Public Notary Transparency Working Group   B. Laurie
Internet-Draft   A. Langley
Intended status: Standards Track   E. Kasper
Expires: November 28, 2016   E. Messeri
                                                              Google
                                                          R. Stradling
                                                              Comodo
                                                          May 27, 2016

Certificate Transparency
draft-ietf-trans-rfc6962-bis-16

Abstract

   This document describes a protocol for publicly logging the existence
   of Transport Layer Security (TLS) 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.

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

Status of This Memo

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 need to be trusted because they are publicly
   auditable. Anyone may verify the correctness of each log and monitor
   when new certificates are added to it. The logs do not themselves
   prevent misissue, but they ensure that interested parties
   (particularly those named in certificates) can detect such
   misissuance. Note that this is a general mechanism; but in this
   document, we only describe its use for public TLS server certificates
   issued by public certification authorities (CAs).

   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.
   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; but, 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 show that any particular instance of the log is
a superset of any particular previous instance. Likewise, Merkle
Trees avoid the need to blindly trust logs: if a log attempts to show
different things to different people, this can be efficiently
detected by comparing tree roots and consistency proofs. Similarly,
other misbehaviors of any log (e.g., issuing signed timestamps for
certificates they then don't log) can be efficiently detected and
proved to the world at large.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in RFC 2119 [RFC2119].

1.2. Data Structures

Data structures are defined according to the conventions laid out in
Section 4 of [RFC5246].

2. Cryptographic Components

2.1. Merkle Hash Trees

Logs use a binary Merkle Hash Tree for efficient auditing. The
hashing algorithm used by each log is expected to be specified as
part of the metadata relating to that log (see Section 10.1). We
have established a registry of acceptable algorithms, see
Section 12.2. The hashing algorithm in use is referred to as HASH
throughout this document 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 || is concatenation and D[k1:k2] denotes the list {d(k1),
d(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.)

2.1.1. 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.

Given an ordered list of n inputs to the tree, D[n] = {d(0), ...,
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,

where : is concatenation of lists and D[k1:k2] denotes the length (k2
- k1) list {d(k1), d(k1+1),..., d(k2-1)} as before.

2.1.2. 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.

Given an ordered list of n inputs to the tree, D[n] = {d(0), ...,
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.

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])

Here, : is a concatenation of lists, and D[k1:k2] denotes the length
(k2 - k1) list {d(k1), d(k1+1),..., d(k2-1)} as before.

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

2.1.3. 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
```
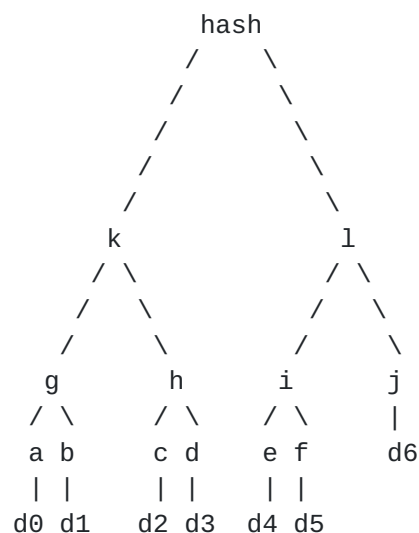
The inclusion proof for d0 is [b, h, l].

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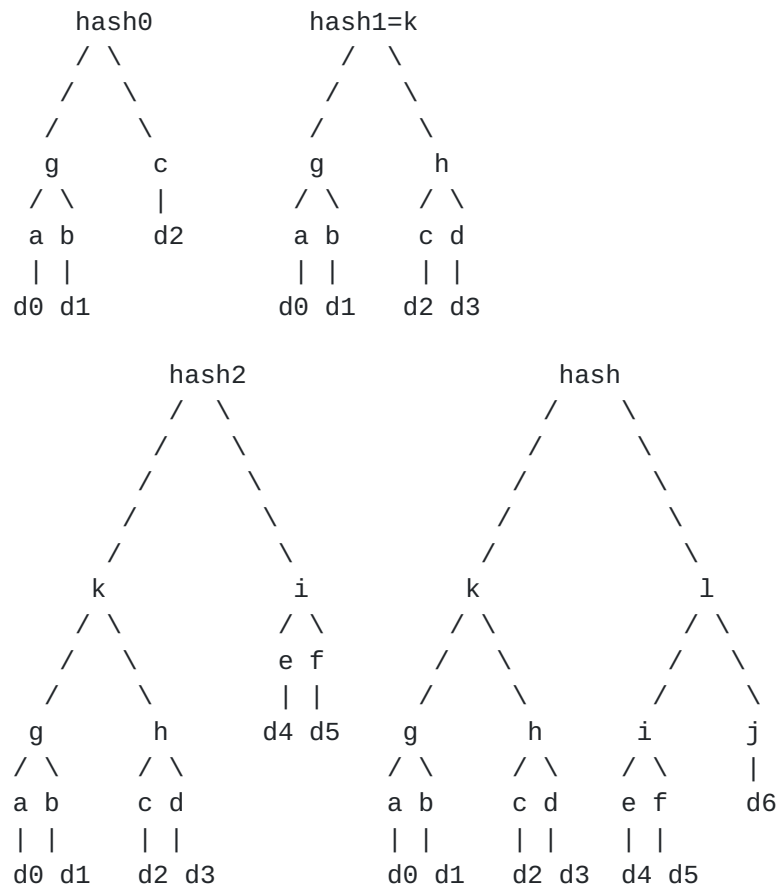
The consistency proof between hash0 and hash is PROOF(3, D[7]) = [c, d, g, l]. c, g are used to verify hash0, and d, l are additionally used to show hash is consistent with hash0.

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.

2.1.4. Signatures

Various data structures are signed. A log MUST use one of the
signature algorithms defined in the Section 12.3.

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. 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 5.6). The submitter SHOULD validate the
returned SCT as described in Section 10.2 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 6.1) to a log. Since
certificates may not be accepted by TLS clients unless logged, it is
expected that certificate owners or their CAs will usually submit
them.

3.2. Precertificates

Alternatively, (root as well as intermediate) CAs may preannounce a
certificate prior to issuance by submitting a precertificate
(Section 6.2) 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. Examples of situations where
the returned SCT is not incorporated into the issued certificate
would be when a CA sends the precertificate to multiple logs, but
only incorporates the SCTs that are returned first, or the CA is
using domain name redaction (Section 4.2) and intends to use another
mechanism to publish SCTs (such as an OCSP response (Section 9.1.1)
or the TLS extension (Section 8.5)).

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

o It MUST be DER encoded.

o "SignedData.encapContentInfo.eContentType" MUST be the OID
   1.3.101.78.

o "SignedData.encapContentInfo.eContent" MUST contain a
   TBSCertificate [RFC5280], which MAY redact certain domain name
   labels that will be present in the issued certificate (see
   Section 4.2) and MUST NOT contain any SCTs, but which will be
   otherwise identical to the TBSCertificate in the issued
   certificate.

o "SignedData.signerInfos" MUST contain a signature from the same
   (root or intermediate) CA that will ultimately issue the
   certificate. This signature indicates the CA's intent to issue
   the certificate. This intent is considered binding (i.e.
   misissuance of the precertificate is considered equivalent to
   misissuance of the certificate). (Note that, because of the
   structure of CMS, the signature on the CMS object will not be a
   valid X.509v3 signature and so cannot be used to construct a
   certificate from the precertificate).

o "SignedData.certificates" SHOULD be omitted.

4. Private Domain Name Labels

Some regard some DNS domain name labels within their registered
domain space as private and security sensitive. Even though these
domains are often only accessible within the domain owner's private
network, it's common for them to be secured using publicly trusted
TLS server certificates. We define a mechanism to allow these
private labels to not appear in public logs.

4.1. Wildcard Certificates

A certificate containing a DNS-ID [RFC6125] of "*.example.com" could
be used to secure the domain "topsecret.example.com", without
revealing the string "topsecret" publicly.

Since TLS clients only match the wildcard character to the complete
leftmost label of the DNS domain name (see Section 6.4.3 of
[RFC6125]), a different approach is needed when more than one of the
leftmost labels in a DNS-ID are considered private (e.g.
"top.secret.example.com"). Also, wildcard certificates are
prohibited in some cases, such as Extended Validation Certificates
[EVSSLGuidelines].

4.2. Redaction of Domain Name Labels

4.2.1. Redacting Labels in Precertificates

   When creating a precertificate, the CA MAY substitute one or more
   labels in each DNS-ID and CN-ID [RFC6125] with a corresponding number
   of "?" labels. Every label to the left of a "?" label MUST also be
   redacted. For example, if a certificate contains a DNS-ID of
   "top.secret.example.com", then the corresponding DNS-ID in the
   precertificate could contain "?.?.example.com" instead, but not
   "top.?.example.com" instead.

   Wildcard "*" labels MUST NOT be redacted. However, if the complete
   leftmost label of a DNS-ID or CN-ID is "*", it is considered redacted
   for the purposes of determining if the label to the right may be
   redacted. For example, if a certificate contains a DNS-ID of
   "*.top.secret.example.com", then the corresponding DNS-ID in the
   precertificate could contain "*.?.?.example.com" instead, but not
   "?.?.?.example.com" instead.

4.2.2. Redacted Labels Certificate Extension

   When a precertificate contains one or more "?" labels, a non-critical
   extension (OID 1.3.101.77, whose extnValue OCTET STRING contains an
   ASN.1 SEQUENCE OF INTEGERs) MUST be added to the corresponding
   certificate. The purpose of this extension is to enable TLS clients
   to reconstruct the TBSCertificate component of the precertificate
   from the certificate, as described in Section 10.2.2.

   Each INTEGER MUST have a value of zero or more. The first INTEGER
   indicates the total number of "?" labels in the precertificate's
   first DNS-ID; the second INTEGER does the same for the
   precertificate's second DNS-ID; etc. The last INTEGER does the same
   for the precertificate's zero or more CN-IDs. There MUST NOT be more
   INTEGERs than there are DNS-IDs (plus one, if any CN-IDs are
   present); if there are fewer INTEGERs than this, the shortfall is
   made up by implicitly repeating the last INTEGER.

4.3. Using a Name-Constrained Intermediate CA

   An intermediate CA certificate or intermediate CA precertificate that
   contains the critical or non-critical Name Constraints [RFC5280]
   extension MAY be logged in place of end-entity certificates issued by
   that intermediate CA, as long as all of the following conditions are
   met:

   o there MUST be a non-critical extension (OID 1.3.101.76, whose
      extnValue OCTET STRING contains ASN.1 NULL data (0x05 0x00)).

This extension is an explicit indication that it is acceptable to
not log certificates issued by this intermediate CA.

o permittedSubtrees MUST specify one or more dNSNames.

o excludedSubtrees MUST specify the entire IPv4 and IPv6 address
   ranges.

Below is an example Name Constraints extension that meets these
conditions:

```
SEQUENCE {
  OBJECT IDENTIFIER '2 5 29 30'
  OCTET STRING, encapsulates {
    SEQUENCE {
      [0] {
        SEQUENCE {
          [2] 'example.com'
          }
        }
      [1] {
        SEQUENCE {
          [7] 00 00 00 00 00 00 00 00
          }
        SEQUENCE {
          [7]
            00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
            00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
          }
        }
      }
    }
  }
```

5. Log Format and Operation

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

When it receives 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 (to reduce the ability to track
clients as described in Section 13.5). 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 incorporate the submitted entry in its
Merkle Tree no later than a fixed amount of time, known as the
Maximum Merge Delay (MMD), after the issuance of the SCT.
Periodically, the log MUST append all its new entries to its Merkle
Tree and sign the root of the tree.

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

5.1. Accepting Submissions

Logs MUST verify that each submitted certificate or precertificate
has a valid signature chain to an accepted trust anchor, using the
chain of intermediate CA certificates provided by the submitter.
Logs MUST accept certificates and precertificates that are fully
valid according to RFC 5280 [RFC5280] verification rules and are
submitted with such a chain. Logs MAY accept certificates and
precertificates that have expired, are not yet valid, have been
revoked, or are otherwise not fully valid according to RFC 5280
verification rules in order to accommodate quirks of CA certificate-
issuing software. However, logs MUST reject submissions without a
valid signature chain to an accepted trust anchor. Logs MUST also
reject precertificates that do not conform to the requirements in
Section 3.2.

Logs SHOULD limit the length of chain they will accept. The maximum
chain length is specified in the log's metadata.

The log SHALL allow retrieval of its list of accepted trust anchors
(see Section 6.8), 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.

5.2. 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 present this chain for auditing upon
request (see Section 6.7). This chain is required to prevent a CA
from avoiding blame by logging a partial or empty chain.

Each certificate entry in a log MUST include a "X509ChainEntry"
structure, and each precertificate entry MUST include a
"PrecertChainEntryV2" structure:

```
opaque ASN.1Cert<1..2^24-1>;

struct {
    ASN.1Cert leaf_certificate;
    ASN.1Cert certificate_chain<0..2^24-1>;
} X509ChainEntry;

opaque CMSPrecert<1..2^24-1>;

struct {
    CMSPrecert pre_certificate;
    ASN.1Cert precertificate_chain<1..2^24-1>;
} PrecertChainEntryV2;
```

"leaf_certificate" is a submitted certificate that has been accepted
by the log.

"certificate_chain" is a vector of 0 or more additional certificates
required to verify "leaf_certificate". The first certificate MUST
certify "leaf_certificate". Each following certificate MUST directly
certify the one preceding it. The final certificate MUST be a trust
anchor accepted by the log. If "leaf_certificate" is an accepted
trust anchor, then this vector is empty.

"pre_certificate" is a submitted precertificate that has been
accepted by the log.

"precertificate_chain" is a vector of 1 or more additional
certificates required to verify "pre_certificate". The first
certificate MUST certify "pre_certificate". Each following
certificate MUST directly certify the one preceding it. The final
certificate MUST be a trust anchor accepted by the log.

5.3. Log ID

Each log is uniquely identified by an OID. A log's operator MUST
either allocate the OID themselves or request an OID from one of the
two Log ID Registries (see Section 12.6.1 and Section 12.6.2). The
OID is specified in the log's metadata. 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 DER encoding of the OID can be
reproduced simply by prepending an OBJECT IDENTIFIER tag (0x06) to
the opaque vector length and contents.

5.4. 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 {
    reserved(0),
    x509_entry_v2(1), precert_entry_v2(2),
    x509_sct_v2(3), precert_sct_v2(4),
    tree_head_v2(5), signed_tree_head_v2(6),
    consistency_proof_v2(7), inclusion_proof_v2(8),
    x509_sct_with_proof_v2(9), precert_sct_with_proof_v2(10),
    (65535)
} 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 tree_head_v2: TreeHeadDataV2;
        case signed_tree_head_v2: SignedTreeHeadDataV2;
        case consistency_proof_v2: ConsistencyProofDataV2;
        case inclusion_proof_v2: InclusionProofDataV2;
        case x509_sct_with_proof_v2: SCTWithProofDataV2;
        case precert_sct_with_proof_v2: SCTWithProofDataV2;
    } data;
} TransItem;
```

"versioned_type" is 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.

5.5. Merkle Tree Leaves

The leaves of a log's Merkle Tree correspond to the log's entries
(see Section 5.2). 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 hashing algorithm is specified in the log's metadata.

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

    struct {
        uint64 timestamp;
        opaque issuer_key_hash[HASH_SIZE];
        TBSCertificate tbs_certificate;
        SctExtension sct_extensions<0..2^16-1>;
    } TimestampedCertificateEntryDataV2;
```

"timestamp" is the NTP Time [RFC5905] at which the certificate or
precertificate was accepted by the log, measured in milliseconds
since the epoch (January 1, 1970, 00:00), ignoring leap seconds.
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".

"tbs_certificate" is the DER encoded TBSCertificate from either the
"leaf_certificate" (in the case of an "X509ChainEntry") or the
"pre_certificate" (in the case of a "PrecertChainEntryV2"). (Note
that a precertificate's TBSCertificate can be reconstructed from the
corresponding certificate as described in Section 10.2.2).

"sct_extensions" matches the SCT extensions of the corresponding SCT.

5.6. Signed Certificate Timestamp (SCT)

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

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

       struct {
           SctExtensionType sct_extension_type;
           opaque sct_extension_data<0..2^16-1>;
       } SctExtension;

       struct {
           LogID log_id;
           uint64 timestamp;
           SctExtension sct_extensions<0..2^16-1>;
           digitally-signed struct {
               TransItem timestamped_entry;
           } signature;
       } SignedCertificateTimestampDataV2;
```

   "log_id" is this log's unique ID, encoded in an opaque vector as
   described in Section 5.3.

   "timestamp" is equal to the timestamp from the
   "TimestampedCertificateEntryDataV2" structure encapsulated in the
   "timestamped_entry".

   "sct_extension_type" identifies a single extension from the IANA
   registry in Section 12.4. At the time of writing, no extensions are
   specified.

   The interpretation of the "sct_extension_data" field is determined
   solely by the value of the "sct_extension_type" field. Each document
   that registers a new "sct_extension_type" must describe how to
   interpret the corresponding "sct_extension_data".

   "sct_extensions" is a vector of 0 or more SCT extensions. This
   vector MUST NOT include more than one extension with the same
   "sct_extension_type". The extensions in the vector MUST be ordered
   by the value of the "sct_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.

The encoding of the digitally-signed element is defined in [RFC5246].

"timestamped_entry" is a "TransItem" structure that MUST be of type "x509_entry_v2" or "precert_entry_v2" (see Section 5.5).

5.7. Merkle Tree Head

The log stores information about its Merkle Tree in a "TransItem" structure of type "tree_head_v2", which encapsulates a "TreeHeadDataV2" structure:

```
opaque NodeHash[HASH_SIZE];

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

"timestamp" is the current NTP Time [RFC5905], measured in milliseconds since the epoch (January 1, 1970, 00:00), ignoring leap seconds.

"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" matches the STH extensions of the corresponding STH.

5.8. Signed Tree Head (STH)

Periodically each log SHOULD sign its current tree head information (see Section 5.7) to produce an STH. When a client requests a log's latest STH (see Section 6.3), the log MUST return an STH that is no older than the log's MMD. However, STHs could be used to mark individual clients (by producing a new one for each query), so logs MUST NOT produce them more frequently than is declared in their metadata. In general, there is no need to produce a new STH unless there are new entries in the log; however, in the unlikely event that it receives no new submissions during an MMD period, the log SHALL 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:

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

struct {
    SthExtensionType sth_extension_type;
    opaque sth_extension_data<0..2^16-1>;
} SthExtension;

struct {
    LogID log_id;
    uint64 timestamp;
    uint64 tree_size;
    NodeHash root_hash;
    SthExtension sth_extensions<0..2^16-1>;
    digitally-signed struct {
        TransItem merkle_tree_head;
    } signature;
} SignedTreeHeadDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as
described in Section 5.3.

"timestamp" is equal to the timestamp from the "TreeHeadDataV2"
structure encapsulated in "merkle_tree_head". This timestamp 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_size" is equal to the tree size from the "TreeHeadDataV2"
structure encapsulated in "merkle_tree_head".

"root_hash" is equal to the root hash from the "TreeHeadDataV2"
structure encapsulated in "merkle_tree_head".

"sth_extension_type" identifies a single extension from the IANA
registry in Section 12.5. At the time of writing, no extensions are
specified.

The interpretation of the "sth_extension_data" field is determined
solely by the value of the "sth_extension_type" field. Each document
that registers a new "sth_extension_type" must describe how to
interpret the corresponding "sth_extension_data".

"sth_extensions" is a vector of 0 or more STH extensions. This
vector MUST NOT include more than one extension with the same
"sth_extension_type". The extensions in the vector MUST be ordered
by the value of the "sth_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.

"merkle_tree_head" is a "TransItem" structure that MUST be of type
"tree_head_v2" (see Section 5.7).

5.9. 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<1..2^8-1>;
    } ConsistencyProofDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as
described in Section 5.3.

"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.

5.10. 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<1..2^8-1>;
    } InclusionProofDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as described in Section 5.3.

"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.

5.11. 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. To avoid that, the following actions are suggested:

o Make it known to clients and monitors that the log will be frozen.

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

o Once MMD from the last accepted submission has passed and all pending submissions are incorporated, issue a final STH and publish it as a part of the log's metadata. 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.

o 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).

6. 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 [RFC4627]. 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 [RFC4648] as specified in the individual messages.

Note that JSON objects and URL parameters may contain fields not
specified here. These extra fields should be ignored.

The <log server> prefix, which is part of the log's metadata, MAY
include a path as well as a server name and a port.

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 [RFC2616]), and, in place of the
responses outlined in the subsections below, the body SHOULD be a
JSON structure containing at least the following field:

error_message: A human-readable string describing the error which
   prevented the log from processing the request.

   In the case of a malformed request, the string SHOULD provide
   sufficient detail for the error to be rectified.

error_code: An error code readable by the client. Some codes are
   generic and are detailed here. Others are detailed in the
   individual requests. Error codes are fixed text strings.

   not compliant The request is not compliant with this RFC.

e.g. In response to a request of "/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:

```
{
    "error_message": "'start' cannot be greater than 'end'",
    "error_code": "not compliant",
}
```

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
[RFC2616], in the case of a 503 response the log MAY include a
"Retry-After:" header in order to request a minimum time for the
client to wait before retrying the request.

6.1. Add Chain to Log

POST https://<log server>/ct/v2/add-chain

Inputs:

   chain: An array of base64 encoded certificates. The first
      element is the certificate for which the submitter desires an
      SCT; the second certifies the first and so on to the last,
      which either is, or is certified by, an accepted trust anchor.

Outputs:

   sct: A base64 encoded "TransItem" of type "x509_sct_v2", signed
      by this log, that corresponds to the submitted certificate.

Error codes:

   unknown anchor The last certificate in the chain both is not, and
      is not certified by, an accepted trust anchor.

   bad chain The alleged chain is not actually a chain of
      certificates.

   bad certificate One or more certificates in the chain are not
      valid (e.g. not properly encoded).

   shutdown The log has ceased operation and is not accepting new
      submissions.

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 10.3) can then detect these encoding errors, which may be
accepted by some TLS clients.

6.2. Add PreCertChain to Log

POST https://<log server>/ct/v2/add-pre-chain

Inputs:

   precertificate: The base64 encoded precertificate.

   chain: An array of base64 encoded CA certificates. The first
      element is the signer of the precertificate; the second
      certifies the first and so on to the last, which either is, or
      is certified by, an accepted trust anchor.

Outputs:

   sct: A base64 encoded "TransItem" of type "precert_sct_v2",
      signed by this log, that corresponds to the submitted
      precertificate.

Errors are the same as in Section 6.1.

6.3. Retrieve Latest Signed Tree Head

GET https://<log server>/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.

6.4. Retrieve Merkle Consistency Proof between Two Signed Tree Heads

GET https://<log server>/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.

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:

first unknown "first" is before the latest known STH but is not
   from an existing STH.

second unknown "second" is before the latest known STH but is not
   from an existing STH.

See Section 10.4.2 for an outline of how to use the "consistency"
output.

6.5. Retrieve Merkle Inclusion Proof from Log by Leaf Hash

GET https://<log server>/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 5.5. The
"tree_size" must designate an existing v2 STH. Because of skew,
the front-end may not know the requested STH. 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 STH 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 chosen certificate 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:

   hash unknown "hash" is not the hash of a known leaf (may be
      caused by skew or by a known certificate not yet merged).

   tree_size unknown "hash" is before the latest known STH but is
      not from an existing STH.

See Section 10.4.1 for an outline of how to use the "inclusion"
output.

6.6. Retrieve Merkle Inclusion Proof, Signed Tree Head and Consistency Proof by Leaf Hash

GET https://<log server>/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 5.5. The
   "tree_size" must designate an existing v2 STH.

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

latest STH < requested STH Return latest STH.

latest STH > requested STH Return latest STH and a consistency
   proof between it and the requested STH (see Section 6.4).

index of requested hash < latest STH Return "inclusion".

Note that more than one case can be true, in which case the
returned data is their concatenation. 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 chosen certificate 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 STH 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 6.5.

See Section 10.4.1 for an outline of how to use the "inclusion"
output, and see Section 10.4.2 for an outline of how to use the
"consistency" output.

6.7. Retrieve Entries and STH from Log

GET https://<log server>/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

leaf_input: The base64 encoded "TransItem" structure of type
"x509_entry_v2" or "precert_entry_v2" (see Section 5.5).

log_entry: The base64 encoded log entry (see Section 5.2). In
the case of an "x509_entry_v2" entry, this is the whole
"X509ChainEntry"; and in the case of a "precert_entry_v2",
this is the whole "PrecertChainEntryV2".

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 6.3.

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

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".

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 10.4.3 for an outline of how to use a complete list of
"leaf_input" entries to verify the "root_hash".

6.8. Retrieve Accepted Trust Anchors

GET https://<log server>/ct/v2/get-anchors

No inputs.

Outputs:

   certificates: An array of base64 encoded trust anchors that are
      acceptable to the log.

   max_chain: 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.

7. Optional Client Messages

Logs MAY implement these messages. They are not required for correct
operation of logs or their clients, but may be convenient in some
circumstances. Note that mirrors can implement these even if the log
they are mirroring does not.

7.1. Get Entry Number for SCT

GET https://<log server>/ct/v2/get-entry-for-sct

Inputs:

   sct: A base64 encoded "TransItem" of type "x509_sct_v2" or
      "precert_sct_v2" signed by this log.

Outputs:

   entry: 0-based index of the log entry corresponding to the
      supplied SCT.

Error codes:

   bad signature "sct" is not signed by this log.

   not found "sct" does not correspond to an entry that is currently
      available.

Note that any SCT signed by a log MUST have a corresponding entry in
the log, but it may not be retrievable until the MMD has passed since
the SCT was issued.

7.2. Get Entry Numbers for Certificate

   GET https://<log server>/ct/v2/get-entry-for-certificate

   Inputs:

   hash: A base64 encoded HASH of a "TBSCertificate". Note that if
      the certificate has redacted labels then the "TBSCertificate"
      must be constructed as described above (Section 4.2).

   Outputs:

   entries: An array of 0-based indices of log entries corresponding
      to the supplied HASH.

   Error codes:

   bad hash "hash" is not the right size or format.

   not found "sct" does not correspond to an entry that is currently
      available.

Note that it is possible for a certificate to be logged more than
once. If that is the case, the log MAY return more than one entry
index. If the certificate is present in the log, then the log MUST
return at least one entry index.

8. TLS Servers

TLS servers MUST use at least one of the three mechanisms listed
below to present one or more SCTs from one or more logs to each TLS
client during full TLS handshakes, where each SCT corresponds to the
server certificate or to a name-constrained intermediate the server
certificate chains to. TLS servers SHOULD also present corresponding
inclusion proofs and STHs (see Section 8.3).

Three mechanisms are provided because they have different tradeoffs.

o A TLS extension (Section 7.4.1.4 of [RFC5246]) with type
   "transparency_info" (see Section 8.5). 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.

o An Online Certificate Status Protocol (OCSP) [RFC6960] response
   extension (see Section 9.1.1), where the OCSP response is provided
   in the "CertificateStatus" message, provided that the TLS client
   included the "status_request" extension in the (extended)
   "ClientHello" (Section 8 of [RFC6066]). This mechanism, popularly
   known as OCSP stapling, is already widely (but not universally)
   implemented. It also allows SCTs and inclusion proofs to be
   updated on the fly.

o An X509v3 certificate extension (see Section 9.1.2). 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 where 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 will
   subsequently consider the certificate to be non-compliant and in
   need of re-issuance.

   Additionally, a TLS server which supports presenting SCTs using an
   OCSP response MAY provide it when the TLS client included the
   "status_request_v2" extension ([RFC6961]) in the (extended)
   "ClientHello", but only in addition to at least one of the three
   mechanisms listed above.

8.1. Multiple SCTs

   TLS servers SHOULD send SCTs from multiple logs in case one or more
   logs are not acceptable to the TLS client (for example, if a log has
   been struck off for misbehavior, has had a key compromise, or is not
   known to the TLS client). For example:

o If a CA and a log collude, it is possible to temporarily hide
   misissuance from clients. Including SCTs from different logs
   makes it more difficult to mount this attack.

o If a log misbehaves, 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), servers may wish to reduce the probability of their
   certificates being rejected as a result by including SCTs from
   different logs.

o TLS clients may have policies related to the above risks requiring
   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.

8.2. 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).

8.3. Presenting SCTs, inclusion proofs and STHs

When constructing a "TransItemList" structure, a TLS server SHOULD
construct and include "TransItem" structures of type
"x509_sct_with_proof_v2" (for an SCT of type "x509_sct_v2") or
"precert_sct_with_proof_v2" (for an SCT of type "precert_sct_v2"),
both of which encapsulate a "SCTWithProofDataV2" structure:

```
struct {
    SignedCertificateTimestampDataV2 sct;
    SignedTreeHeadDataV2 sth;
    InclusionProofDataV2 inclusion_proof;
} SCTWithProofDataV2;
```

"sct" is the encapsulated data structure from an SCT that corresponds
to the server certificate or to a name-constrained intermediate the
server certificate chains to.

"sth" is the encapsulated data structure from an STH that was signed
by the same log as "sct".

"inclusion_proof" is the encapsulated data structure from an
inclusion proof that corresponds to "sct" and can be used to compute
the root in "sth".

8.4. Presenting SCTs only

Presenting inclusion proofs and STHs in the TLS handshake helps to
protect the client's privacy (see Section 10.2.4) and reduces load on
log servers. However, if a TLS server is unable to obtain an
inclusion proof and STH that correspond to an SCT, then it MUST
include "TransItem" structures of type "x509_sct_v2" or
"precert_sct_v2" in the "TransItemList".

8.5. transparency_info TLS Extension

Provided that a TLS client includes the "transparency_info" extension
type in the ClientHello, the TLS server SHOULD include the
"transparency_info" extension in the ServerHello with
"extension_data" set to a "TransItemList". The TLS server SHOULD
ignore any "extension_data" sent by the TLS client. Additionally,
the TLS server MUST NOT process or include this extension when a TLS
session is resumed, since session resumption uses the original
session information.

9. Certification Authorities

9.1. 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 9.1.1) and certificates (see
Section 9.1.2). 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".

9.1.1. OCSP Response Extension

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

or for a precertificate that corresponds to that certificate, or for
a name-constrained intermediate to which that certificate chains.

9.1.2. Certificate Extension

A certification authority MAY include a Transparency Information
X.509v3 extension in a certificate. Any included SCTs or inclusion
proofs MUST be either for a precertificate that corresponds to this
certificate, or for a name-constrained intermediate to which this
certificate chains.

9.2. TLS Feature Extension

A certification authority MAY include the transparency_info
(Section 8.5) TLS extension identifier in the TLS Feature [RFC7633]
certificate extension in root, intermediate and end-entity
certificates. When a certificate chain includes such a certificate,
this indicates that CT compliance is required.

10. 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 metadata in order to communicate with logs
and verify their responses. This metadata is described below, but
note that this document does not describe how the metadata is
obtained, which is implementation dependent (see, for example,
[Chromium.Policy]).

Clients should somehow exchange STHs they see, or make them available
for scrutiny, in order to ensure that they all have a consistent
view. The exact mechanisms will be in separate documents, but it is
expected there will be a variety.

10.1. Metadata

In order to communicate with and verify a log, clients need metadata
about the log.

Base URL: The URL to substitute for <log server> in Section 6.

Hash Algorithm The hash algorithm used for the Merkle Tree (see
     Section 12.2).

Signing Algorithm The signing algorithm used (see Section 2.1.4).

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.

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

Maximum Chain Length The longest chain submission the log is willing
     to accept, if the log chose to limit it.

STH Frequency Count The maximum number of STHs the log may produce
     in any period equal to the "Maximum Merge Delay" (see
     Section 5.8).

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.

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

10.2. TLS Client

10.2.1. Receiving SCTs

TLS clients receive SCTs alongside or in certificates. TLS clients
MUST implement all of the three mechanisms by which TLS servers may
present SCTs (see Section 8). TLS clients MAY also accept SCTs via
the "status_request_v2" extension ([RFC6961]). TLS clients that
support the "transparency_info" TLS extension SHOULD include it in
ClientHello messages, with empty "extension_data". TLS clients may
also receive inclusion proofs in addition to SCTs, which should be
checked once the SCTs are validated.

10.2.2. Reconstructing the TBSCertificate

To reconstruct the TBSCertificate component of a precertificate from
a certificate, TLS clients should:

o Remove the Redacted Labels extension described in Section 4.2.2

o Replace leftmost labels of each DNS-ID with "?", based on the
   INTEGER value corresponding to that DNS-ID in the extension.

A certificate with redacted labels where one of the redacted labels
is "*" MUST NOT be considered compliant.

If the SCT checked is for a Precertificate (where the "type" of the
"TransItem" is "precert_sct_v2"), then the client SHOULD also remove
embedded v1 SCTs, identified by OID 1.3.6.1.4.1.11129.2.4.2 (See
Section 3.3. of [RFC6962]), in the process of reconstructing the
TBSCertificate. That is to allow embedded v1 and v2 SCTs to co-exist
in a certificate (See Appendix A).

10.2.3. Validating SCTs

In addition to normal validation of the server certificate and its
chain, TLS clients SHOULD validate each received SCT for which they
have the corresponding log's metadata. To validate an SCT, a TLS
client computes the signature input from the SCT data and the
corresponding certificate, and then verifies the signature using the
corresponding log's public key. TLS clients MUST NOT consider valid
any SCT whose timestamp is in the future.

Before considering any SCT to be invalid, the TLS client MUST attempt
to validate it against the server certificate and against each of the
zero or more suitable name-constrained intermediates (Section 4.3) in
the chain. These certificates may be evaluated in the order they
appear in the chain, or, indeed, in any order.

10.2.4. Validating inclusion proofs

After validating a received SCT, a TLS client MAY request a
corresponding inclusion proof (if one is not already available) and
then verify it. An inclusion proof can be requested directly from a
log using "get-proof-by-hash" (Section 6.5) or "get-all-by-hash"
(Section 6.6), but note that this will disclose to the log which TLS
server the client has been communicating with.

Alternatively, if the TLS client has received an inclusion proof (and
an STH) alongside the SCT, it can proceed to verifying the inclusion
proof to the provided STH. The client then has to verify consistency
between the provided STH and an STH it knows about, which is less
sensitive from a privacy perspective.

TLS clients SHOULD also verify each received inclusion proof (see
Section 10.4.1) for which they have the corresponding log's metadata,
to audit the log and gain confidence that the certificate is logged.

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 6.3),
then verify it by requesting a consistency proof (Section 6.4). Note
that if the TLS client uses "get-all-by-hash", then it will already
have the new STH.

10.2.5. Evaluating compliance

To be considered compliant, a certificate MUST be accompanied by at
least one valid SCT. A certificate not accompanied by any valid SCTs
MUST NOT be considered compliant by TLS clients.

10.2.6. TLS Feature Extension

If any certificate in a chain includes the transparency_info
(Section 8.5) TLS extension identifier in the TLS Feature [RFC7633]
certificate extension, then CT compliance (using any of the
mechanisms from Section 8) is required.

TLS clients MUST treat certificates which fail this requirement as
non-compliant.

10.2.7. Handling of Non-compliance

If a TLS server presents a certificate chain that is non-compliant,
there are two possibilities.

o In the case that use of TLS with a valid certificate is mandated
   by explicit security policy, application protocol specification,
   or other means, the TLS client MUST refuse the connection.

o If the use of TLS with a valid certificate is optional, the TLS
   client MAY accept the connection but MUST NOT treat the
   certificate as valid.

10.3. 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 needs to, at least, inspect every new entry in each log it
watches. It may also want to keep copies of entire logs. In order
to do this, it should follow these steps for each log:

1. Fetch the current STH (Section 6.3).

2. Verify the STH signature.

3. Fetch all the entries in the tree corresponding to the STH
    (Section 6.7).

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

5. Fetch the current STH (Section 6.3). Repeat until the STH
    changes.

6. Verify the STH signature.

7. Fetch all the new entries in the tree corresponding to the STH
    (Section 6.7). If they remain unavailable for an extended
    period, then this should be viewed as misbehavior on the part of
    the log.

8. 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 6.4).

    2. Verify the consistency proof.

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

9. Go to Step 5.

10.4. 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 behaviour properties that should be
checked:

o The Maximum Merge Delay (MMD).

o The STH Frequency Count.

o The append-only property.

o 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 10.3) 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 10.2) 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 6.5). It can also verify that the SCT corresponds to the certificate it arrived with (i.e. the log entry is that certificate, is a precertificate for that certificate or is an appropriate name-constrained intermediate [see Section 4.3]).

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-aware entities, and is discussed in Section 13.4.

The following algorithm outlines may be useful for clients that wish to perform various audit operations.

10.4.1. Verifying an inclusion proof

When a client has received a "TransItem" of type "inclusion_proof_v2" and wishes to verify inclusion of an input "hash" for an STH with 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" against "tree_size". If "leaf_index" is greater than or equal to "tree_size" 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 "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:

       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.

10.4.2. Verifying consistency between two STHs

When a client has an STH "first_hash" for tree size "first", an STH
"second_hash" for tree size "second" where "0 < first < second", and
has received a "TransItem" of type "consistency_proof_v2" that they
wish to use to verify both hashes, the following algorithm may be
used:

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

2. Set "fn" to "first - 1" and "sn" to "second - 1".

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

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

5. 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:

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

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

6. 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.

10.4.3. Verifying root hash given entries

When a client has a complete list of leaf input "entries" from "0" up
to "tree_size - 1" and wishes to verify this list against an STH
"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. In other words, set
        "merge_count" to the number of consecutive "1"s 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 (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".

11. 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 should be frozen as specified in Section 10.1
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.

12. IANA Considerations

12.1. TLS Extension Type

IANA is asked to allocate an RFC 5246 ExtensionType value for the
"transparency_info" TLS extension. IANA should update this extension
type to point at this document.

12.2. Hash Algorithms

IANA is asked to establish a registry of hash values, initially
consisting of:

```
             +-------+---------------------+
             | Index | Hash                |
             +-------+---------------------+
             | 0     | SHA-256 [FIPS.180-4] |
             +-------+---------------------+
```

12.3. Signature Algorithms

IANA is asked to establish a registry of signature algorithm values,
initially consisting of:

```
+-------+--------------------------------------------------------+
| Index | Signature Algorithm                                    |
+-------+--------------------------------------------------------+
| 0     | deterministic ECDSA [RFC6979] using the NIST P-256 curve |
|       | (Section D.1.2.3 of the Digital Signature Standard [DSS]) |
|       | and HMAC-SHA256                                        |
| 1     | RSA signatures (RSASSA-PKCS1-v1_5 with SHA-256, Section |
|       | 8.2 of [RFC3447]) using a key of at least 2048 bits.   |
+-------+--------------------------------------------------------+
```

12.4. SCT Extensions

IANA is asked to establish a registry of SCT extensions, initially consisting of:

```
                    +-------+-----------+
                    | Type  | Extension |
                    +-------+-----------+
                    | 65535 | reserved  |
                    +-------+-----------+
```

TBD: policy for adding to the registry

12.5. STH Extensions

IANA is asked to establish a registry of STH extensions, initially consisting of:

```
                    +-------+-----------+
                    | Type  | Extension |
                    +-------+-----------+
                    | 65535 | reserved  |
                    +-------+-----------+
```

TBD: policy for adding to the registry

12.6. Object Identifiers

This document uses object identifiers (OIDs) to identify Log IDs (see Section 5.3), the precertificate CMS "eContentType" (see Section 3.2), and X.509v3 extensions in certificates (see Section 4.2.2, Section 4.3 and Section 9.1.2) and OCSP responses (see Section 9.1.1). The OIDs are defined in an arc that was selected due to its short encoding.

12.6.1. Log ID Registry 1

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

IANA is requested to establish a registry that will allocate Log IDs
from this range.

TBD: policy for adding to the registry. Perhaps "Expert Review"?

12.6.2. Log ID Registry 2

The 1.3.101.80 arc has been delegated. 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.

IANA is requested to establish a registry that will allocate Log IDs
from this arc.

TBD: policy for adding to the registry. Perhaps "Expert Review"?

13. 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 misissue and take some
action, such as asking a CA to revoke a misissued certificate, or
that the log has misbehaved, which will be discovered when the SCT is
audited. A signed timestamp is not a guarantee that the certificate
is not misissued, 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.

13.1. Misissued Certificates

Misissued certificates that have not been publicly logged, and thus
do not have a valid SCT, are not considered compliant (so TLS clients
may decide, for example, to reject them). 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.
Thus, the maximum period of time during which a misissued certificate
can be used without being available for audit is the MMD.

13.2. 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.

13.3. Avoiding Overly Redacting Domain Name Labels

Redaction of domain name labels carries the same risks as the use of
wildcards (See Section 7.2 of [RFC6125], for example). If the
entirety of the domain space below the unredacted part of a domain
name is not controlled by a single entity (e.g. "?.com", "?.co.uk"
and other public suffixes [Public.Suffix.List]), then the domain name
may be considered by clients to be overly redacted.

CAs should take care to avoid overly redacting domain names in
precertificates. It is expected that monitors will treat
precertificates that contain overly redacted domain names as
potentially misissued. TLS clients MAY consider a certificate to be
non-compliant if the reconstructed TBSCertificate (Section 10.2.2)
contains any overly redacted domain names.

13.4. 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 and issuing STHs too
frequently. Such misbehavior is detectable and the
[I-D.ietf-trans-threat-analysis] provides more details on how this
can be done.

Violation of the MMD contract is detected by log clients requesting a
Merkle inclusion proof (Section 6.5) for each observed SCT. These
checks can be asynchronous and need only be done once per each
certificate. In order to protect the clients' privacy, these checks
need not reveal the exact certificate to the log. Instead, clients
can request the proof from a trusted auditor (since anyone can
compute the proofs from the log) or communicate with the log via
proxies.

Violation of the append-only property or the STH issuance rate limit
can be detected by clients comparing their instances of the Signed
Tree Heads. There are various ways this could be done, for example

via gossip (see [I-D.ietf-trans-gossip]) or peer-to-peer
communications or by sending STHs to monitors (who could then
directly check against their own copy of the relevant log). A 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.

13.5. Deterministic Signatures

Logs are required to use deterministic signatures for the following
reasons:

o Using non-deterministic ECDSA with a predictable source of
   randomness means that each signature can potentially expose the
   secret material of the signing key.

o Clients that gossip STHs or report back SCTs can be tracked or
   traced if a log was to produce multiple STHs or SCTs with the same
   timestamp and data but different signatures.

13.6. Multiple SCTs

By offering multiple SCTs, each from a different log, TLS servers
reduce the effectiveness of an attack where a CA and a log collude
(see Section 8.1).

13.7. Threat Analysis

[I-D.ietf-trans-threat-analysis] provides a more detailed threat
analysis of the Certificate Transparency architecture.

14. Acknowledgements

The authors would like to thank Erwann Abelea, Robin Alden, Al
Cutter, 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, 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.

15. References

15.1. Normative References

[DSS]   National Institute of Standards and Technology, "Digital
           Signature Standard (DSS)", FIPS 186-3, June 2009,
           <http://csrc.nist.gov/publications/fips/fips186-3/
           fips_186-3.pdf>.

[FIPS.180-4]
           National Institute of Standards and Technology, "Secure
           Hash Standard", FIPS PUB 180-4, March 2012,
           <http://csrc.nist.gov/publications/fips/fips180-4/
           fips-180-4.pdf>.

[HTML401] Raggett, D., Le Hors, A., and I. Jacobs, "HTML 4.01
           Specification", World Wide Web Consortium Recommendation
           REC-html401-19991224, 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, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H.,
           Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext
           Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography
           Standards (PKCS) #1: RSA Cryptography Specifications
           Version 2.1", RFC 3447, February 2003.

[RFC4627] Crockford, D., "The application/json Media Type for
           JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data
           Encodings", RFC 4648, October 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
           (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[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, May 2008.

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

   [RFC5905] Mills, D., Martin, J., Burbank, J., and W. Kasch, "Network
              Time Protocol Version 4: Protocol and Algorithms
              Specification", RFC 5905, June 2010.

   [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions:
              Extension Definitions", RFC 6066, January 2011.

   [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and
              Verification of Domain-Based Application Service Identity
              within Internet Public Key Infrastructure Using X.509
              (PKIX) Certificates in the Context of Transport Layer
              Security (TLS)", RFC 6125, March 2011.

   [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,
              <http://www.rfc-editor.org/info/rfc6960>.

   [RFC6961] Pettersen, Y., "The Transport Layer Security (TLS)
              Multiple Certificate Status Request Extension", RFC 6961,
              DOI 10.17487/RFC6961, June 2013,
              <http://www.rfc-editor.org/info/rfc6961>.

   [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, <http://www.rfc-editor.org/info/rfc6979>.

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

15.2. Informative References

   [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>.

   [EVSSLGuidelines]
              CA/Browser Forum, "Guidelines For The Issuance And
              Management Of Extended Validation Certificates", 2007,
              <https://cabforum.org/wp-content/uploads/
              EV_Certificate_Guidelines.pdf>.

   [I-D.ietf-trans-gossip]
              Nordberg, L., Gillmor, D., and T. Ritter, "Gossiping in
              CT", draft-ietf-trans-gossip-01 (work in progress),
              October 2015.

   [I-D.ietf-trans-threat-analysis]
              Kent, S., "Attack Model and Threat for Certificate
              Transparency", draft-ietf-trans-threat-analysis-03 (work
              in progress), October 2015.

   [JSON.Metadata]
              The Chromium Projects, "Chromium Log Metadata JSON
              Schema", 2014, <http://www.certificate-transparency.org/
              known-logs/log_list_schema.json>.

   [Public.Suffix.List]
              Mozilla Foundation, "Public Suffix List", 2016, <https://
              publicsuffix.org>.

   [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate
              Transparency", RFC 6962, June 2013.

Appendix A. Supporting v1 and v2 simultaneously

   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:

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

   o Embed the obtained v2 SCTs in the TBSCertificate, as described in
      Section 9.1.2.

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

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

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

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
   Comodo CA, Ltd.

   EMail: rob.stradling@comodo.com