Workgroup: TRANS (Public Notary Transparency)
Internet-Draft:
draft-ietf-trans-rfc6962-bis-42
Obsoletes: 6962 (if approved)
Published: 31 August 2021
Intended Status: Experimental
Expires: 4 March 2022
Authors: B. Laurie (Google), A. Langley (Google), E. Kasper (Google), E. Messeri (Google), R. Stradling (Sectigo)

Certificate Transparency Version 2.0

Abstract

This document describes version 2.0 of the Certificate Transparency (CT) protocol for publicly logging the existence of Transport Layer Security (TLS) server certificates as they are issued or observed, in a manner that allows anyone to audit certification authority (CA) activity and notice the issuance of suspect certificates as well as to audit the certificate logs themselves. The intent is that eventually clients would refuse to honor certificates that do not appear in a log, effectively forcing CAs to add all issued certificates to the logs.

This document obsoletes RFC 6962. It also specifies a new TLS extension that is used to send various CT log artifacts.

Logs are network services that implement the protocol operations for submissions and queries that are defined in this document.

[RFC Editor: please update 'RFCXXXX' to refer to this document, once its RFC number is known, through the document. Also, the OID assigned below must also appear in the appendix as indicated.]

Status of This Memo

This Internet-Draft will expire on 4 March 2022.

Copyright Notice

Table of Contents

1. Introduction

Certificate Transparency aims to mitigate the problem of misissued certificates by providing append-only logs of issued certificates. The logs do not themselves prevent misissuance, but they ensure that interested parties (particularly those named in certificates) can detect such misissuance. Note that this is a general mechanism that could be used for transparently logging any form of binary data, subject to some kind of inclusion criteria. In this document, we only describe its use for public TLS server certificates (i.e., where the inclusion criteria is a valid certificate issued by a public certification authority (CA)). A typical definition of "public" can be found in [CABBR].

Each log contains certificate chains, which can be submitted by anyone. It is expected that public CAs will contribute all their newly issued certificates to one or more logs; however, certificate holders can also contribute their own certificate chains, as can third parties. In order to avoid logs being rendered useless by the submission of large numbers of spurious certificates, it is required that each chain ends with a trust anchor that is accepted by the log. A log may also limit the length of the chain it is willing to accept; such chains must also end with an acceptable trust anchor. When a chain is accepted by a log, a signed timestamp is returned, which can later be used to provide evidence to TLS clients that the chain has been submitted. TLS clients can thus require that all certificates they accept as valid are accompanied by signed timestamps.

Those who are concerned about misissuance can monitor the logs, asking them regularly for all new entries, and can thus check whether domains for which they are responsible have had certificates issued that they did not expect. What they do with this information, particularly when they find that a misissuance has happened, is beyond the scope of this document. However, broadly speaking, they can invoke existing business mechanisms for dealing with misissued certificates, such as working with the CA to get the certificate revoked, or with maintainers of trust anchor lists to get the CA removed. Of course, anyone who wants can monitor the logs and, if they believe a certificate is incorrectly issued, take action as they see fit.

Similarly, those who have seen signed timestamps from a particular log can later demand a proof of inclusion from that log. If the log is unable to provide this (or, indeed, if the corresponding certificate is absent from monitors' copies of that log), that is evidence of the incorrect operation of the log. The checking operation is asynchronous to allow clients to proceed without delay, despite possible issues such as network connectivity and the vagaries of firewalls.

The append-only property of each log is achieved using Merkle Trees, which can be used to efficiently prove that any particular instance of the log is a superset of any particular previous instance and to efficiently detect various misbehaviors of the log (e.g., issuing a signed timestamp for a certificate that is not subsequently logged).

It is necessary to treat each log as a trusted third party, because the log auditing mechanisms described in this document can be circumvented by a misbehaving log that shows different, inconsistent views of itself to different clients. While mechanisms are being developed to address these shortcomings and thereby avoid the need to blindly trust logs, such mechanisms are outside the scope of this document.

1.1. 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.

1.2. Data Structures

Data structures are defined and encoded according to the conventions laid out in Section 3 of [RFC8446].

This document uses object identifiers (OIDs) to identify Log IDs (see Section 4.4), the precertificate CMS eContentType (see Section 3.2), and X.509v3 extensions in certificates (see Section 7.1.2) and OCSP responses (see Section 7.1.1). The OIDs are defined in an arc that was selected due to its short encoding.

1.3. Major Differences from CT 1.0

This document revises and obsoletes the CT 1.0 [RFC6962] protocol, drawing on insights gained from CT 1.0 deployments and on feedback from the community. The major changes are:

* Hash and signature algorithm agility: permitted algorithms are now specified in IANA registries.
* Precertificate format: precertificates are now CMS objects rather than X.509 certificates, which avoids violating the certificate serial number uniqueness requirement in Section 4.1.2.2 of [RFC5280].
* Removed precertificate signing certificates and the precertificate poison extension: the change of precertificate format means that these are no longer needed.
* Logs IDs: each log is now identified by an OID rather than by the hash of its public key. OID allocations are managed by an IANA registry.
* TransItem structure: this new data structure is used to encapsulate most types of CT data. A TransItemList, consisting of one or more TransItem structures, can be used anywhere that SignedCertificateTimestampList was used in [RFC6962].
* Merkle tree leaves: the MerkleTreeLeaf structure has been replaced by the TransItem structure, which eases extensibility and simplifies the leaf structure by removing one layer of abstraction.
* Unified leaf format: the structure for both certificate and precertificate entries now includes only the TBSCertificate (whereas certificate entries in [RFC6962] included the entire certificate).
* Log Artifact Extensions: these are now typed and managed by an IANA registry, and they can now appear not only in SCTs but also in STHs.
* API outputs: complete TransItem structures are returned, rather than the constituent parts of each structure.
* get-all-by-hash: new client API for obtaining an inclusion proof and the corresponding consistency proof at the same time.
* submit-entry: new client API, replacing add-chain and add-pre-chain.

* Presenting SCTs with proofs: TLS servers may present SCTs together with the corresponding inclusion proofs using any of the mechanisms that [RFC6962] defined for presenting SCTs only. (Presenting SCTs only is still supported).

* CT TLS extension: the signed_certificate_timestamp TLS extension has been replaced by the transparency_info TLS extension.

* Verification algorithms: added detailed algorithms for verifying inclusion proofs, for verifying consistency between two STHs, and for verifying a root hash given a complete list of the relevant leaf input entries.

* Extensive clarifications and editorial work.

2. Cryptographic Components

2.1. Merkle Hash Trees

A full description of Merkle Hash Tree is beyond the scope of this document. Briefly, it is a binary tree where each non-leaf node is a hash of its children. For CT, the number of children is at most two. Additional information can be found in the Introduction and Reference section of [RFC8391].

2.1.1. Definition of the Merkle Tree

The log uses a binary Merkle Hash Tree for efficient auditing. The hash algorithm used is one of the log's parameters (see Section 4.1). This document establishes a registry of acceptable hash algorithms (see Section 10.2.1). Throughout this document, the hash algorithm in use is referred to as HASH and the size of its output in bytes as HASH_SIZE. The input to the Merkle Tree Hash is a list of data entries; these entries will be hashed to form the leaves of the Merkle Hash Tree. The output is a single HASH_SIZE Merkle Tree Hash. Given an ordered list of n inputs, D_n = {d[0], d[1], ..., d[n-1]}, the Merkle Tree Hash (MTH) is thus defined as follows:

The hash of an empty list is the hash of an empty string:

```
MTH({}) = HASH().
```

The hash of a list with one entry (also known as a leaf hash) is:

```
MTH({d[0]}) = HASH(0x00 || d[0]).
```

For n > 1, let k be the largest power of two smaller than n (i.e., k < n <= 2k). The Merkle Tree Hash of an n-element list D_n is then defined recursively as

MTH(D_n) = HASH(0x01 || MTH(D[0:k]) || MTH(D[k:n])),

where:

*|| denotes concatenation

*: denotes concatenation of lists

*D[k1:k2] = D'_(k2-k1) denotes the list {d'[0] = d[k1], d'[1] = d[k1+1], ..., d'[k2-k1-1] = d[k2-1]} of length (k2 - k1).

Note that the hash calculations for leaves and nodes differ; this domain separation is required to give second preimage resistance.

Note that we do not require the length of the input list to be a power of two. The resulting Merkle Tree may thus not be balanced; however, its shape is uniquely determined by the number of leaves. (Note: This Merkle Tree is essentially the same as the history tree [CrosbyWallach] proposal, except our definition handles non-full trees differently).

2.1.2. Verifying a Tree Head Given Entries

When a client has a complete list of entries from 0 up to tree_size - 1 and wishes to verify this list against a tree head root_hash returned by the log for the same tree_size, the following algorithm may be used:

1. Set stack to an empty stack.

2. For each i from 0 up to tree_size - 1:

    1. Push HASH(0x00 || entries[i]) to stack.

    2. Set merge_count to the lowest value (0 included) such that LSB(i >> merge_count) is not set, where LSB means the least significant bit. In other words, set merge_count to the number of consecutive 1s found starting at the least significant bit of i.

    3. Repeat merge_count times:

        1. Pop right from stack.

        2. Pop left from stack.

        3. Push HASH(0x01 || left || right) to stack.

3. If there is more than one element in the stack, repeat the same merge procedure (the sub-items of Step 2.3 above) until only a single element remains.

4. The remaining element in stack is the Merkle Tree hash for the given tree_size and should be compared by equality against the supplied root_hash.

2.1.3. Merkle Inclusion Proofs

A Merkle inclusion proof for a leaf in a Merkle Hash Tree is the shortest list of additional nodes in the Merkle Tree required to compute the Merkle Tree Hash for that tree. Each node in the tree is either a leaf node or is computed from the two nodes immediately below it (i.e., towards the leaves). At each step up the tree (towards the root), a node from the inclusion proof is combined with the node computed so far. In other words, the inclusion proof consists of the list of missing nodes required to compute the nodes leading from a leaf to the root of the tree. If the root computed from the inclusion proof matches the true root, then the inclusion proof proves that the leaf exists in the tree.

2.1.3.1. Generating an Inclusion Proof

Given an ordered list of n inputs to the tree, D_n = {d[0], d[1], ..., d[n-1]}, the Merkle inclusion proof PATH(m, D_n) for the (m+1)th input d[m], 0 <= m < n, is defined as follows:

The proof for the single leaf in a tree with a one-element input list D[1] = {d[0]} is empty:

```
PATH(0, {d[0]}) = {}
```

For n > 1, let k be the largest power of two smaller than n. The proof for the (m+1)th element d[m] in a list of n > m elements is then defined recursively as

```
PATH(m, D_n) = PATH(m, D[0:k]) : MTH(D[k:n]) for m < k; and

PATH(m, D_n) = PATH(m - k, D[k:n]) : MTH(D[0:k]) for m >= k,
```

The : operator and D[k1:k2] are defined the same as in Section 2.1.1.

2.1.3.2. Verifying an Inclusion Proof

When a client has received an inclusion proof (e.g., in a TransItem of type inclusion_proof_v2) and wishes to verify inclusion of an input hash for a given tree_size and root_hash, the following algorithm may be used to prove the hash was included in the root_hash:

1. Compare leaf_index from the inclusion_proof_v2 structure against tree_size. If leaf_index is greater than or equal to tree_size then fail the proof verification.

2. Set fn to leaf_index and sn to tree_size - 1.

3. Set r to hash.

4. For each value p in the inclusion_path array:

   If sn is 0, stop the iteration and fail the proof verification.

   If LSB(fn) is set, or if fn is equal to sn, then:

   1. Set r to HASH(0x01 || p || r)

   2. If LSB(fn) is not set, then right-shift both fn and sn equally until either LSB(fn) is set or fn is 0.

   Otherwise:

   1. Set r to HASH(0x01 || r || p)

   Finally, right-shift both fn and sn one time.

5. Compare sn to 0. Compare r against the root_hash. If sn is equal to 0, and r and the root_hash are equal, then the log has proven the inclusion of hash. Otherwise, fail the proof verification.

2.1.4. Merkle Consistency Proofs

Merkle consistency proofs prove the append-only property of the tree. A Merkle consistency proof for a Merkle Tree Hash MTH(D_n) and a previously advertised hash MTH(D[0:m]) of the first m leaves, m <= n, is the list of nodes in the Merkle Tree required to verify that the first m inputs D[0:m] are equal in both trees. Thus, a consistency proof must contain a set of intermediate nodes (i.e., commitments to inputs) sufficient to verify MTH(D_n), such that (a subset of) the same nodes can be used to verify MTH(D[0:m]). We define an algorithm that outputs the (unique) minimal consistency proof.

2.1.4.1. Generating a Consistency Proof

Given an ordered list of n inputs to the tree, D_n = {d[0], d[1], ..., d[n-1]}, the Merkle consistency proof PROOF(m, D_n) for a previous Merkle Tree Hash MTH(D[0:m]), 0 < m < n, is defined as:

```
PROOF(m, D_n) = SUBPROOF(m, D_n, true)
```

In SUBPROOF, the boolean value represents whether the subtree created from D[0:m] is a complete subtree of the Merkle Tree created from D_n, and, consequently, whether the subtree Merkle Tree Hash MTH(D[0:m]) is known. The initial call to SUBPROOF sets this to be true, and SUBPROOF is then defined as follows:

The subproof for m = n is empty if m is the value for which PROOF was originally requested (meaning that the subtree created from D[0:m] is a complete subtree of the Merkle Tree created from the original D_n for which PROOF was requested, and the subtree Merkle Tree Hash MTH(D[0:m]) is known):

```
SUBPROOF(m, D_m, true) = {}
```

Otherwise, the subproof for m = n is the Merkle Tree Hash committing inputs D[0:m]:

```
SUBPROOF(m, D_m, false) = {MTH(D_m)}
```

For m < n, let k be the largest power of two smaller than n. The subproof is then defined recursively, using the appropriate step below:

If m <= k, the right subtree entries D[k:n] only exist in the current tree. We prove that the left subtree entries D[0:k] are consistent and add a commitment to D[k:n]:

```
SUBPROOF(m, D_n, b) = SUBPROOF(m, D[0:k], b) : MTH(D[k:n])
```

If m > k, the left subtree entries D[0:k] are identical in both trees. We prove that the right subtree entries D[k:n] are consistent and add a commitment to D[0:k].

```
SUBPROOF(m, D_n, b) = SUBPROOF(m - k, D[k:n], false) : MTH(D[0:k])
```

The number of nodes in the resulting proof is bounded above by ceil(log2(n)) + 1.

The : operator and D[k1:k2] are defined the same as in Section 2.1.1.

2.1.4.2. Verifying Consistency between Two Tree Heads

When a client has a tree head first_hash for tree size first, a tree head second_hash for tree size second where 0 < first < second, and has received a consistency proof between the two (e.g., in a TransItem of type consistency_proof_v2), the following algorithm may be used to verify the consistency proof:

1. If consistency_path is an empty array, stop and fail the proof verification.

2. If first is an exact power of 2, then prepend first_hash to the consistency_path array.

3. Set fn to first - 1 and sn to second - 1.

4. If LSB(fn) is set, then right-shift both fn and sn equally until LSB(fn) is not set.

5. Set both fr and sr to the first value in the consistency_path array.

6. For each subsequent value c in the consistency_path array:

   If sn is 0, stop the iteration and fail the proof verification.

   If LSB(fn) is set, or if fn is equal to sn, then:

   1. Set fr to HASH(0x01 || c || fr)

      Set sr to HASH(0x01 || c || sr)

   2. If LSB(fn) is not set, then right-shift both fn and sn equally until either LSB(fn) is set or fn is 0.

   Otherwise:

   1. Set sr to HASH(0x01 || sr || c)

   Finally, right-shift both fn and sn one time.

7. After completing iterating through the consistency_path array as described above, verify that the fr calculated is equal to the first_hash supplied, that the sr calculated is equal to the second_hash supplied and that sn is 0.

2.1.5. Example

The binary Merkle Tree with 7 leaves:

```
            hash
           /    \
          /      \
         /        \
        /          \
       /            \
      k              l
     / \            / \
    /   \          /   \
   /     \        /     \
  g       h      i      j
 / \     / \    / \     |
 a b     c d    e f     d6
 | |     | |    | |
d0 d1   d2 d3  d4 d5
```

The inclusion proof for d0 is [b, h, l].

The inclusion proof for d3 is [c, g, l].

The inclusion proof for d4 is [f, j, k].

The inclusion proof for d6 is [i, k].

The same tree, built incrementally in four steps:

```
    hash0          hash1=k
    / \              /  \
   /   \            /    \
  /     \          /      \
  g      c         g       h
 / \     |        / \     / \
 a b     d2       a b     c d
 | |              | |     | |
d0 d1            d0 d1   d2 d3
```

```
          hash2                    hash
          /  \                    /    \
         /    \                  /      \
        /      \                /        \
       /        \              /          \
      /          \            /            \
     k            i          k              l
    / \          / \        / \            / \
   /   \         e f       /   \          /   \
  /     \        | |      /     \        /     \
 g       h      d4 d5    g       h      i      j
/ \     / \             / \     / \    / \     |
a b     c d             a b     c d    e f     d6
| |     | |             | |     | |    | |
d0 d1   d2 d3           d0 d1   d2 d3  d4 d5
```

The consistency proof between hash0 and hash is PROOF(3, D[7]) = [c, d, g, l]. c, g are used to verify hash0, and d, l are additionally used to show hash is consistent with hash0.

The consistency proof between hash1 and hash is PROOF(4, D[7]) = [l]. hash can be verified using hash1=k and l.

The consistency proof between hash2 and hash is PROOF(6, D[7]) = [i, j, k]. k, i are used to verify hash2, and j is additionally used to show hash is consistent with hash2.

2.2. Signatures

When signing data structures, a log MUST use one of the signature algorithms from the IANA CT Signature Algorithms registry, described in Section 10.2.2.

3. Submitters

Submitters submit certificates or preannouncements of certificates prior to issuance (precertificates) to logs for public auditing, as described below. In order to enable attribution of each logged certificate or precertificate to its issuer, each submission MUST be accompanied by all additional certificates required to verify the chain up to an accepted trust anchor (Section 5.7). The trust anchor (a root or intermediate CA certificate) MAY be omitted from the submission.

If a log accepts a submission, it will return a Signed Certificate Timestamp (SCT) (see Section 4.8). The submitter SHOULD validate the returned SCT as described in Section 8.1 if they understand its format and they intend to use it directly in a TLS handshake or to construct a certificate. If the submitter does not need the SCT (for example, the certificate is being submitted simply to make it available in the log), it MAY validate the SCT.

3.1. Certificates

Any entity can submit a certificate (Section 5.1) to a log. Since it is anticipated that TLS clients will reject certificates that are not logged, it is expected that certificate issuers and subjects will be strongly motivated to submit them.

3.2. Precertificates

CAs may preannounce a certificate prior to issuance by submitting a precertificate (Section 5.1) that the log can use to create an entry that will be valid against the issued certificate. The CA MAY incorporate the returned SCT in the issued certificate. One example of where the returned SCT is not incorporated in the issued certificate is when a CA sends the precertificate to multiple logs, but only incorporates the SCTs that are returned first.

A precertificate is a CMS [RFC5652] signed-data object that conforms to the following profile:

* It MUST be DER encoded as described in [X690].

* SignedData.version MUST be v3(3).

* SignedData.digestAlgorithms MUST be the same as the SignerInfo.digestAlgorithm OID value (see below).

* SignedData.encapContentInfo:

  - eContentType MUST be the OID 1.3.101.78.

  - eContent MUST contain a TBSCertificate [RFC5280] that will be identical to the TBSCertificate in the issued certificate, except that the Transparency Information (Section 7.1) extension MUST be omitted.

* SignedData.certificates MUST be omitted.

* SignedData.crls MUST be omitted.

* SignedData.signerInfos MUST contain one SignerInfo:

  - version MUST be v3(3).

  - sid MUST use the subjectKeyIdentifier option.

  - digestAlgorithm MUST be one of the hash algorithm OIDs listed in the IANA CT Hash Algorithms Registry, described in Section 10.2.1.

  - signedAttrs MUST be present and MUST contain two attributes:

    o A content-type attribute whose value is the same as SignedData.encapContentInfo.eContentType.

    o A message-digest attribute whose value is the message digest of SignedData.encapContentInfo.eContent.

  - signatureAlgorithm MUST be the same OID as TBSCertificate.signature.

  - signature MUST be from the same (root or intermediate) CA that intends to issue the corresponding certificate (see Section 3.2.1).

  - unsignedAttrs MUST be omitted.

SignerInfo.signedAttrs is included in the message digest calculation process (see Section 5.4 of [RFC5652]), which ensures that the SignerInfo.signature value will not be a valid X.509v3 signature that could be used in conjunction with the TBSCertificate (from SignedData.encapContentInfo.eContent) to construct a valid certificate.

3.2.1. Binding Intent to Issue

Under normal circumstances, there will be a short delay between precertificate submission and issuance of the corresponding certificate. Longer delays are to be expected occasionally (e.g., due to log server downtime), and in some cases the CA might not actually issue the corresponding certificate. Nevertheless, a precertificate's signature indicates the CA's binding intent to issue the corresponding certificate, which means that:

* Misissuance of a precertificate is considered equivalent to misissuance of the corresponding certificate. The CA should expect to be held to account, even if the corresponding certificate has not actually been issued.

* Upon observing a precertificate, a client can reasonably presume that the corresponding certificate has been issued. A client may wish to obtain status information (e.g., by using the Online Certificate Status Protocol [RFC6960] or by checking a Certificate Revocation List [RFC5280]) about a certificate that is presumed to exist, especially if there is evidence or suspicion that the corresponding precertificate was misissued.

* TLS clients may have policies that require CAs to be able to revoke, and to provide certificate status services for, each certificate that is presumed to exist based on the existence of a corresponding precertificate.

4. Log Format and Operation

A log is a single, append-only Merkle Tree of submitted certificate and precertificate entries.

When it receives and accepts a valid submission, the log MUST return an SCT that corresponds to the submitted certificate or precertificate. If the log has previously seen this valid submission, it SHOULD return the same SCT as it returned before, as discussed in Section 11.3. If different SCTs are produced for the same submission, multiple log entries will have to be created, one for each SCT (as the timestamp is a part of the leaf structure). Note that if a certificate was previously logged as a precertificate, then the precertificate's SCT of type precert_sct_v2 would not be appropriate; instead, a fresh SCT of type x509_sct_v2 should be generated.

An SCT is the log's promise to append to its Merkle Tree an entry for the accepted submission. Upon producing an SCT, the log MUST fulfil this promise by performing the following actions within a fixed amount of time known as the Maximum Merge Delay (MMD), which is one of the log's parameters (see Section 4.1):

* Allocate a tree index to the entry representing the accepted submission.

* Calculate the root of the tree.

* Sign the root of the tree (see Section 4.10).

The log may append multiple entries before signing the root of the tree.

Log operators SHOULD NOT impose any conditions on retrieving or sharing data from the log.

4.1. Log Parameters

A log is defined by a collection of immutable parameters, which are used by clients to communicate with the log and to verify log artifacts. Except for the Final Signed Tree Head (STH), each of these parameters MUST be established before the log operator begins to operate the log.

Base URL: The prefix used to construct URLs ([RFC3986]) for client messages (see Section 5). The base URL MUST be an "https" URL, MAY contain a port, MAY contain a path with any number of path segments, but MUST NOT contain a query string, fragment, or trailing "/". Example: https://ct.example.org/blue

Hash Algorithm: The hash algorithm used for the Merkle Tree (see Section 10.2.1).

Signature Algorithm: The signature algorithm used (see Section 2.2).

Public Key: The public key used to verify signatures generated by the log. A log MUST NOT use the same keypair as any other log.

Log ID: The OID that uniquely identifies the log.

Maximum Merge Delay: The MMD the log has committed to. This document deliberately does not specify any limits on the value, to allow for experimentation.

Version: The version of the protocol supported by the log (currently 1 or 2).

Maximum Chain Length: The longest certificate chain submission the log is willing to accept, if the log imposes any limit.

STH Frequency Count: The maximum number of STHs the log may produce in any period equal to the Maximum Merge Delay (see Section 4.10).

Final STH: If a log has been closed down (i.e., no longer accepts new entries), existing entries may still be valid. In this case, the client should know the final valid STH in the log to ensure no new entries can be added without detection. This value MUST be provided in the form of a TransItem of type signed_tree_head_v2. If a log is still accepting entries, this value should not be provided.

[JSON.Metadata] is an example of a metadata format which includes the above elements.

4.2. Evaluating Submissions

A log determines whether to accept or reject a submission by evaluating it against the minimum acceptance criteria (see Section 4.2.1) and against the log's discretionary acceptance criteria (see Section 4.2.2).

If the acceptance criteria are met, the log SHOULD accept the submission. (A log may decide, for example, to temporarily reject acceptable submissions to protect itself against denial-of-service attacks).

The log SHALL allow retrieval of its list of accepted trust anchors (see Section 5.7), each of which is a root or intermediate CA certificate. This list might usefully be the union of root certificates trusted by major browser vendors.

4.2.1. Minimum Acceptance Criteria

To ensure that logged certificates and precertificates are attributable to an accepted trust anchor, to set clear expectations for what monitors would find in the log, and to avoid being overloaded by invalid submissions, the log MUST reject a submission if any of the following conditions are not met:

* The submission, type and chain inputs MUST be set as described in Section 5.1. The log MUST NOT accommodate misordered CA certificates or use any other source of intermediate CA certificates to attempt certification path construction.

* Each of the zero or more intermediate CA certificates in the chain MUST have one or both of the following features:

  - The Basic Constraints extension with the cA boolean asserted.

  - The Key Usage extension with the keyCertSign bit asserted.

* Each certificate in the chain MUST fall within the limits imposed by the zero or more Basic Constraints pathLenConstraint values found higher up the chain.

* Precertificate submissions MUST conform to all of the requirements in Section 3.2.

4.2.2. Discretionary Acceptance Criteria

If the minimum acceptance criteria are met but the submission is not fully valid according to [RFC5280] verification rules (e.g., the certificate or precertificate has expired, is not yet valid, has been revoked, exhibits ASN.1 DER encoding errors but the log can still parse it, etc), then the acceptability of the submission is left to the log's discretion. It is useful for logs to accept such submissions in order to accommodate quirks of CA certificate-issuing software and to facilitate monitoring of CA compliance with applicable policies and technical standards. However, it is impractical for this document to enumerate, and for logs to consider, all of the ways that a submission might fail to comply with [RFC5280].

Logs SHOULD limit the length of chain they will accept. The maximum chain length is one of the log's parameters (see Section 4.1).

4.3. Log Entries

If a submission is accepted and an SCT issued, the accepting log MUST store the entire chain used for verification. This chain MUST include the certificate or precertificate itself, the zero or more intermediate CA certificates provided by the submitter, and the trust anchor used to verify the chain (even if it was omitted from the submission). The log MUST provide this chain for auditing upon request (see Section 5.6) so that the CA cannot avoid blame by logging a partial or empty chain. Each log entry is a TransItem structure of type x509_entry_v2 or precert_entry_v2. However, a log may store its entries in any format. If a log does not store this TransItem in full, it must store the timestamp and sct_extensions of the corresponding TimestampedCertificateEntryDataV2 structure. The TransItem can be reconstructed from these fields and the entire chain that the log used to verify the submission.

4.4. Log ID

Each log is identified by an OID, which is one of the log's parameters (see Section 4.1) and which MUST NOT be used to identify any other log. A log's operator MUST either allocate the OID themselves or request an OID from the Log ID registry (see Section 10.2.5). One way to get an OID arc, from which OIDs can be allocated, is to request a Private Enterprise Number from IANA, by completing the registration form. The only advantage of the registry is that the DER encoding can be small. (Recall that OID allocations do not require a central registration, although logs will most likely want to make themselves known to potential clients through out of band means.) Various data structures include the DER encoding of this OID, excluding the ASN.1 tag and length bytes, in an opaque vector:

```
opaque LogID<2..127>;
```

Note that the ASN.1 length and the opaque vector length are identical in size (1 byte) and value, so the full DER encoding (including the tag and length) of the OID can be reproduced simply by prepending an OBJECT IDENTIFIER tag (0x06) to the opaque vector length and contents.

The OID used to identify a log is limited such that the DER encoding of its value, excluding the tag and length, MUST be no longer than 127 octets.

4.5. TransItem Structure

Various data structures are encapsulated in the TransItem structure to ensure that the type and version of each one is identified in a common fashion:

```
enum {
    x509_entry_v2(0x0100), precert_entry_v2(0x0101),
    x509_sct_v2(0x0102), precert_sct_v2(0x0103),
    signed_tree_head_v2(0x0104), consistency_proof_v2(0x0105),
    inclusion_proof_v2(0x0106),

    /* Reserved Code Points */
    reserved_rfc6962(0x0000..0x00FF),
    reserved_experimentaluse(0xE000..0xEFFF),
    reserved_privateuse(0xF000..0xFFFF),
    (0xFFFF)
} VersionedTransType;

struct {
    VersionedTransType versioned_type;
    select (versioned_type) {
        case x509_entry_v2: TimestampedCertificateEntryDataV2;
        case precert_entry_v2: TimestampedCertificateEntryDataV2;
        case x509_sct_v2: SignedCertificateTimestampDataV2;
        case precert_sct_v2: SignedCertificateTimestampDataV2;
        case signed_tree_head_v2: SignedTreeHeadDataV2;
        case consistency_proof_v2: ConsistencyProofDataV2;
        case inclusion_proof_v2: InclusionProofDataV2;
    } data;
} TransItem;
```

versioned_type is a value from the IANA registry in Section 10.2.3 that identifies the type of the encapsulated data structure and the earliest version of this protocol to which it conforms. This document is v2.

data is the encapsulated data structure. The various structures named with the DataV2 suffix are defined in later sections of this document.

Note that VersionedTransType combines the v1 [RFC6962] type enumerations Version, LogEntryType, SignatureType and MerkleLeafType. Note also that v1 did not define TransItem, but this document provides guidelines (see Appendix A) on how v2 implementations can co-exist with v1 implementations.

Future versions of this protocol may reuse VersionedTransType values defined in this document as long as the corresponding data structures are not modified, and may add new VersionedTransType values for new or modified data structures.

4.6. Log Artifact Extensions

```
enum {
    reserved(65535)
} ExtensionType;

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
```

The Extension structure provides a generic extensibility for log artifacts, including SCTs (Section 4.8) and STHs (Section 4.10). The interpretation of the extension_data field is determined solely by the value of the extension_type field.

This document does not define any extensions, but it does establish a registry for future ExtensionType values (see Section 10.2.4). Each document that registers a new ExtensionType must specify the context in which it may be used (e.g., SCT, STH, or both) and describe how to interpret the corresponding extension_data.

4.7. Merkle Tree Leaves

The leaves of a log's Merkle Tree correspond to the log's entries (see Section 4.3). Each leaf is the leaf hash (Section 2.1) of a TransItem structure of type x509_entry_v2 or precert_entry_v2, which encapsulates a TimestampedCertificateEntryDataV2 structure. Note that leaf hashes are calculated as HASH(0x00 || TransItem), where the hash algorithm is one of the log's parameters.

```
opaque TBSCertificate<1..2^24-1>;

struct {
    uint64 timestamp;
    opaque issuer_key_hash<32..2^8-1>;
    TBSCertificate tbs_certificate;
    Extension sct_extensions<0..2^16-1>;
} TimestampedCertificateEntryDataV2;
```

timestamp is the date and time at which the certificate or precertificate was accepted by the log, in the form of a 64-bit unsigned number of milliseconds elapsed since the Unix Epoch (1 January 1970 00:00:00 UTC - see [UNIXTIME]), ignoring leap seconds, in network byte order. Note that the leaves of a log's Merkle Tree are not required to be in strict chronological order.

issuer_key_hash is the HASH of the public key of the CA that issued the certificate or precertificate, calculated over the DER encoding of the key represented as SubjectPublicKeyInfo [RFC5280]. This is needed to bind the CA to the certificate or precertificate, making it impossible for the corresponding SCT to be valid for any other certificate or precertificate whose TBSCertificate matches tbs_certificate. The length of the issuer_key_hash MUST match HASH_SIZE.

tbs_certificate is the DER encoded TBSCertificate from the submission. (Note that a precertificate's TBSCertificate can be reconstructed from the corresponding certificate as described in Section 8.1.2).

sct_extensions is byte-for-byte identical to the SCT extensions of the corresponding SCT.

The type of the TransItem corresponds to the value of the type parameter supplied in the Section 5.1 call.

4.8. Signed Certificate Timestamp (SCT)

An SCT is a TransItem structure of type x509_sct_v2 or precert_sct_v2, which encapsulates a SignedCertificateTimestampDataV2 structure:

```
struct {
    LogID log_id;
    uint64 timestamp;
    Extension sct_extensions<0..2^16-1>;
    opaque signature<1..2^16-1>;
} SignedCertificateTimestampDataV2;
```

log_id is this log's unique ID, encoded in an opaque vector as described in Section 4.4.

timestamp is equal to the timestamp from the corresponding TimestampedCertificateEntryDataV2 structure.

sct_extensions is a vector of 0 or more SCT extensions. This vector MUST NOT include more than one extension with the same extension_type. The extensions in the vector MUST be ordered by the value of the extension_type field, smallest value first. All SCT extensions are similar to non-critical X.509v3 extensions (i.e., the mustUnderstand field is not set), and a recipient SHOULD ignore any extension it does not understand. Furthermore, an implementation MAY choose to ignore any extension(s) that it does understand.

signature is computed over a TransItem structure of type x509_entry_v2 or precert_entry_v2 (see Section 4.7) using the signature algorithm declared in the log's parameters (see Section 4.1).

4.9. Merkle Tree Head

The log stores information about its Merkle Tree in a TreeHeadDataV2:

```
opaque NodeHash<32..2^8-1>;

struct {
    uint64 timestamp;
    uint64 tree_size;
    NodeHash root_hash;
    Extension sth_extensions<0..2^16-1>;
} TreeHeadDataV2;
```

The length of NodeHash MUST match HASH_SIZE of the log.

timestamp is the current date and time, using the format defined in Section 4.7.

tree_size is the number of entries currently in the log's Merkle Tree.

root_hash is the root of the Merkle Hash Tree.

sth_extensions is a vector of 0 or more STH extensions. This vector MUST NOT include more than one extension with the same extension_type. The extensions in the vector MUST be ordered by the value of the extension_type field, smallest value first. If an implementation sees an extension that it does not understand, it SHOULD ignore that extension. Furthermore, an implementation MAY choose to ignore any extension(s) that it does understand.

4.10. Signed Tree Head (STH)

Periodically each log SHOULD sign its current tree head information (see Section 4.9) to produce an STH. When a client requests a log's latest STH (see Section 5.2), the log MUST return an STH that is no older than the log's MMD. However, since STHs could be used to mark individual clients (by producing a new STH for each query), a log MUST NOT produce STHs more frequently than its parameters declare (see Section 4.1). In general, there is no need to produce a new STH unless there are new entries in the log; however, in the event that a log does not accept any submissions during an MMD period, the log MUST sign the same Merkle Tree Hash with a fresh timestamp.

An STH is a TransItem structure of type signed_tree_head_v2, which encapsulates a SignedTreeHeadDataV2 structure:

```
struct {
    LogID log_id;
    TreeHeadDataV2 tree_head;
    opaque signature<1..2^16-1>;
} SignedTreeHeadDataV2;
```

log_id is this log's unique ID, encoded in an opaque vector as described in Section 4.4.

The timestamp in tree_head MUST be at least as recent as the most recent SCT timestamp in the tree. Each subsequent timestamp MUST be more recent than the timestamp of the previous update.

tree_head contains the latest tree head information (see Section 4.9).

signature is computed over the tree_head field using the signature algorithm declared in the log's parameters (see Section 4.1).

4.11. Merkle Consistency Proofs

To prepare a Merkle Consistency Proof for distribution to clients, the log produces a TransItem structure of type consistency_proof_v2, which encapsulates a ConsistencyProofDataV2 structure:

```
struct {
    LogID log_id;
    uint64 tree_size_1;
    uint64 tree_size_2;
    NodeHash consistency_path<0..2^16-1>;
} ConsistencyProofDataV2;
```

log_id is this log's unique ID, encoded in an opaque vector as described in Section 4.4.

tree_size_1 is the size of the older tree.

tree_size_2 is the size of the newer tree.

consistency_path is a vector of Merkle Tree nodes proving the consistency of two STHs as described in Section 2.1.4.

4.12. Merkle Inclusion Proofs

To prepare a Merkle Inclusion Proof for distribution to clients, the log produces a TransItem structure of type inclusion_proof_v2, which encapsulates an InclusionProofDataV2 structure:

```
struct {
    LogID log_id;
    uint64 tree_size;
    uint64 leaf_index;
    NodeHash inclusion_path<0..2^16-1>;
} InclusionProofDataV2;
```

log_id is this log's unique ID, encoded in an opaque vector as described in Section 4.4.

tree_size is the size of the tree on which this inclusion proof is based.

leaf_index is the 0-based index of the log entry corresponding to this inclusion proof.

inclusion_path is a vector of Merkle Tree nodes proving the inclusion of the chosen certificate or precertificate as described in Section 2.1.3.

4.13. Shutting down a log

Log operators may decide to shut down a log for various reasons, such as deprecation of the signature algorithm. If there are entries in the log for certificates that have not yet expired, simply making TLS clients stop recognizing that log will have the effect of invalidating SCTs from that log. In order to avoid that, the following actions SHOULD be taken:

* Make it known to clients and monitors that the log will be frozen. This is not part of the API, so it will have to be done via a relevant out-of-band mechanism.

* Stop accepting new submissions (the error code "shutdown" should be returned for such requests).

* Once MMD from the last accepted submission has passed and all pending submissions are incorporated, issue a final STH and publish it as one of the log's parameters. Having an STH with a timestamp that is after the MMD has passed from the last SCT issuance allows clients to audit this log regularly without special handling for the final STH. At this point the log's private key is no longer needed and can be destroyed.

* Keep the log running until the certificates in all of its entries have expired or exist in other logs (this can be determined by scanning other logs or connecting to domains mentioned in the certificates and inspecting the SCTs served).

5. Log Client Messages

Messages are sent as HTTPS GET or POST requests. Parameters for POSTs and all responses are encoded as JavaScript Object Notation (JSON) objects [RFC8259]. Parameters for GETs are encoded as order-independent key/value URL parameters, using the "application/x-www-form-urlencoded" format described in the "HTML 4.01 Specification" [HTML401]. Binary data is base64 encoded according to section 4 of [RFC4648] as specified in the individual messages.

Clients are configured with a log's base URL, which is one of the log's parameters. Clients construct URLs for requests by appending suffixes to this base URL. This structure places some degree of restriction on how log operators can deploy these services, as noted in [RFC8820]. However, operational experience with version 1 of this protocol has not indicated that these restrictions are a problem in practice.

Note that JSON objects and URL parameters may contain fields not specified here, to allow for experimentation. Any fields that are not understood SHOULD be ignored.

In practice, log servers may include multiple front-end machines. Since it is impractical to keep these machines in perfect sync, errors may occur that are caused by skew between the machines. Where such errors are possible, the front-end will return additional information (as specified below) making it possible for clients to make progress, if progress is possible. Front-ends MUST only serve data that is free of gaps (that is, for example, no front-end will respond with an STH unless it is also able to prove consistency from all log entries logged within that STH).

For example, when a consistency proof between two STHs is requested, the front-end reached may not yet be aware of one or both STHs. In the case where it is unaware of both, it will return the latest STH it is aware of. Where it is aware of the first but not the second, it will return the latest STH it is aware of and a consistency proof from the first STH to the returned STH. The case where it knows the second but not the first should not arise (see the "no gaps" requirement above).

If the log is unable to process a client's request, it MUST return an HTTP response code of 4xx/5xx (see [RFC7231]), and, in place of the responses outlined in the subsections below, the body SHOULD be a JSON Problem Details Object (see [RFC7807] Section 3), containing:

type: A URN reference identifying the problem. To facilitate automated response to errors, this document defines a set of standard tokens for use in the type field, within the URN namespace of: "urn:ietf:params:trans:error:".

detail: A human-readable string describing the error that prevented the log from processing the request, ideally with sufficient detail to enable the error to be rectified.

e.g., In response to a request of <Base URL>/ct/v2/get-entries?start=100&end=99, the log would return a 400 Bad Request response code with a body similar to the following:

```
{
    "type": "urn:ietf:params:trans:error:endBeforeStart",
    "detail": "'start' cannot be greater than 'end'"
}
```

Most error types are specific to the type of request and are defined in the respective subsections below. The one exception is the "malformed" error type, which indicates that the log server could not parse the client's request because it did not comply with this document:

type	detail
malformed	The request could not be parsed.

Table 1

Clients SHOULD treat 500 Internal Server Error and 503 Service Unavailable responses as transient failures and MAY retry the same request without modification at a later date. Note that as per [RFC7231], in the case of a 503 response the log MAY include a Retry-After: header field in order to request a minimum time for the client to wait before retrying the request. In the absence of this header field, this document does not specify a minimum.

Clients SHOULD treat any 4xx error as a problem with the request and not attempt to resubmit without some modification to the request. The full status code MAY provide additional details.

This document deliberately does not provide more specific guidance on the use of HTTP status codes.

5.1. Submit Entry to Log

POST <Base URL>/ct/v2/submit-entry

Inputs:

submission: The base64 encoded certificate or precertificate.

type: The VersionedTransType integer value that indicates the type of the submission: 1 for x509_entry_v2, or 2 for precert_entry_v2.

chain: An array of zero or more JSON strings, each of which is a base64 encoded CA certificate. The first element is the certifier of the submission; the second certifies the first; etc. The last element of chain (or, if chain is an empty array, the submission) is certified by an accepted trust anchor.

Outputs:

sct: A base64 encoded TransItem of type x509_sct_v2 or precert_sct_v2, signed by this log, that corresponds to the submission.

If the submitted entry is immediately appended to (or already exists in) this log's tree, then the log SHOULD also output:

sth: A base64 encoded TransItem of type signed_tree_head_v2, signed by this log.

inclusion: A base64 encoded TransItem of type inclusion_proof_v2 whose inclusion_path array of Merkle Tree nodes proves the inclusion of the submission in the returned sth.

Error codes:

type	**detail**
badSubmission	submission is neither a valid certificate nor a valid precertificate.
badType	type is neither 1 nor 2.
badChain	The first element of chain is not the certifier of the submission, or the second element does not certify the first, etc.
badCertificate	One or more certificates in the chain are not valid (e.g., not properly encoded).
unknownAnchor	The last element of chain (or, if chain is an empty array, the submission) both is not, and is not certified by, an accepted trust anchor.
shutdown	The log is no longer accepting submissions.

Table 2

If the version of sct is not v2, then a v2 client may be unable to verify the signature. It MUST NOT construe this as an error. This is to avoid forcing an upgrade of compliant v2 clients that do not use the returned SCTs.

If a log detects bad encoding in a chain that otherwise verifies correctly then the log MUST either log the certificate or return the "bad certificate" error. If the certificate is logged, an SCT MUST be issued. Logging the certificate is useful, because monitors (Section 8.2) can then detect these encoding errors, which may be accepted by some TLS clients.

If submission is an accepted trust anchor whose certifier is neither an accepted trust anchor nor the first element of chain, then the log MUST return the "unknown anchor" error. A log is not able to generate an SCT for a submission if it does not have access to the issuer's public key.

If the returned sct is intended to be provided to TLS clients, then sth and inclusion (if returned) SHOULD also be provided to TLS clients. For example, if type was 2 (indicating precert_sct_v2) then all three TransItems could be embedded in the certificate.

5.2. Retrieve Latest STH

GET <Base URL>/ct/v2/get-sth

No inputs.

Outputs:

sth: A base64 encoded TransItem of type signed_tree_head_v2, signed by this log, that is no older than the log's MMD.

5.3. Retrieve Merkle Consistency Proof between Two STHs

GET <Base URL>/ct/v2/get-sth-consistency

Inputs:

first: The tree_size of the older tree, in decimal.

second: The tree_size of the newer tree, in decimal (optional).

Both tree sizes must be from existing v2 STHs. However, because of skew, the receiving front-end may not know one or both of the existing STHs. If both are known, then only the consistency output is returned. If the first is known but the second is not (or has been omitted), then the latest known STH is returned, along with a consistency proof between the first STH and the latest. If neither are known, then the latest known STH is returned without a consistency proof.

Outputs:

consistency: A base64 encoded TransItem of type consistency_proof_v2, whose tree_size_1 MUST match the first input. If the sth output is omitted, then tree_size_2 MUST match the second input. If first and second are equal and correspond to a known STH, the returned consistency proof MUST be empty (a consistency_path array with zero elements).

sth: A base64 encoded TransItem of type signed_tree_head_v2, signed by this log.

Note that no signature is required for the consistency output as it is used to verify the consistency between two STHs, which are signed.

Error codes:

type	detail
firstUnknown	first is before the latest known STH but is not from an existing STH.
secondUnknown	second is before the latest known STH but is not from an existing STH.
secondBeforeFirst	second is smaller than first.

Table 3

See Section 2.1.4.2 for an outline of how to use the consistency output.

5.4. Retrieve Merkle Inclusion Proof from Log by Leaf Hash

GET <Base URL>/ct/v2/get-proof-by-hash

Inputs:

hash: A base64 encoded v2 leaf hash.

tree_size: The tree_size of the tree on which to base the proof, in decimal.

The hash must be calculated as defined in Section 4.7. A v2 STH must exist for the tree_size. Because of skew, the front-end may not know the requested tree head. In that case, it will return the latest STH it knows, along with an inclusion proof to that STH. If the front-end knows the requested tree head then only inclusion is returned.

Outputs:

inclusion: A base64 encoded TransItem of type inclusion_proof_v2 whose inclusion_path array of Merkle Tree nodes proves the inclusion of the certificate (as specified by the hash parameter) in the selected STH.

sth: A base64 encoded TransItem of type signed_tree_head_v2, signed by this log.

Note that no signature is required for the inclusion output as it is used to verify inclusion in the selected STH, which is signed.

Error codes:

type	detail
hashUnknown	hash is not the hash of a known leaf (may be caused by skew or by a known certificate not yet merged).
treeSizeUnknown	hash is before the latest known STH but is not from an existing STH.

Table 4

See Section 2.1.3.2 for an outline of how to use the inclusion output.

5.5. Retrieve Merkle Inclusion Proof, STH and Consistency Proof by Leaf Hash

GET <Base URL>/ct/v2/get-all-by-hash

Inputs:

hash: A base64 encoded v2 leaf hash.

tree_size: The tree_size of the tree on which to base the proofs, in decimal.

The hash must be calculated as defined in Section 4.7. A v2 STH must exist for the tree_size.

Because of skew, the front-end may not know the requested tree head or the requested hash, which leads to a number of cases:

Case	Response
latest STH < requested tree head	Return latest STH
latest STH > requested tree head	Return latest STH and a consistency proof between it and the requested tree head (see Section 5.3)
	Return inclusion

Case	Response
index of requested hash < latest STH	

Table 5

Note that more than one case can be true, in which case the returned data is their union. It is also possible for none to be true, in which case the front-end MUST return an empty response.

Outputs:

inclusion: A base64 encoded TransItem of type inclusion_proof_v2 whose inclusion_path array of Merkle Tree nodes proves the inclusion of the certificate (as specified by the hash parameter) in the selected STH.

sth: A base64 encoded TransItem of type signed_tree_head_v2, signed by this log.

consistency: A base64 encoded TransItem of type consistency_proof_v2 that proves the consistency of the requested tree head and the returned STH.

Note that no signature is required for the inclusion or consistency outputs as they are used to verify inclusion in and consistency of STHs, which are signed.

Errors are the same as in Section 5.4.

See Section 2.1.3.2 for an outline of how to use the inclusion output, and see Section 2.1.4.2 for an outline of how to use the consistency output.

5.6. Retrieve Entries and STH from Log

GET <Base URL>/ct/v2/get-entries

Inputs:

start: 0-based index of first entry to retrieve, in decimal.

end: 0-based index of last entry to retrieve, in decimal.

Outputs:

entries: An array of objects, each consisting of

log_entry: The base64 encoded TransItem structure of type x509_entry_v2 or precert_entry_v2 (see Section 4.3).

submitted_entry: JSON object equivalent to inputs that were submitted to submit-entry, with the addition of the trust anchor to the chain field if the submission did not include it.

sct: The base64 encoded TransItem of type x509_sct_v2 or precert_sct_v2 corresponding to this log entry.

sth: A base64 encoded TransItem of type signed_tree_head_v2, signed by this log.

Note that this message is not signed -- the entries data can be verified by constructing the Merkle Tree Hash corresponding to a retrieved STH. All leaves MUST be v2. However, a compliant v2 client MUST NOT construe an unrecognized TransItem type as an error. This means it may be unable to parse some entries, but note that each client can inspect the entries it does recognize as well as verify the integrity of the data by treating unrecognized leaves as opaque input to the tree.

The start and end parameters SHOULD be within the range 0 <= x < tree_size as returned by get-sth in Section 5.2.

The start parameter MUST be less than or equal to the end parameter.

Each submitted_entry output parameter MUST include the trust anchor that the log used to verify the submission, even if that trust anchor was not provided to submit-entry (see Section 5.1). If the submission does not certify itself, then the first element of chain MUST be present and MUST certify the submission.

Log servers MUST honor requests where 0 <= start < tree_size and end >= tree_size by returning a partial response covering only the valid entries in the specified range. end >= tree_size could be caused by skew. Note that the following restriction may also apply:

Logs MAY restrict the number of entries that can be retrieved per get-entries request. If a client requests more than the permitted number of entries, the log SHALL return the maximum number of entries permissible. These entries SHALL be sequential beginning with the entry specified by start. Note that limit on the number of entries is not immutable and therefore the restriction may be changed or lifted at any time and is not listed with the other Log Parameters in Section 4.1.

Because of skew, it is possible the log server will not have any entries between start and end. In this case it MUST return an empty entries array.

In any case, the log server MUST return the latest STH it knows about.

See Section 2.1.2 for an outline of how to use a complete list of log_entry entries to verify the root_hash.

Error codes:

type	detail
startUnknown	start is greater than the number of entries in the Merkle tree.
endBeforeStart	start cannot be greater than end.

Table 6

5.7. Retrieve Accepted Trust Anchors

GET <Base URL>/ct/v2/get-anchors

No inputs.

Outputs:

certificates: An array of JSON strings, each of which is a base64 encoded CA certificate that is acceptable to the log.

max_chain_length: If the server has chosen to limit the length of chains it accepts, this is the maximum number of certificates in the chain, in decimal. If there is no limit, this is omitted.

This data is not signed and the protocol depends on the security guarantees of TLS to ensure correctness.

6. TLS Servers

CT-using TLS servers MUST use at least one of the mechanisms described below to present one or more SCTs from one or more logs to each TLS client during full TLS handshakes, when requested by the client, where each SCT corresponds to the server certificate. (Of course, a server can only send a TLS extension if the client has specified it first.) Servers SHOULD also present corresponding inclusion proofs and STHs.

A server can provide SCTs using a TLS 1.3 extension (Section 4.2 of [RFC8446]) with type transparency_info (see Section 6.5). This mechanism allows TLS servers to participate in CT without the cooperation of CAs, unlike the other two mechanisms. It also allows SCTs and inclusion proofs to be updated on the fly.

The server may also use an Online Certificate Status Protocol (OCSP) [RFC6960] response extension (see Section 7.1.1), providing the OCSP response as part of the TLS handshake. Providing a response during a TLS handshake is popularly known as "OCSP stapling." For TLS 1.3, the information is encoded as an extension in the status_request extension data; see Section 4.4.2.1 of [RFC8446]. For TLS 1.2 ([RFC5246]), the information is encoded in the CertificateStatus message; see Section 8 of [RFC6066]. Using stapling also allows SCTs and inclusion proofs to be updated on the fly.

CT information can also be encoded as an extension in the X.509v3 certificate (see Section 7.1.2). This mechanism allows the use of unmodified TLS servers, but the SCTs and inclusion proofs cannot be updated on the fly. Since the logs from which the SCTs and inclusion proofs originated won't necessarily be accepted by TLS clients for the full lifetime of the certificate, there is a risk that TLS clients may subsequently consider the certificate to be non-compliant and in need of re-issuance or the use of one of the other two methods for delivering CT information.

6.1. TLS Client Authentication

This specification includes no description of how a TLS server can use CT for TLS client certificates. While this may be useful, it is not documented here for the following reasons:

* The greater security exposure is for clients to end up interacting with an illegitimate server.

* In general, TLS client certificates are not expected to be submitted to CT logs, particularly those intended for general public use.

A future version could include such information.

6.2. Multiple SCTs

CT-using TLS servers SHOULD send SCTs from multiple logs, because:

* One or more logs may not have become acceptable to all CT-using TLS clients. Note that client discovery, trust, and distrust of logs is expected to be handled out-of-band and is out of scope of this document.

* If a CA and a log collude, it is possible to temporarily hide misissuance from clients. When a TLS client requires SCTs from multiple logs to be provided, it is more difficult to mount this attack.

* If a log misbehaves or suffers a key compromise, a consequence may be that clients cease to trust it. Since the time an SCT may be in use can be considerable (several years is common in current practice when embedded in a certificate), including SCTs from multiple logs reduces the probability of the certificate being rejected by TLS clients.

* TLS clients may have policies related to the above risks requiring TLS servers to present multiple SCTs. For example, at the time of writing, Chromium [Chromium.Log.Policy] requires multiple SCTs to be presented with EV certificates in order for the EV indicator to be shown.

To select the logs from which to obtain SCTs, a TLS server can, for example, examine the set of logs popular TLS clients accept and recognize.

6.3. TransItemList Structure

Multiple SCTs, inclusion proofs, and indeed TransItem structures of any type, are combined into a list as follows:

```
opaque SerializedTransItem<1..2^16-1>;

struct {
    SerializedTransItem trans_item_list<1..2^16-1>;
} TransItemList;
```

Here, SerializedTransItem is an opaque byte string that contains the serialized TransItem structure. This encoding ensures that TLS clients can decode each TransItem individually (so, for example, if there is a version upgrade, out-of-date clients can still parse old TransItem structures while skipping over new TransItem structures whose versions they don't understand).

6.4. Presenting SCTs, inclusions proofs and STHs

In each TransItemList that is sent during a TLS handshake, the TLS server MUST include a TransItem structure of type x509_sct_v2 or precert_sct_v2.

Presenting inclusion proofs and STHs in the TLS handshake helps to protect the client's privacy (see Section 8.1.4) and reduces load on log servers. Therefore, if the TLS server can obtain them, it SHOULD also include TransItems of type inclusion_proof_v2 and signed_tree_head_v2 in the TransItemList.

6.5. transparency_info TLS Extension

Provided that a TLS client includes the transparency_info extension type in the ClientHello and the TLS server supports the transparency_info extension:

* The TLS server MUST verify that the received extension_data is empty.

* The TLS server MUST construct a TransItemList of relevant TransItems (see Section 6.4), which SHOULD omit any TransItems that are already embedded in the server certificate or the stapled OCSP response (see Section 7.1). If the constructed TransItemList is not empty, then the TLS server MUST include the transparency_info extension with the extension_data set to this TransItemList. If the list is empty then the server SHOULD omit the extension_data element, but MAY send it with an empty array.

TLS servers MUST only include this extension in the following messages:

* the ServerHello message (for TLS 1.2 or earlier).

* the Certificate or CertificateRequest message (for TLS 1.3).

TLS servers MUST NOT process or include this extension when a TLS session is resumed, since session resumption uses the original session information.

7. Certification Authorities

7.1. Transparency Information X.509v3 Extension

The Transparency Information X.509v3 extension, which has OID 1.3.101.75 and SHOULD be non-critical, contains one or more TransItem structures in a TransItemList. This extension MAY be included in OCSP responses (see Section 7.1.1) and certificates (see Section 7.1.2). Since RFC5280 requires the extnValue field (an OCTET STRING) of each X.509v3 extension to include the DER encoding of an ASN.1 value, a TransItemList MUST NOT be included directly. Instead, it MUST be wrapped inside an additional OCTET STRING, which is then put into the extnValue field:

```
TransparencyInformationSyntax ::= OCTET STRING
```

TransparencyInformationSyntax contains a TransItemList.

7.1.1. OCSP Response Extension

A certification authority MAY include a Transparency Information X.509v3 extension in the singleExtensions of a SingleResponse in an OCSP response. All included SCTs and inclusion proofs MUST be for the certificate identified by the certID of that SingleResponse, or for a precertificate that corresponds to that certificate.

7.1.2. Certificate Extension

A certification authority MAY include a Transparency Information X.509v3 extension in a certificate. All included SCTs and inclusion proofs MUST be for a precertificate that corresponds to this certificate.

7.2. TLS Feature X.509v3 Extension

A certification authority SHOULD NOT issue any certificate that identifies the transparency_info TLS extension in a TLS feature extension [RFC7633], because TLS servers are not required to support the transparency_info TLS extension in order to participate in CT (see Section 6).

8. Clients

There are various different functions clients of logs might perform. We describe here some typical clients and how they should function. Any inconsistency may be used as evidence that a log has not behaved correctly, and the signatures on the data structures prevent the log from denying that misbehavior.

All clients need various parameters in order to communicate with logs and verify their responses. These parameters are described in Section 4.1, but note that this document does not describe how the parameters are obtained, which is implementation-dependent (see, for example, [Chromium.Policy]).

8.1. TLS Client

8.1.1. Receiving SCTs and inclusion proofs

TLS clients receive SCTs and inclusion proofs alongside or in certificates. CT-using TLS clients MUST implement all of the three mechanisms by which TLS servers may present SCTs (see Section 6).

TLS clients that support the transparency_info TLS extension (see Section 6.5) SHOULD include it in ClientHello messages, with empty extension_data. If a TLS server includes the transparency_info TLS extension when resuming a TLS session, the TLS client MUST abort the handshake.

8.1.2. Reconstructing the TBSCertificate

Validation of an SCT for a certificate (where the type of the TransItem is x509_sct_v2) uses the unmodified TBSCertificate component of the certificate.

Before an SCT for a precertificate (where the type of the TransItem is precert_sct_v2) can be validated, the TBSCertificate component of the precertificate needs to be reconstructed from the TBSCertificate component of the certificate as follows:

* Remove the Transparency Information extension (see Section 7.1).

* Remove embedded v1 SCTs, identified by OID 1.3.6.1.4.1.11129.2.4.2 (see section 3.3 of [RFC6962]). This allows embedded v1 and v2 SCTs to co-exist in a certificate (see Appendix A).

8.1.3. Validating SCTs

In order to make use of a received SCT, the TLS client MUST first validate it as follows:

* Compute the signature input by constructing a TransItem of type x509_entry_v2 or precert_entry_v2, depending on the SCT's TransItem type. The TimestampedCertificateEntryDataV2 structure is constructed in the following manner:

  - timestamp is copied from the SCT.

  - tbs_certificate is the reconstructed TBSCertificate portion of the server certificate, as described in Section 8.1.2.

  - issuer_key_hash is computed as described in Section 4.7.

  - sct_extensions is copied from the SCT.

* Verify the SCT's signature against the computed signature input using the public key of the corresponding log, which is identified by the log_id. The required signature algorithm is one of the log's parameters.

If the TLS client does not have the corresponding log's parameters, it cannot attempt to validate the SCT. When evaluating compliance (see Section 8.1.6), the TLS client will consider only those SCTs that it was able to validate.

Note that SCT validation is not a substitute for the normal validation of the server certificate and its chain.

8.1.4. Fetching inclusion proofs

When a TLS client has validated a received SCT but does not yet possess a corresponding inclusion proof, the TLS client MAY request the inclusion proof directly from a log using get-proof-by-hash (Section 5.4) or get-all-by-hash (Section 5.5).

Note that fetching inclusion proofs directly from a log will disclose to the log which TLS server the client has been communicating with. This may be regarded as a significant privacy concern, and so it is preferable for the TLS server to send the inclusion proofs (see Section 6.4).

8.1.5. Validating inclusion proofs

When a TLS client has received, or fetched, an inclusion proof (and an STH), it SHOULD proceed to verifying the inclusion proof to the provided STH. The TLS client SHOULD also verify consistency between the provided STH and an STH it knows about.

If the TLS client holds an STH that predates the SCT, it MAY, in the process of auditing, request a new STH from the log (Section 5.2), then verify it by requesting a consistency proof (Section 5.3). Note that if the TLS client uses get-all-by-hash, then it will already have the new STH.

8.1.6. Evaluating compliance

It is up to a client's local policy to specify the quantity and form of evidence (SCTs, inclusion proofs or a combination) needed to achieve compliance and how to handle non-compliance.

A TLS client can only evaluate compliance if it has given the TLS server the opportunity to send SCTs and inclusion proofs by any of the three mechanisms that are mandatory to implement for CT-using TLS clients (see Section 8.1.1). Therefore, a TLS client MUST NOT evaluate compliance if it did not include both the transparency_info and status_request TLS extensions in the ClientHello.

8.2. Monitor

Monitors watch logs to check that they behave correctly, for certificates of interest, or both. For example, a monitor may be configured to report on all certificates that apply to a specific domain name when fetching new entries for consistency validation.

A monitor MUST at least inspect every new entry in every log it watches, and it MAY also choose to keep copies of entire logs.

To inspect all of the existing entries, the monitor SHOULD follow these steps once for each log:

1. Fetch the current STH (Section 5.2).

2. Verify the STH signature.

3. Fetch all the entries in the tree corresponding to the STH (Section 5.6).

4. If applicable, check each entry to see if it's a certificate of interest.

5. Confirm that the tree made from the fetched entries produces the same hash as that in the STH.

To inspect new entries, the monitor SHOULD follow these steps repeatedly for each log:

1. Fetch the current STH (Section 5.2). Repeat until the STH changes. This document does not specify the polling frequency, to allow for experimentation.

2. Verify the STH signature.

3. Fetch all the new entries in the tree corresponding to the STH (Section 5.6). If they remain unavailable for an extended period, then this should be viewed as misbehavior on the part of the log.

4. If applicable, check each entry to see if it's a certificate of interest.

5. Either:

    1. Verify that the updated list of all entries generates a tree with the same hash as the new STH.

    Or, if it is not keeping all log entries:

    1. Fetch a consistency proof for the new STH with the previous STH (Section 5.3).

    2. Verify the consistency proof.

    3. Verify that the new entries generate the corresponding elements in the consistency proof.

6. Repeat from step 1.

8.3. Auditing

Auditing ensures that the current published state of a log is reachable from previously published states that are known to be good, and that the promises made by the log in the form of SCTs have been kept. Audits are performed by monitors or TLS clients.

In particular, there are four log behavior properties that should be checked:

* The Maximum Merge Delay (MMD).

* The STH Frequency Count.

* The append-only property.

* The consistency of the log view presented to all query sources.

A benign, conformant log publishes a series of STHs over time, each derived from the previous STH and the submitted entries incorporated into the log since publication of the previous STH. This can be proven through auditing of STHs. SCTs returned to TLS clients can be audited by verifying against the accompanying certificate, and using Merkle Inclusion Proofs, against the log's Merkle tree.

The action taken by the auditor if an audit fails is not specified, but note that in general if audit fails, the auditor is in possession of signed proof of the log's misbehavior.

A monitor (Section 8.2) can audit by verifying the consistency of STHs it receives, ensure that each entry can be fetched and that the STH is indeed the result of making a tree from all fetched entries.

A TLS client (Section 8.1) can audit by verifying an SCT against any STH dated after the SCT timestamp + the Maximum Merge Delay by requesting a Merkle inclusion proof (Section 5.4). It can also verify that the SCT corresponds to the server certificate it arrived with (i.e., the log entry is that certificate, or is a precertificate corresponding to that certificate).

Checking of the consistency of the log view presented to all entities is more difficult to perform because it requires a way to share log responses among a set of CT-using entities, and is discussed in Section 11.3.

9. Algorithm Agility

It is not possible for a log to change any of its algorithms part way through its lifetime:

Signature algorithm: SCT signatures must remain valid so signature algorithms can only be added, not removed.

Hash algorithm: A log would have to support the old and new hash algorithms to allow backwards-compatibility with clients that are not aware of a hash algorithm change.

Allowing multiple signature or hash algorithms for a log would require that all data structures support it and would significantly complicate client implementation, which is why it is not supported by this document.

If it should become necessary to deprecate an algorithm used by a live log, then the log MUST be frozen as specified in Section 4.13 and a new log SHOULD be started. Certificates in the frozen log that have not yet expired and require new SCTs SHOULD be submitted to the new log and the SCTs from that log used instead.

10. IANA Considerations

The assignment policy criteria mentioned in this section refer to the policies outlined in [RFC8126].

10.1. Additions to existing registries

This sub-section defines additions to existing registries.

10.1.1. New Entry to the TLS ExtensionType Registry

IANA is asked to add the following entry to the "TLS ExtensionType Values" registry defined in [RFC8446], with an assigned Value:

Value	Extension Name	TLS 1.3	Recommended	Reference
TBD	transparency_info	CH, CR, CT	Y	RFCXXXX

Table 7

10.1.2. URN Sub-namespace for TRANS (urn:ietf:params:trans)

IANA is requested to add a new entry in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry, following the template in [RFC3553]:

Registry name: trans

Specification: RFCXXXX

Repository: https://www.iana.org/assignments/trans

Index value: No transformation needed.

10.2. New CT-Related registries

IANA is requested to add a new protocol registry, "Public Notary Transparency", to the list that appears at https://www.iana.org/assignments/

The rest of this section defines sub-registries to be created within the new Public Notary Transparency registry.

10.2.1. Hash Algorithms

IANA is asked to establish a registry of hash algorithm values, named "Hash Algorithms", that initially consists of:

Value	Hash Algorithm	OID	Reference / Assignment Policy
0x00	SHA-256	2.16.840.1.101.3.4.2.1	[RFC6234]
0x01 - 0xDF	Unassigned		Specification Required
0xE0 - 0xEF	Reserved		Experimental Use
0xF0 - 0xFF	Reserved		Private Use

Table 8

The Designated Expert(s) should ensure that the proposed algorithm has a public specification and is suitable for use as a cryptographic hash algorithm with no known preimage or collision attacks. These attacks can damage the integrity of the log.

10.2.2. Signature Algorithms

IANA is asked to establish a registry of signature algorithm values, named "Signature Algorithms".

The following notes should be added:

* This is a subset of the TLS SignatureScheme Registry, limited to those algorithms that are appropriate for CT. A major advantage of this is leveraging the expertise of the TLS working group and its Designated Expert(s).

* The value 0x0403 appears twice. While this may be confusing, it is okay because the verification process is the same for both algorithms, and the choice of which to use when generating a signature is purely internal to the log server.

The registry should initially consist of:

SignatureScheme Value	Signature Algorithm	Reference / Assignment Policy
0x0000 - 0x0402	Unassigned	Specification Required
ecdsa_secp256r1_sha256(0x0403)	ECDSA (NIST P-256) with SHA-256	[FIPS186-4]
ecdsa_secp256r1_sha256(0x0403)	Deterministic ECDSA (NIST P-256) with HMAC-SHA256	[RFC6979]
0x0404 - 0x0806	Unassigned	Specification Required
ed25519(0x0807)	Ed25519 (PureEdDSA with the edwards25519 curve)	[RFC8032]
0x0808 - 0xFDFF	Unassigned	Expert Review
0xFE00 - 0xFEFF	Reserved	Experimental Use
0xFF00 - 0xFFFF	Reserved	Private Use

Table 9

The Designated Expert(s) should ensure that the proposed algorithm has a public specification, has a value assigned to it in the TLS SignatureScheme Registry (that IANA was asked to establish in [RFC8446]), and is suitable for use as a cryptographic signature algorithm.

10.2.3. VersionedTransTypes

IANA is asked to establish a registry of VersionedTransType values, named "VersionedTransTypes".

The following note should be added:

*The range 0x0000..0x00FF is reserved so that v1 SCTs are distinguishable from v2 SCTs and other TransItem structures.

The registry should initially consist of:

Value	Type and Version	Reference / Assignment Policy
0x0000 - 0x00FF	Reserved	[RFC6962]
0x0100	x509_entry_v2	RFCXXXX
0x0101	precert_entry_v2	RFCXXXX
0x0102	x509_sct_v2	RFCXXXX
0x0103	precert_sct_v2	RFCXXXX
0x0104	signed_tree_head_v2	RFCXXXX
0x0105	consistency_proof_v2	RFCXXXX
0x0106	inclusion_proof_v2	RFCXXXX

Value	Type and Version	Reference / Assignment Policy
0x0107 - 0xDFFF	Unassigned	Specification Required
0xE000 - 0xEFFF	Reserved	Experimental Use
0xF000 - 0xFFFF	Reserved	Private Use

Table 10

The Designated Expert(s) should review the public specification to ensure that it is detailed enough to ensure implementation interoperability.

10.2.4. Log Artifact Extension Registry

IANA is asked to establish a registry of ExtensionType values, named "Log Artifact Extensions", that initially consists of:

ExtensionType	Status	Use	Reference / Assignment Policy
0x0000 - 0xDFFF	Unassigned	n/a	Specification Required
0xE000 - 0xEFFF	Reserved	n/a	Experimental Use
0xF000 - 0xFFFF	Reserved	n/a	Private Use

Table 11

The "Use" column should contain one or both of the following values:

* "SCT", for extensions specified for use in Signed Certificate Timestamps.

* "STH", for extensions specified for use in Signed Tree Heads.

The Designated Expert(s) should review the public specification to ensure that it is detailed enough to ensure implementation interoperability. They should also verify that the extension is appropriate to the contexts in which it is specified to be used (SCT, STH, or both).

10.2.5. Log IDs Registry

IANA is asked to establish a registry of Log IDs, named "Log IDs", that initially consists of:

Log ID	Log Base URL	Log Operator	Reference / Assignment Policy
1.3.101.8192 - 1.3.101.16383	Unassigned	Unassigned	First Come First Served
1.3.101.80.0 - 1.3.101.80.*	Unassigned	Unassigned	First Come First Served

Table 12

All OIDs in the range from 1.3.101.8192 to 1.3.101.16383 have been set aside for Log IDs. This is a limited resource of 8,192 OIDs, each of which has an encoded length of 4 octets.

The 1.3.101.80 arc has also been set aside for Log IDs. This is an unlimited resource, but only the 128 OIDs from 1.3.101.80.0 to 1.3.101.80.127 have an encoded length of only 4 octets.

Each application for the allocation of a Log ID MUST be accompanied by:

* the Log's Base URL (see Section 4.1).

* the Log Operator's contact details.

IANA is asked to reject any request to update a Log ID or Log Base URL in this registry, because these fields are immutable (see Section 4.1).

IANA is asked to accept requests from log operators to update their contact details in this registry.

Since log operators can choose to not use this registry (see Section 4.4), it is not expected to be a global directory of all logs.

10.2.6. Error Types Registry

IANA is requested to create a new registry for errors, the "Error Types" registry.

Requirements for this registry are Specification Required.

This registry should have the following three fields:

Field Name	Type	Reference
identifier	string	RFCXXXX
meaning	string	RFCXXXX
reference	string	RFCXXXX

Table 13

The initial values are as follows, taken from the text above:

Identifier	Meaning	Reference
malformed	The request could not be parsed.	RFCXXXX
badSubmission	submission is neither a valid certificate nor a valid precertificate	RFCXXXX
badType	type is neither 1 nor 2	RFCXXXX
badChain	The first element of chain is not the certifier of the submission, or the	RFCXXXX

Identifier	Meaning	Reference
	second element does not certify the first, etc.	
badCertificate	One or more certificates in the chain are not valid (e.g., not properly encoded)	RFCXXXX
unknownAnchor	The last element of chain (or, if chain is an empty array, the submission) both is not, and is not certified by, an accepted trust anchor	RFCXXXX
shutdown	The log is no longer accepting submissions	RFCXXXX
firstUnknown	first is before the latest known STH but is not from an existing STH.	RFCXXXX
secondUnknown	second is before the latest known STH but is not from an existing STH.	RFCXXXX
secondBeforeFirst	second is smaller than first.	RFCXXXX
hashUnknown	hash is not the hash of a known leaf (may be caused by skew or by a known certificate not yet merged).	RFCXXXX
treeSizeUnknown	hash is before the latest known STH but is not from an existing STH.	RFCXXXX
startUnknown	start is greater than the number of entries in the Merkle tree.	RFCXXXX
endBeforeStart	start cannot be greater than end.	RFCXXXX

Table 14

10.3. OID Assignment

IANA is asked to assign one object identifier from the "SMI Security for PKIX Module Identifier" registry to identify the ASN.1 module in Appendix B of this document with an assigned Decimal value.

Decimal	Description	References
TBD	id-mod-public-notary-v2	RFCXXXX

Table 15

11. Security Considerations

With CAs, logs, and servers performing the actions described here, TLS clients can use logs and signed timestamps to reduce the likelihood that they will accept misissued certificates. If a server presents a valid signed timestamp for a certificate, then the client knows that a log has committed to publishing the certificate. From this, the client knows that monitors acting for the subject of the certificate have had some time to notice the misissuance and take some action, such as asking a CA to revoke a misissued certificate. A signed timestamp does not guarantee this though, since appropriate monitors might not have checked the logs or the CA might have refused to revoke the certificate.

In addition, if TLS clients will not accept unlogged certificates, then site owners will have a greater incentive to submit certificates to logs, possibly with the assistance of their CA, increasing the overall transparency of the system.

11.1. Misissued Certificates

Misissued certificates that have not been publicly logged, and thus do not have a valid SCT, are not considered compliant. Misissued certificates that do have an SCT from a log will appear in that public log within the Maximum Merge Delay, assuming the log is operating correctly. Since a log is allowed to serve an STH of any age up to the MMD, the maximum period of time during which a misissued certificate can be used without being available for audit is twice the MMD.

11.2. Detection of Misissue

The logs do not themselves detect misissued certificates; they rely instead on interested parties, such as domain owners, to monitor them and take corrective action when a misissue is detected.

11.3. Misbehaving Logs

A log can misbehave in several ways. Examples include: failing to incorporate a certificate with an SCT in the Merkle Tree within the MMD; presenting different, conflicting views of the Merkle Tree at different times and/or to different parties; issuing STHs too frequently; mutating the signature of a logged certificate; and failing to present a chain containing the certifier of a logged certificate.

Violation of the MMD contract is detected by log clients requesting a Merkle inclusion proof (Section 5.4) for each observed SCT. These checks can be asynchronous and need only be done once per certificate. However, note that there may be privacy concerns (see Section 8.1.4).

Violation of the append-only property or the STH issuance rate limit can be detected by multiple clients comparing their instances of the STHs. This technique, known as "gossip," is an active area of research and not defined here. Proof of misbehavior in such cases would be: a series of STHs that were issued too closely together, proving violation of the STH issuance rate limit; or an STH with a root hash that does not match the one calculated from a copy of the log, proving violation of the append-only property.

Clients that report back SCTs can be tracked or traced if a log produces multiple STHs or SCTs with the same timestamp and data but different signatures. Logs SHOULD mitigate this risk by either:

* Using deterministic signature schemes, or

* Producing no more than one SCT for each distinct submission and no more than one STH for each distinct tree_size. Each of these SCTs and STHs can be stored by the log and served to other clients that submit the same certificate or request the same STH.

11.4. Multiple SCTs

By requiring TLS servers to offer multiple SCTs, each from a different log, TLS clients reduce the effectiveness of an attack where a CA and a log collude (see Section 6.2).

11.5. Leakage of DNS Information

Malicious monitors can use logs to learn about the existence of domain names that might not otherwise be easy to discover. Some subdomain labels may reveal information about the service and software for which the subdomain is used, which in turn might facilitate targeted attacks.

12. Acknowledgements

The authors would like to thank Erwann Abelea, Robin Alden, Andrew Ayer, Richard Barnes, Al Cutter, David Drysdale, Francis Dupont, Adam Eijdenberg, Stephen Farrell, Daniel Kahn Gillmor, Paul Hadfield, Brad Hill, Jeff Hodges, Paul Hoffman, Jeffrey Hutzelman, Kat Joyce, Stephen Kent, SM, Alexey Melnikov, Linus Nordberg, Chris Palmer, Trevor Perrin, Pierre Phaneuf, Eric Rescorla, Rich Salz, Melinda Shore, Ryan Sleevi, Martin Smith, Carl Wallace and Paul Wouters for their valuable contributions.

A big thank you to Symantec for kindly donating the OIDs from the 1.3.101 arc that are used in this document.

13. References

13.1. Normative References

[FIPS186-4] NIST, "FIPS PUB 186-4", 1 July 2013, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.

[HTML401] Raggett, D., Le Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, 24 December 1999, <http://www.w3.org/TR/1999/REC-html401-19991224>.

[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/info/rfc2119>.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June 2003, <https://www.rfc-editor.org/info/rfc3553>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

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

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/info/rfc5652>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

[RFC6979]
Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[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/info/rfc7633>.

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <https://www.rfc-editor.org/info/rfc7807>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[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/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8391] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <https://www.rfc-editor.org/info/rfc8391>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[UNIXTIME] IEEE, "The Open Group Base Specifications Issue 7 IEEE Std 1003.1-2008, 2016 Edition", n.d., <http://pubs.opengroup.org/onlinepubs/9699919799.2016edition/basedefs/V1_chap04.html#tag_04_16>.

[X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical

Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2002, November 2015.

13.2. Informative References

[CABBR] CA/Browser Forum, "Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates", 2020, <https://cabforum.org/wp-content/uploads/CA-Browser-Forum-BR-1.7.3.pdf>.

[Chromium.Log.Policy] The Chromium Projects, "Chromium Certificate Transparency Log Policy", 2014, <http://www.chromium.org/Home/chromium-security/certificate-transparency/log-policy>.

[Chromium.Policy] The Chromium Projects, "Chromium Certificate Transparency", 2014, <http://www.chromium.org/Home/chromium-security/certificate-transparency>.

[CrosbyWallach] Crosby, S. and D. Wallach, "Efficient Data Structures for Tamper-Evident Logging", Proceedings of the 18th USENIX Security Symposium, Montreal, August 2009, <http://static.usenix.org/event/sec09/tech/full_papers/crosby.pdf>.

[JSON.Metadata] The Chromium Projects, "Chromium Log Metadata JSON Schema", 2014, <https://www.gstatic.com/ct/log_list/log_list_schema.json>.

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

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8820] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 8820, DOI 10.17487/RFC8820, June 2020, <https://www.rfc-editor.org/info/rfc8820>.

Appendix A. Supporting v1 and v2 simultaneously (Informative)

Certificate Transparency logs have to be either v1 (conforming to [RFC6962]) or v2 (conforming to this document), as the data structures are incompatible and so a v2 log could not issue a valid v1 SCT.

CT clients, however, can support v1 and v2 SCTs, for the same certificate, simultaneously, as v1 SCTs are delivered in different TLS, X.509 and OCSP extensions than v2 SCTs.

v1 and v2 SCTs for X.509 certificates can be validated independently. For precertificates, v2 SCTs should be embedded in the TBSCertificate before submission of the TBSCertificate (inside a v1 precertificate, as described in Section 3.1. of [RFC6962]) to a v1 log so that TLS clients conforming to [RFC6962] but not this document are oblivious to the embedded v2 SCTs. An issuer can follow these steps to produce an X.509 certificate with embedded v1 and v2 SCTs:

* Create a CMS precertificate as described in Section 3.2 and submit it to v2 logs.

* Embed the obtained v2 SCTs in the TBSCertificate, as described in Section 7.1.2.

* Use that TBSCertificate to create a v1 precertificate, as described in Section 3.1. of [RFC6962] and submit it to v1 logs.

* Embed the v1 SCTs in the TBSCertificate, as described in Section 3.3 of [RFC6962].

* Sign that TBSCertificate (which now contains v1 and v2 SCTs) to issue the final X.509 certificate.

Appendix B. An ASN.1 Module (Informative)

The following ASN.1 module may be useful to implementors.

```
CertificateTransparencyV2Module-2021
 -- { id-mod-public-notary-v2 from above, in
        iso(1) identified-organization(3) ...
    form }
DEFINITIONS IMPLICIT TAGS ::= BEGIN

-- EXPORTS ALL --

IMPORTS
  EXTENSION
  FROM PKIX-CommonTypes-2009 -- RFC 5912
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkixCommon-02(57) }

  CONTENT-TYPE
  FROM CryptographicMessageSyntax-2010  -- RFC 6268
    { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)
      pkcs-9(9) smime(16) modules(0) id-mod-cms-2009(58) }

  TBSCertificate
  FROM PKIX1Explicit-2009 -- RFC 5912
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkix1-explicit-02(51) }
;

--
-- Section 3.2.  Precertificates
--

ct-tbsCertificate CONTENT-TYPE ::= {
  TYPE TBSCertificate
  IDENTIFIED BY id-ct-tbsCertificate }

id-ct-tbsCertificate OBJECT IDENTIFIER ::= { 1 3 101 78 }

--
-- Section 7.1.  Transparency Information X.509v3 Extension
--

ext-transparencyInfo EXTENSION ::= {
   SYNTAX TransparencyInformationSyntax
   IDENTIFIED BY id-ce-transparencyInfo
   CRITICALITY { FALSE } }

id-ce-transparencyInfo OBJECT IDENTIFIER ::= { 1 3 101 75 }

TransparencyInformationSyntax ::= OCTET STRING
```

```
--
-- Section 7.1.1.  OCSP Response Extension
--

ext-ocsp-transparencyInfo EXTENSION ::= {
   SYNTAX TransparencyInformationSyntax
   IDENTIFIED BY id-pkix-ocsp-transparencyInfo
   CRITICALITY { FALSE } }

id-pkix-ocsp-transparencyInfo OBJECT IDENTIFIER ::=
   id-ce-transparencyInfo

--
-- Section 8.1.2.  Reconstructing the TBSCertificate
--

ext-embeddedSCT-CTv1 EXTENSION ::= {
   SYNTAX SignedCertificateTimestampList
   IDENTIFIED BY id-ce-embeddedSCT-CTv1
   CRITICALITY { FALSE } }

id-ce-embeddedSCT-CTv1 OBJECT IDENTIFIER ::= {
   1 3 6 1 4 1 11129 2 4 2 }

SignedCertificateTimestampList ::= OCTET STRING

END
```

Authors' Addresses

Ben Laurie
Google UK Ltd.

Email: benl@google.com

Adam Langley
Google Inc.

Email: agl@google.com

Emilia Kasper
Google Switzerland GmbH

Email: ekasper@google.com

Eran Messeri
Google UK Ltd.

Email: eranm@google.com

Rob Stradling
Sectigo Ltd.

Email: rob@sectigo.com