Public Notary Transparency Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 10, 2015

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

March 9, 2015

Certificate Transparency
draft-ietf-trans-rfc6962-bis-06

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

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 September 10, 2015.

Table of Contents

1. 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 certification 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, however certificate holders can also contribute their own certificate chains, as can third parties. In order to avoid logs being rendered useless by submitting large numbers of spurious certificates, it is required that each chain is rooted in a CA certificate accepted by the log. When a chain is submitted to 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 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, 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 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 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. We have established a registry of acceptable algorithms, see Section 7.2. The hashing algorithm in use is referred to as HASH throughout this document. 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({}) = 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)

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].

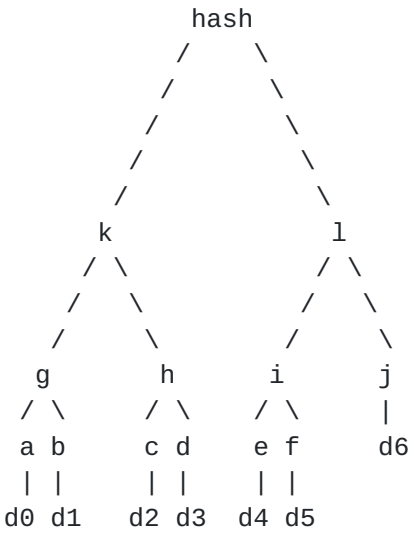
SUBPROOF(m, D[n], b) = SUBPROOF(m - k, D[k:n], false) : MTH(D[0:k])

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:



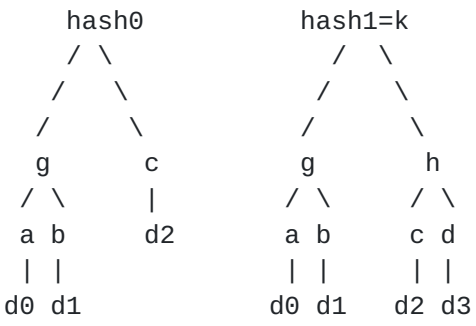
The inclusion proof for d0 is [b, h, l].

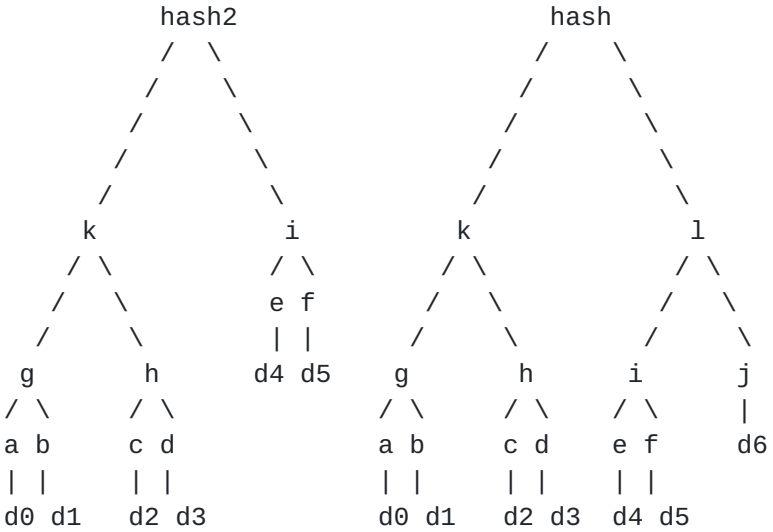
The inclusion proof for d3 is [c, g, l].

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:





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 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 (note that if a certificate was previously logged as a precertificate, then the precertificate's SCT would not be appropriate, instead a fresh SCT of type x509_entry should be generated). TLS servers MUST present an SCT from one or more logs to the TLS client together with the certificate. A certificate not accompanied by an SCT (either for the end-entity certificate or for a name-constrained intermediate the end-entity certificate chains to) MUST NOT be considered compliant by TLS clients.

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

In order to enable attribution of each logged certificate to its issuer, 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. The log SHALL allow retrieval of a list of accepted root certificates (this list

might usefully be the union of root certificates trusted by major browser vendors).

Alternatively, (root as well as intermediate) certification authorities may preannounce a certificate to logs prior to issuance in order to incorporate the SCT in the issued certificate. To do this, the CA submits a precertificate that the log can use to create an entry that will be valid against the issued certificate. A precertificate is a CMS [RFC5652] "signed-data" object that contains a TBSCertificate [RFC5280] in its "SignedData.encapContentInfo.eContent" field, identified by the OID <TBD> in the "SignedData.encapContentInfo.eContentType" field. This TBSCertificate MAY redact certain domain name labels that will be present in the issued certificate (see Section 3.2.2) and MUST NOT contain any SCTs, but it will be otherwise identical to the TBSCertificate in the issued certificate. "SignedData.signerInfos" MUST contain a signature from the same (root or intermediate) CA that will ultimately issue the certificate. This signature indicates the certification authority'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). As above, the precertificate submission MUST be accompanied by all the additional certificates required to verify the chain up to an accepted root certificate. This does not involve using the "SignedData.certificates" field, so that field SHOULD be omitted.

Logs MUST verify that the submitted certificate or precertificate has a valid signature chain to an accepted root certificate, using the chain of intermediate CA certificates provided by the submitter. Logs MUST accept certificates that are fully valid according to X.509 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 X.509 verification rules in order to accommodate quirks of CA certificate-issuing software. However, logs MUST reject certificates without a valid signature chain to an accepted root certificate. 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 limit the submission of spurious certificates is found. The authors welcome suggestions.)

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

```
enum { x509_entry(0), precert_entry_V2(3), (65535) } LogEntryType;

struct {
    LogEntryType entry_type;
    select (entry_type) {
        case x509_entry: X509ChainEntry;
        case precert_entry_V2: PrecertChainEntryV2;
    } entry;
} LogEntry;

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<0..2^24-1>;
} PrecertChainEntryV2;
```

Logs SHOULD limit the length of chain they will accept.

"entry_type" is the type of this entry. Future revisions of this protocol 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 either be, or be issued by, 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 MUST certify the precertificate. Each following certificate MUST directly

certify the one preceding it. The final certificate MUST be a root certificate accepted by the log.

3.2. 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.

3.2.1. 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]), this approach would not work for a DNS-ID such as "top.secret.example.com". Also, wildcard certificates are prohibited in some cases, such as Extended Validation Certificates [EVSSLGuidelines].

3.2.2. Redacting Domain Name Labels in Precertificates

When creating a precertificate, the CA MAY substitute one or more labels in each DNS-ID 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 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 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 precertificate could contain "*.?.?.example.com" instead, but not "?.?.?.example.com" instead.

When a precertificate contains one or more redacted labels, a non-critical extension (OID 1.3.6.1.4.1.11129.2.4.6, whose extnValue OCTET STRING contains an ASN.1 SEQUENCE OF INTEGERs) MUST be added to the corresponding certificate: the first INTEGER indicates the total number of redacted labels and wildcard "*" labels in the

precertificate's first DNS-ID; the second INTEGER does the same for the precertificate's second DNS-ID; etc. There MUST NOT be more INTEGERs than there are DNS-IDs. If there are fewer INTEGERs than there are DNS-IDs, the shortfall is made up by implicitly repeating the last INTEGER. Each INTEGER MUST have a value of zero or more. The purpose of this extension is to enable TLS clients to accurately reconstruct the TBSCertificate component of the precertificate from the certificate without having to perform any guesswork.

When a precertificate contains that extension and contains a CN-ID [RFC6125], the CN-ID MUST match the first DNS-ID and have the same labels redacted. TLS clients will use the first entry in the SEQUENCE OF INTEGERs to reconstruct both the first DNS-ID and the CN-ID.

3.2.3. 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:

- there MUST be a non-critical extension (OID 1.3.6.1.4.1.11129.2.4.7, 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.

- permittedSubtrees MUST specify one or more dNSNames.

- 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
          }
        }
      }
    }
  }
```

3.3. Structure of the Signed Certificate Timestamp

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

enum { v1(0), v2(1), (255) }
  Version;

  struct {
      opaque key_id[32];
  } LogID;

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

  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.

"tbs_certificate" is the DER-encoded TBSCertificate component of the precertificate. Note that it is also possible to reconstruct this TBSCertificate from the issued certificate by extracting the TBSCertificate from it, redacting the domain name labels indicated by the redacted labels extension, and deleting the SCT list extension and redacted labels extension.

```
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_V2: TBSCertificate;
        } 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 v2.

"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 TBSCertificate (in the case of a PrecertChainEntryV2), as described above.

"extensions" are future extensions to SignedCertificateTimestamp v2. Currently, no extensions are specified.

3.4. Including the Signed Certificate Timestamp in the TLS Handshake

The SCT data corresponding to at least one certificate in the chain 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 a non-critical 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 certification authority MAY submit a precertificate to more than one log, and all obtained SCTs can be directly embedded in the issued certificate, by encoding the SignedCertificateTimestampList structure as an ASN.1 OCTET STRING and inserting the resulting data in the TBSCertificate as a non-critical 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 SCT 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, has had a key compromise or is not known to the client).

The three mechanisms are provided because they have different tradeoffs. Embedding the SCTs in the certificate allows the use of unmodified TLS servers, but, because they cannot be changed without

re-issuing the certificate, increases the risk that the certificate will be refused if the SCTs become invalid. OCSP Stapling is already widely (but not universally) implemented, and provides a mechanism by which TLS servers that already support it can serve SCTs that are generated on the fly. Finally, the TLS extension permits TLS servers to participate in CT without the cooperation of CAs, unlike the other two mechanisms. It also allows SCTs to be updated on the fly.

3.4.1. 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 in this TLS extension 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.

3.5. Merkle Tree

The hashing algorithm for the Merkle Tree Hash is specified in the log's metadata.

Structure of the Merkle Tree input:

```
enum { v1(0), v2(1), (255) }
  LeafVersion;

struct {
    uint64 timestamp;
    LogEntryType entry_type;
    select(entry_type) {
        case x509_entry: ASN.1Cert;
        case precert_entry_V2: TBSCertificate;
    } signed_entry;
    CtExtensions extensions;
} TimestampedEntry;

struct {
    LeafVersion version;
    TimestampedEntry timestamped_entry;
} MerkleTreeLeaf;
```

Here, "version" is the version of the MerkleTreeLeaf structure. This version is v2. Note that MerkleTreeLeaf v1 [RFC6962] had another layer of indirection which is removed in v2.

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

"entry_type" is the type of entry stored in "signed_entry". New "LogEntryType" values may be added to "signed_entry" without increasing the "MerkleTreeLeaf" version. Section 4 explains how clients should handle unknown entry types.

"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.

3.6. 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:

```
enum { v1(0), (255) } TreeHeadVersion;

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

"version" is the version of the TreeHeadSignature structure. 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 MAY 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.

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/v1/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.

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.

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.

Error codes:

unknown root The root of the chain is not one accepted by the log.

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).

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. This is to avoid forcing an upgrade of compliant v1 clients that do not use the returned SCTs.

If a log detects bad encoding in a chain that otherwise verifies correctly (e.g. some software will accept BER instead of DER encodings in certificates, or incorrect character encodings, even though these are technically incorrect) then the log MAY still log the certificate but SHOULD NOT return an SCT. It should instead return the "bad certificate" error. Logging the certificate is useful, because monitors can then detect these encoding errors, which may be accepted by some TLS clients.

Note that not all certificate handling software is capable of detecting all encoding errors.

4.2. Add PreCertChain to Log

POST https://<log server>/ct/v1/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 chains to the first and so on to the last, which is either the root certificate or a certificate that chains to an accepted root certificate.

Outputs and errors 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/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 v1 STHs (Signed Tree Heads). 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: An array of Merkle Tree nodes, base64 encoded.

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.

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

4.5. Retrieve Merkle Inclusion Proof from Log by Leaf Hash

GET https://<log server>/ct/v2/get-proof-by-hash

Inputs:

hash: A base64-encoded 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 3.5. The "tree_size" must designate an existing v1 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 "leaf_index" and "audit_path" are returned.

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.

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.

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.

4.6. 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 v1 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 3.5. The "tree_size" must designate an existing v1 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 4.4).

index of requested hash < latest STH Return "leaf_index" and "audit_path".

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:

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.

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.

consistency: An array of base64-encoded Merkle Tree nodes proving the consistency of the requested STH and the returned STH.

Errors are the same as in Section 4.5.

4.7. 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 PrecertChainEntryV2, this is the whole "PrecertChainEntryV2".

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 or v2. However, a compliant v1 client MUST NOT construe an unrecognized 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".

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.

4.8. 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.

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.

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 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.

5.1. 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 4.

Hash Algorithm The hash algorithm used for the Merkle Tree (see Section 7.2).

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

Public Key The public key used for signing.

Maximum Merge Delay The MMD the log has committed to.

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.

5.2. Submitters

Submitters submit certificates or precertificates to the log as described above. When a Submitter intends to use the returned SCT directly in a TLS handshake or to construct a certificate, they SHOULD validate the SCT as described in Section 5.3 if they understand its format.

5.3. TLS Client

TLS clients receive SCTs alongside or in certificates, either for the server certificate itself or for intermediate CA precertificates. In addition to normal validation of the certificate and its chain, TLS clients 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.

A TLS client MAY audit the corresponding log by requesting, and verifying, a Merkle audit proof for said certificate. 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 4.3), then verify it by requesting a consistency proof (Section 4.4).

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

5.4. Monitor

Monitors watch logs and check that they behave correctly. Monitors may additionally watch for certificates of interest. 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 4.3).

2. Verify the STH signature.

3. Fetch all the entries in the tree corresponding to the STH (Section 4.7).

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 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 4.7). 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 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.

5.5. Auditing

Auditing is taking partial information about a log as input and verifying that this information is consistent with other partial information held. All clients described above may perform auditing as an additional function. The action taken by the client if audit fails is not specified, but note that in general if audit fails, the client is in possession of signed proof of the log's misbehavior.

A monitor (Section 5.4) 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 5.3) 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 4.5). 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 3.2.3]).

6. Algorithm Agility

It is not possible for a log to change any of its algorithms part way through its lifetime. If it should become necessary to deprecate an algorithm used by a live log, then the log should be frozen as specified in Section 5.1 and a new log should be started. If necessary, the new log can contain existing entries from the frozen log, which monitors can verify are an exact match.

7. IANA Considerations

7.1. TLS Extension Type

IANA has allocated an RFC 5246 ExtensionType value (18) for the SCT TLS extension. The extension name is "signed_certificate_timestamp". IANA should update this extension type to point at this document.

7.2. Hash Algorithms

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

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

8. 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 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, 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 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.

8.1. 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.

8.2. 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.

8.3. Redaction of Public Domain Name Labels

CAs SHOULD NOT redact domain name labels in precertificates such that the entirety of the domain space below the unredacted part of the domain name is not owned or controlled by a single entity (e.g. "?.com" and "?.co.uk" would both be problematic). Logs MUST NOT reject any precertificate that is overly redacted but which is otherwise considered compliant. It is expected that monitors will treat overly redacted precertificates as potentially misissued. TLS clients MAY reject a certificate whose corresponding precertificate would be overly redacted, perhaps using the same mechanism for determining whether a wildcard in a domain name of a certificate is too broad.

8.4. 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 can be detected by clients comparing their instances of the 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. There are various ways this could be done, for example via gossip (see http://trac.tools.ietf.org/id/draft-linus-trans-gossip-00.txt) or peer-to-peer communications or by sending STHs to monitors (who could then directly check against their own copy of the relevant log).

8.5. Multiple SCTs

TLS servers may wish to offer multiple SCTs, each from a different log.

- 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.

- 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 the SCT is 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.

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

9. 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.

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 and Carl Wallace for their valuable contributions.

11. References

11.1. 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.

[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.

[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, September 2009.

[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.

11.2. 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>.

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

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

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