Network Working Group
Internet-Draft
Expires: April 21, 2013

B. Laurie
A. Langley
E. Kasper

October 18, 2012

Certificate Transparency
draft-laurie-pki-sunlight-02

Abstract

The aim of Certificate Transparency is to have every public end-entity and intermediate TLS certificate issued by a known Certificate Authority recorded in one or more certificate logs. In order to detect mis-issuance of certificates, all logs are publicly auditable. In particular, domain owners or their agents will be able to monitor logs for certificates issued on their own domain.

To protect clients from unlogged mis-issued certificates, logs sign all recorded certificates, and clients can choose not to trust certificates that are not accompanied by an appropriate log signature. For privacy and performance reasons log signatures are embedded in the TLS handshake via the TLS authorization extension [RFC5878], or in the certificate itself via an X.509v3 certificate extension [RFC5280].

To ensure a globally consistent view of the log, logs also provide a global signature over the entire log. Any inconsistency of logs can be detected through cross-checks on the global signature. Consistency between any pair of global signatures, corresponding to snapshots of the log at different times, can be efficiently shown.

Logs are only expected to certify that they have seen a certificate, and thus we do not specify any revocation mechanism for log signatures in this document. Logs are append-only, and log signatures will be valid indefinitely.

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 April 21, 2013.

Copyright Notice

Table of Contents

1. Cryptographic components

1.1. Merkle Hash Trees

Logs use a binary Merkle hash tree for efficient auditing. The hashing algorithm is SHA-256. 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 root 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 is:

MTH({d(0)}) = SHA-256(0 || d(0)).

For n > 1, let k be the largest power of two smaller than n. The Merkle Tree Hash of an n-element list D[n] is then defined recursively as

MTH(D[n]) = SHA-256(1 || MTH(D[0:k]) || MTH(D[k:n])),

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

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. [This Merkle tree is essentially the same as the history tree [1] proposal, except our definition omits dummy leaves.]

1.1.1. 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.

1.1.2. 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 root 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 root hash MTH(D[0:m]) is known):

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

The subproof for m = n is the root hash committing inputs D[0:m] otherwise:

SUBPROOF(m, D[m], false) = {MTH(D[m])}

For m &lt 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].

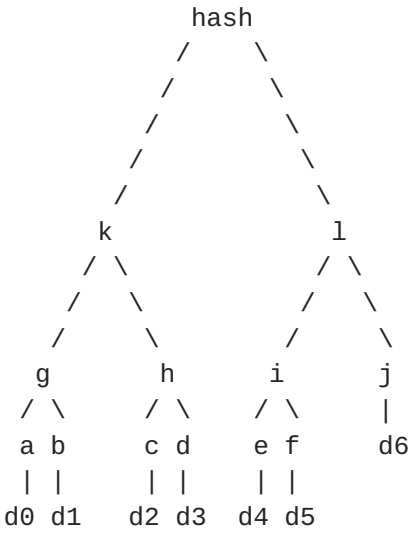
SUBPROOF(m, D[n], b) = SUBPROOF(m - k, D[k:n], false) : MTH(D[0:k]).

Here : is 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.

1.1.3. Example

The binary Merkle tree with 7 leaves:



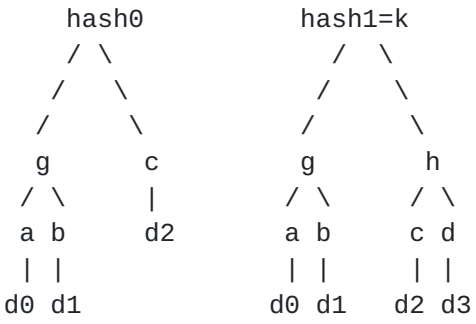
The audit path for d0 is [b, h, l].

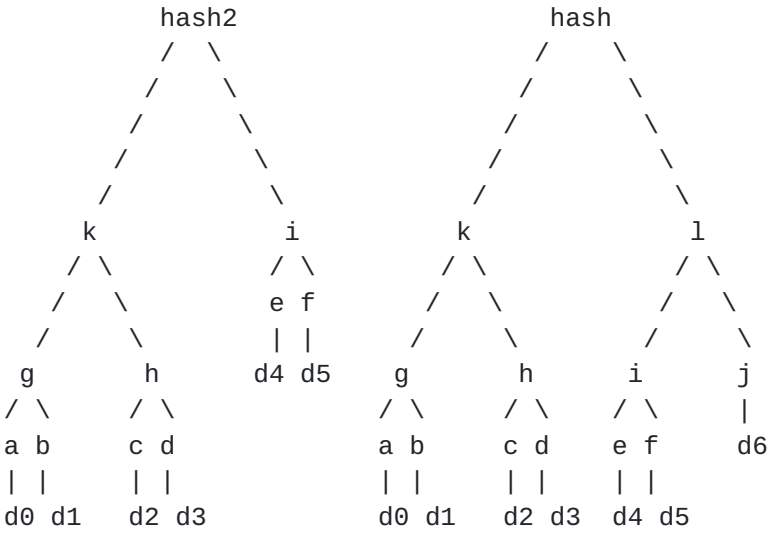
The audit path for d3 is [c, g, l].

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 hash1, and j is additionally used to show hash is consistent with hash2.

2. Log Format

Anyone can submit certificates to certificate logs for public auditing, however, since certificates will not be accepted by 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.

After accepting a certificate submission, 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). Servers MUST present an SCT from one or more logs to the client together with the certificate. Clients MUST reject certificates that do not have a valid Signed Certificate Timestamp.

Periodically, the log appends all new entries to the Merkle Tree, and signs the root of the tree. Clients and auditors can thus verify that each certificate for which an SCT has been issued indeed appears in the log. The log MUST incorporate a certificate in its Merkle Tree within the Maximum Merge Delay period after the issuance of the SCT.

2.1. Log Entries

Anyone can submit a certificate to the log. In order to attribute each logged certificate to its issuer, the log shall publish a list of acceptable root certificates (this list should 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 self-signed root certificate itself MAY be omitted from this list.

In this case, the SCT must be included in the TLS handshake, either by using an Authorization Extension [RFC5878] with type [TBD] containing the SCT, or by using OCSP Stapling (section 8 of [RFC6066]), where the response includes an OCSP extension [RFC5280] with OID 1.3.6.1.4.1.11129.2.4.5 and body:

```
SignedCertificateTimestamp ::= OCTET STRING
```

Alternatively, (root as well as intermediate) Certificate Authorities may submit a certificate to the log prior to issuance. To do so, a Certificate Authority constructs a Precertificate by signing the leaf TBSCertificate [RFC5280] with a special-purpose (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

certified by the CA certificate. As above, the Precertificate submission MUST be accompanied by the Precertificate Signing Certificate 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). The log verifies the Precertificate signature chain, and issues a Signed Certificate Timestamp on the corresponding TBSCertificate. The SCT can then be directly embedded in the final certificate, by inserting it 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 log MUST verify that the submitted leaf 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. In case of Precertificates, the log MUST also verify that the Precertificate Signing Certificate has the correct Extended Key Usage extension. The log 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. However, the log MUST refuse to publish certificates without a valid chain to a known root CA. If a certificate is accepted and an SCT issued, the log MUST store the chain used for verification including the certificate or Precertificate itself, and MUST present this chain for auditing upon request.

Each certificate entry in the 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 tbs_certificate;
    ASN.1Cert precertificate_chain<1..2^24-1>;
} PrecertChainEntry;
```

Logs MAY limit the length of chain they will accept.

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

"certificate_chain" is a chain of additional certificates required to verify the leaf certificate. The first certificate MUST certify the leaf certificate. Each following certificate MUST directly certify the one preceding it. The self-signed root certificate MAY be omitted from the chain.

"tbs_certificate" is the TBSCertificate component of the Precertificate (i.e., the original TBSCertificate, without the Precertificate signature and the SCT extension).

"precertificate_chain" is a chain of certificates required to verify the Precertificate submission. The first certificate MUST be the original Precertificate, with its unsigned part matching the "tbs_certificate". The second certificate MUST be a valid Precertificate Signing Certificate, and MUST certify the first certificate. Each following certificate MUST directly certify the one preceding it. The self-signed root certificate MAY be omitted from the chain.

Structure of the Signed Certificate Timestamp:

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

      enum { v1(0) }
        Version;

        struct {
            opaque key_id[32];
        } LogID;
```

"key_id" is the SHA-256 hash of the log's public key [TODO: define how to calculate this].

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

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

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

"timestamp" is the current UTC time since epoch (January 1, 1970, 00:00), in milliseconds.

"signed_entry" is the "leaf_certificate" (in case of an X509ChainEntry), or "tbs_certificate" (in case of a PrecertChainEntry).

2.2. Merkle Tree

A certificate log MUST periodically append all new log entries to the log Merkle Tree. The log MUST sign these entries by constructing a binary Merkle Tree with log entries as consecutive inputs to the tree, signing the corresponding Merkle Tree Hash, and publishing each update to the tree in a Signed Merkle Tree Update. The hashing algorithm for the Merkle Tree Hash is SHA-256.

Structure of the Merkle Tree input:

```
    struct {
        uint64 timestamp;
        LogEntryType entry_type;
        select(entry_type) {
            case x509_entry: ASN.1Cert;
            case precert_entry: ASN.1Cert;
        } signed_entry;
    } MerkleTreeLeaf;
```

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

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

Structure of the Signed Merkle Tree Update:

```
    struct {
        Version version;
        LogID id;
        uint64 old_tree_size;
        uint64 timestamp;
        MerkleTreeLeaf new_leaves<0..2^64-1>;
        digitally-signed struct {
            SignatureType signature_type = tree_hash;
            uint64 timestamp;
            uint64 tree_size;
            opaque sha256_root_hash[32];
        } TreeHeadSignature;
    } SignedMerkleTreeUpdate;
```

"version" is the version of the protocol the SignedMerkleTreeUpdate conforms to.

"old_tree_size" is the size of the tree prior to this update.

"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.

"new_leaves" is the list of leaves added to the tree in this update, ordered by leaf index. This order can be fixed arbitrarily amongst new entries.

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

The log MUST produce a Signed Merkle Tree Update at least as often as 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.

2.3. Audit Proofs

It is possible to audit the entire log by computing the current "sha256_root_hash" value from consecutive Signed Merkle Tree Updates, and verifying the Tree Head Signature. We rely on cross-checks of the Signed Tree Head between auditors to verify that their views of the log are consistent.

Additionally, logs provide Merkle audit proofs for efficient partial checks. (In fact, anyone can compute audit proofs from the full log.) Merkle audit proofs allow auditors to efficiently verify that a certificate for which an SCT has been issued indeed appears in the log, without inspecting the entire log.

Structure of the Merkle audit proof:

```
struct {
    opaque sha256_hash[32];
} MerkleNode;


struct {
    Version version;
    LogID id;
    uint64 tree_size;
    uint64 timestamp;
    uint64 leaf_index;
    MerkleNode audit_path<0..2^16-1>;
    TreeHeadSignature tree_head_signature;
} MerkleAuditProof;
```

"tree_size" is the generation of the tree that this proof is for.

"timestamp" is the corresponding timestamp.

"leaf_index" is the index of the audited node in the Merkle tree, from 0 to "tree_size - 1".

"audit_path" is a list of additional nodes in the Merkle tree required for reconstructing the root hash corresponding to the "tree_size". Nodes must be listed from leaf to root level, i.e., in

the order they are used in the Merkle Tree Hash computation, as defined in Section 1.1.1. _Notice that the left-right ordering is determined by the position of the leaf._ The leaf node under audit as well as the root node shall be omitted from the path.

"tree_head_signature" is the TreeHeadSignature for generation "tree_size".

A valid audit proof for a Merkle Tree Leaf MUST satisfy the following:

- The "tree_size" MUST be at least 1;
- The "leaf_index" MUST NOT exceed "tree_size - 1";
- The "tree_signature" MUST be a valid signature on the corresponding "timestamp", "tree_size", and the root hash reconstructed from the Merkle Tree Leaf, "leaf_index" and "audit_path".

3. Client Messages

TBD. Messages that clients send to logs, e.g. to request an SCT or retrieve entries in the log.

4. Security and Privacy Considerations

4.1. Misissued Certificates

Misissued certificates that have not been publicly logged, and thus do not have a valid SCT, will be rejected by clients. Misissued certificates that do have an SCT from a log will appear in the 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.

4.2. Detection of Misissue

The log does not itself detect misissued certificate, it relies instead on interested parties, such as domain owners, to monitor it and take corrective action when a misissue is detected.

4.3. 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 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 the log comparing their versions of the latest signed tree head. As soon as two conflicting signed tree heads are detected, this is cryptographic proof of the log's misbehaviour.

5. Efficiency Considerations

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

Auditing the log for integrity does not require third parties to maintain a copy of the entire log. The Signed Tree Head root hash can be updated incrementally as new entries become available, without recomputing the entire tree. Third party auditors need only store a logarithmic number of intermediate nodes in the Merkle Tree.

Additionally, the Merkle consistency proofs defined in Section 1.1.2 can be used to efficiently prove the append-only property of an incremental update to the Merkle Tree, without auditing the entire tree.

6. References

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

[RFC5878] Brown, M. and R. Housley, "The Transport Layer Security (TLS) Authorization Extensions", RFC 5280, May 2010.

[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.

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

[1] <http://tamperevident.cs.rice.edu/Logging.html/>

Authors' Addresses

Ben Laurie

Email: benl@google.com

Adam Langley

Email: agl@google.com

Emilia Kasper

Email: ekasper@google.com