the provisional tree head (if any) associated with bearer_token, or be a log entry issued less than or equal to max_behind milliseconds in the past.

The response body is RefreshProofResponse. The prefix field contains an inclusion proof from the Prefix Tree stored at the requested log entry, specifically for the reference identifiers associated with bearer_token. The subtree field contains inclusion proofs for the

certificate chain in the Certificate Subtree of each reference identifier associated with bearer_token, in the order the reference identifiers were provided.

6.4. Issue Token

Site Operators access this endpoint to obtain a bearer token to use when accessing the Transparency Log endpoints described in subsequent sections.

```
POST /issue-token

struct {
  Certificate chain<1..2^8-1>;
  opaque signature<0..2^16-1>;
} IssueTokenRequest;

struct {
  BearerToken bearer_token;
} IssueTokenResponse;
```

The request body is IssueTokenRequest. The chain field contains a certificate chain possessed by the client. The first entry in chain is assumed to be the leaf certificate and each subsequent certificate is assumed to authenticate the one prior, ending at a certificate which is authenticated by a trust anchor. The signature field contains a signature from the public key in the leaf certificate over the challenge specified in Section 6.2.

The Transparency Log verifies the certificate chain and the signature challenge. If verification is successful, the response body is IssueTokenResponse. The bearer_token field is used to authenticate requests to the Transparency Log endpoints described below. The bearer token will stop working once the associated certificate chain has expired, regardless of revocation status.

6.5. Get Certificates

Site Operators access this endpoint for the purpose of auditing certificates that have been issued for their domain names or IP addresses.

```
GET /get-certificates?bearer_token=X&reference_id=Y{&start=Z}
```

```
struct {
  select (start query parameter not present) {
    case true:
      uint32 first_valid;
      NodeValue full_subtrees<0..2^8-1>;
  }
  SubtreeLogLeaf leaves<0..2^8-1>;
} GetCertificatesResponse;
```

The bearer_token query parameter contains a bearer token obtained from the Issue Token endpoint encoded in URL-safe base64. The reference_id query parameter may contain any domain name or IP address authenticated by the leaf certificate associated with the provided bearer token, or any domain name that is suffixed by any authenticated domain name. The optional start query parameter contains the requested start position in the Certificate Subtree.

Note that reference_id may be a domain name that is not authenticated by the certificate. For example, a certificate that only authenticates example.com would not be accepted by TLS clients for sub.example.com. However, sub.example.com would still be an acceptable input to this endpoint since it has the suffix .example.com.

The response body is GetCertificatesResponse. If the start query parameter is not present, the first_valid field contains the lowest value for first_valid in any DomainCertificates structure for the reference identifier that is currently published (meaning, excluding DomainCertificates structures in log entries that are past their maximum lifetime). The full_subtrees field contains the full subtrees of the Certificate Subtree, in left-to-right order, up to but excluding the first_valid entry.

The leaves field contains SubtreeLogLeaf structures in the same order that they were sequenced in the Certificate Subtree for the requested reference identifier, starting at position start or starting at position first_valid is start is not present. If the start query parameter is present, it MUST be greater than first_valid.

6.6. Get Subdomains

Site Operators access this endpoint to learn about subdomains of domains they control that have logged certificates.

GET /get-subdomains?bearer_token=W&reference_id=X&position=Y&start=Z

```
struct {
  opaque search_key<0..2^8-1>;
  uint32 size;
  uint32 first_valid;
  uint32 invalid_entries<0..2^8-1>;
} Subdomain;

struct {
  PrefixProof prefix;
  Subdomain subdomains<0..2^8-1>;
} GetSubdomainsResponse;
```

The bearer_token query parameter contains a bearer token obtained from the Issue Token endpoint. The reference_id query parameter may contain the zero-indexed offset of any domain name in the subjectAltName extension of the leaf certificate associated with the provided bearer token. The position query parameter is the position of a log entry that has not passed its maximum lifetime. The start query parameter contains the number of subdomains to skip in the endpoint’s output.

The response body is GetSubdomainsResponse. Each entry of subdomains corresponds to a search key stored in the Prefix Tree that is prefixed with the search key corresponding to reference_id plus an additional dot. The subdomains array is sorted by lexicographic order of the corresponding Prefix Tree search keys. In each entry, the search_key field contains the Prefix Tree search key while the size, first_valid, and invalid_entries fields match the corresponding DomainCertificates structure for the search key as it exists in the log entry at position.

If any search keys with the required prefix exist in the Prefix Tree in the log entry at position, the prefix field contains an inclusion proof for all the search keys returned in subdomains. If no search keys with the required prefix exist, the prefix field contains a non-inclusion proof for the prefix.

Clients rely on the structure of the Prefix Tree to authenticate whether or not all subdomains have been provided yet by the Transparency Log. Clients may need to make multiple requests to this endpoint, each time increasing the start query parameter to reflect subdomains already consumed, if the total number of subdomains exceeds a per-response limit.

6.7. Add Revocation

Site Operators or Certificate Authorities access this endpoint for the purpose of distributing revocations for their certificates.

```
POST /add-revocation

struct {
  Revocation revocation;
} AddRevocationRequest;
```

The request body is AddRevocationRequest. There is no response body; an HTTP response status code of 204 indicates success, and a response status code in the 400-599 range indicates failure of the submission. The Transparency Log applies the revocation by updating any DomainCertificates structures to exclude chains that are affected by the revocation. The revocation SHOULD be applied to all DomainCertificates structures within max_behind milliseconds.

TODO define Revocation type

6.8. Get Revocations

Site Operators access this endpoint to audit revocations affecting their certificate chains.

```
GET /get-revocations?bearer_token=X&reference_id=Y{&page=Z}

struct {
  Revocation revocations<0..2^8-1>;
  optional<uint32> next;
} GetRevocationsResponse;
```

The bearer_token query parameter contains a bearer token obtained from the Issue Token endpoint. The reference_id query parameter may contain the zero-indexed offset of any domain name or IP address in the subjectAltName extension of the leaf certificate associated with the provided bearer token. An optional page number may be provided in the page query parameter. The page parameter MUST be a value observed in the next field of a prior GetRevocationsResponse.

The response body is GetRevocationsResponse. The revocations field of the response contains Revocation structures corresponding one-to-one with the invalid_entries field of the most recent DomainCertificates structure in the Certificate Subtree for the requested reference identifier. Note that the most recent DomainCertificates structure may only have been published in a provisional tree head at the time of the request. If the last entry of revocations does not correspond to the last entry of invalid_entries, a page number in next is provided to allow the client to make a follow-up request for the next subset of revocations. The Transparency Log MUST set next such that, if the invalid_entries field is modified while a client is requesting a series of pages, the client will not miss any revocations that existed as of the first request (with no page parameter) as a result.

7. Extended Resolution Mechanisms

When clients observe a provisional inclusion proof, they retain condensed state about the proof until they observe that the associated certificate chain was properly included in the subsequent log entry. The primary mechanism by which clients are expected to observe this is through future connections to the same server. However, this may fail to happen for various reasons: the client may not wish to contact the server again within the allotted time, the server may go offline, or the server may simply decline to respond with a proof from the same Transparency Log. As such, additional mechanisms are described to enable the client to reach resolution on all provisional inclusion proofs it observes.

7.1. Background Requests

When a client has an unresolved provisional inclusion proof, where the rightmost log entry’s timestamp is between 5*max_behind and 10*max_behind milliseconds in the past, the client SHOULD attempt a single background request to the server that provided the proof. A *background request* is a connection attempt that is not initiated by a user and will not carry user request data. The client advertises in its ClientHello only provisional tree head types where the rightmost log entry has a timestamp more than max_behind milliseconds in the past. If the connection succeeds, any certificate the server responds with will necessarily provide the information the client needs to purge a previously observed provisional inclusion proof from its state. Any additional provisional inclusion proof provided by the server in such a connection SHOULD be disregarded.

7.2. Oblivious Third Party

If provided, clients may attempt to contact a third-party service, potentially operated by their software vendor, to request proof that a subsequent log entry is constructed correctly. Such an endpoint could be contacted over Oblivious HTTP [RFC9458] to preserve the client's privacy.

7.3. Final Reporting

If a client has been unable to resolve a provisional inclusion proof on its own, and the rightmost log entry's timestamp is more than 10*max_behind milliseconds in the past, the client MUST report the provisional inclusion proof to a party distinct from the issuing Certificate Authority and the operator of the Transparency Log. After this, the client MAY delete the associated state.

The purpose of reporting provisional inclusion proofs that are unable to be resolved is to ensure that there is broader ecosystem awareness of a potential issue with the Transparency Log. The extent to which there was any malicious behavior or operational errors, and any corrective action to be taken, would need to be decided out-of-band.

8. Operational Considerations

8.1. Client State

Clients retain the following state:

* For each fork of each trusted Transparency Log:
  - The timestamp and Prefix Tree root hash for any observed log entries that may be useful to advertise in the client's ClientHello:
    - All observed log entries with a timestamp less than or equal to max_behind milliseconds in the past.
    - The rightmost observed log entry with a timestamp more than max_behind milliseconds in the past.
  - Any intermediate Log Tree nodes that are necessary to compute the most recent Log Tree root hash from the retained log entries.
* For each provisional inclusion proof observed that has not been replaced with a superseding provisional inclusion proof, or been shown to be included in the subsequent log entry:

- The domain or IP address of the host that presented the provisional proof.
- The ProvisionalTreeHead structure.
- The full subtrees of the Log Tree, as presented in the provisional proof.
- The authenticated reference identifiers
- The size, first_valid, and invalid_entries fields of each SubtreeInclusionProof structure.
- The full subtrees of each Certificate Subtree.
- The bearer token and pre-shared key associated with the provisional inclusion proof.

Clients MUST only update their stored state once a proof has been fully and successfully verified. In addition, clients MUST be able to handle being shown forked views of a Transparency Log.

When a client advertises a provisional tree head to the server and the server responds with a standard proof type, the server's response will necessarily include proof that the certificate in the provisional tree head was correctly included in the subsequent log entry. As such, the information retained about the provisional inclusion proof is deleted. Similarly, when a client advertises a provisional tree head to the server and the server responds with a provisional proof type, the server's response will contain proof that the new provisional inclusion proof supersedes (i.e., contains all the same certificates as) the previous one. As such, state for the previous provisional inclusion proof is deleted and replaced with state for the new one.

Regardless of whether the server responds with a standard or provisional proof type, if the server presents a new view of the Log Tree that the client was previously unaware of, the client retains this new view for later use. If possible, the client stores the new view as an extension of a previously-observed view of the Transparency Log. However, if the view is inconsistent with what the client has previously observed, then it is stored as a new independent fork.

8.2. Server Behavior

To prevent connection failure, it's critical that servers that implement the TLS extension in Section 5 always have satisfactory proof to offer to clients. Servers MUST implement the automatic refreshing of proofs and MUST implement automatic failover between multiple trusted Transparency Logs in the event that one is temporarily unavailable.

Along this same line, it's also possible for a Transparency Log to sequence incorrect information and no longer be able to provide acceptable proofs. This can happen, for example, if a Transparency Log issues a provisional inclusion proof for a certificate and then neglects to include the certificate in the subsequent log entry. This is functionally equivalent to a prolonged outage, as the server is unable to obtain an acceptable inclusion proof for its certificate. As such, servers MUST verify proofs in the same manner as clients and avoid serving proofs that fail verification, failing over to another Transparency Log if necessary to get an acceptable proof.

Given the substantial load that may be placed on Transparency Logs, especially in scenarios where one log's traffic is failing over to others, servers have a responsibility to minimize their individual impact on logs. Servers SHOULD NOT attempt to refresh an inclusion proof from a Transparency Log until the rightmost log entry's timestamp in the current inclusion proof is more than max_behind*3/4 milliseconds in the past. Servers SHOULD NOT contact a Transparency Log about the same inclusion proof more than 3 times within max_behind milliseconds.

Finally, servers MUST generate the bearer tokens that are provided to clients in a way that, when the bearer token is advertised back to the server, does not degrade the client's privacy. One suggested way to do this would be to make the bearer token a symmetric encryption of an identifier for the associated provisional certificate along with a 12- or 16-byte random value. The random value would then be used to compute the pre-shared key to give the client. When the client later advertises the bearer token back, it can be decrypted by the server to identify the provisional certificate to respond with and to re-compute the pre-shared key for the connection. This minimizes the server operational burden and also effectively preserves client privacy.

8.3. Handling Forks

There are several long-term expectations placed on Transparency Logs that, in practice, will almost certainly fail to be upheld. In particular, Transparency Logs are generally expected to be append-only, meaning that log entries are never removed once they've been added. However, enforcing this strictly is incompatible with many of the standard best-practices for operating reliable software systems. For example, if this property needed to be enforced strictly, it would be pointless to back up such a Transparency Log. Restoring from backup would imply some degree of rollback and this would be unacceptable. As such, it is necessary for the protocol to handle reasonable violations of these expectations gracefully, and in a way that preserves overall security.

As discussed in Section 8.1, clients MUST be able to handle forked views of a Transparency Log gracefully. When a client becomes aware of a fork, it MUST report the fork to a party distinct from the issuing Certificate Authority and the operator of the Transparency Log. Going forward, as long as the client is aware of multiple forks where the timestamps of the rightmost log entries differ by less than max_behind milliseconds, the client MUST NOT advertise in its ClientHello a standard tree head type with a tree size past where any such fork occurs. This minimizes the risk of connection failure when connecting to servers that have inclusion proofs from different forks.

If a client advertised a non-zero standard tree head type and proof verification failed only at the final signature verification step (or at direct comparison of the Prefix Tree root hash, in the case of SameHeadProof), this may indicate that there is a fork the client is unaware of. The client MAY attempt to reconnect to the server while advertising a zero standard tree head type. This will prompt the server to provide additional intermediate nodes, allowing the client to compute the correct Log Tree root hash for signature verification to succeed.

Excluding the production of forks, the other ways that a Transparency Log can fail are much simpler to handle because we know that the client and server agree on the state of the log. These cases are handled by the provisions of Section 8.2.

9. Certificate Authority

9.1. Poison Extension

A trivial downgrade attack is possible where a host refuses to acknowledge the presence of the "transparency_revocation" TLS extension in an attempt to circumvent the stronger transparency or non-revocation requirements. Customers of a Certificate Authority can mitigate such an attack by including a special non-critical poison extension (OID TODO, whose extnValue OCTET STRING contains ASN.1 NULL data (0x05 0x00)) in all certificates they issue.

Clients that advertise the "transparency_revocation" extension in their ClientHello MUST reject a certificate that contains the extension if it is not provided with an appropriate inclusion proof.

10. Performance Considerations

This section contains brief a analysis of the performance impacts of the system.

10.1. Transparency Log

Storage. The storage of a Transparency Log is long-term bounded (i.e., Transparency Logs will not grow infinitely) and is dominated by the cost of storing _unexpired_ certificate chains. Certificate chains can be purged once they have expired and are at the leftmost edge of the Certificate Subtree. Any associated revocations can be purged once the associated certificate chains have been. Subtrees of the Log Tree and Prefix Tree may be purged once they are no longer necessary to the protocol. For specific leaves of the Log Tree, this occurs once they are older than maximum_lifetime milliseconds. For subtrees of the Prefix Tree, this occurs once all certificate chains stored in the subtree have expired.

Bandwidth. A TLS server maintaining a valid inclusion proof for a single certificate chain is expected to make the following requests to a Transparency Log:

1. When a new certificate chain is first issued, the server will initialize its state (a Get Tree request) and submit the chain to receive a provisional inclusion proof (an Add Chain request).

2. From then on, the server will regularly obtain the most recent tree head (a Get Tree request with the tree_size parameter provided) and refresh its inclusion proof (a Refresh Proof request).

The table below provides formulas for the size in bytes of the Transparency Log's response to each query. Each variable in the formulas is also given a "typical" value which is used to estimate the total bandwidth commitment of a Transparency Log.

S The size of one cryptographic signature in bytes. For the purpose of computing an estimate, S is assumed to be 64 bytes as this is the size of a typical elliptic curve signature.

N The height of the Log Tree. Based on a Transparency Log that sequences two new log entries per day, and that has been operating for 365 days, N is estimated to be 11.

P The average height of the Prefix Tree. Based on a Transparency Log that stores 300 million unique identifiers, P is estimated to be 30.

M The average height of a Certificate Subtree. Based on an assumption that a typical Site Operator will issue several hundred certificates, M is estimated to be 10.

Response Type	Formula	Estimated Size	Count
GetTreeResponse (full)	694 + S + 52*N	1330	1
GetTreeResponse (abridged)	54 + S	118	179
AddChainResponse	51 + S + 32_P + 32_M	1395	1
RefreshProofResponse	15 + 32*P	975	179

Table 1

The "Count" column of the table contains the number of times each response will be served to a client, assuming a certificate lifetime of 3 months and that the server refreshes its proof twice every day and a half. As such, it's estimated that supporting a single Site Operator requires an outbound bandwidth commitment from a Transparency Log of approximately 0.2 bits per second.

10.2. TLS Server

One particularly salient concern about the future of the [RFC6962] ecosystem is that it requires a multitude of signatures from different co-signers and this may not translate well to a world where those signatures are required to come from post-quantum signature schemes. Post-quantum signature schemes produce signatures that are substantially larger than classical ones, and transmitting a significant number of them in the TLS handshake will degrade performance.

The appeal of the system described in this document is that it replaces up to three signatures (two from distinct Certificate Transparency logs and one from an OCSP staple) with either one or zero signatures.

In the TLS extension described in Section 5, the most common response one would expect in the TLS server's Certificate message is a SameHeadProof. This is because clients make an effort to advertise the specific version of the Transparency Log that the server can provide a proof for without also transmitting a signature.

Using the variables from the previous subsection, the size in bytes of a SameHeadProof is 15 + 32*P + 32*M, which would then be estimated at 1295 bytes in a typical deployment. Compared to transmitting three ML-DSA-44 signatures, which would add to 7680 bytes, this is an 83% reduction.

11. Security Considerations

11.1. ClientHello Extension

Given that ClientHello extensions are sent unencrypted, this portion of the extension was designed to avoid unnecessary privacy leaks. In particular, care was taken to avoid leaking what certificate(s) the client may have been shown in previous connections and what other hosts the client may have contacted recently.

Clients advertise a recently observed tree size for each Transparency Log that they support receiving inclusion proofs from. Since clients will generally only "observe" various tree sizes of a Transparency Log by communicating with hosts that provide proofs from that Transparency Log, and since different hosts will update their proofs at different times, this may cause a privacy leak. Specifically, it could happen that a client communicates with a host that uses proofs from a very recently-created tree size. If the client advertised this very recently-created tree size to other hosts, it would reveal who they previously communicated with.

To mitigate this, clients only advertise tree sizes where the timestamp of the rightmost log entry is sufficiently old. This time delay ensures that the inclusion proof provided by almost any host could've conveyed the same tree size, creating a large anonymity set.

When a client observes a provisional inclusion proof from a host, they retain condensed information about it to allow them to later verify that the information it contained was properly integrated into the Transparency Log. The primary avenue for obtaining this verification is advertising knowledge of the provisional proof back to the host that it came from, hoping to get the necessary information in-band.

Since provisional inclusion proofs must be issued quickly, they don't have time to build up a large anonymity set with other hosts. Instead of having clients advertise knowledge of a specific provisional proof in their ClientHello, they instead use a bearer token that was provided by the host. These bearer tokens are provided in the encrypted Certificate message whenever a provisional proof is shown by the host. Similarly, when the bearer token is redeemed (i.e., when the host shows that the provisional proof was correctly integrated into the Transparency Log), this information is provided in the encrypted Certificate message. As such, if the host generates the bearer token in a secure way, a passive network observer never sees anything that would identify the certificate shown to the client.

Each bearer token is additionally associated with a pre-shared key which is provided to the TLS key schedule. This prevents an active attacker from establishing a TLS connection to the host, advertising an observed bearer token, and learning which certificate is provided.

Finally, note that it is not a goal to prevent an attacker from learning whether a client has previously contacted a host _at all_ before. The protocol explicitly relies on the client's stored state to remain secure while sending minimal data over the wire. A passive observer of network traffic could trivially determine from the size of the encrypted portion of the handshake messages whether such state was present or not, and therefore whether the host had been contacted before. Similarly, it is not a goal to prevent a host from identifying the same client over many connections.

11.2. Downgrade Prevention

The stronger transparency and non-revocation guarantees this protocol provides would be irrelevant if a malicious actor could cause the TLS client to disable them at-will. An attacker may have, for example, a revoked certificate to which they know the private key. This would allow them to fully intercept a client's connection to a host and attempt to impersonate the host. In a downgraded version of TLS, the client may not enforce revocation at all and therefore the attacker's interception would succeed.

Site Operators that have deployed this protocol and wish to prevent capable TLS clients from being downgraded can include a poison extension in their TLS certificates, as described in Section 9.1. The poison extension will be silently ignored by TLS clients that genuinely do not support it, but will cause updated clients to abort the protocol in downgrade scenarios.

Site Operators can monitor the existing [RFC6962] ecosystem to detect any certificates that have been issued without the poison extension, potentially permitting downgrades. However, the creation of such a certificate without the Site Operator's consent would imply mis-issuance by a Certificate Authority rather than abuse of a compromised/revoked certificate.

12. IANA Considerations

* Codepoint for TLS extension "transparency_revocation"

* Registry for transparency_log_id(?)

* OID for poison extension

13. References

13.1. Normative References

[KEYTRANS] McMillion, B. and F. Linker, "Key Transparency Protocol", Work in Progress, Internet-Draft, draft-ietf-keytrans-protocol-01, 4 June 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-keytrans-protocol-01>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/rfc/rfc4291>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

13.2. Informative References

[AppleKT] Apple, "Advancing iMessage security: iMessage Contact Key Verification", 27 October 2023, <https://security.apple.com/blog/imessage-contact-key-verification/>.

[CertLifetimes] "SC-081v3: Introduce Schedule of Reducing Validity and Data Reuse Periods", n.d., <https://groups.google.com/a/groups.cabforum.org/g/servercert-wg/c/bvWh5RN6tYI>.

[Clubcards] Schanck, J. M. and Mozilla, "Clubcards for the WebPKI: smaller certificate revocation tests in theory and practice", 1 April 2025, <https://research.mozilla.org/files/2025/04/clubcards_for_the_webpki.pdf>.

[CRLite] Jones, J.C. and Mozilla, "Introducing CRLite: All of the Web PKI’s revocations, compressed", 9 January 2020, <https://blog.mozilla.org/security/2020/01/09/crlite-part-1-all-web-pki-revocations-compressed/>.

[CRLSets] "Chromium Security > CRLSets", n.d., <https://www.chromium.org/Home/chromium-security/crlsets/>.

[KeyTransWG] "Key Transparency (keytrans) Working Group", n.d., <https://datatracker.ietf.org/wg/keytrans/about/>.

[MetaKT] Meta, "Deploying key transparency at WhatsApp", 13 April 2023, <https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/>.

[ProtonKT] Proton, "What is Key Transparency?", n.d., <https://proton.me/support/key-transparency>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/rfc/rfc6962>.

[RFC7633] Hallam-Baker, P., "X.509v3 Transport Layer Security (TLS) Feature Extension", RFC 7633, DOI 10.17487/RFC7633, October 2015, <https://www.rfc-editor.org/rfc/rfc7633>.

[RFC9458] Thomson, M. and C. A. Wood, "Oblivious HTTP", RFC 9458, DOI 10.17487/RFC9458, January 2024, <https://www.rfc-editor.org/rfc/rfc9458>.

Acknowledgments

TODO acknowledge.

Authors' Addresses

Brendan McMillion
Email: brendanmcmillion@gmail.com

Devon O'Brien
Google LLC
Email: asymmetric@google.com

Dennis Jackson
Mozilla
Email: ietf@dennis-jackson.uk