Public Notary Transparency Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 23, 2014

B. Laurie
A. Langley
E. Kasper
Google
R. Stradling
Comodo

February 19, 2014

# Certificate Transparency
# draft-ietf-trans-rfc6962-bis-00

## Abstract

This document describes an experimental 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 certificate 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

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

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

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 23, 2014.

## Copyright Notice

Table of Contents

## 1. Informal Introduction

Certificate transparency aims to mitigate the problem of misissued certificates by providing publicly auditable, append-only, untrusted logs of all issued certificates. The logs are publicly auditable so that it is possible for anyone to verify the correctness of each log and to 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 certificate authorities (CAs).

Each log consists of 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; it is also expected that certificate holders will contribute their own certificate chains. In order to avoid logs being spammed into uselessness, it is required that each chain is rooted in a known CA certificate. When a chain is submitted to a log, a signed timestamp is returned, which can later be used to provide evidence to clients that the chain has been submitted. TLS clients can thus require that all certificates they see have been logged.

Those who are concerned about misissue can monitor the logs, asking them regularly for all new entries, and can thus check whether domains they are responsible for 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. 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 TLS connections to proceed without delay, despite network connectivity issues and the vagaries of firewalls.

The append-only property of each log is technically achieved using Merkle Trees, which can be used to show that any particular version of the log is a superset of any particular previous version. 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 is SHA-256 [FIPS.180-4] (note that this is fixed for this experiment, but it is anticipated that each log would be able to specify a hash algorithm). 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 32-byte 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({}) = SHA-256().

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

MTH({d(0)}) = SHA-256(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]) = SHA-256(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 Audit Paths

A Merkle audit path 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 audit path is combined with the node computed so far. In other words, the audit path 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 audit path matches the true root, then the audit path is proof 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 audit path PATH(m, D[n]) for the (m+1)th input d(m), 0 <= m < n, is defined as follows:

The path 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 path 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)

The subproof for m = n is empty if m is the value for which PROOF was originally requested (meaning that the subtree Merkle Tree Hash MTH(D[0:m]) is known):

SUBPROOF(m, D[m], true) = {}

The subproof for m = n is the Merkle Tree Hash committing inputs D[0:m]; otherwise:

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].

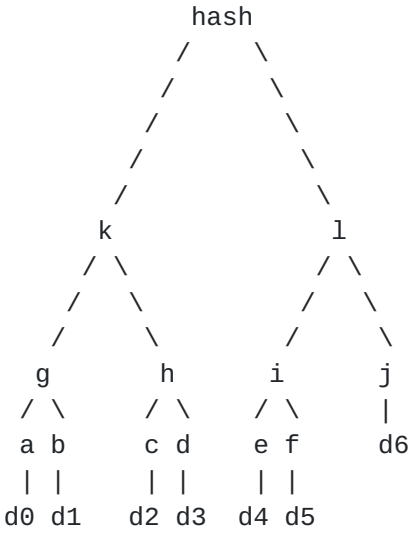
SUBPROOF(m, D[n], b) = SUBPROOF(m - k, D[k:n], false) : MTH(D[0:k])

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:



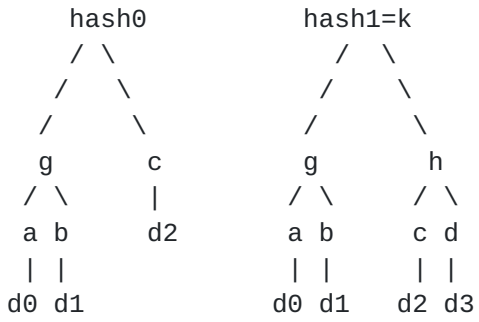

The audit path for d0 is [b, h, l].

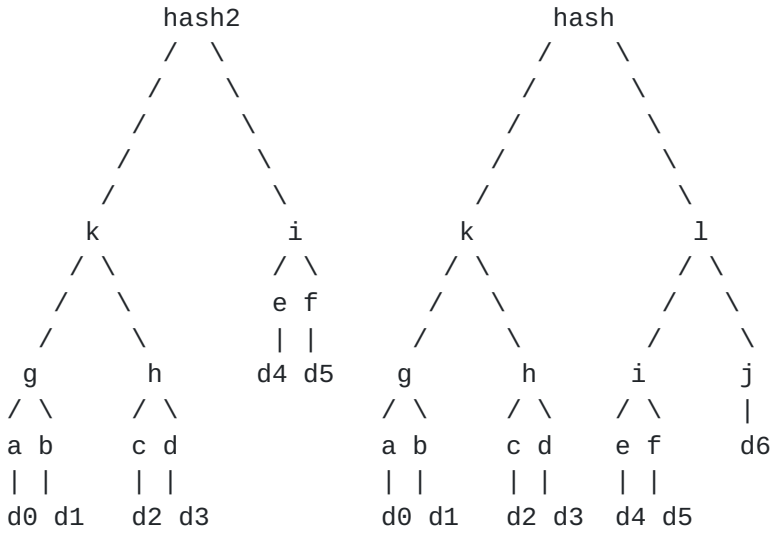
The audit path for d3 is [c, g, l].

The audit path for d4 is [f, j, k].

The audit path for d6 is [i, k].

The same tree, built incrementally in four steps:




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 either elliptic curve signatures using the NIST P-256 curve (Section D.1.2.3 of the Digital Signature Standard [DSS]) or RSA signatures (RSASSA-PKCS1-V1_5 with SHA-256, Section 8.2 of [RFC3447]) using a key of at least 2048 bits.

## 3. Log Format and Operation

Anyone can submit certificates to certificate logs for public auditing; however, since certificates will not be accepted by TLS clients unless logged, it is expected that certificate owners or their CAs will usually submit them. A log is a single, ever-growing, append-only Merkle Tree of such certificates.

When a valid certificate is submitted to a log, the log MUST immediately return a Signed Certificate Timestamp (SCT). The SCT is the log's promise to incorporate the certificate in the Merkle Tree within a fixed amount of time known as the Maximum Merge Delay (MMD). If the log has previously seen the certificate, it MAY return the same SCT as it returned before. TLS servers MUST present an SCT from one or more logs to the TLS client together with the certificate. TLS clients MUST reject certificates that do not have a valid SCT for the end-entity certificate.

Periodically, each log appends all its new entries to the Merkle Tree and signs the root of the tree. The log MUST incorporate a certificate in its Merkle Tree within the Maximum Merge Delay period after the issuance of the SCT. When encountering an SCT, an Auditor can verify that the certificate was added to the Merkle Tree within that timeframe.

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

### 3.1. Log Entries

Anyone can submit a certificate to any log. In order to enable attribution of each logged certificate to its issuer, the log SHALL publish a list of acceptable root certificates (this list might usefully be the union of root certificates trusted by major browser vendors). Each submitted certificate MUST be accompanied by all additional certificates required to verify the certificate chain up to an accepted root certificate. The root certificate itself MAY be omitted from the chain submitted to the log server.

Alternatively, (root as well as intermediate) certificate authorities may submit a certificate to logs prior to issuance in order to incorporate the SCT in the issued certificate. To do so, the CA submits a Precertificate that the log can use to create an entry that will be valid against the issued certificate. The Precertificate is an X.509v3 certificate for simplicity, but, since it isn't used for anything but logging, could equally be some other data structure. The Precertificate is constructed from the certificate to be issued by adding a special critical poison extension (OID

1.3.6.1.4.1.11129.2.4.3, whose extnValue OCTET STRING contains ASN.1 NULL data (0x05 0x00)) to the end-entity TBSCertificate, minus the SCT extension, which is obviously unknown until after the Precertificate has been submitted to the log. The poison extension is to ensure that the Precertificate cannot be validated by a standard X.509v3 client. The resulting TBSCertificate [RFC5280] is then signed with either

- a special-purpose (CA:true, Extended Key Usage: Certificate Transparency, OID 1.3.6.1.4.1.11129.2.4.4) Precertificate Signing Certificate. The Precertificate Signing Certificate MUST be directly certified by the (root or intermediate) CA certificate that will ultimately sign the end-entity TBSCertificate yielding the end-entity certificate (note that the log may relax standard validation rules to allow this, so long as the issued certificate will be valid),

- or, the CA certificate that will sign the final certificate.

As above, the Precertificate submission MUST be accompanied by the Precertificate Signing Certificate, if used, and all additional certificates required to verify the chain up to an accepted root certificate. The signature on the TBSCertificate indicates the certificate authority's intent to issue a certificate. This intent is considered binding (i.e., misissuance of the Precertificate is considered equal to misissuance of the final certificate). Each log verifies the Precertificate signature chain and issues a Signed Certificate Timestamp on the corresponding TBSCertificate.

Logs MUST verify that the submitted end-entity certificate or Precertificate has a valid signature chain leading back to a trusted root CA certificate, using the chain of intermediate CA certificates provided by the submitter. Logs MAY accept certificates that have expired, are not yet valid, have been revoked, or are otherwise not fully valid according to X.509 verification rules in order to accommodate quirks of CA certificate-issuing software. However, logs MUST refuse to publish certificates without a valid chain to a known root CA. If a certificate is accepted and an SCT issued, the accepting log MUST store the entire chain used for verification, including the certificate or Precertificate itself and including the root certificate used to verify the chain (even if it was omitted from the submission), and MUST present this chain for auditing upon request. This chain is required to prevent a CA from avoiding blame by logging a partial or empty chain. (Note: This effectively excludes self-signed and DANE-based certificates until some mechanism to control spam for those certificates is found. The authors welcome suggestions.)

Each certificate entry in a log MUST include the following components:

```
enum { x509_entry(0), precert_entry(1), (65535) } LogEntryType;

struct {
    LogEntryType entry_type;
    select (entry_type) {
        case x509_entry: X509ChainEntry;
        case precert_entry: PrecertChainEntry;
    } entry;
} LogEntry;

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

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

struct {
    ASN.1Cert pre_certificate;
    ASN.1Cert precertificate_chain<0..2^24-1>;
} PrecertChainEntry;
```

Logs MAY limit the length of chain they will accept.

"entry_type" is the type of this entry. Future revisions of this protocol version may add new LogEntryType values. Section 4 explains how clients should handle unknown entry types.

"leaf_certificate" is the end-entity certificate submitted for auditing.

"certificate_chain" is a chain of additional certificates required to verify the end-entity certificate. The first certificate MUST certify the end-entity certificate. Each following certificate MUST directly certify the one preceding it. The final certificate MUST be a root certificate accepted by the log.

"pre_certificate" is the Precertificate submitted for auditing.

"precertificate_chain" is a chain of additional certificates required to verify the Precertificate submission. The first certificate MAY be a valid Precertificate Signing Certificate and MUST certify the first certificate. Each following certificate MUST directly certify the one preceding it. The final certificate MUST be a root certificate accepted by the log.

### 3.2. Structure of the Signed Certificate Timestamp

```
enum { certificate_timestamp(0), tree_hash(1), (255) }
  SignatureType;

enum { v1(0), (255) }
  Version;

  struct {
      opaque key_id[32];
  } LogID;

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

  struct {
    opaque issuer_key_hash[32];
    TBSCertificate tbs_certificate;
  } PreCert;

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

"key_id" is the SHA-256 hash of the log's public key, calculated over the DER encoding of the key represented as SubjectPublicKeyInfo.

"issuer_key_hash" is the SHA-256 hash of the certificate issuer's public key, calculated over the DER encoding of the key represented as SubjectPublicKeyInfo. This is needed to bind the issuer to the final certificate.

"tbs_certificate" is the DER-encoded TBSCertificate (see [RFC5280]) component of the Precertificate -- that is, without the signature and the poison extension. If the Precertificate is not signed with the CA certificate that will issue the final certificate, then the TBSCertificate also has its issuer changed to that of the CA that will issue the final certificate. Note that it is also possible to reconstruct this TBSCertificate from the final certificate by extracting the TBSCertificate from it and deleting the SCT extension. Also note that since the TBSCertificate contains an AlgorithmIdentifier that must match both the Precertificate signature algorithm and final certificate signature algorithm, they must be signed with the same algorithm and parameters. If the Precertificate is issued using a Precertificate Signing Certificate and an Authority Key Identifier extension is present in the TBSCertificate, the corresponding extension must also be present in the Precertificate Signing Certificate -- in this case, the TBSCertificate also has its Authority Key Identifier changed to match the final issuer.

```
struct {
    Version sct_version;
    LogID id;
    uint64 timestamp;
    CtExtensions extensions;
    digitally-signed struct {
        Version sct_version;
        SignatureType signature_type = certificate_timestamp;
        uint64 timestamp;
        LogEntryType entry_type;
        select(entry_type) {
            case x509_entry: ASN.1Cert;
            case precert_entry: PreCert;
        } signed_entry;
       CtExtensions extensions;
    };
} SignedCertificateTimestamp;
```

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

"sct_version" is the version of the protocol to which the SCT conforms. This version is v1.

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

"entry_type" may be implicit from the context in which the SCT is presented.

"signed_entry" is the "leaf_certificate" (in the case of an X509ChainEntry) or is the PreCert (in the case of a PrecertChainEntry), as described above.

"extensions" are future extensions to this protocol version (v1). Currently, no extensions are specified.

### 3.3. Including the Signed Certificate Timestamp in the TLS Handshake

The SCT data corresponding to the end-entity certificate from at least one log must be included in the TLS handshake, either by using an X509v3 certificate extension as described below, by using a TLS extension (Section 7.4.1.4 of [RFC5246]) with type "signed_certificate_timestamp", or by using Online Certificate Status Protocol (OCSP) Stapling (also known as the "Certificate Status Request" TLS extension; see [RFC6066]), where the OCSP response includes an extension with OID 1.3.6.1.4.1.11129.2.4.5 (see [RFC2560]) and body:

```
SignedCertificateTimestampList ::= OCTET STRING
```

in the singleExtensions component of the SingleResponse pertaining to the end-entity certificate.

At least one SCT MUST be included. Server operators MAY include more than one SCT.

Similarly, a certificate authority MAY submit a Precertificate to more than one log, and all obtained SCTs can be directly embedded in the final certificate, by encoding the SignedCertificateTimestampList structure as an ASN.1 OCTET STRING and inserting the resulting data in the TBSCertificate as an X.509v3 certificate extension (OID 1.3.6.1.4.1.11129.2.4.2). Upon receiving the certificate, clients can reconstruct the original TBSCertificate to verify the SCT signature.

The contents of the ASN.1 OCTET STRING embedded in an OCSP extension or X509v3 certificate extension are as follows:

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

struct {
    SerializedSCT sct_list <1..2^16-1>;
} SignedCertificateTimestampList;
```

Here, "SerializedSCT" is an opaque byte string that contains the serialized TLS structure. This encoding ensures that TLS clients can decode each SCT individually (i.e., if there is a version upgrade, out-of-date clients can still parse old SCTs while skipping over new SCTs whose versions they don't understand).

Likewise, SCTs can be embedded in a TLS extension. See below for details.

TLS clients MUST implement all three mechanisms. Servers MUST implement at least one of the three mechanisms. Note that existing TLS servers can generally use the certificate extension mechanism without modification.

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

### 3.3.1. TLS Extension

The SCT can be sent during the TLS handshake using a TLS extension with type "signed_certificate_timestamp".

Clients that support the extension SHOULD send a ClientHello extension with the appropriate type and empty "extension_data".

Servers MUST only send SCTs to clients who have indicated support for the extension in the ClientHello, in which case the SCTs are sent by setting the "extension_data" to a "SignedCertificateTimestampList".

Session resumption uses the original session information: clients SHOULD include the extension type in the ClientHello, but if the session is resumed, the server is not expected to process it or include the extension in the ServerHello.

## 3.4. Merkle Tree

The hashing algorithm for the Merkle Tree Hash is SHA-256.

Structure of the Merkle Tree input:

```
    enum { timestamped_entry(0), (255) }
      MerkleLeafType;

    struct {
        uint64 timestamp;
        LogEntryType entry_type;
        select(entry_type) {
            case x509_entry: ASN.1Cert;
            case precert_entry: PreCert;
        } signed_entry;
        CtExtensions extensions;
    } TimestampedEntry;

    struct {
        Version version;
        MerkleLeafType leaf_type;
        select (leaf_type) {
            case timestamped_entry: TimestampedEntry;
        }
    } MerkleTreeLeaf;
```

Here, "version" is the version of the protocol to which the MerkleTreeLeaf corresponds. This version is v1.

"leaf_type" is the type of the leaf input. Currently, only "timestamped_entry" (corresponding to an SCT) is defined. Future revisions of this protocol version may add new MerkleLeafType types. Section 4 explains how clients should handle unknown leaf types.

"timestamp" is the timestamp of the corresponding SCT issued for this certificate.

"signed_entry" is the "signed_entry" of the corresponding SCT.

"extensions" are "extensions" of the corresponding SCT.

The leaves of the Merkle Tree are the leaf hashes of the corresponding "MerkleTreeLeaf" structures.

### 3.5. Signed Tree Head

Every time a log appends new entries to the tree, the log SHOULD sign the corresponding tree hash and tree information (see the corresponding Signed Tree Head client message in Section 4.3). The signature for that data is structured as follows:

```
digitally-signed struct {
    Version version;
    SignatureType signature_type = tree_hash;
    uint64 timestamp;
    uint64 tree_size;
    opaque sha256_root_hash[32];
} TreeHeadSignature;
```

"version" is the version of the protocol to which the TreeHeadSignature conforms. This version is v1.

"timestamp" is the current time. The 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" equals the number of entries in the new tree.

"sha256_root_hash" is the root of the Merkle Hash Tree.

Each log MUST produce on demand a Signed Tree Head that is no older than the Maximum Merge Delay. 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.

## 4. 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 can include a path as well as a server name and a port.

In general, where needed, the "version" is v1 and the "id" is the log id for the log server queried.

Any errors will be returned as HTTP 4xx or 5xx responses, with human-readable error messages.

### 4.1. Add Chain to Log

POST https://<log server>/ct/v1/add-chain

Inputs:

chain: An array of base64-encoded certificates. The first element is the end-entity certificate; the second chains to the first and so on to the last, which is either the root certificate or a certificate that chains to a known root certificate.

Outputs:

sct_version: The version of the SignedCertificateTimestamp structure, in decimal. A compliant v1 implementation MUST NOT expect this to be 0 (i.e., v1).

id: The log ID, base64 encoded. Since log clients who request an SCT for inclusion in TLS handshakes are not required to verify it, we do not assume they know the ID of the log.

timestamp: The SCT timestamp, in decimal.

extensions: An opaque type for future expansion. It is likely that not all participants will need to understand data in this field. Logs should set this to the empty string. Clients should decode the base64-encoded data and include it in the SCT.

signature: The SCT signature, base64 encoded.

If the "sct_version" is not v1, then a v1 client may be unable to verify the signature. It MUST NOT construe this as an error. (Note: Log clients don't need to be able to verify this structure; only TLS clients do. If we were to serve the structure as a binary blob, then we could completely change it without requiring an upgrade to v1 clients.)

### 4.2. Add PreCertChain to Log

POST https://<log server>/ct/v1/add-pre-chain

Inputs:

chain: An array of base64-encoded Precertificates. The first element is the end-entity certificate; the second chains to the first and so on to the last, which is either the root certificate or a certificate that chains to a known root certificate.

Outputs are the same as in Section 4.1.

### 4.3. Retrieve Latest Signed Tree Head

GET https://<log server>/ct/v1/get-sth

No inputs.

Outputs:

tree_size: The size of the tree, in entries, in decimal.

timestamp: The timestamp, in decimal.

sha256_root_hash: The Merkle Tree Hash of the tree, in base64.

tree_head_signature: A TreeHeadSignature for the above data.

### 4.4. Retrieve Merkle Consistency Proof between Two Signed Tree Heads

GET https://<log server>/ct/v1/get-sth-consistency

Inputs:

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

second: The tree_size of the second tree, in decimal.

Both tree sizes must be from existing v1 STHs (Signed Tree Heads).

Outputs:

consistency: An array of Merkle Tree nodes, base64 encoded.

Note that no signature is required on this data, as it is used to verify an STH, which is signed.

### 4.5. Retrieve Merkle Audit Proof from Log by Leaf Hash

GET https://<log server>/ct/v1/get-proof-by-hash

Inputs:

hash: A base64-encoded v1 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 3.4. The "tree_size" must designate an existing v1 STH.

Outputs:

leaf_index: The 0-based index of the entry corresponding to the "hash" parameter.

audit_path: An array of base64-encoded Merkle Tree nodes proving the inclusion of the chosen certificate.

### 4.6. Retrieve Entries from Log

GET https://<log server>/ct/v1/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 MerkleTreeLeaf structure.

extra_data: The base64-encoded unsigned data pertaining to the log entry. In the case of an X509ChainEntry, this is the "certificate_chain". In the case of a PrecertChainEntry, this is the whole "PrecertChainEntry".

Note that this message is not signed -- the retrieved data can be verified by constructing the Merkle Tree Hash corresponding to a retrieved STH. All leaves MUST be v1. However, a compliant v1 client MUST NOT construe an unrecognized MerkleLeafType or LogEntryType value 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 4.3.

Logs MAY 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. 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".

### 4.7. Retrieve Accepted Root Certificates

GET https://<log server>/ct/v1/get-roots

No inputs.

Outputs:

certificates: An array of base64-encoded root certificates that are acceptable to the log.

### 4.8. Retrieve Entry+Merkle Audit Proof from Log

GET https://<log server>/ct/v1/get-entry-and-proof

Inputs:

leaf_index: The index of the desired entry.

tree_size: The tree_size of the tree for which the proof is desired.

The tree size must designate an existing STH.

Outputs:

leaf_input: The base64-encoded MerkleTreeLeaf structure.

extra_data: The base64-encoded unsigned data, same as in Section 4.6.

audit_path: An array of base64-encoded Merkle Tree nodes proving the inclusion of the chosen certificate.

This API is probably only useful for debugging.

## 5. Clients

There are various different functions clients of logs might perform. We describe here some typical clients and how they could 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 should gossip with each other, exchanging STHs at least; this is all that is required to ensure that they all have a consistent view. The exact mechanism for gossip will be described in a separate document, but it is expected there will be a variety.

### 5.1. Submitters

Submitters submit certificates or Precertificates to the log as described above. They may go on to use the returned SCT to construct a certificate or use it directly in a TLS handshake.

### 5.2. TLS Client

TLS clients are not directly clients of the log, but they receive SCTs alongside or in server certificates. In addition to normal validation of the certificate and its chain, they should validate the SCT by computing the signature input from the SCT data as well as the certificate and verifying the signature, using the corresponding log's public key. Note that this document does not describe how clients obtain the logs' public keys.

TLS clients MUST reject SCTs whose timestamp is in the future.

### 5.3. Monitor

Monitors watch logs and check that they behave correctly. They also watch for certificates of interest.

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 4.3).

2. Verify the STH signature.

3. Fetch all the entries in the tree corresponding to the STH (Section 4.6).

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 4.3). Repeat until the STH changes.

6. Verify the STH signature.

7. Fetch all the new entries in the tree corresponding to the STH (Section 4.6). 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 4.4).

    2. Verify the consistency proof.

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

9. Go to Step 5.

### 5.4. Auditor

Auditors take partial information about a log as input and verify that this information is consistent with other partial information they have. An auditor might be an integral component of a TLS client; it might be a standalone service; or it might be a secondary function of a monitor.

Any pair of STHs from the same log can be verified by requesting a consistency proof (Section 4.4).

A certificate accompanied by an SCT can be verified against any STH dated after the SCT timestamp + the Maximum Merge Delay by requesting a Merkle audit proof (Section 4.5).

Auditors can fetch STHs from time to time of their own accord, of course (Section 4.3).

## 6. IANA Considerations

IANA has allocated an RFC 5246 ExtensionType value (18) for the SCT TLS extension. The extension name is "signed_certificate_timestamp".

## 7. 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 the certificate has been published in a log. From this, the client knows that the subject of the certificate has had some time to notice the misissue and take some action, such as asking a CA to revoke a misissued certificate. A signed timestamp is not a guarantee that the certificate is not misissued, since the subject of the certificate 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.

### 7.1. Misissued Certificates

Misissued certificates that have not been publicly logged, and thus do not have a valid SCT, will be rejected by TLS clients. 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.

### 7.2. 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.

### 7.3. Misbehaving Logs

A log can misbehave in two ways: (1) by failing to incorporate a certificate with an SCT in the Merkle Tree within the MMD and (2) by violating its append-only property by presenting two different, conflicting views of the Merkle Tree at different times and/or to different parties. Both forms of violation will be promptly and publicly detectable.

Violation of the MMD contract is detected by log clients requesting a Merkle audit proof 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. Clients can instead request the proof from a trusted auditor (since anyone can compute the audit proofs from the log) or request Merkle proofs for a batch of certificates around the SCT timestamp.

Violation of the append-only property is detected by global gossiping, i.e., everyone auditing logs comparing their versions of the latest Signed Tree Heads. As soon as two conflicting Signed Tree Heads for the same log are detected, this is cryptographic proof of that log's misbehavior.

## 8. Efficiency Considerations

The Merkle Tree design serves the purpose of keeping communication overhead low.

Auditing logs for integrity does not require third parties to maintain a copy of each entire log. The Signed Tree Heads can be updated as new entries become available, without recomputing entire trees. Third-party auditors need only fetch the Merkle consistency proofs against a log's existing STH to efficiently verify the append-only property of updates to their Merkle Trees, without auditing the entire tree.

## 9. Future Changes

This section lists things we might address in a Standards Track version of this document.

- Rather than forcing a log operator to create a new log in order to change the log signing key, we may allow some key roll mechanism.
- We may add hash and signing algorithm agility.
- We may describe some gossip protocols.

## 10. Acknowledgements

The authors would like to thank Erwann Abelea, Robin Alden, Al Cutter, Francis Dupont, Stephen Farrell, Brad Hill, Jeff Hodges, Paul Hoffman, Jeffrey Hutzelman, SM, Alexey Melnikov, Chris Palmer, Trevor Perrin, Ryan Sleevi, Rob Stradling, and Carl Wallace for their valuable contributions.

## 11. References

### 11.1. Normative Reference

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

### 11.2. Informative References

[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>.

[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>.

[RFC2560] Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 2560, 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.

[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.

## 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

Rob Stradling
Comodo CA, Ltd.

EMail: rob.stradling@comodo.com