Network Working Group
Internet-Draft
Expires: June 2, 2013

B. Laurie
A. Langley
E. Kasper

November 29, 2012

Certificate Transparency
draft-laurie-pki-sunlight-03

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], in a stapled [RFC6066] OCSP extension [RFC2560], 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 June 2, 2013.

Copyright Notice

Table of Contents

1. Informal introduction

Certificate Transparency aims to solve the problem of mis-issued certificates by providing a publicly auditable, append-only, untrusted log of all issued certificates. The logs are publicly auditable so that it is possible for anyone to verify the correct operation of the log, and to monitor when new certificates added to it. The logs do not themselves prevent mis-issue, but they ensure that interested parties (particularly those named in certificates) can detect such mis-issuance. Note that this is a general mechanism, but in this document we only decsribe its use for public TLS certificates issued by public CAs.

The log consists of certificate chains, which can be submitted by anyone. It is expected that most public CAs will contribute all their newly-issued certificates to the log; it is also expected that certificate holders will also contribute their own certificate chains. In order to avoid the log being spammed into uselessness, it is required that the chain is rooted in a known CA certificate. When a chain is submitted to the log, a signed timestamp is returned, which can later be used to prove to clients that the chain has been submitted. Clients can thus require that all certificates they see have been logged.

Those who are concerned about mis-issue can monitor the log, asking it 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 mis-issuance has happened, is beyond the scope of this document, but broadly speaking they can invoke existing business mechanisms for dealing with mis-issued certificates. Of course, anyone who wants can monitor the log, and if they believe a certificate is incorrectly issued, take action as they see fit.

Similarly, those who have seen signed timestamps from the log can later demand a proof of inclusion from the log. If the log is unable to provide this (or, indeed, if the corresponding certificate is absent from monitors' copies of the log), that is evidence of the incorrect operation of the log. This 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 a 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 trust the log: if the log attempts to show different things to different people, this can be efficiently detected by comparing tree roots and consistency proofs.

2. Cryptographic components

2.1. 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.]

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 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 < 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].

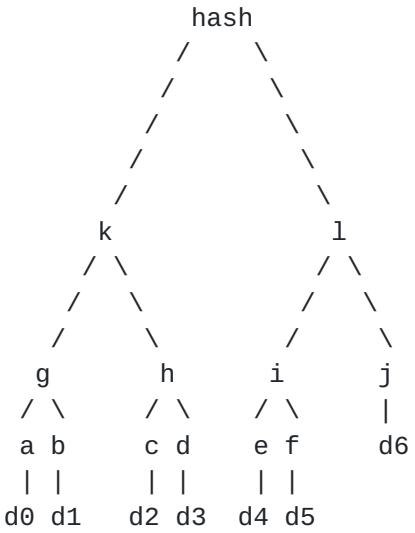
SUBPROOF(m, D[n], b) = SUBPROOF(m - k, D[k:n], false) : MTH(D[0:k]).

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.

2.1.3. Example

The binary Merkle tree with 7 leaves:



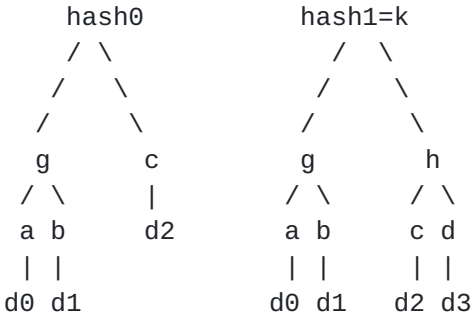
The audit path for d0 is [b, h, l].

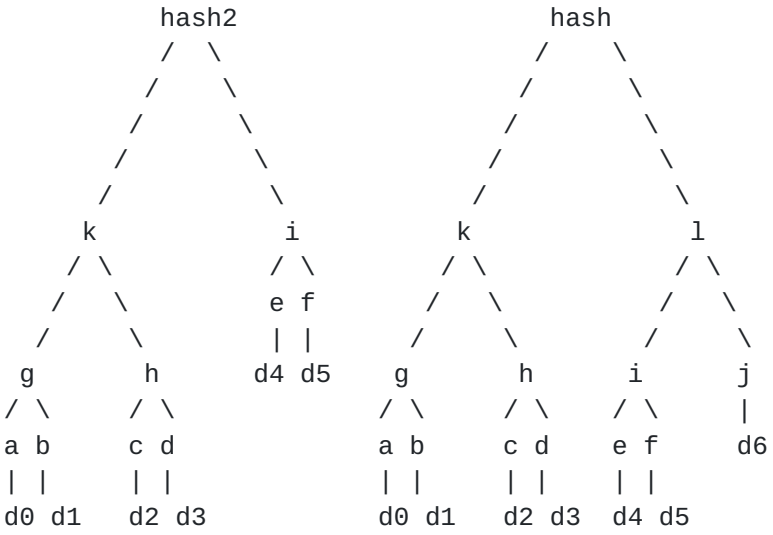
The audit path for d3 is [c, g, l].

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 can use either elliptic curve signatures using the NIST P-256 curve (http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf section D.1.2.3) or RSA signatures using a key of at least 2048 bits.

3. 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). If the log has previously seen the certificate, it MAY return the same SCT as it returned before. 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 SCT for the end-entity certificate.

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.

3.1. 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.

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 adding a special critical poison extension (OID 1.3.6.1.4.1.11129.2.4.3, ASN.1 NULL data) to the leaf TBSCertificate, and signing the resulting TBSCertificate [RFC5280] with a special-purpose (Extended Key Usage: Certificate Transparency, OID 1.3.6.1.4.1.11129.2.4.4, basicConstraints=critical,CA:FALSE) 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 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.

"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 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), 255 }
      Version;

      struct {
          opaque key_id[32];
      } LogID;

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

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

```
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: ASN.1Cert;
        } signed_entry;
       CtExtensions extensions;
    };
} SignedCertificateTimestamp;
```

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

"sct_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.

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

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

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

3.2. Including the Signed Certificate Timestamp in the TLS Handshake

The SCT data from at least one log must be included in the TLS handshake, either by using an Authorization Extension [RFC5878] with type 182, or by using OCSP Stapling (section 8 of [RFC6066]), where the response includes an OCSP extension with OID 1.3.6.1.4.1.11129.2.4.5 (see [RFC2560]) and body:

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

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

Similarly, the Certificate Authority MAY submit the 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 bytestring that contains the serialized TLS structure. This encoding ensures that 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 version they don't understand).

SCTs embedded in the TLS Authorization Extension are each encoded as an individual AuthorizationDataEntry [RFC5878].

3.3. 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:

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

struct {
    uint64 timestamp;
    LogEntryType entry_type;
    select(entry_type) {
        case x509_entry: ASN.1Cert;
        case precert_entry: ASN.1Cert;
    } 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 the MerkleTreeLeaf corresponds to. 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 hashes of the corresponding "MerkleTreeLeaf" structures.

3.4. Tree Head Signature

Every time the log appends new entries to the tree, the log MUST sign the corresponding tree hash and tree information (see also the corresponding Signed Tree Head client message in Section 4.3). The signature input 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 the TreeHeadSignature conforms to. 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.

The log MUST produce a Tree Head Signature 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.

4. Client Messages

Messages are sent as HTTPS GET or POST requests. Parameters for POSTs and all responses are encoded as JSON objects. Parameters for GETs are encoded as URL parameters. Binary data is base64 encoded as specified in the individual messages.

The <log server> prefix can include a path as well as a server name and a port. It must map one-to-one to a known public key (how this mapping is distributed is out of scope for this document).

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

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 leaf 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 clients who request an SCT for inclusion in the TLS handshake are not required to verify it, we do not assume they know the ID of the log.

timestamp The SCT timestamp, in decimal.

extensions [TBD]

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.

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 leaf 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 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 root 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 published v1 STHs.

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 to base the proof on, in decimal.

The "hash" must be calculated as defined in Section 3.3. The "tree_size" must designate a published v1 STH.

Outputs

timestamp The tree's timestamp, in decimal.

leaf_index The index of the leaf 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 Index of first entry to retrieve, in decimal.

end 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 "precertificate_chain".

Note that this message is not signed - the retrieved data can be verified by constructing the root 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.

4.7. 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 a published STH.

Outputs

entries An array of objects, each consisting of

leaf_input The base64-encoded MerkleTreeLeaf structure.

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

timestamp The tree's timestamp, in decimal.

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 the log 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 misbehaviour.

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 is TBD, but it is expected there will be a variety.

5.1. Monitor

Monitors watch the log and check that it behaves correctly. They also watch for certificates of interest.

A monitor needs to, at least, inspect every new entry in the log. It may also want to keep a copy of the entire log. In order to do this, it should follow these steps:

1. Fetch the current STH using Section 4.3.

2. Verify the STH signature.

3. Fetch all the entries in the tree corresponding to the STH using 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 using Section 4.3. Repeat until STH changes.

6. Verify the STH signature.

7. Fetch all the new entries in the tree corresponding to the STH using Section 4.6. If they remain unavailable for an extended period, then this should be viewed as misbehaviour 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:

2. Fetch a consistency proof for the new STH with the previous STH using Section 4.4.

3. Verify the consistency proof.

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

9. Go to Step 5.

5.2. 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 using 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 using Section 4.5.

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

6. Security and Privacy Considerations

6.1. 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.

6.2. 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.

6.3. 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.

7. 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 can be updated as new entries become available, without recomputing the entire tree. Third party auditors need only fetch the Merkle consistency proof against an existing STH to efficiently verify the append-only property of an update to the Merkle Tree, without auditing the entire tree.

8. References

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

[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.

[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