TRANS (Public Notary Transparency)
Internet-Draft
Obsoletes: 6962 (if approved)
Intended status: Experimental
Expires: May 10, 2019

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

November 06, 2018

Certificate Transparency Version 2.0
draft-ietf-trans-rfc6962-bis-30

Abstract

This document describes version 2.0 of the Certificate Transparency (CT) protocol for publicly logging the existence of Transport Layer Security (TLS) server 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.

This document obsoletes RFC 6962. It also specifies a new TLS extension that is used to send various CT log artifacts.

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 https://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 May 10, 2019.

Table of Contents

1. Introduction

Certificate Transparency aims to mitigate the problem of misissued certificates by providing append-only logs of issued certificates. The logs do not themselves prevent misissuance, but they ensure that interested parties (particularly those named in certificates) can detect such misissuance. Note that this is a general mechanism that could be used for transparently logging any form of binary data, subject to some kind of inclusion criteria. In this document, we only describe its use for public TLS server certificates (i.e., where the inclusion criteria is a valid certificate issued by a public certification authority (CA)).

Each log contains certificate chains, which can be submitted by anyone. It is expected that public CAs will contribute all their newly issued certificates to one or more logs; however certificate holders can also contribute their own certificate chains, as can third parties. In order to avoid logs being rendered useless by the submission of large numbers of spurious certificates, it is required that each chain ends with a trust anchor that is accepted by the log. When a chain is accepted by a log, a signed timestamp is returned, which can later be used to provide evidence to TLS clients that the chain has been submitted. TLS clients can thus require that all certificates they accept as valid are accompanied by signed timestamps.

Those who are concerned about misissuance can monitor the logs, asking them regularly for all new entries, and can thus check whether domains for which they are responsible have had certificates issued that they did not expect. What they do with this information, particularly when they find that a misissuance has happened, is beyond the scope of this document. However, broadly speaking, they can invoke existing business mechanisms for dealing with misissued certificates, such as working with the CA to get the certificate revoked, or with maintainers of trust anchor lists to get the CA removed. Of course, anyone who wants can monitor the logs and, if they believe a certificate is incorrectly issued, take action as they see fit.

Similarly, those who have seen signed timestamps from a particular log can later demand a proof of inclusion from that log. If the log is unable to provide this (or, indeed, if the corresponding certificate is absent from monitors' copies of that log), that is evidence of the incorrect operation of the log. The checking operation is asynchronous to allow clients to proceed without delay, despite possible issues such as network connectivity and the vagaries of firewalls.

The append-only property of each log is achieved using Merkle Trees, which can be used to efficiently prove that any particular instance of the log is a superset of any particular previous instance and to efficiently detect various misbehaviors of the log (e.g., issuing a signed timestamp for a certificate that is not subsequently logged).

It is necessary to treat each log as a trusted third party, because the log auditing mechanisms described in this document can be circumvented by a misbehaving log that shows different, inconsistent views of itself to different clients. Whilst it is anticipated that additional mechanisms could be developed to address these shortcomings and thereby avoid the need to blindly trust logs, such mechanisms are outside the scope of this document.

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 [RFC2119].

1.2. Data Structures

Data structures are defined and encoded according to the conventions laid out in Section 3 of [RFC8446].

1.3. Major Differences from CT 1.0

This document revises and obsoletes the experimental CT 1.0 [RFC6962] protocol, drawing on insights gained from CT 1.0 deployments and on feedback from the community. The major changes are:

- Hash and signature algorithm agility: permitted algorithms are now specified in IANA registries.
- Precertificate format: precertificates are now CMS objects rather than X.509 certificates, which avoids violating the certificate serial number uniqueness requirement in Section 4.1.2.2 of [RFC5280].

- Removed precertificate signing certificates and the precertificate poison extension: the change of precertificate format means that these are no longer needed.

- Logs IDs: each log is now identified by an OID rather than by the hash of its public key. OID allocations are managed by an IANA registry.

- "TransItem" structure: this new data structure is used to encapsulate most types of CT data. A "TransItemList", consisting of one or more "TransItem" structures, can be used anywhere that "SignedCertificateTimestampList" was used in [RFC6962].

- Merkle tree leaves: the "MerkleTreeLeaf" structure has been replaced by the "TransItem" structure, which eases extensibility and simplifies the leaf structure by removing one layer of abstraction.

- Unified leaf format: the structure for both certificate and precertificate entries now includes only the TBSCertificate (whereas certificate entries in [RFC6962] included the entire certificate).

- Log Artifact Extensions: these are now typed and managed by an IANA registry, and they can now appear not only in SCTs but also in STHs.

- API outputs: complete "TransItem" structures are returned, rather than the constituent parts of each structure.

- get-all-by-hash: new client API for obtaining an inclusion proof and the corresponding consistency proof at the same time.

- submit-entry: new client API, replacing add-chain and add-pre-chain.

- Presenting SCTs with proofs: TLS servers may present SCTs together with the corresponding inclusion proofs using any of the mechanisms that [RFC6962] defined for presenting SCTs only. (Presenting SCTs only is still supported).

- CT TLS extension: the "signed_certificate_timestamp" TLS extension has been replaced by the "transparency_info" TLS extension.

- Other TLS extensions: "status_request_v2" may be used (in the same manner as "status_request"); "cached_info" may be used to avoid sending the same complete SCTs and inclusion proofs to the same TLS clients multiple times.

- Verification algorithms: added detailed algorithms for verifying inclusion proofs, for verifying consistency between two STHs, and for verifying a root hash given a complete list of the relevant leaf input entries.

- Extensive clarifications and editorial work.

2. Cryptographic Components

2.1. Merkle Hash Trees

2.1.1. Definition of the Merkle Tree

The log uses a binary Merkle Hash Tree for efficient auditing. The hash algorithm used is one of the log's parameters (see Section 4.1). We have established a registry of acceptable hash algorithms (see Section 10.3). Throughout this document, the hash algorithm in use is referred to as HASH and the size of its output in bytes as HASH_SIZE. The input to the Merkle Tree Hash is a list of data entries; these entries will be hashed to form the leaves of the Merkle Hash Tree. The output is a single HASH_SIZE Merkle Tree Hash. Given an ordered list of n inputs, D_n = {d[0], d[1], ..., d[n-1]}, the Merkle Tree Hash (MTH) is thus defined as follows:

The hash of an empty list is the hash of an empty string:

MTH({}) = HASH().

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

MTH({d[0]}) = HASH(0x00 || d[0]).

For n > 1, let k be the largest power of two smaller than n (i.e., k < n <= 2k). The Merkle Tree Hash of an n-element list D_n is then defined recursively as

MTH(D_n) = HASH(0x01 || MTH(D[0:k]) || MTH(D[k:n])),

Where || is concatenation and D[k1:k2] = D'_(k2-k1) denotes the list {d'[0] = d[k1], d'[1] = d[k1+1], ..., d'[k2-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).

2.1.2. Verifying a Tree Head Given Entries

When a client has a complete list of n input "entries" from "0" up to "tree_size - 1" and wishes to verify this list against a tree head "root_hash" returned by the log for the same "tree_size", the following algorithm may be used:

1. Set "stack" to an empty stack.
2. For each "i" from "0" up to "tree_size - 1":
    1. Push "HASH(0x00 || entries[i])" to "stack".
    2. Set "merge_count" to the lowest value ("0" included) such that "LSB(i >> merge_count)" is not set. In other words, set "merge_count" to the number of consecutive "1"s found starting at the least significant bit of "i".
    3. Repeat "merge_count" times:
        1. Pop "right" from "stack".
        2. Pop "left" from "stack".
        3. Push "HASH(0x01 || left || right)" to "stack".
3. If there is more than one element in the "stack", repeat the same merge procedure (Step 2.3 above) until only a single element remains.
4. The remaining element in "stack" is the Merkle Tree hash for the given "tree_size" and should be compared by equality against the supplied "root_hash".

2.1.3. 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.

2.1.3.1. Generating an Inclusion Proof

Given an ordered list of n inputs to the tree, D_n = {d[0], d[1], ..., 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,

The : operator and D[k1:k2] are defined the same as in Section 2.1.1.

2.1.3.2. Verifying an Inclusion Proof

When a client has received an inclusion proof (e.g., in a "TransItem" of type "inclusion_proof_v2") and wishes to verify inclusion of an input "hash" for a given "tree_size" and "root_hash", the following algorithm may be used to prove the "hash" was included in the "root_hash":

1. Compare "leaf_index" against "tree_size". If "leaf_index" is greater than or equal to "tree_size" then fail the proof verification.

2. Set "fn" to "leaf_index" and "sn" to "tree_size - 1".

3. Set "r" to "hash".

4. For each value "p" in the "inclusion_path" array:

   If "sn" is 0, stop the iteration and fail the proof verification.

   If "LSB(fn)" is set, or if "fn" is equal to "sn", then:

   1. Set "r" to "HASH(0x01 || p || r)"

2. If "LSB(fn)" is not set, then right-shift both "fn" and "sn" equally until either "LSB(fn)" is set or "fn" is "0".

Otherwise:

1. Set "r" to "HASH(0x01 || r || p)"

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

5. Compare "sn" to 0. Compare "r" against the "root_hash". If "sn" is equal to 0, and "r" and the "root_hash" are equal, then the log has proven the inclusion of "hash". Otherwise, fail the proof verification.

2.1.4. 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.

2.1.4.1. Generating a Consistency Proof

Given an ordered list of n inputs to the tree, D_n = {d[0], d[1], ..., d[n-1]}, the Merkle consistency proof PROOF(m, D_n) for a previous Merkle Tree Hash MTH(D[0:m]), 0 < m < n, is defined as:

PROOF(m, D_n) = SUBPROOF(m, D_n, true)

In SUBPROOF, the boolean value represents whether the subtree created from D[0:m] is a complete subtree of the Merkle Tree created from D_n, and, consequently, whether the subtree Merkle Tree Hash MTH(D[0:m]) is known. The initial call to SUBPROOF sets this to be true, and SUBPROOF is then defined as follows:

The subproof for m = n is empty if m is the value for which PROOF was originally requested (meaning that the subtree created from D[0:m] is a complete subtree of the Merkle Tree created from the original D_n for which PROOF was requested, and the subtree Merkle Tree Hash MTH(D[0:m]) is known):

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

Otherwise, the subproof for m = n is the Merkle Tree Hash committing inputs D[0:m]:

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

For m < n, let k be the largest power of two smaller than n. The subproof is then defined recursively.

If m <= k, the right subtree entries D[k:n] only exist in the current tree. We prove that the left subtree entries D[0:k] are consistent and add a commitment to D[k:n]:

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

If m > k, the left subtree entries D[0:k] are identical in both trees. We prove that the right subtree entries D[k:n] are consistent and add a commitment to D[0:k].

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

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

The : operator and D[k1:k2] are defined the same as in Section 2.1.1.

2.1.4.2. Verifying Consistency between Two Tree Heads

When a client has a tree head "first_hash" for tree size "first", a tree head "second_hash" for tree size "second" where "0 < first < second", and has received a consistency proof between the two (e.g., in a "TransItem" of type "consistency_proof_v2"), the following algorithm may be used to verify the consistency proof:

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

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

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

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

5. For each subsequent value "c" in the "consistency_path" array:

   If "sn" is 0, stop the iteration and fail the proof verification.

If "LSB(fn)" is set, or if "fn" is equal to "sn", then:

1. Set "fr" to "HASH(0x01 || c || fr)"
   Set "sr" to "HASH(0x01 || c || sr)"

2. If "LSB(fn)" is not set, then right-shift both "fn" and "sn" equally until either "LSB(fn)" is set or "fn" is "0".

Otherwise:

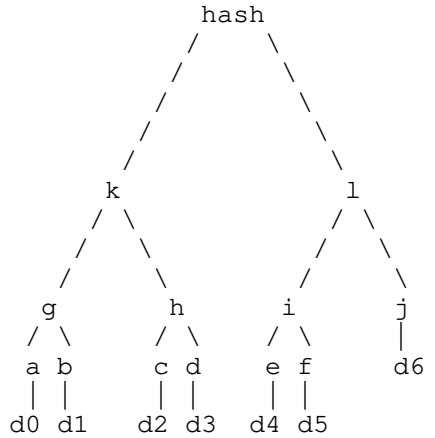
1. Set "sr" to "HASH(0x01 || sr || c)"

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

6. After completing iterating through the "consistency_path" array as described above, verify that the "fr" calculated is equal to the "first_hash" supplied, that the "sr" calculated is equal to the "second_hash" supplied and that "sn" is 0.

2.1.5. Example

The binary Merkle Tree with 7 leaves:



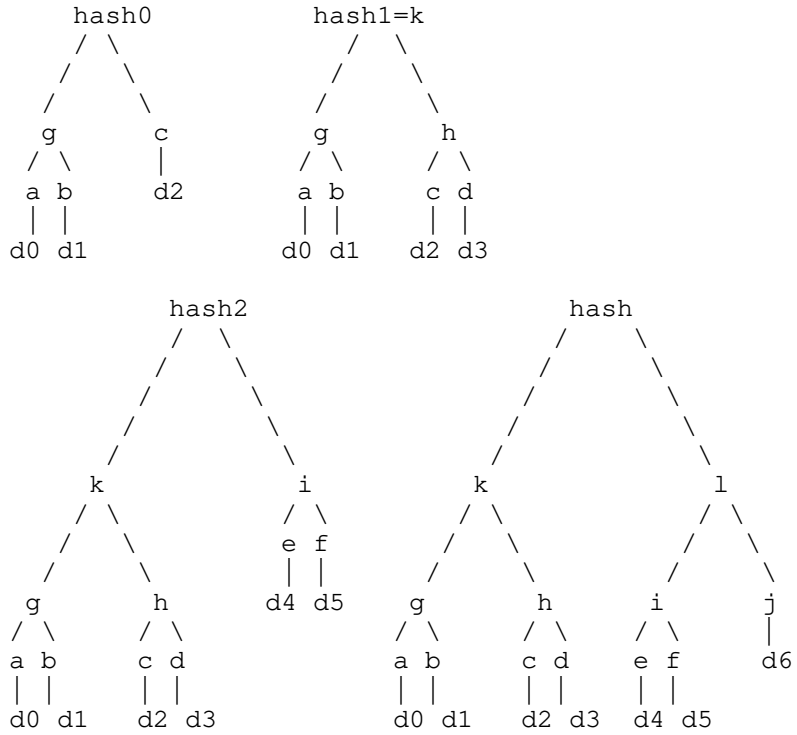
The inclusion proof for d0 is [b, h, l].

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.

2.2. Signatures

Various data structures Section 1.2 are signed. A log MUST use one of the signature algorithms defined in Section 10.4.

3. Submitters

Submitters submit certificates or preannouncements of certificates prior to issuance (precertificates) to logs for public auditing, as described below. In order to enable attribution of each logged certificate or precertificate to its issuer, each submission MUST be

accompanied by all additional certificates required to verify the chain up to an accepted trust anchor (Section 5.7). The trust anchor (a root or intermediate CA certificate) MAY be omitted from the submission.

If a log accepts a submission, it will return a Signed Certificate Timestamp (SCT) (see Section 4.8). The submitter SHOULD validate the returned SCT as described in Section 8.1 if they understand its format and they intend to use it directly in a TLS handshake or to construct a certificate. If the submitter does not need the SCT (for example, the certificate is being submitted simply to make it available in the log), it MAY validate the SCT.

3.1. Certificates

Any entity can submit a certificate (Section 5.1) to a log. Since it is anticipated that TLS clients will reject certificates that are not logged, it is expected that certificate issuers and subjects will be strongly motivated to submit them.

3.2. Precertificates

CAs may preannounce a certificate prior to issuance by submitting a precertificate (Section 5.1) that the log can use to create an entry that will be valid against the issued certificate. The CA MAY incorporate the returned SCT in the issued certificate. One example of where the returned SCT is not incorporated in the issued certificate is when a CA sends the precertificate to multiple logs, but only incorporates the SCTs that are returned first.

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

- It MUST be DER encoded.
- "SignedData.version" MUST be v3(3).
- "SignedData.digestAlgorithms" MUST only include the "SignerInfo.digestAlgorithm" OID value (see below).
- "SignedData.encapContentInfo":
  - "eContentType" MUST be the OID 1.3.101.78.
  - "eContent" MUST contain a TBSCertificate [RFC5280] that will be identical to the TBSCertificate in the issued certificate, except that the Transparency Information (Section 7.1) extension MUST be omitted.

- "SignedData.certificates" MUST be omitted.
- "SignedData.crls" MUST be omitted.
- "SignedData.signerInfos" MUST contain one "SignerInfo":
  - "version" MUST be v3(3).
  - "sid" MUST use the "subjectKeyIdentifier" option.
  - "digestAlgorithm" MUST be one of the hash algorithm OIDs listed in Section 10.3.
  - "signedAttrs" MUST be present and MUST contain two attributes:
    - A content-type attribute whose value is the same as "SignedData.encapContentInfo.eContentType".
    - A message-digest attribute whose value is the message digest of "SignedData.encapContentInfo.eContent".
  - "signatureAlgorithm" MUST be the same OID as "TBSCertificate.signature".
  - "signature" MUST be from the same (root or intermediate) CA that will ultimately issue the certificate. This signature indicates the CA's intent to issue the certificate. This intent is considered binding (i.e., misissuance of the precertificate is considered equivalent to misissuance of the corresponding certificate).
  - "unsignedAttrs" MUST be omitted.

"SignerInfo.signedAttrs" is included in the message digest calculation process (see Section 5.4 of [RFC5652]), which ensures that the "SignerInfo.signature" value will not be a valid X.509v3 signature that could be used in conjunction with the TBSCertificate (from "SignedData.encapContentInfo.eContent") to construct a valid certificate.

4. Log Format and Operation

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

When it receives and accepts a valid submission, the log MUST return an SCT that corresponds to the submitted certificate or precertificate. If the log has previously seen this valid

submission, it SHOULD return the same SCT as it returned before (to reduce the ability to track clients as described in Section 11.4). If different SCTs are produced for the same submission, multiple log entries will have to be created, one for each SCT (as the timestamp is a part of the leaf structure). Note that if a certificate was previously logged as a precertificate, then the precertificate's SCT of type "precert_sct_v2" would not be appropriate; instead, a fresh SCT of type "x509_sct_v2" should be generated.

An SCT is the log's promise to append to its Merkle Tree an entry for the accepted submission. Upon producing an SCT, the log MUST fulfil this promise by performing the following actions within a fixed amount of time known as the Maximum Merge Delay (MMD), which is one of the log's parameters (see Section 4.1):

- Allocate a tree index to the entry representing the accepted submission.
- Calculate the root of the tree.
- Sign the root of the tree (see Section 4.10).

The log may append multiple entries before signing the root of the tree.

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

4.1. Log Parameters

A log is defined by a collection of parameters, which are used by clients to communicate with the log and to verify log artifacts.

Base URL: The URL to substitute for <log server> in Section 5.

Hash Algorithm: The hash algorithm used for the Merkle Tree (see Section 10.3).

Signature Algorithm: The signature algorithm used (see Section 2.2).

Public Key: The public key used to verify signatures generated by the log. A log MUST NOT use the same keypair as any other log.

Log ID: The OID that uniquely identifies the log.

Maximum Merge Delay: The MMD the log has committed to.

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

Maximum Chain Length: The longest chain submission the log is willing to accept, if the log imposes any limit.

STH Frequency Count: The maximum number of STHs the log may produce in any period equal to the "Maximum Merge Delay" (see Section 4.10).

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. The final STH should be provided in the form of a TransItem of type "signed_tree_head_v2".

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

4.2. Accepting Submissions

To ensure that logged certificates and precertificates are attributable to a known trust anchor, and to set clear expectations for what monitors would find in a log, and to avoid being overloaded by invalid submissions, the log MUST NOT accept any submission until it has verified that the submitted certificate or precertificate chains to an accepted trust anchor.

The log MUST NOT use other sources of intermediate CA certificates to attempt certification path construction; instead, it MUST only use the intermediate CA certificates provided in the submission, in the order provided.

Logs SHOULD accept certificates and precertificates that are fully valid according to RFC 5280 [RFC5280] verification rules and are submitted with such a chain. (A log may decide, for example, to temporarily reject valid submissions to protect itself against denial-of-service attacks).

Logs MAY accept certificates and precertificates that have expired, are not yet valid, have been revoked, or are otherwise not fully valid according to RFC 5280 verification rules in order to accommodate quirks of CA certificate-issuing software. However, logs MUST reject submissions without a valid signature chain to an accepted trust anchor. Logs MUST also reject precertificates that do not conform to the requirements in Section 3.2.

Logs SHOULD limit the length of chain they will accept. The maximum chain length is one of the log's parameters (see Section 4.1).

The log SHALL allow retrieval of its list of accepted trust anchors (see Section 5.7), each of which is a root or intermediate CA certificate. This list might usefully be the union of root certificates trusted by major browser vendors.

4.3. Log Entries

If a submission is accepted and an SCT issued, the accepting log MUST store the entire chain used for verification. This chain MUST include the certificate or precertificate itself, the zero or more intermediate CA certificates provided by the submitter, and the trust anchor used to verify the chain (even if it was omitted from the submission). The log MUST present this chain for auditing upon request (see Section 5.6). This prevents the CA from avoiding blame by logging a partial or empty chain. Each log entry is a "TransItem" structure of type "x509_entry_v2" or "precert_entry_v2". However, a log may store its entries in any format. If a log does not store this "TransItem" in full, it must store the "timestamp" and "sct_extensions" of the corresponding "TimestampedCertificateEntryDataV2" structure. The "TransItem" can be reconstructed from these fields and the entire chain that the log used to verify the submission.

4.4. Log ID

Each log is identified by an OID, which is one of the log's parameters (see Section 4.1) and which MUST NOT be used to identify any other log. A log's operator MUST either allocate the OID themselves or request an OID from the Log ID Registry (see Section 10.7.1). Various data structures include the DER encoding of this OID, excluding the ASN.1 tag and length bytes, in an opaque vector:

```
opaque LogID<2..127>;
```

Note that the ASN.1 length and the opaque vector length are identical in size (1 byte) and value, so the DER encoding of the OID can be reproduced simply by prepending an OBJECT IDENTIFIER tag (0x06) to the opaque vector length and contents.

OIDs used to identify logs are limited such that the DER encoding of their value is less than or equal to 127 octets.

4.5. TransItem Structure

Various data structures are encapsulated in the "TransItem" structure to ensure that the type and version of each one is identified in a common fashion:

```
enum {
    reserved(0),
    x509_entry_v2(1), precert_entry_v2(2),
    x509_sct_v2(3), precert_sct_v2(4),
    signed_tree_head_v2(5), consistency_proof_v2(6),
    inclusion_proof_v2(7),
    (65535)
} VersionedTransType;

struct {
    VersionedTransType versioned_type;
    select (versioned_type) {
        case x509_entry_v2: TimestampedCertificateEntryDataV2;
        case precert_entry_v2: TimestampedCertificateEntryDataV2;
        case x509_sct_v2: SignedCertificateTimestampDataV2;
        case precert_sct_v2: SignedCertificateTimestampDataV2;
        case signed_tree_head_v2: SignedTreeHeadDataV2;
        case consistency_proof_v2: ConsistencyProofDataV2;
        case inclusion_proof_v2: InclusionProofDataV2;
    } data;
} TransItem;
```

"versioned_type" is a value from the IANA registry in Section 10.5 that identifies the type of the encapsulated data structure and the earliest version of this protocol to which it conforms. This document is v2.

"data" is the encapsulated data structure. The various structures named with the "DataV2" suffix are defined in later sections of this document.

Note that "VersionedTransType" combines the v1 [RFC6962] type enumerations "Version", "LogEntryType", "SignatureType" and "MerkleLeafType". Note also that v1 did not define "TransItem", but this document provides guidelines (see Appendix A) on how v2 implementations can co-exist with v1 implementations.

Future versions of this protocol may reuse "VersionedTransType" values defined in this document as long as the corresponding data structures are not modified, and may add new "VersionedTransType" values for new or modified data structures.

4.6. Log Artifact Extensions

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

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
```

The "Extension" structure provides a generic extensibility for log artifacts, including Signed Certificate Timestamps (Section 4.8) and Signed Tree Heads (Section 4.10). The interpretation of the "extension_data" field is determined solely by the value of the "extension_type" field.

This document does not define any extensions, but it does establish a registry for future "ExtensionType" values (see Section 10.6). Each document that registers a new "ExtensionType" must specify the context in which it may be used (e.g., SCT, STH, or both) and describe how to interpret the corresponding "extension_data".

4.7. Merkle Tree Leaves

The leaves of a log's Merkle Tree correspond to the log's entries (see Section 4.3). Each leaf is the leaf hash (Section 2.1) of a "TransItem" structure of type "x509_entry_v2" or "precert_entry_v2", which encapsulates a "TimestampedCertificateEntryDataV2" structure. Note that leaf hashes are calculated as HASH(0x00 || TransItem), where the hash algorithm is one of the log's parameters.

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

struct {
    uint64 timestamp;
    opaque issuer_key_hash<32..2^8-1>;
    TBSCertificate tbs_certificate;
    Extension sct_extensions<0..2^16-1>;
} TimestampedCertificateEntryDataV2;
```

"timestamp" is the date and time at which the certificate or precertificate was accepted by the log, in the form of a 64-bit unsigned number of milliseconds elapsed since the Unix Epoch (1 January 1970 00:00:00 UTC - see [UNIXTIME]), ignoring leap seconds, in network byte order. Note that the leaves of a log's Merkle Tree are not required to be in strict chronological order.

"issuer_key_hash" is the HASH of the public key of the CA that issued the certificate or precertificate, calculated over the DER encoding of the key represented as SubjectPublicKeyInfo [RFC5280]. This is needed to bind the CA to the certificate or precertificate, making it impossible for the corresponding SCT to be valid for any other certificate or precertificate whose TBSCertificate matches "tbs_certificate". The length of the "issuer_key_hash" MUST match HASH_SIZE.

"tbs_certificate" is the DER encoded TBSCertificate from the submission. (Note that a precertificate's TBSCertificate can be reconstructed from the corresponding certificate as described in Section 8.1.2).

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

The type of the "TransItem" corresponds to the value of the "type" parameter supplied in the Section 5.1 call.

4.8. Signed Certificate Timestamp (SCT)

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

```
struct {
    LogID log_id;
    uint64 timestamp;
    Extension sct_extensions<0..2^16-1>;
    opaque signature<0..2^16-1>;
} SignedCertificateTimestampDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as described in Section 4.4.

"timestamp" is equal to the timestamp from the corresponding "TimestampedCertificateEntryDataV2" structure.

"sct_extensions" is a vector of 0 or more SCT extensions. This vector MUST NOT include more than one extension with the same "extension_type". The extensions in the vector MUST be ordered by the value of the "extension_type" field, smallest value first. If an implementation sees an extension that it does not understand, it SHOULD ignore that extension. Furthermore, an implementation MAY choose to ignore any extension(s) that it does understand.

"signature" is computed over a "TransItem" structure of type "x509_entry_v2" or "precert_entry_v2" (see Section 4.7) using the

signature algorithm declared in the log's parameters (see Section 4.1).

4.9. Merkle Tree Head

The log stores information about its Merkle Tree in a "TreeHeadDataV2":

```
opaque NodeHash<32..2^8-1>;

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

The length of NodeHash MUST match HASH_SIZE of the log.

"timestamp" is the current date and time, in the form of a 64-bit unsigned number of milliseconds elapsed since the Unix Epoch (1 January 1970 00:00:00 UTC - see [UNIXTIME]), ignoring leap seconds, in network byte order.

"tree_size" is the number of entries currently in the log's Merkle Tree.

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

"sth_extensions" is a vector of 0 or more STH extensions. This vector MUST NOT include more than one extension with the same "extension_type". The extensions in the vector MUST be ordered by the value of the "extension_type" field, smallest value first. If an implementation sees an extension that it does not understand, it SHOULD ignore that extension. Furthermore, an implementation MAY choose to ignore any extension(s) that it does understand.

4.10. Signed Tree Head (STH)

Periodically each log SHOULD sign its current tree head information (see Section 4.9) to produce an STH. When a client requests a log's latest STH (see Section 5.2), the log MUST return an STH that is no older than the log's MMD. However, since STHs could be used to mark individual clients (by producing a new STH for each query), a log MUST NOT produce STHs more frequently than its parameters declare (see Section 4.1). In general, there is no need to produce a new STH unless there are new entries in the log; however, in the event that a

log does not accept any submissions during an MMD period, the log MUST sign the same Merkle Tree Hash with a fresh timestamp.

An STH is a "TransItem" structure of type "signed_tree_head_v2", which encapsulates a "SignedTreeHeadDataV2" structure:

```
struct {
    LogID log_id;
    TreeHeadDataV2 tree_head;
    opaque signature<0..2^16-1>;
} SignedTreeHeadDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as described in Section 4.4.

The "timestamp" in "tree_head" 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_head" contains the latest tree head information (see Section 4.9).

"signature" is computed over the "tree_head" field using the signature algorithm declared in the log's parameters (see Section 4.1).

4.11. Merkle Consistency Proofs

To prepare a Merkle Consistency Proof for distribution to clients, the log produces a "TransItem" structure of type "consistency_proof_v2", which encapsulates a "ConsistencyProofDataV2" structure:

```
struct {
    LogID log_id;
    uint64 tree_size_1;
    uint64 tree_size_2;
    NodeHash consistency_path<1..2^16-1>;
} ConsistencyProofDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as described in Section 4.4.

"tree_size_1" is the size of the older tree.

"tree_size_2" is the size of the newer tree.

"consistency_path" is a vector of Merkle Tree nodes proving the consistency of two STHs.

4.12. Merkle Inclusion Proofs

To prepare a Merkle Inclusion Proof for distribution to clients, the log produces a "TransItem" structure of type "inclusion_proof_v2", which encapsulates an "InclusionProofDataV2" structure:

```
struct {
    LogID log_id;
    uint64 tree_size;
    uint64 leaf_index;
    NodeHash inclusion_path<1..2^16-1>;
} InclusionProofDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as described in Section 4.4.

"tree_size" is the size of the tree on which this inclusion proof is based.

"leaf_index" is the 0-based index of the log entry corresponding to this inclusion proof.

"inclusion_path" is a vector of Merkle Tree nodes proving the inclusion of the chosen certificate or precertificate.

4.13. Shutting down a log

Log operators may decide to shut down a log for various reasons, such as deprecation of the signature algorithm. If there are entries in the log for certificates that have not yet expired, simply making TLS clients stop recognizing that log will have the effect of invalidating SCTs from that log. To avoid that, the following actions are suggested:

- Make it known to clients and monitors that the log will be frozen.
- Stop accepting new submissions (the error code "shutdown" should be returned for such requests).
- Once MMD from the last accepted submission has passed and all pending submissions are incorporated, issue a final STH and publish it as one of the log's parameters. Having an STH with a timestamp that is after the MMD has passed from the last SCT issuance allows clients to audit this log regularly without

special handling for the final STH. At this point the log's private key is no longer needed and can be destroyed.

- Keep the log running until the certificates in all of its entries have expired or exist in other logs (this can be determined by scanning other logs or connecting to domains mentioned in the certificates and inspecting the SCTs served).

5. 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 [RFC8259]. 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.

Clients are configured with a base URL for a log and construct URLs for requests by appending suffixes to this base URL. This structure places some degree of restriction on how log operators can deploy these services, as noted in [RFC7320]. However, operational experience with version 1 of this protocol has not indicated that these restrictions are a problem in practice.

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

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

In practice, log servers may include multiple front-end machines. Since it is impractical to keep these machines in perfect sync, errors may occur that are caused by skew between the machines. Where such errors are possible, the front-end will return additional information (as specified below) making it possible for clients to make progress, if progress is possible. Front-ends MUST only serve data that is free of gaps (that is, for example, no front-end will respond with an STH unless it is also able to prove consistency from all log entries logged within that STH).

For example, when a consistency proof between two STHs is requested, the front-end reached may not yet be aware of one or both STHs. In the case where it is unaware of both, it will return the latest STH it is aware of. Where it is aware of the first but not the second, it will return the latest STH it is aware of and a consistency proof from the first STH to the returned STH. The case where it knows the

second but not the first should not arise (see the "no gaps" requirement above).

If the log is unable to process a client's request, it MUST return an HTTP response code of 4xx/5xx (see [RFC7231]), and, in place of the responses outlined in the subsections below, the body SHOULD be a JSON Problem Details Object (see [RFC7807] Section 3), containing:

type: A URN reference identifying the problem. To facilitate automated response to errors, this document defines a set of standard tokens for use in the "type" field, within the URN namespace of: "urn:ietf:params:trans:error:".

detail: A human-readable string describing the error that prevented the log from processing the request, ideally with sufficient detail to enable the error to be rectified.

e.g., In response to a request of "/ct/v2/get-entries?start=100&end=99", the log would return a "400 Bad Request" response code with a body similar to the following:

```
{
    "type": "urn:ietf:params:trans:error:endBeforeStart",
    "detail": "'start' cannot be greater than 'end'"
}
```

Most error types are specific to the type of request and are defined in the respective subsections below. The one exception is the "malformed" error type, which indicates that the log server could not parse the client's request because it did not comply with this document:

type	detail
malformed	The request could not be parsed.

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 [RFC7231], 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.

5.1. Submit Entry to Log

POST https://<log server>/ct/v2/submit-entry

Inputs:

submission: The base64 encoded certificate or precertificate.

type: The "VersionedTransType" integer value that indicates the type of the "submission": 1 for "x509_entry_v2", or 2 for "precert_entry_v2".

chain: An array of zero or more base64 encoded CA certificates. The first element is the certifier of the "submission"; the second certifies the first; etc. The last element of "chain" (or, if "chain" is an empty array, the "submission") is certified by an accepted trust anchor.

Outputs:

sct: A base64 encoded "TransItem" of type "x509_sct_v2" or "precert_sct_v2", signed by this log, that corresponds to the "submission".

If the submitted entry is immediately appended to (or already exists in) this log's tree, then the log SHOULD also output:

sth: A base64 encoded "TransItem" of type "signed_tree_head_v2", signed by this log.

inclusion: A base64 encoded "TransItem" of type "inclusion_proof_v2" whose "inclusion_path" array of Merkle Tree nodes proves the inclusion of the "submission" in the returned "sth".

Error codes:

type	detail
badSubmission	"submission" is neither a valid certificate nor a valid precertificate.
badType	"type" is neither 1 nor 2.
badChain	The first element of "chain" is not the certifier of the "submission", or the second element does not certify the first, etc.
badCertificate	One or more certificates in the "chain" are not valid (e.g., not properly encoded).
unknownAnchor	The last element of "chain" (or, if "chain" is an empty array, the "submission") both is not, and is not certified by, an accepted trust anchor.
shutdown	The log is no longer accepting submissions.

If the version of "sct" is not v2, then a v2 client may be unable to verify the signature. It MUST NOT construe this as an error. This is to avoid forcing an upgrade of compliant v2 clients that do not use the returned SCTs.

If a log detects bad encoding in a chain that otherwise verifies correctly then the log MUST either log the certificate or return the "bad certificate" error. If the certificate is logged, an SCT MUST be issued. Logging the certificate is useful, because monitors (Section 8.2) can then detect these encoding errors, which may be accepted by some TLS clients.

If "submission" is an accepted trust anchor whose certifier is neither an accepted trust anchor nor the first element of "chain", then the log MUST return the "unknown anchor" error. A log cannot generate an SCT for a submission if it does not have access to the issuer's public key.

If the returned "sct" is intended to be provided to TLS clients, then "sth" and "inclusion" (if returned) SHOULD also be provided to TLS clients (e.g., if "type" was 2 (for "precert_sct_v2") then all three "TransItem"s could be embedded in the certificate).

5.2. Retrieve Latest Signed Tree Head

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

No inputs.

Outputs:

sth: A base64 encoded "TransItem" of type "signed_tree_head_v2", signed by this log, that is no older than the log's MMD.

5.3. Retrieve Merkle Consistency Proof between Two Signed Tree Heads

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

Inputs:

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

second: The tree_size of the newer tree, in decimal (optional).

Both tree sizes must be from existing v2 STHs. However, because of skew, the receiving front-end may not know one or both of the existing STHs. If both are known, then only the "consistency" output is returned. If the first is known but the second is not (or has been omitted), then the latest known STH is returned, along with a consistency proof between the first STH and the latest. If neither are known, then the latest known STH is returned without a consistency proof.

Outputs:

consistency: A base64 encoded "TransItem" of type "consistency_proof_v2", whose "tree_size_1" MUST match the "first" input. If the "sth" output is omitted, then "tree_size_2" MUST match the "second" input. If "first" and "second" are equal and correspond to a known STH, the returned consistency proof MUST be empty (a "consistency_path" array with zero elements).

sth: A base64 encoded "TransItem" of type "signed_tree_head_v2", signed by this log.

Note that no signature is required for the "consistency" output as it is used to verify the consistency between two STHs, which are signed.

Error codes:

type	detail
firstUnknown	"first" is before the latest known STH but is not from an existing STH.
secondUnknown	"second" is before the latest known STH but is not from an existing STH.
secondBeforeFirst	"second" is smaller than "first".

See Section 2.1.4.2 for an outline of how to use the "consistency" output.

5.4. Retrieve Merkle Inclusion Proof from Log by Leaf Hash

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

Inputs:

hash: A base64 encoded v2 leaf hash.

tree_size: The tree_size of the tree on which to base the proof, in decimal.

The "hash" must be calculated as defined in Section 4.7. The "tree_size" must designate an existing v2 STH. Because of skew, the front-end may not know the requested STH. In that case, it will return the latest STH it knows, along with an inclusion proof to that STH. If the front-end knows the requested STH then only "inclusion" is returned.

Outputs:

inclusion: A base64 encoded "TransItem" of type "inclusion_proof_v2" whose "inclusion_path" array of Merkle Tree nodes proves the inclusion of the chosen certificate in the selected STH.

sth: A base64 encoded "TransItem" of type "signed_tree_head_v2", signed by this log.

Note that no signature is required for the "inclusion" output as it is used to verify inclusion in the selected STH, which is signed.

Error codes:

type	detail
hashUnknown	"hash" is not the hash of a known leaf (may be caused by skew or by a known certificate not yet merged).
treeSizeUnknown	"hash" is before the latest known STH but is not from an existing STH.

See Section 2.1.3.2 for an outline of how to use the "inclusion" output.

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

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

Inputs:

hash: A base64 encoded v2 leaf hash.

tree_size: The tree_size of the tree on which to base the proofs, in decimal.

The "hash" must be calculated as defined in Section 4.7. The "tree_size" must designate an existing v2 STH.

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

Case	Response
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 5.3)
index of requested hash < latest STH	Return "inclusion"

Note that more than one case can be true, in which case the returned data is their union. It is also possible for none to be true, in which case the front-end MUST return an empty response.

Outputs:

inclusion: A base64 encoded "TransItem" of type "inclusion_proof_v2" whose "inclusion_path" array of Merkle Tree nodes proves the inclusion of the chosen certificate in the returned STH.

sth: A base64 encoded "TransItem" of type "signed_tree_head_v2", signed by this log.

consistency: A base64 encoded "TransItem" of type "consistency_proof_v2" that proves the consistency of the requested STH and the returned STH.

Note that no signature is required for the "inclusion" or "consistency" outputs as they are used to verify inclusion in and consistency of STHs, which are signed.

Errors are the same as in Section 5.4.

See Section 2.1.3.2 for an outline of how to use the "inclusion" output, and see Section 2.1.4.2 for an outline of how to use the "consistency" output.

5.6. Retrieve Entries and STH from Log

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

Inputs:

start: 0-based index of first entry to retrieve, in decimal.

end: 0-based index of last entry to retrieve, in decimal.

Outputs:

entries: An array of objects, each consisting of

log_entry: The base64 encoded "TransItem" structure of type "x509_entry_v2" or "precert_entry_v2" (see Section 4.3).

submitted_entry: JSON object representing the inputs that were submitted to "submit-entry", with the addition of the trust

anchor to the "chain" field if the submission did not include it.

sct: The base64 encoded "TransItem" of type "x509_sct_v2" or "precert_sct_v2" corresponding to this log entry.

sth: A base64 encoded "TransItem" of type "signed_tree_head_v2", signed by this log.

Note that this message is not signed -- the "entries" data can be verified by constructing the Merkle Tree Hash corresponding to a retrieved STH. All leaves MUST be v2. However, a compliant v2 client MUST NOT construe an unrecognized TransItem type as an error. This means it may be unable to parse some entries, but note that each client can inspect the entries it does recognize as well as verify the integrity of the data by treating unrecognized leaves as opaque input to the tree.

The "start" and "end" parameters SHOULD be within the range 0 <= x < "tree_size" as returned by "get-sth" in Section 5.2.

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

Each "submitted_entry" output parameter MUST include the trust anchor that the log used to verify the "submission", even if that trust anchor was not provided to "submit-entry" (see Section 5.1). If the "submission" does not certify itself, then the first element of "chain" MUST be present and MUST certify the "submission".

Log servers MUST honor requests where 0 <= "start" < "tree_size" and "end" >= "tree_size" by returning a partial response covering only the valid entries in the specified range. "end" >= "tree_size" could be caused by skew. Note that the following restriction may also apply:

Logs MAY restrict the number of entries that can be retrieved per "get-entries" request. If a client requests more than the permitted number of entries, the log SHALL return the maximum number of entries permissible. These entries SHALL be sequential beginning with the entry specified by "start".

Because of skew, it is possible the log server will not have any entries between "start" and "end". In this case it MUST return an empty "entries" array.

In any case, the log server MUST return the latest STH it knows about.

See Section 2.1.2 for an outline of how to use a complete list of "log_entry" entries to verify the "root_hash".

Error codes:

type	detail
startUnknown	"start" is greater than the number of entries in the Merkle tree.
endBeforeStart	"start" cannot be greater than "end".

5.7. Retrieve Accepted Trust Anchors

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

No inputs.

Outputs:

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

max_chain_length: 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.

6. TLS Servers

CT-using TLS servers MUST use at least one of the three mechanisms listed below to present one or more SCTs from one or more logs to each TLS client during full TLS handshakes, where each SCT corresponds to the server certificate. They SHOULD also present corresponding inclusion proofs and STHs.

Three mechanisms are provided because they have different tradeoffs.

- A TLS extension (Section 4.2 of [RFC8446]) with type "transparency_info" (see Section 6.4). This mechanism allows TLS servers to participate in CT without the cooperation of CAs, unlike the other two mechanisms. It also allows SCTs and inclusion proofs to be updated on the fly.

- An Online Certificate Status Protocol (OCSP) [RFC6960] response extension (see Section 7.1.1), where the OCSP response is provided

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

- An X509v3 certificate extension (see Section 7.1.2). This mechanism allows the use of unmodified TLS servers, but the SCTs and inclusion proofs cannot be updated on the fly. Since the logs from which the SCTs and inclusion proofs originated won't necessarily be accepted by TLS clients for the full lifetime of the certificate, there is a risk that TLS clients will subsequently consider the certificate to be non-compliant and in need of re-issuance.

6.1. Multiple SCTs

CT-using TLS servers SHOULD send SCTs from multiple logs, because:

- One or more logs may not have become acceptable to all CT-using TLS clients.

- If a CA and a log collude, it is possible to temporarily hide misissuance from clients. When a TLS client requires SCTs from multiple logs to be provided, it is more difficult to mount this attack.

- If a log misbehaves or suffers a key compromise, a consequence may be that clients cease to trust it. Since the time an SCT may be in use can be considerable (several years is common in current practice when embedded in a certificate), including SCTs from multiple logs reduces the probability of the certificate being rejected by TLS clients.

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

To select the logs from which to obtain SCTs, a TLS server can, for example, examine the set of logs popular TLS clients accept and recognize.

6.2. TransItemList Structure

Multiple SCTs, inclusion proofs, and indeed "TransItem" structures of any type, are combined into a list as follows:

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

struct {
    SerializedTransItem trans_item_list<1..2^16-1>;
} TransItemList;
```

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

6.3. Presenting SCTs, inclusions proofs and STHs

In each "TransItemList" that is sent to a client during a TLS handshake, the TLS server MUST include a "TransItem" structure of type "x509_sct_v2" or "precert_sct_v2" (except as described in Section 6.5).

Presenting inclusion proofs and STHs in the TLS handshake helps to protect the client's privacy (see Section 8.1.4) and reduces load on log servers. Therefore, if the TLS server can obtain them, it SHOULD also include "TransItem"s of type "inclusion_proof_v2" and "signed_tree_head_v2" in the "TransItemList".

6.4. transparency_info TLS Extension

Provided that a TLS client includes the "transparency_info" extension type in the ClientHello and the TLS server supports the "transparency_info" extension:

- The TLS server MUST verify that the received "extension_data" is empty.

- The TLS server MUST construct a "TransItemList" of relevant "TransItem"s (see Section 6.3), which SHOULD omit any "TransItem"s that are already embedded in the server certificate or the stapled OCSP response (see Section 7.1). If the constructed "TransItemList" is not empty, then the TLS server MUST include the "transparency_info" extension with the "extension_data" set to this "TransItemList".

TLS servers MUST only include this extension in the following messages:

- the ServerHello message (for TLS 1.2 or earlier).
- the Certificate or CertificateRequest message (for TLS 1.3).

TLS servers MUST NOT process or include this extension when a TLS session is resumed, since session resumption uses the original session information.

6.5. cached_info TLS Extension

When a TLS server includes the "transparency_info" extension, it SHOULD NOT include any "TransItem" structures of type "x509_sct_v2" or "precert_sct_v2" in the "TransItemList" if all of the following conditions are met:

- The TLS client includes the "cached_info" ([RFC7924]) extension type in the ClientHello, with a "CachedObject" of type "ct_compliant" (see Section 8.1.7) and at least one "CachedObject" of type "cert".
- The TLS server sends a modified Certificate message (as described in section 4.1 of [RFC7924]).

If the "hash_value" of any "CachedObject" of type "ct_compliant" sent by a TLS client is not 1 byte long with the value 0, the CT-using TLS server MUST abort the handshake.

7. Certification Authorities

7.1. Transparency Information X.509v3 Extension

The Transparency Information X.509v3 extension, which has OID 1.3.101.75 and SHOULD be non-critical, contains one or more "TransItem" structures in a "TransItemList". This extension MAY be included in OCSP responses (see Section 7.1.1) and certificates (see Section 7.1.2). Since RFC5280 requires the "extnValue" field (an OCTET STRING) of each X.509v3 extension to include the DER encoding of an ASN.1 value, a "TransItemList" MUST NOT be included directly. Instead, it MUST be wrapped inside an additional OCTET STRING, which is then put into the "extnValue" field:

```
TransparencyInformationSyntax ::= OCTET STRING
```

"TransparencyInformationSyntax" contains a "TransItemList".

7.1.1. OCSP Response Extension

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

7.1.2. Certificate Extension

A certification authority MAY include a Transparency Information X.509v3 extension in a certificate. All included SCTs and inclusion proofs MUST be for a precertificate that corresponds to this certificate.

7.2. TLS Feature X.509v3 Extension

A certification authority SHOULD NOT issue any certificate that identifies the "transparency_info" TLS extension in a TLS feature extension [RFC7633], because TLS servers are not required to support the "transparency_info" TLS extension in order to participate in CT (see Section 6).

8. Clients

There are various different functions clients of logs might perform. We describe here some typical clients and how they should function. Any inconsistency may be used as evidence that a log has not behaved correctly, and the signatures on the data structures prevent the log from denying that misbehavior.

All clients need various parameters in order to communicate with logs and verify their responses. These parameters are described in Section 4.1, but note that this document does not describe how the parameters are obtained, which is implementation-dependent (see, for example, [Chromium.Policy]).

8.1. TLS Client

8.1.1. Receiving SCTs and inclusion proofs

TLS clients receive SCTs and inclusion proofs alongside or in certificates. CT-using TLS clients MUST implement all of the three mechanisms by which TLS servers may present SCTs (see Section 6).

TLS clients that support the "transparency_info" TLS extension (see Section 6.4) SHOULD include it in ClientHello messages, with empty "extension_data". If a TLS server includes the "transparency_info"

TLS extension when resuming a TLS session, the TLS client MUST abort the handshake.

8.1.2. Reconstructing the TBSCertificate

Validation of an SCT for a certificate (where the "type" of the "TransItem" is "x509_sct_v2") uses the unmodified TBSCertificate component of the certificate.

Before an SCT for a precertificate (where the "type" of the "TransItem" is "precert_sct_v2") can be validated, the TBSCertificate component of the precertificate needs to be reconstructed from the TBSCertificate component of the certificate as follows:

- Remove the Transparency Information extension (see Section 7.1).
- Remove embedded v1 SCTs, identified by OID 1.3.6.1.4.1.11129.2.4.2 (see section 3.3 of [RFC6962]). This allows embedded v1 and v2 SCTs to co-exist in a certificate (see Appendix A).

8.1.3. Validating SCTs

In addition to normal validation of the server certificate and its chain, CT-using TLS clients MUST validate each received SCT for which they have the corresponding log's parameters. To validate an SCT, a TLS client computes the signature input by constructing a "TransItem" of type "x509_entry_v2" or "precert_entry_v2", depending on the SCT's "TransItem" type. The "TimestampedCertificateEntryDataV2" structure is constructed in the following manner:

- "timestamp" is copied from the SCT.
- "tbs_certificate" is the reconstructed TBSCertificate portion of the server certificate, as described in Section 8.1.2.
- "issuer_key_hash" is computed as described in Section 4.7.
- "sct_extensions" is copied from the SCT.

The SCT's "signature" is then verified using the public key of the corresponding log, which is identified by the "log_id". The required signature algorithm is one of the log's parameters.

8.1.4. Fetching inclusion proofs

When a TLS client has validated a received SCT but does not yet possess a corresponding inclusion proof, the TLS client MAY request

the inclusion proof directly from a log using "get-proof-by-hash" (Section 5.4) or "get-all-by-hash" (Section 5.5).

Note that fetching inclusion proofs directly from a log will disclose to the log which TLS server the client has been communicating with. This may be regarded as a significant privacy concern, and so it is preferable for the TLS server to send the inclusion proofs (see Section 6.3).

8.1.5. Validating inclusion proofs

When a TLS client has received, or fetched, an inclusion proof (and an STH), it SHOULD proceed to verifying the inclusion proof to the provided STH. The TLS client SHOULD also verify consistency between the provided STH and an STH it knows about.

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 5.2), then verify it by requesting a consistency proof (Section 5.3). Note that if the TLS client uses "get-all-by-hash", then it will already have the new STH.

8.1.6. Evaluating compliance

It is up to a client's local policy to specify the quantity and form of evidence (SCTs, inclusion proofs or a combination) needed to achieve compliance and how to handle non-compliance.

A TLS client can only evaluate compliance if it has given the TLS server the opportunity to send SCTs and inclusion proofs by any of the three mechanisms that are mandatory to implement for CT-using TLS clients (see Section 8.1.1). Therefore, a TLS client MUST NOT evaluate compliance if it did not include both the "transparency_info" and "status_request" TLS extensions in the ClientHello.

8.1.7. cached_info TLS Extension

If a TLS client uses the "cached_info" TLS extension ([RFC7924]) to indicate 1 or more cached certificates, all of which it already considers to be CT compliant, the TLS client MAY also include a "CachedObject" of type "ct_compliant" in the "cached_info" extension. Its "hash_value" field MUST have the value 0 and be 1 byte long (the minimum length permitted by [RFC7924]).

8.2. Monitor

Monitors watch logs to check that they behave correctly, for certificates of interest, or both. For example, a monitor may be configured to report on all certificates that apply to a specific domain name when fetching new entries for consistency validation.

A monitor MUST at least inspect every new entry in every log it watches, and it MAY also choose to keep copies of entire logs.

To inspect all of the existing entries, the monitor SHOULD follow these steps once for each log:

1. Fetch the current STH (Section 5.2).

2. Verify the STH signature.

3. Fetch all the entries in the tree corresponding to the STH (Section 5.6).

4. If applicable, check each entry to see if it’s a certificate of interest.

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

To inspect new entries, the monitor SHOULD follow these steps repeatedly for each log:

1. Fetch the current STH (Section 5.2). Repeat until the STH changes.

2. Verify the STH signature.

3. Fetch all the new entries in the tree corresponding to the STH (Section 5.6). If they remain unavailable for an extended period, then this should be viewed as misbehavior on the part of the log.

4. If applicable, check each entry to see if it’s a certificate of interest.

5. 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 5.3).

2. Verify the consistency proof.

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

6. Repeat from step 1.

8.3. Auditing

Auditing ensures that the current published state of a log is reachable from previously published states that are known to be good, and that the promises made by the log in the form of SCTs have been kept. Audits are performed by monitors or TLS clients.

In particular, there are four log behavior properties that should be checked:

- The Maximum Merge Delay (MMD).
- The STH Frequency Count.
- The append-only property.
- The consistency of the log view presented to all query sources.

A benign, conformant log publishes a series of STHs over time, each derived from the previous STH and the submitted entries incorporated into the log since publication of the previous STH. This can be proven through auditing of STHs. SCTs returned to TLS clients can be audited by verifying against the accompanying certificate, and using Merkle Inclusion Proofs, against the log's Merkle tree.

The action taken by the auditor if an audit fails is not specified, but note that in general if audit fails, the auditor is in possession of signed proof of the log's misbehavior.

A monitor (Section 8.2) 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 8.1) 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 5.4). It can also verify that the SCT corresponds to the server certificate it arrived

with (i.e., the log entry is that certificate, or is a precertificate corresponding to that certificate).

Checking of the consistency of the log view presented to all entities is more difficult to perform because it requires a way to share log responses among a set of CT-using entities, and is discussed in Section 11.3.

9. Algorithm Agility

It is not possible for a log to change any of its algorithms part way through its lifetime:

Signature algorithm: SCT signatures must remain valid so signature algorithms can only be added, not removed.

Hash algorithm: A log would have to support the old and new hash algorithms to allow backwards-compatibility with clients that are not aware of a hash algorithm change.

Allowing multiple signature or hash algorithms for a log would require that all data structures support it and would significantly complicate client implementation, which is why it is not supported by this document.

If it should become necessary to deprecate an algorithm used by a live log, then the log MUST be frozen as specified in Section 4.13 and a new log SHOULD be started. Certificates in the frozen log that have not yet expired and require new SCTs SHOULD be submitted to the new log and the SCTs from that log used instead.

10. IANA Considerations

The assignment policy criteria mentioned in this section refer to the policies outlined in [RFC8126].

10.1. New Entry to the TLS ExtensionType Registry

IANA is asked to add an entry for "transparency_info(TBD)" to the "TLS ExtensionType Values" registry defined in [RFC8446], setting the "Recommended" value to "Y", setting the "TLS 1.3" value to "CH, CR, CT", and citing this document as the "Reference".

10.2. New Entry to the TLS CachedInformationType registry

IANA is asked to add an entry for "ct_compliant(TBD)" to the "TLS CachedInformationType Values" registry defined in [RFC7924], citing this document as the "Reference".

10.3. Hash Algorithms

IANA is asked to establish a registry of hash algorithm values, named "CT Hash Algorithms", that initially consists of:

Value	Hash Algorithm	OID	Reference / Assignment Policy
0x00	SHA-256	2.16.840.1.101.3.4.2.1	[RFC6234]
0x01 - 0xDF	Unassigned		Specification Required and Expert Review
0xE0 - 0xEF	Reserved		Experimental Use
0xF0 - 0xFF	Reserved		Private Use

10.3.1. Expert Review guidelines

The appointed Expert should ensure that the proposed algorithm has a public specification and is suitable for use as a cryptographic hash algorithm with no known preimage or collision attacks. These attacks can damage the integrity of the log.

10.4. Signature Algorithms

IANA is asked to establish a registry of signature algorithm values, named "CT Signature Algorithms", that initially consists of:

SignatureScheme Value	Signature Algorithm	Reference / Assignment Policy
ecdsa_secp256r1_sha256(0x0403)	ECDSA (NIST P-256) with SHA-256	[FIPS186-4]
ecdsa_secp256r1_sha256(0x0403)	Deterministic ECDSA (NIST P-256) with HMAC-SHA256	[RFC6979]
ed25519(0x0807)	Ed25519 (PureEdDSA with the edwards25519 curve)	[RFC8032]
private_use(0xFE00..0xFFFF)	Reserved	Private Use

10.4.1. Expert Review guidelines

The appointed Expert should ensure that the proposed algorithm has a public specification, has a value assigned to it in the TLS SignatureScheme Registry (that IANA is asked to establish in [RFC8446]) and is suitable for use as a cryptographic signature algorithm.

10.5. VersionedTransTypes

IANA is asked to establish a registry of "VersionedTransType" values, named "CT VersionedTransTypes", that initially consists of:

Value	Type and Version	Reference / Assignment Policy
0x0000	Reserved	[RFC6962] (*)
0x0001	x509_entry_v2	RFCXXXX
0x0002	precert_entry_v2	RFCXXXX
0x0003	x509_sct_v2	RFCXXXX
0x0004	precert_sct_v2	RFCXXXX
0x0005	signed_tree_head_v2	RFCXXXX
0x0006	consistency_proof_v2	RFCXXXX
0x0007	inclusion_proof_v2	RFCXXXX
0x0008 - 0xDFFF	Unassigned	Specification Required and Expert Review
0xE000 - 0xEFFF	Reserved	Experimental Use
0xF000 - 0xFFFF	Reserved	Private Use

(*) The 0x0000 value is reserved so that v1 SCTs are distinguishable from v2 SCTs and other "TransItem" structures.

[RFC Editor: please update 'RFCXXXX' to refer to this document, once its RFC number is known.]

10.5.1. Expert Review guidelines

The appointed Expert should review the public specification to ensure that it is detailed enough to ensure implementation interoperability.

10.6. Log Artifact Extension Registry

IANA is asked to establish a registry of "ExtensionType" values, named "CT Log Artifact Extensions", that initially consists of:

ExtensionType	Status	Use	Reference / Assignment Policy
0x0000 - 0xDFFF	Unassigned	n/a	Specification Required and Expert Review
0xE000 - 0xEFFF	Reserved	n/a	Experimental Use
0xF000 - 0xFFFF	Reserved	n/a	Private Use

The "Use" column should contain one or both of the following values:

- "SCT", for extensions specified for use in Signed Certificate Timestamps.

- "STH", for extensions specified for use in Signed Tree Heads.

10.6.1. Expert Review guidelines

The appointed Expert should review the public specification to ensure that it is detailed enough to ensure implementation interoperability. The Expert should also verify that the extension is appropriate to the contexts in which it is specified to be used (SCT, STH, or both).

10.7. Object Identifiers

This document uses object identifiers (OIDs) to identify Log IDs (see Section 4.4), the precertificate CMS "eContentType" (see Section 3.2), and X.509v3 extensions in certificates (see Section 7.1.2) and OCSP responses (see Section 7.1.1). The OIDs are defined in an arc that was selected due to its short encoding.

10.7.1. Log ID Registry

IANA is asked to establish a registry of Log IDs, named "CT Log ID Registry", that initially consists of:

Value	Log	Reference / Assignment Policy
1.3.101.8192 - 1.3.101.16383	Unassigned	Parameters Required and First Come First Served
1.3.101.80.0 - 1.3.101.80.*	Unassigned	Parameters Required and First Come First Served

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

The 1.3.101.80 arc has been delegated. This is an unlimited resource, but only the 128 OIDs from 1.3.101.80.0 to 1.3.101.80.127 have an encoded length of only 4 octets.

Each application for the allocation of a Log ID should be accompanied by all of the required parameters (except for the Log ID) listed in Section 4.1.

11. Security Considerations

With CAs, logs, and servers performing the actions described here, TLS clients can use logs and signed timestamps to reduce the likelihood that they will accept misissued certificates. If a server presents a valid signed timestamp for a certificate, then the client knows that a log has committed to publishing the certificate. From this, the client knows that monitors acting for the subject of the certificate have had some time to notice the misissuance and take some action, such as asking a CA to revoke a misissued certificate. A signed timestamp does not guarantee this though, since appropriate monitors might not have checked the logs or the CA might have refused to revoke the certificate.

In addition, if TLS clients will not accept unlogged certificates, then site owners will have a greater incentive to submit certificates to logs, possibly with the assistance of their CA, increasing the overall transparency of the system.

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

11.1. Misissued Certificates

Misissued certificates that have not been publicly logged, and thus do not have a valid SCT, are not considered compliant. 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. Since a log is allowed to serve an STH of any age up to the MMD, the maximum period of time during which a misissued certificate can be used without being available for audit is twice the MMD.

11.2. 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.

11.3. Misbehaving Logs

A log can misbehave in several ways. Examples include: failing to incorporate a certificate with an SCT in the Merkle Tree within the MMD; presenting different, conflicting views of the Merkle Tree at different times and/or to different parties; issuing STHs too frequently; mutating the signature of a logged certificate; and failing to present a chain containing the certifier of a logged certificate. Such misbehavior is detectable and [I-D.ietf-trans-threat-analysis] provides more details on how this can be done.

Violation of the MMD contract is detected by log clients requesting a Merkle inclusion proof (Section 5.4) for each observed SCT. These checks can be asynchronous and need only be done once per certificate. However, note that there may be privacy concerns (see Section 8.1.4).

Violation of the append-only property or the STH issuance rate limit can be detected by clients comparing their instances of the Signed Tree Heads. There are various ways this could be done, for example via gossip (see [I-D.ietf-trans-gossip]) or peer-to-peer communications or by sending STHs to monitors (who could then directly check against their own copy of the relevant log). Proof of misbehavior in such cases would be: a series of STHs that were issued too closely together, proving violation of the STH issuance rate limit; or an STH with a root hash that does not match the one calculated from a copy of the log, proving violation of the append-only property.

11.4. Preventing Tracking Clients

Clients that gossip STHs or report back SCTs can be tracked or traced if a log produces multiple STHs or SCTs with the same timestamp and data but different signatures. Logs SHOULD mitigate this risk by either:

- Using deterministic signature schemes, or
- Producing no more than one SCT for each distinct submission and no more than one STH for each distinct tree_size. Each of these SCTs and STHs can be stored by the log and served to other clients that submit the same certificate or request the same STH.

11.5. Multiple SCTs

By requiring TLS servers to offer multiple SCTs, each from a different log, TLS clients reduce the effectiveness of an attack where a CA and a log collude (see Section 6.1).

12. Acknowledgements

The authors would like to thank Erwann Abelea, Robin Alden, Andrew Ayer, Richard Barnes, Al Cutter, David Drysdale, Francis Dupont, Adam Eijdenberg, Stephen Farrell, Daniel Kahn Gillmor, Paul Hadfield, Brad Hill, Jeff Hodges, Paul Hoffman, Jeffrey Hutzelman, Kat Joyce, Stephen Kent, SM, Alexey Melnikov, Linus Nordberg, Chris Palmer, Trevor Perrin, Pierre Phaneuf, Eric Rescorla, Melinda Shore, Ryan Sleevi, Martin Smith, Carl Wallace and Paul Wouters for their valuable contributions.

A big thank you to Symantec for kindly donating the OIDs from the 1.3.101 arc that are used in this document.

13. References

13.1. Normative References

[FIPS186-4] NIST, "FIPS PUB 186-4", July 2013, <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.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, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[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, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

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

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

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

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <https://www.rfc-editor.org/info/rfc7807>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[UNIXTIME] IEEE, "The Open Group Base Specifications Issue 7 IEEE Std 1003.1-2008, 2016 Edition", n.d., <http://pubs.opengroup.org/onlinepubs/9699919799.2016edition/basedefs/V1_chap04.html#tag_04_16>.

13.2. 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>.

[I-D.ietf-trans-gossip] Nordberg, L., Gillmor, D., and T. Ritter, "Gossiping in CT", draft-ietf-trans-gossip-05 (work in progress), January 2018.

[I-D.ietf-trans-threat-analysis] Kent, S., "Attack and Threat Model for Certificate Transparency", draft-ietf-trans-threat-analysis-16 (work in progress), October 2018.

[JSON.Metadata]
The Chromium Projects, "Chromium Log Metadata JSON Schema", 2014, <https://www.gstatic.com/ct/log_list/log_list_schema.json>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7320] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 7320, DOI 10.17487/RFC7320, July 2014, <https://www.rfc-editor.org/info/rfc7320>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

Appendix A. Supporting v1 and v2 simultaneously

Certificate Transparency logs have to be either v1 (conforming to [RFC6962]) or v2 (conforming to this document), as the data structures are incompatible and so a v2 log could not issue a valid v1 SCT.

CT clients, however, can support v1 and v2 SCTs, for the same certificate, simultaneously, as v1 SCTs are delivered in different TLS, X.509 and OCSP extensions than v2 SCTs.

v1 and v2 SCTs for X.509 certificates can be validated independently. For precertificates, v2 SCTs should be embedded in the TBSCertificate before submission of the TBSCertificate (inside a v1 precertificate, as described in Section 3.1. of [RFC6962]) to a v1 log so that TLS clients conforming to [RFC6962] but not this document are oblivious to the embedded v2 SCTs. An issuer can follow these steps to produce an X.509 certificate with embedded v1 and v2 SCTs:

- Create a CMS precertificate as described in Section 3.2 and submit it to v2 logs.
- Embed the obtained v2 SCTs in the TBSCertificate, as described in Section 7.1.2.
- Use that TBSCertificate to create a v1 precertificate, as described in Section 3.1. of [RFC6962] and submit it to v1 logs.
- Embed the v1 SCTs in the TBSCertificate, as described in Section 3.3 of [RFC6962].
- Sign that TBSCertificate (which now contains v1 and v2 SCTs) to issue the final X.509 certificate.

Authors' Addresses

Ben Laurie
Google UK Ltd.

Email: benl@google.com

Adam Langley
Google Inc.

Email: agl@google.com

Emilia Kasper
Google Switzerland GmbH

Email: ekasper@google.com

Eran Messeri
Google UK Ltd.

Email: eranm@google.com

Rob Stradling
Sectigo Ltd.

Email: rob@sectigo.com

Public Notary Transparency
Internet Draft
Expires: December 2015
Intended Status: Informational

Kent, S.
BBN Technologies

June 2015

Attack Model for Certificate Transparency
draft-ietf-trans-threat-analysis-01

Abstract

This document describes an attack model for the Web PKI context in which security mechanisms to detect mis-issuance of web site certificates will be developed. The model provides an analysis of detection and remediation mechanisms for both syntactic and semantic mis-issuance. The model introduces an outline of attacks to organize the discussion. The model also describes the roles played by the elements of the Certificate Transparency (CT) system, to establish a context for the model.

Table of Contents

1. Introduction

Certificate transparency (CT) is a set of mechanisms designed to detect, deter, and facilitate remediation of certificate mis-issuance. The term certificate mis-issuance is defined here to encompass violations of either semantic or syntactic constraints. The fundamental semantic constraint for a certificate is that it was issued to an entity that is authorized to represent the Subject (or Subject Alternative) named in the certificate. (It is also assumed that the entity requested the certificate from the CA that issued it.) Throughout the remainder of this document we refer to a semantically mis-issued certificate as "bogus.")

A certificate is characterized as syntactically mis-issued if it violates syntax constraints associated with the class of certificate that it purports to represent. Syntax constraints for certificates are established by certificate profiles, and typically are application-specific. For example, certificates used in the Web PKI environment might be characterized as domain validation (DV) or extended validation (EV) certificates. Certificates used with applications such as IPsec or S/MIME have different syntactic constraints from those in the Web PKI context.

There are two classes of beneficiaries of CT: certificate Subjects and relying parties (RPs). In the initial focus context of CT, the Web PKI, Subjects are web sites and RPs are browsers employing HTTPS to access these web sites.

A certificate Subject benefits from CT because CT helps detect certificates that have been mis-issued in the name of that Subject. A Subject learns of a bogus certificate (issued in its name), via the Monitor function of CT. The Monitor function may be provided by the Subject itself, i.e., self-monitoring, or by a third party trusted by the Subject. When a Subject is informed of certificate mis-issuance by a Monitor, the Subject is expected to request/demand revocation of the bogus certificate. Revocation of a bogus certificate is the primary means of remedying mis-issuance. (If a browser vendor distributes a "blacklist" of bogus certificates or CAs that mis-issued certificates and modifies the browser to reject any certificates on the list or for which the issuing CA is on the list, this too remedies mis-issuance. Throughout the remainder of this document references to certificate revocation as a remedy encompass this and analogous forms of browser behavior, if available. Note: there are no IETF standards defining a browser blacklist capability.)

Note that a Subject can benefit from the Monitor function of CT even if the Subject’s certificate has not been logged. Monitoring of logs for certificates issued in the Subject’s name suffices to detect mis-issuance targeting the Subject, if the bogus certificate is logged.

A relying party (e.g., browser) benefits from CT if it rejects a bogus certificate, i.e., treats it as invalid. An RP is protected from accepting a bogus certificate if that certificate is revoked, and if the RP checks the revocation status of the certificate. (An RP is also protected if a browser vendor "blacklists" a certificate or a CA as noted above.) An RP also may benefit from CT if the RP validates an SCT associated with a certificate, and rejects the certificate if the SCT is invalid. If an RP verified that a

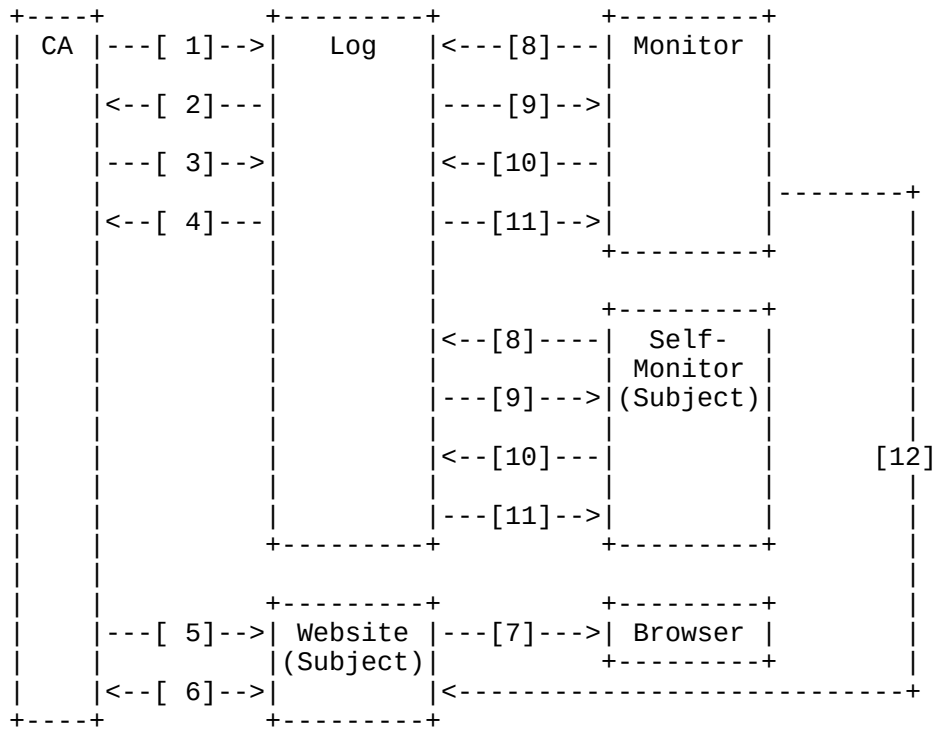
certificate that claims to have been logged has a valid log entry, the RP would have a higher degree of confidence that the certificate is genuine. However, checking logs in this fashion imposes a burden on RPs and on logs. Moreover, the existence of a log entry does not ensure that the certificate is not mis-issued. Unless the certificate Subject is monitoring the log(s) in question, a bogus certificate will not be detected by CT mechanisms. Finally, if an RP were to check logs for individual certificates that would disclose to logs the identity of web sites being visited by the RP, a privacy violation. Thus this attack model assumes that RPs will not check log entries.

Note that all RPs may benefit from CT even if they do nothing with SCTs. If Monitors inform Subjects of mis-issuance, and if a CA revokes a certificate in response to a request from the certificate's legitimate Subject, then an RP benefits without having to implement any CT-specific mechanisms.

Logging [TRANS] is the central element of CT. Logging enables a Monitor to detect a bogus certificate based on reference information provided by the certificate Subject. Logging of certificates is intended to deter mis-issuance, by creating a publicly-accessible record that associates a CA with any certificates that it mis-issues. Logging does not remedy mis-issuance; but it does facilitate remediation by providing the information needed to enable revocation of bogus certificates in some circumstances.

Auditing is a function employed by CT to detect mis-behavior by logs. Auditing is intended to deter mis-issuance that is abetted by misbehaving logs. Auditing detects log mis-behavior by noting inconsistencies between SCTs issued by a log and the log entries published by the log. (A failure to make available log entries in a timely fashion, consistent with the MMD for the log is also a form of misbehavior that can be detected by the Audit function.) Such inconsistencies indicate log misbehavior and suggest that Monitors ought not trust such logs. Thus the Audit function does not detect mis-issuance per se. There is no agreed-upon Audit function design for CT at the time of this writing. As a result, the model merely notes where Auditing is needed to detect certain classes of attacks.

Figure 1 (below) illustrates the data exchanges among the major elements of the CT system, based on the log specification [TRANS] and on the assumed behavior of other CT system elements as described above. This Figure does not include the Audit function, because there is not yet agreement on how that function will work in a distributed, privacy-preserving fashion.



[1] Retrieve accepted root certs
[2] accepted root certs
[3] Add chain to log/add PreCertChain to log
[4] SCT (embedded in pre-cert, if applicable)
[5] send cert + SCTs (or cert with embedded SCTs)
[6] Revocation request/response (in response to detected mis-issuance)
[7] cert + SCTs (or cert with embedded SCTs)
[8] Retrieve entries from Log
[9] returned entries from log
[10] Retrieve latest STH
[11] returned STH
[12] bogus/erroneous cert notification

Figure 1. Data Exchanges Between Major Elements of the CT System

Certificate mis-issuance may arise in one of several ways. The ways by which CT enables a Subject (or others) to detect and redress mis-issuance depends on the context and the entities involved in the mis-issuance. This attack model applies to use of CT in the Web PKI context. If CT is extended to apply to other contexts, each context will require its own attack model, although most elements of the model described here are likely to be applicable.

Because certificates are issued by CAs, the top level differentiation in this analysis is whether the CA that mis-issued a certificate did so maliciously or not. Next, for each scenario, the model considers whether or not the certificate was logged. Scenarios are further differentiated based on whether the logs and monitors are benign or malicious and whether a certificate's Subject is self-monitoring or is using a third party Monitoring service. Finally, the analysis considers whether a browser is performing checking relevant to CT. The scenarios are organized as illustrated by the following outline:

```
Web PKI CA - malicious vs non-malicious
   Certificate - logged vs not logged
         Log - benign vs malicious
         Third party Monitor - benign vs malicious
         Certificate's Subject - self-monitoring (or not)
         Browser - careful (or not)
```

The following sections examine each of these cases. As noted above, the focus here is on the Web PKI context, although most of the analysis is applicable to other PKI contexts.

Conventions used in this document

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 [RFC2119].

2. Semantic mis-issuance

2.1. Non-malicious Web PKI CA context

In this section, we address the case where the CA has no intent to issue a bogus certificate.

A CA may have mis-issued a certificate as a result of an error or, in the case of a bogus certificate, because it was the victim of a

social engineering attack or an attack such as the one that affected DigiNotar [https://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx]. In the case of an error, the CA should have a record of the erroneous certificate and be prepared to revoke this certificate once it has discovered and confirmed the error. In the event of an attack, a CA may have no record of a bogus certificate.

2.1.1. Certificate logged

2.1.1.1. Benign log

The log (or logs) is benign and thus is presumed to provide consistent, accurate responses to requests from all clients.

If a bogus (pre-)certificate has been submitted to one or more logs prior to issuance to acquire an embedded SCT, or post-issuance to acquire a standalone SCT, detection of mis-issuance is the responsibility of a Monitor.

2.1.1.1.1. Self-monitoring Subject

If a Subject is tracking the log(s) to which a certificate was submitted, and is performing self-monitoring, then it will be able to detect a bogus (pre-)certificate and request revocation. In this case, the CA will make use of the log entry (supplied by the Subject) to determine the serial number of the bogus certificate, and investigate/revoke it. (See Sections 4.1, 4.2 and 4.3.)

2.1.1.1.2. Benign third party Monitor

If a benign third party monitor is checking the logs to which a certificate was submitted and is protecting the targeted Subject, it will detect a bogus certificate and will alert the Subject. The Subject, in turn, will ask the CA to revoke the bogus certificate. In this case, the CA will make use of the log entry (supplied by the Subject) to determine the serial number of the bogus certificate, and revoke it (after investigation). (See Sections 4.1, 4.2 and 4.3.)

2.1.1.2. Mis-behaving log

In this case, the bogus (pre-)certificate has been submitted to one or more logs, each of which generate an SCT for the submission. In this scenario, a log probably will suppress a bogus certificate log entry, or it may create an entry for the certificate but report it selectively. (A mis-behaving log also could create and report

entries for bogus certificates that have not been issued by the indicated CA (hereafter called "fake"). Unless a Monitor validates the associated certificate chains up to roots that it trusts, these fake bogus certificates could cause the Monitors to report non-existent semantic problems to the Subject who would in turn report them to the purported issuing CA. This might cause the CA to do needless investigative work or perhaps incorrectly revoke and re-issue the Subject's real certificate. Note that for every certificate submitted to a log, the log MUST verify a complete certificate chain up to one of the roots it accepts. So creating a log entry for a fake bogus certificate marks the log as mis-behaving.)

2.1.1.2.1. Self-monitoring Subject

If a mis-behaving log suppresses a bogus certificate log entry, a Subject performing self-monitoring will not detect the bogus certificate. CT relies on an Audit mechanism to detect log misbehavior, as a deterrent. (It is anticipated that logs that are identified as persistently mis-behaving will cease to be trusted by Monitors and by non-malicious CAs.) It is not clear if such a mechanism will be viable if there are a very large number of self-monitoring Subjects.

2.1.1.2.2. Benign third party Monitor

Because a mis-behaving log will suppress a bogus certificate log entry (or report such entries inconsistently) a benign third party Monitor that is protecting the targeted Subject also will not detect a bogus certificate. In this scenario, CT relies on a distributed Auditing mechanism [gossip] to detect log misbehavior, as a deterrent. (See Section 4.6 below.) However, a Monitor (third-party or self) must participate in the Audit mechanism in order to become aware of log misbehavior.

If the mis-behaving log has logged the bogus certificate when it issuing the associated SCT, it will try to hide this from the Subject (if self-monitoring) or from the Monitor protecting the Subject. It does so by presenting them with a view of its log entries and STH that does not contain the bogus certificate. To other entities, the log presents log entries and an STH that include the bogus certificate. This discrepancy can be detected if there is an exchange of information about the log entries and STH between the entities receiving the view that excludes the bogus certificate and entities that receive a view that includes it, i.e., a distributed Audit mechanism.

If a malicious log does not create an entry for a bogus certificate (for which an SCT has been issued), then any Monitor/Auditor that sees the bogus certificate will detect this when it checks with the log for log entries and STH (see Section 2.1.2.)

2.1.1.3. Mis-behaving third party Monitor

A third party Monitor that mis-behaves will not notify the targeted Subject of a bogus certificate. This is true irrespective of whether the Monitor checks the logs or whether the logs are benign or malicious/conspiring.

Note that independent of any mis-issuance on the part of the CA, a mis-behaving Monitor could issue false warnings to a Subject that it protects. These could cause the Subject to report non-existent semantic problems to the issuing CA and cause the CA to do needless investigative work or perhaps incorrectly revoke and re-issue the Subject’s certificate.

2.1.2. Certificate not logged

If the CA does not submit a pre-certificate to a log, whether a log is benign or mis-behaving does not matter. The same is true if a Subject is issued a certificate without an SCT and does not log the certificate itself, to acquire an SCT. Also, since there is no log entry in this scenario, there is no difference in outcome between a benign and a mis-behaving third party Monitor. In both cases, no Monitor (self or third-party) will detect a bogus certificate based on Monitor functions and there will be no consequent reporting of the problem to the Subject or by the Subject to the CA based on examination of log entries.

2.1.2.1. Self-monitoring Subject

A Subject performing self-monitoring will be able to detect the lack of an embedded SCT in the certificate it received from the CA. The Subject SHOULD notify the CA if the Subject believes that its certificate was supposed to be logged. If the certificate was supposed to be logged, but was not, the CA can use the certificate supplied by the Subject to investigate and remedy the problem. In the context of a benign CA, a failure to log the certificate might be the result of an operations error, or evidence of an attack on the CA.

2.1.2.2. Careful browser

If a browser rejects certificates without SCTs (see Section 4.4.), CAs may be "encouraged" to log the certificates they issue. This, in

turn, would make it easier for Monitors to detect bogus certificates. However, it is not clear how such behavior by browsers can be deployed incrementally throughout the Internet. As a result, this model does not assume that browsers will reject a certificate that is not accompanied by an SCT. In the absence of logging, browsers generally do not benefit from CT, since certificates have to be logged to enable detection of mis-issuance by Monitors, and to trigger subsequent revocation.

2.2. Malicious Web PKI CA context

In this section, we address the scenario in which the mis-issuance is intentional, not due to error. The CA is not the victim but the attacker.

2.2.1. Certificate logged

2.2.1.1. Benign log

A bogus (pre-)certificate may be submitted to one or more benign logs prior to issuance, to acquire an embedded SCT, or post-issuance to acquire a standalone SCT. The log (or logs) replies correctly to requests from clients.

2.2.1.1.1. Self-monitoring Subject

If a Subject is checking the logs to which a certificate was submitted and is performing self-monitoring, it will be able to detect the bogus certificate and will request revocation. The CA may refuse to revoke, or may substantially delay revoking, the bogus certificate. The CA could make excuses about inadequate proof that the certificate is bogus, or argue that it cannot quickly revoke the certificate because of legal concerns, etc. In this case, the CT mechanisms will have detected mis-issuance, but the information logged by CT does not help remedy the problem. (See Sections 3 and 5.)

2.2.1.1.2. Benign third party Monitor

If a benign third party monitor is checking the logs to which a certificate was submitted and is protecting the targeted Subject, it will detect the bogus certificate and will alert the Subject. The Subject will then ask the CA to revoke the bogus certificate. As in 2.2.1.1.1, the CA may or may not revoke the certificate.

2.2.1.2. Mis-behaving log

A bogus (pre-)certificate may have been submitted to one or more logs that are mis-behaving, e.g., conspiring with an attacker. These logs may or may not issue SCTs, but will hide the log entries from some or all Monitors.

2.2.1.2.1. Monitors - third party and self

If log entries are hidden from a Monitor (third party or self), the Monitor will not be able to detect issuance of a bogus certificate.

The Audit function of CT is intended to detect logs that conspire to delay or suppress log entries (potentially selectively), based on consistency checking of logs. (See 2.1.1.2.2.) If a Monitor learns of misbehaving log operation, it alerts the Subjects that it is protecting, so that they no longer acquire SCTs from that log. The Monitor also avoids relying upon such a log in the future. However, unless a distributed Audit mechanism proves effective in detecting such misbehavior, CT cannot be relied upon to detect this form of mis-issuance. (See Section 4.6 below.)

2.2.1.3. Mis-behaving third party Monitor

If the third party Monitor that is "protecting" the targeted Subject is mis-behaving, then it will not notify the targeted Subject of any mis-issuance or of any malfeasant log behavior that it detects irrespective of whether the logs it checks are benign or malicious/conspiring.

2.2.2. Certificate not logged

Because the CA is presumed malicious, it may choose to not submit a (pre-)certificate to a log. This means there is no SCT for the certificate.

When a CA does not submit a certificate to a log, whether a log is benign or mis-behaving does not matter. Also, since there is no log entry, there is no difference in behavior between a benign and a mis-behaving third-party Monitor. Neither will report a problem to the Subject.

A bogus certificate would not be delivered to the (legitimate) Subject. So the Subject, acting as a self-Monitor, cannot detect the issuance of a bogus certificate in this case.

2.2.2.1. Careful browser

If careful browsers reject certificates without SCTs, CAs may be "encouraged" to log certificates (see section 4.4.) However, it is not clear how such behavior by browsers can be deployed incrementally throughout the Internet. As a result, this model does not assume that browsers will reject a certificate that is not accompanied by an SCT. In the absence of logging, browsers generally do not benefit from CT, since certificates have to be logged to enable detection of mis-issuance by Monitors, and to trigger subsequent revocation.

3. Syntactic mis-issuance

3.1. Non-malicious Web PKI CA context

This section analyzes the scenario in which the CA has no intent to issue a syntactically incorrect certificate. Throughout the remainder of this document we refer to a syntactically incorrect certificate as "erroneous".

3.1.1. Certificate logged

3.1.1.1. Benign log

If a (pre-)certificate is submitted to a benign log, syntactic mis-issuance can (optionally) be detected, and noted. This will happen only if the log performs syntactic checks in general, and if the log is capable of performing the checks applicable to the submitted (pre-)certificate. (A (pre-)certificate SHOULD be logged even if it fails syntactic validation; logging takes precedence over detection of syntactic mis-issuance.) If syntactic validation fails, this can be noted in an SCT extension returned to the submitter.

. If the (pre-)certificate is submitted by the non-malicious issuing CA, and if the CA has a record of the certificate content, then the CA SHOULD remedy the syntactic problem and re-submit the (pre-)certificate to a log or logs. If this is a pre-certificate submitted prior to issuance, syntactic checking by a log helps avoid issuance of an erroneous certificate. If the CA does not have a record of the certificate contents, then presumably it was a bogus certificate and the CA SHOULD revoke it.

. If a certificate is submitted by its Subject, and is deemed erroneous, then the Subject SHOULD contact the issuing CA and request a new certificate. If the Subject is a legitimate subscriber of the CA, then the CA will either have a record of

the certificate content or can obtain a copy of the certificate from the Subject. The CA will remedy the syntactic problem and either re-submit a corrected (pre-)certificate to a log and send it to the Subject or the Subject will re-submit it to a log. Here too syntactic checking by a log enables a Subject to be informed that its certificate is erroneous and thus may hasten issuance of a replacement certificate.

. If a certificate is submitted by a third party, that party might contact the Subject or the issuing CA, but because the party is not the Subject of the certificate it is not clear how the CA will respond.

Bottom line: Syntactic mis-issuance of a certificate can be avoided by a CA if it makes use of logs that are capable of performing these checks for the types of certificates that are submitted, and if the CA acts on the feedback it receives. If a CA uses a log that does not perform such checks, or if the CA requests checking relative to criteria not supported by the log, then syntactic mis-issuance will not be detected or avoided by this mechanism. Similarly, syntactic mis-issuance can be remedied if a Subject submits a certificate to a log that performs syntactic checks, and if the Subject asks the issuing CA to fix problems detected by the log. (The issuer is presumed to be willing to re-issue the certificate, correcting any problems, because the issuing CA is not malicious.)

3.1.1.2. Mis-behaving log or third party Monitor

A log or Monitor that is conspiring with the attacker or is independently malicious, will either not perform syntactic checks, even though it claims to do so, or simply not report errors. The log entry and the SCT for an erroneous certificate will assert that the certificate syntax was verified.

As with detection of semantic mis-issuance, a distributed Audit mechanism could, in principle, detect mis-behavior by logs or Monitors with respect to syntactic checking. For example, if for a given certificate, some logs (or Monitors) are reporting syntactic errors and some which claim to do syntactic checking, are not reporting these errors, this is indicative of misbehavior by these logs and/or Monitors.

Note that a malicious log (or Monitor) could report syntactic errors for a syntactically valid certificate. This could result in reporting of non-existent syntactic problems to the issuing CA, which might cause the CA to do needless investigative work or perhaps incorrectly revoke and re-issue the Subject’s certificate.

3.1.1.3. Self-monitoring Subject and Benign third-party Monitor

If a Subject or benign Monitor performs syntactic checks, it will detect the erroneous certificate and the issuing CA will be notified (by the Subject). If the Subject is a legitimate subscriber of the CA, then the CA will either have a record of the certificate content or can obtain a copy of the certificate from the Subject. The CA SHOULD revoke the erroneous certificate (after investigation) and remedy the syntactic problem. The CA SHOULD either re-submit the corrected (pre-)certificate to one or more logs and then send the result to the Subject, or send the corrected certificate to the Subject, who will re-submit it to one or more logs.

3.1.1.4. Careful browser

If a browser rejects an erroneous certificate and notifies the Subject and/or the issuing CA, then syntactic mis-issuance will be detected (see Section 4.) Unfortunately, experience suggests that many browsers do not perform thorough syntactic checks on certificates, and so it seems unlikely that browsers will be a reliable way to detect erroneous certificates. Moreover, a protocol used by a browser to notify a Subject and/or CA of an erroneous certificate represents a DoS potential, and thus may not be appropriate. This argues for syntactic checking to be performed by other elements of the CT system, e.g., logs and/or Monitors.

3.1.2. Certificate not logged

If a CA does not submit a certificate to a log, there can be no syntactic checking by the log. Detection of syntactic errors will depend on a Subject performing the requisite checks when it receives its certificate from a CA.

3.2. Malicious Web PKI CA context

This section analyzes the scenario in which the CA’s issuance of a syntactically incorrect certificate is intentional, not due to error. The CA is not the victim but the attacker.

3.2.1. Certificate logged

3.2.1.1. Benign log

Because the CA is presumed to be malicious, the CA may cause the log to not perform checks, in one of several ways. (See [DOMVAL] and [EXTVAL] for more details on validation checks and CCIDs).

1. The CA may assert that the certificate is being issued w/o regard to any guidelines (the "no guidelines" reserved CCID).

2. The CA may assert a CCID that has not been registered, and thus no log will be able to perform a check.

3. The CA may check to see which CCIDs a log declares it can check, and chose a registered CCID that is not checked by the log in question. In this fashion the CA can prevent the log from performing checks, and the SCT and log entry will not contain an indication of a failed check.

4. The CA may submit a (pre-) certificate to a log that is known to not perform any syntactic checks, and thus avoid syntactic checking.

3.2.1.2. Mis-behaving log or third party Monitor

A mis-behaving log or third party Monitor will either not perform syntactic checks or not report any problems that it discovers. (See 3.1.1.2 for further problems).

3.2.1.3. Self-monitoring Subject and Benign third party Monitor

Irrespective of whether syntactic checks are performed by a log, a malicious CA will acquire an embedded SCT, or post-issuance will acquire a standalone SCT. If Subjects or Monitors perform syntactic checks that detect the syntactic mis-issuance and report the problem to the CA, a malicious CA may do nothing or may delay action to remedy the problem.

3.2.1.4. Careful browser

As noted above (3.1.1.4) many browsers fail to perform thorough syntax checks on certificates. Such browsers would benefit from having such checks performed by a log and reported in the SCT. (Remember, in this scenario, the log is benign.) However, if a browser does not discriminate against certificates that do not contain SCTs (or that are not accompanied by an SCT in the TLS handshake), only minimal benefits might accrue to them from syntax checks perform by logs.

If a browser accepts certificates that do not appear to have been syntactically checked by a log (as indicated by the SCT), a malicious CA need not worry about failing a log-based check. Similarly, if there is no requirement for a browser to reject a certificate that was logged by an operator that does not perform

syntactic checks, the fourth approach noted in 3.2.1.1 will succeed as well. If a browser were configured to know which versions of certificate types are applicable to its use of a certificate, the second and third strategies noted above could be thwarted.

3.2.2. Certificate is not logged

Since certificates are not logged in this scenario, a Monitor (third-party or self) cannot detect the issuance of an erroneous certificate. Thus there is no difference between a benign or a malicious/conspiring log or a benign or conspiring/malicious Monitor. (A Subject MAY detect a syntax error by examining the certificate returned to it by the Issuer.) However, even if errors are detected and reported to the CA, a malicious/conspiring CA may do nothing to fix the problem or may delay action.

4. Issues Applicable to Sections 2 and 3

4.1. How does a Subject know which Monitor(s) to use?

If a CA submits a bogus certificate to one or more logs, but these logs are not tracked by a Monitor that is protecting the targeted Subject, CT will not remedy this type of mis-issuance attack. It is not clear whether every Monitor MUST offer to track every Subject that requests protection. Absent such a guarantee, how do Subjects know which set of Monitors will provide "sufficient" coverage? If a Subject acts as its own Monitor, this problem is solved for that Subject.

4.2. How does a Monitor discover new logs?

It is not clear how a (self-)Monitor becomes aware of all (relevant?) logs, including newly created logs. The means by which Monitors become aware of new logs MUST accommodate self-monitoring by a potentially very large number of web site operators. If there are many logs, it may not be feasible for a (self-) Monitor to track all of them.

4.3. CA response to report of a bogus or erroneous certificate

A CA being presented with evidence of a bogus or erroneous certificate, in the form of a log entry and/or SCT, will need to examine its records to determine if it has knowledge of the certificate in question. It also will likely require the targeted Subject to provide assurances that it is the authorized entity representing the Subject name (subjectAltname) in question. Thus a Subject should not expect immediate revocation of a contested

certificate. The time frame in which a CA will respond to a revocation request usually is described in the CPS for the CA. Other certificate fields and extensions may be of interest for forensic purposes, but are not required to effect revocation nor to verify that the certificate to be revoked is bogus or erroneous, based on applicable criteria. The SCT and log entry, because each contains a timestamp from a third party, is probably valuable for forensic purposes (assuming a non-conspiring log operator).

4.4. Browser behavior

If a browser is to reject a certificate that lacks an embedded SCT, or is not accompanied by an SCT transported via the TLS handshake, this behavior needs to be defined in a way that is compatible with incremental deployment. Issuing a warning to a (human) user is probably insufficient, based on experience with warnings displayed for expired certificates, lack of certificate revocation status information, and similar errors that violate RFC 5280 path validation rules. Unless a mechanism is defined that accommodates incremental deployment of this capability, attackers probably will avoid submitting bogus certificates to (benign) logs as a means of evading detection.

4.5. Remediation for a malicious CA

A targeted Subject might ask the parent of a malicious CA to revoke the certificate of the non-cooperative CA. However, a request of this sort may be rejected, e.g., because of the potential for significant collateral damage. A browser might be configured to reject all certificates issued by the malicious CA, e.g., using a CA hot list distributed by a browser vendor. However, if the malicious CA has a sufficient number of legitimate clients, treating all of their certificates as bogus or erroneous still represents serious collateral damage. If this specification were to require that a browser can be configured to reject a specific, bogus or erroneous certificate identified by a Monitor, then the bogus or erroneous certificate could be rejected in that fashion. This remediation strategy calls for communication between Monitors and browsers, or between Monitors and browser vendors. Such communication has not been specified, i.e., there are no standard ways to configure a browser to reject individual bogus or erroneous certificates based on information provided by an external entity such as a Monitor. Moreover, the same or another malicious CA could issue new bogus or erroneous certificates for the targeted Subject, which would have to be detected and rejected in this (as yet unspecified) fashion. Thus, for now, CT does not seem to provide a way to remedy this form of attack, even though it provides a basis for detecting such attacks.

4.6. Auditing - detecting mis-behaving logs

The combination of a malicious CA and one or more conspiring logs motivates the definition of an audit function, to detect conspiring logs. If a Monitor protecting a Subject does not see bogus certificates, it cannot alert the Subject. If one or more SCTs are present in a certificate, or passed via the TLS handshake, a browser has no way to know that the logged certificate is not visible to Monitors. Only if Monitors and browsers reject certificates that contain SCTs from conspiring logs (based on information from an auditor) will CT be able to detect and deter use of such logs. Thus the means by which a Monitor performing an audit function detects such logs, and informs browsers must be specified for CT to be effective in the context of mis-behaving logs.

Absent a well-defined mechanism that enables Monitors to verify that data from logs are reported in a consistent fashion, CT cannot claim to provide protection against logs that are malicious or may conspire with, or are victims of, attackers effecting certificate mis-issuance. The mechanism needs to protect the privacy of users with respect to which web sites they visit. It needs to scale to accommodate a potentially large number of self-monitoring Subjects and a vast number of browsers, if browsers are part of the mechanism. Even when an Audit mechanism is defined, it will be necessary to describe how the CT system will deal with a mis-behaving or compromised log. For example, will there be a mechanism to alert all browsers to reject SCTs issued by such a log? Absent a description of a remediation strategy to deal with mis-behaving or compromised logs, CT cannot ensure detection of mis-issuance in a wide range of scenarios.

Monitors play a critical role in detecting semantic certificate mis-issuance, for Subjects that have requested monitoring of their certificates. A monitor (including a Subject performing self-monitoring) examines logs for certificates associated with one or more Subjects that are being "protected". A third-party Monitor must obtain a list of valid certificates for the Subject being monitored, in a secure manner, to use as a reference. It also must be able to identify and track a potentially large number of logs on behalf of its Subjects. This may be a daunting task for Subjects that elect to perform self-monitoring.

Note: A Monitor must not rely on a CA or RA database for its reference information or use certificate discovery protocols; this information must be acquired by the Monitor based on reference certificates provided by a Subject. If a Monitor were to rely on a CA or RA database (for the CA that issued a targeted certificate),

the Monitor would not detect mis-issuance due to malfeasance on the part of that CA or the RA, or due to compromise of the CA or the RA. If a CA or RA database is used, it would support detection of mis-issuance by an unauthorized CA. A Monitor must not rely on certificate discovery mechanisms to build the list of valid certificates since such mechanisms might result in bogus or erroneous certificates being added to the list.

As noted above, Monitors represent another target for adversaries who wish to effect certificate mis-issuance. If a Monitor is compromised by, or conspires with, an attacker, it will fail to alert a Subject to a bogus or erroneous certificate targeting that Subject, as noted above. It is suggested that a Subject request certificate monitoring from multiple sources to guard against such failures. Operation of a Monitor by a Subject, on its own behalf, avoids dependence on third party Monitors. However, the burden of Monitor operation may be viewed as too great for many web sites, and thus this mode of operation ought not be assumed to be universal when evaluating protection against Monitor compromise.

5. Security Considerations

A threat model is, by definition, a security-centric document. Unlike a protocol description, a threat model does not create security problems nor does it purport to address security problems. This model postulates a set of threats (i.e., motivated, capable adversaries) and examines classes of attacks that these threats are capable of effecting, based on the motivations ascribed to the threats.

6. IANA Considerations

None.

7. Acknowledgments

The author would like to thank David Mandelberg and Karen Seo for help with the editing and formatting, and other members of the TRANS working group for reviewing this document.

8. References

8.1. Normative References

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

8.2. Informative References

[TRANS] Laurie, B., Langley, A., Kasper, E., Messeri, E., Stradling, R., "Certificate Transparency," draft-ietf-trans-rfc6962-bis-07 (March 9, 2015), work in progress.

[DOMVAL] Kent, S., "Syntactic and Semantic Checks for Domain Validation Certificates," draft-kent-trans-domain-validation-cert-checks-01, (December 2014), work in progress.

[EXTVAL] Kent, S., "Syntactic and Semantic Checks for Extended Validation Certificates," draft-kent-trans-extended-validation-cert-checks-01 (December 2014), work in progress.

[gossip] Nordberg, L, Gillmore, D, "Gossiping in CT," draft-linus-trans-gossip-ct-01, (March 9, 2015), work in progress

Author's Addresses

Stephen Kent
BBN Technologies
10 Moulton Street
Cambridge MA 02138
USA

Phone: +1 (617) 873-3988
Email: skent@bbn.com

Copyright Statement

TRANS
Internet-Draft
Intended status: Experimental
Expires: January 8, 2016

L. Nordberg
NORDUnet
D. Gillmor
ACLU
T. Ritter

July 07, 2015

Gossiping in CT
draft-linus-trans-gossip-ct-02

Abstract

This document describes three gossiping mechanisms for Certificate Transparency (CT) [RFC6962]: SCT Feedback, STH Pollination and Trusted Auditor Relationship.

SCT Feedback enables HTTPS clients to share Signed Certificate Timestamps (SCTs) (Section 3.2 of [RFC6962]) with CT auditors in a privacy-preserving manner by sending SCTs to originating HTTPS servers which in turn share them with CT auditors.

In STH Pollination, HTTPS clients use HTTPS servers as pools sharing Signed Tree Heads (STHs) (Section 3.5 of [RFC6962]) with other connecting clients in the hope that STHs will find their way to auditors and monitors.

HTTPS clients in a Trusted Auditor Relationship share SCTs and STHs with trusted auditors or monitors directly, with expectations of privacy sensitive data being handled according to whatever privacy policy is agreed on between client and trusted party.

Status of This Memo

This Internet-Draft will expire on January 8, 2016.

Table of Contents

1. Introduction

The purpose of the protocols in this document is to detect misbehavior by CT logs. In particular, CT logs can misbehave either by rewriting history or by presenting a "split view" of their operations, also known as a partitioning attack [THREAT-ANALYSIS]. CT provides mechanisms for detection of these misbehaviors, but only if the community dependent on the log knows what to do with them. In order for the community to effectively detect log misbehavior, it needs a well-defined way to "gossip" about the activity of the logs that makes use of the available mechanisms.

One of the major challenges of any gossip protocol is limiting damage to user privacy. The goal of CT gossip is to publish and distribute information about the logs and their operations, but not to leak any additional information about the operation of any of the other particpants. Privacy of consumers of log information (in particular, of web browsers and other TLS clients) should not be damaged by gossip.

This document presents three different, complementary mechanisms for non-log players in the CT ecosystem to exchange information about logs in a manner that preserves the privacy of the non-log players involved. They should provide protective benefits for the system as a whole even if their adoption is not universal.

2. Overview

Public append-only untrusted logs have to be monitored for consistency, i.e., that they should never rewrite history. Additionally, monitors and other log clients need to exchange information about monitored logs in order to be able to detect a partitioning attack.

A partitioning attack is when a log serves different views of the log to different clients. Each client would be able to verify the append-only nature of the log while in the extreme case being the only client seeing this particular view.

Gossiping about what's known about logs helps solve the problem of detecting malicious or compromised logs mounting such a partitioning attack. We want some side of the partitioned tree, and ideally both sides, to see the other side.

Disseminating known information about a log poses a potential threat to the privacy of end users. Some data of interest (e.g. SCTs) are linkable to specific log entries and thereby to specific sites, which makes them privacy-sensitive. Gossip about this data has to take privacy considerations into account in order not to leak associations between users of the log (e.g., web browsers) and certificate holders (e.g., web sites). Even sharing STHs (which do not link to specific log entries) can be problematic - user tracking by fingerprinting through rare STHs is one potential attack.

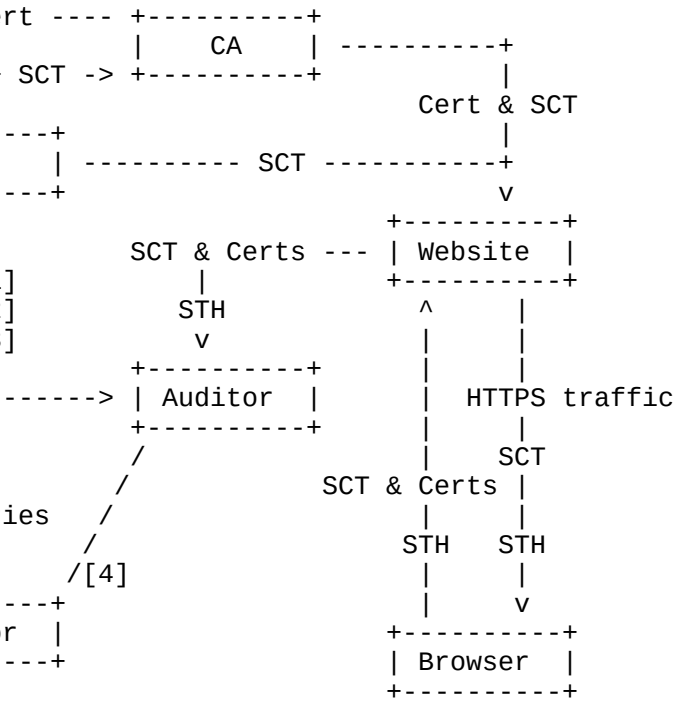
However, there is no loss in privacy if a client sends SCTs for a given site to the site corresponding to the SCT, because the site's access logs would already indicate that the client is accessing that site. In this way a site can accumulate records of SCTs that have been issued by various logs for that site, providing a consolidated repository of SCTs which can be queried by auditors.

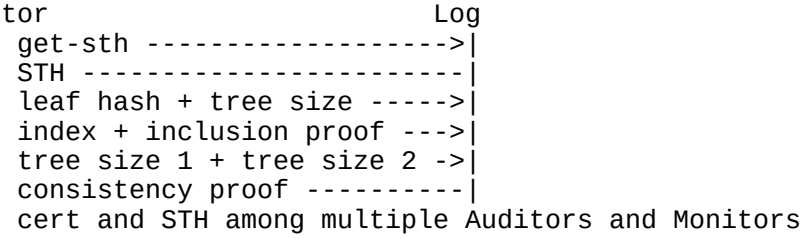
Sharing an STH is considered reasonably safe from a privacy perspective as long as the same STH is shared by a large number of other clients. This "safety in numbers" is achieved by requiring gossip only for STHs of a certain "freshness" and limiting the frequency by which logs can issue STHs.

3. Terminology and data flow

This document relies on terminology and data structures defined in [RFC6962], including STH, SCT, Version, LogID, SCT timestamp, CtExtensions, SCT signature, Merkle Tree Hash.

The following picture shows how certificates, SCTs and STHs flow through a CT system with SCT Feedback and STH Pollination. It does not show what goes in the Trusted Auditor Relationship stream.





4. Who gossips

- HTTPS clients and servers (SCT Feedback and STH Pollination)
- HTTPS servers and CT auditors (SCT Feedback)
- CT auditors and monitors (Trusted Auditor Relationship)

Additionally, some HTTPS clients may engage with an auditor who they trust with their privacy:

- HTTPS clients and CT auditors (Trusted Auditor Relationship)

5. What to gossip about and how

There are three separate gossip streams:

- SCT Feedback, transporting SCTs and certificate chains from HTTPS clients to CT auditors/monitors via HTTPS servers.
- STH Pollination, HTTPS clients and CT auditors/monitors using HTTPS servers as STH pools for exchanging STHs.
- Trusted Auditor Stream, HTTPS clients communicating directly with trusted CT auditors/monitors sharing SCTs, certificate chains and STHs.

5.1. SCT Feedback

The goal of SCT Feedback is for clients to share SCTs and certificate chains with CT auditors and monitors in a privacy-preserving manner.

HTTPS clients store SCTs and certificate chains they see and later send them to the originating HTTPS server by posting them to a .well-known URL. This is described in Section 5.1.1. Note that clients send the same SCTs and chains to servers multiple times with the assumption that a potential man-in-the-middle attack eventually will cease so that an honest server will receive collected malicious SCTs and certificate chains.

HTTPS servers store SCTs and certificate chains received from clients and later share them with CT auditors by either posting them or making them available on a .well-known URL. This is described in Section 5.1.2.

5.1.1. HTTPS client to server

An HTTPS client connects to an HTTPS server for a particular domain. The client receives a set of SCTs as part of the TLS handshake. The client MUST discard SCTs that are not signed by a known log and SHOULD store the remaining SCTs together with the corresponding certificate chain for later use in feedback.

When the client later reconnects to any HTTPS server for the same domain it again receives a set of SCTs. The client MUST add new SCTs from known logs to its store of SCTs for the server. The client MUST send to the server the ones in the store that are for that server and were not received from that server.

Note that the SCT store also contains SCTs received in certificates.

The client MUST NOT send the same set of SCTs to the same server more often than TBD. [benl: "sent to the server" only really counts if the server presented a valid SCT in the handshake and the certificate is known to be unrevoked (which will be hard for a MitM to sustain)] [TODO: expand on rate/resource limiting motivation]

An SCT MUST NOT be sent to any other HTTPS server than one serving the domain that the certificate signed by the SCT refers to. This would lead to two types of privacy leaks. First, the server recieving the SCT would learn about other sites visited by the HTTPS client. Secondly, auditors or monitors recieving SCTs from the HTTPS server would learn information about the other HTTPS servers visited by its clients.

If the HTTPS client has configuration options for not sending cookies to third parties, SCTs MUST be treated as cookies with respect to this setting.

SCTs and corresponding certificates are POSTed to the originating HTTPS server at the well-known URL:

https://<domain>/.well-known/ct/v1/sct-feedback

The data sent in the POST is defined in Section 5.1.3.

HTTPS servers perform a number of sanity checks on SCTs from clients before storing them:

1. if a bit-wise compare of an SCT plus chain matches a pair already in the store, this SCT and chain pair MAY be discarded

2. if the SCT can't be verified to be a valid SCT for the accompanying leaf cert, issued by a known log, the SCT SHOULD be discarded

3. if the leaf cert is not for a domain that the server is authoritative for, the SCT MUST be discarded

Check number 1 is for detecting duplicates. It's important to note that the check must be on pairs of SCT and chain in order to catch different chains accompanied by the same SCT. [XXX why is this important?]

Check number 2 is to prevent spamming attacks where an adversary can fill up the store prior to attacking a client, or a denial of service attack on the server's storage space.

Check number 3 is to help malfunctioning clients from leaking what sites they visit and additionally to prevent spamming attacks.

Note that an HTTPS server MAY perform a certificate chain validation on a submitted certificate chain, and if it matches a trust root configured on the server (but is otherwise unknown to the server), the HTTPS server MAY store the certificate chain and MAY choose to store any submitted SCTs even if they are unable to be verified. The risk of spamming and denial of service can be mitigated by configuring the server with all known acceptable certificates (or certificate hashes).

5.1.2. HTTPS server to auditors

HTTPS servers receiving SCTs from clients SHOULD share SCTs and certificate chains with CT auditors by either providing the well-known URL:

https://<domain>/.well-known/ct/v1/collected-sct-feedback

or by HTTPS POSTing them to a number of preconfigured auditors. This allows an HTTPS server to choose between an active push model or a passive pull model.

The data received in a GET of the well-known URL or sent in the POST is defined in Section 5.1.3.

HTTPS servers SHOULD share all SCTs and accompanying certificate chains they see that pass the checks in Section 5.1.1.

HTTPS servers MUST NOT share any other data that they may learn from the submission of SCT Feedback by HTTPS clients.

Auditors SHOULD provide the following URL accepting HTTPS POSTing of SCT feedback data:

https://<auditor>/ct/v1/sct-feedback

Auditors SHOULD regularly poll HTTPS servers at the well-known collected-sct-feedback URL. The frequency of the polling and how to determine which domains to poll is outside the scope of this document. However, the selection MUST NOT be influenced by potential HTTPS clients connecting directly to the auditor, as it would reveal private information provided by the clients.

5.1.3. SCT Feedback data format

The data shared between HTTPS clients and servers as well as between HTTPS servers and CT auditors/monitors is a JSON object [RFC7159] with the following content:

- sct_feedback: An array of objects consisting of
  - x509_chain: An array of base64-encoded X.509 certificates. The first element is the end-entity certificate, the second chains to the first and so on.
  - sct_data: An array of objects consisting of the base64 representation of the binary SCT data as defined in [RFC6962] Section 3.2.

The 'x509_chain' element MUST contain at the leaf certificate and the full chain to a known root.

5.2. STH pollination

The goal of sharing Signed Tree Heads (STHs) through pollination is to share STHs between HTTPS clients and CT auditors and monitors in a privacy-preserving manner.

HTTPS servers supporting the protocol act as STH pools. HTTPS clients and others in the possesion of STHs should pollinate STH pools by sending STHs to them, and retrieving new STHs to send to new servers. CT auditors and monitors should retrieve STHs from pools by downloading STHs from them.

STH Pollination is carried out by sending STHs to HTTPS servers supporting the protocol, and retrieving new STHs. In the case of HTTPS clients, STHs are sent in an already established TLS session. This makes it hard for an attacker to disrupt STH gossiping without also disturbing ordinary secure browsing (https://).

STHs are sent by POSTing them at the .well-known URL:

https://<domain>/.well-known/ct/v1/sth-pollination

The data sent in the POST is defined in Section 5.2.3.

The response contains zero or more STHs in the same format, described in Section 5.2.3.

An HTTPS client may acquire STHs by several methods:

- in replies to pollination POSTs;
- asking its supported logs for the current STH directly or indirectly;
- via some other (currently unspecified) mechanism.

HTTPS clients (who have STHs), CT auditors and monitors SHOULD pollinate STH pools with STHs. Which STHs to send and how often pollination should happen is regarded as policy and out of scope for this document with exception of certain privacy concerns.

An HTTPS client could be tracked by giving it a unique or rare STH. To address this concern, we place restrictions on different components of the system to ensure an STH will not be rare.

- Logs cannot issue STHs too frequently. This is restricted to 1 per hour.
- HTTPS clients silently ignore STHs which are not fresh.

An STH is considered fresh iff its timestamp is less than 14 days in the past. Given a maximum STH issuance rate of one per hour, an attacker has 336 unique STHs per log for tracking.

When multiplied by the number of logs that a client accepts STHs for, this number of unique STHs grow and the negative privacy implications grow with it. It’s important that this is taken into account when logs are chosen for default settings in HTTPS clients.

[TBD urge HTTPS clients to store STHs retrieved in responses?]

[TBD share inclusion proofs and consistency proofs too?]

5.2.1. HTTPS client STH Fetching

An HTTPS client retrieves SCTs from an HTTPS server, and must obtain an inclusion proof to an STH in order to verify the promise made by the SCT. This retrieval mechanism reveals the client’s browsing habits when the client requests the proof diretly from the log. To mitigate this risk, an HTTPS client MUST retrieve the proof in a manner that disguises the client from the log.

Additionally, for this inclusion proof to be acceptable to the client, the inclusion proof MUST reference a STH that is within the acceptable freshness interval.

Depending on the client's DNS provider, DNS may provide an appropriate intermediate layer that obfuscates the linkability between the user of the client and the request for inclusion (while at the same time providing a caching layer for oft-requested inclusion proofs.)

Also Tor.

5.2.2. Auditor and Monitor Action

Auditors and Monitors participate in STH pollination by retrieving STHs from HTTPS servers. They verify that the STH is valid by checking the signature, and requesting a consistency proof from the STH to the most recent STH.

After retrieving the consistency proof to the most recent STH, they SHOULD pollinate this new STH among participating HTTPS Servers. In this way, as STHs "age out" and are no longer fresh, their "lineage" continues to be tracked in the system.

5.2.3. STH Pollination data format

The data sent from HTTPS clients and CT monitors and auditors to HTTPS servers is a JSON object [RFC7159] with the following content:

- sths - an array of 0 or more fresh STH objects [XXX recently collected] from the log associated with log_id. Each of these objects consists of
  - sth_version: Version as defined in [RFC6962] Section 3.2, as a number. The version of the protocol to which the sth_gossip object conforms.
  - tree_size: The size of the tree, in entries, as a number.
  - timestamp: The timestamp of the STH as defined in [RFC6962] Section 3.2, as a number.
  - sha256_root_hash: The Merkle Tree Hash of the tree as defined in [RFC6962] Section 2.1, as a base64 encoded string.
  - tree_head_signature: A TreeHeadSignature as defined in [RFC6962] Section 3.5 for the above data, as a base64 encoded string.
  - log_id: LogID as defined in [RFC6962] Section 3.2, as a base64 encoded string.

[XXX An STH is considered recently collected iff TBD.]

5.3. Trusted Auditor Stream

HTTPS clients MAY send SCTs and cert chains, as well as STHs, directly to auditors. Note that there are privacy implications of doing so, outlined in Section 6.1.1 and Section 6.1.5.

The most natural trusted auditor arrangement arguably is a web browser that is "logged in to" a provider of various internet services. Another equivalent arrangement is a trusted party like a corporation which an employer is connected to through a VPN or by other similar means. A third might be individuals or smaller groups of people running their own services. In such a setting, retrieving STHs and inclusion proofs from that third party in order to validate SCTs could be considered reasonable from a privacy perspective. The HTTPS client does its own auditing and might additionally share SCTs and STHs with the trusted party to contribute to herd immunity. Here, the ordinary [RFC6962] protocol is sufficient for the client to do the auditing while SCT Feedback and STH Pollination can be used in whole or in parts for the gossip part.

Another well established trusted party arrangement on the internet today is the relation between internet users and their providers of DNS resolver services. DNS resolvers are typically provided by the internet service provider (ISP) used, which by the nature of name resolving already know a great deal about what sites their users visit. As mentioned in Section XXX, in order for HTTPS clients to be able to retrieve inclusion proofs for certificates in a privacy preserving manner, logs could expose a DNS interface in addition to the ordinary HTTPS interface. An informal writeup of such a protocol can be found at XXX.

5.3.1. Trusted Auditor data format

[TBD specify something here or leave this for others?]

6. Security considerations

6.1. Privacy considerations

The most sensitive relationships in the CT ecosystem are the relationships between HTTPS clients and HTTPS servers. Client-server relationships can be aggregated into a network graph with potentially serious implications for correlative de-anonymisation of clients and relationship-mapping or clustering of servers or of clients.

6.1.1. Privacy and SCTs

An SCT contains information that links it to a particular web site. Because the client-server relationship is sensitive, gossip between clients and servers about unrelated SCTs is risky. Therefore, a client with an SCT for a given server should transmit that information in only two channels: to a server associated with the SCT itself; and to a trusted CT auditor, if one exists.

6.1.2. Privacy in SCT Feedback

SCTs introduce yet another mechanism for HTTPS servers to store state on an HTTPS client, and potentially track users. HTTPS clients which allow users to clear history or cookies associated with an origin MUST clear stored SCTs associated with the origin as well.

Auditors should treat all SCTs as sensitive data. SCTs received directly from an HTTPS client are especially sensitive, because the auditor is a trusted by the client to not reveal their associations with servers. Auditors MUST NOT share such SCTs in any way, including sending them to an external log, without first mixing them with multiple other SCTs learned through submissions from multiple other clients. The details of mixing SCTs are TBD.

There is a possible fingerprinting attack where a log issues a unique SCT for targeted log client(s). A colluding log and HTTPS server operator could therefore be a threat to the privacy of an HTTPS client. Given all the other opportunities for HTTPS servers to fingerprint clients - TLS session tickets, HPKP and HSTS headers, HTTP Cookies, etc. - this is acceptable.

The fingerprinting attack described above could be avoided by requiring that logs i) MUST return the same SCT for a given cert chain ([RFC6962] Section 3) and ii) use a deterministic signature scheme when signing the SCT ([RFC6962] Section 2.1.4).

There is another similar fingerprinting attack where an HTTPS server tracks a client by using a variation of cert chains. The risk for this attack is accepted on the same grounds as the unique SCT attack described above. [XXX any mitigations possible here?]

6.1.3. Privacy for HTTPS clients requesting STHs

An HTTPS client that does not act as an auditor should only request an STH from a CT log that it accepts SCTs from. An HTTPS client should regularly request an STH from all logs it is willing to accept, even if it has seen no SCTs from that log.

6.1.4. Privacy in STH Pollination

An STH linked to an HTTPS client may indicate the following about that client:

- that the client gossips;
- that the client been using CT at least until the time that the timestamp and the tree size indicate;
- that the client is talking, possibly indirectly, to the log indicated by the tree hash;
- which software and software version is being used.

There is a possible fingerprinting attack where a log issues a unique STH for a targeted log auditor or HTTPS client. This is similar to the fingerprinting attack described in Section 6.1.2, but it is mitigated by the following factors:

- the relationship between auditors and logs is not sensitive in the way that the relationship between HTTPS clients and HTTPS servers is;
- because auditors regularly exchange STHs with each other, the re-appearance of a targeted STH from some auditor does not imply that the auditor was the original one targeted by the log;
- an HTTPS client's relationship to a log is not sensitive in the way that its relationship to an HTTPS server is. As long as the client does not query the log for more than individual STHs, the client should not leak anything else to the log itself. However, a log and an HTTPS server which are collaborating could use this technique to fingerprint a targeted HTTPS client.

Note that an HTTPS client in the configuration described in this document doesn't make direct use of the STH itself. Its fetching of the STH and reporting via STH Pollination provides a benefit to the CT ecosystem as a whole by providing oversight on logs, but the HTTPS client itself will not necessarily derive direct benefit.

6.1.5. Trusted Auditors for HTTPS Clients

Some HTTPS clients may choose to use a trusted auditor. This trust relationship leaks a certain amount of information from the client to the auditor. In particular, it is likely to identify the web sites that the client has visited to the auditor. Some clients may already share this information to a third party, for example, when using a

server to synchronize browser history across devices in a server-visible way, or when doing DNS lookups through a trusted DNS resolver. For clients with such a relationship already established, sending SCT Feedback to the same organization does not appear to leak any additional information to the trusted third party.

Clients who wish to contact an auditor without associating their identities with their SCT Feedback may wish to use an anonymizing network like Tor to submit SCT Feedback to the auditor. Auditors SHOULD accept SCT Feedback that arrives over such anonymizing networks.

Clients sending feedback to an auditor may prefer to reduce the temporal granularity of the history leakage to the auditor by caching and delaying their SCT Feedback reports. This strategy is only as effective as the granularity of the timestamps embedded in the SCTs and STHs.

6.1.6. HTTPS Clients as Auditors

Some HTTPS Clients may choose to act as Auditors themselves. A Client taking on this role needs to consider the following:

- an Auditing HTTPS Client potentially leaks their history to the logs that they query. Querying the log through a cache or a proxy with many other users may avoid this leakage, but may leak information to the cache or proxy, in the same way that an non-Auditing HTTPS Client leaks information to a trusted Auditor.

- an effective Auditor needs a strategy about what to do in the event that it discovers misbehavior from a log. Misbehavior from a log involves the log being unable to provide either (a) a consistency proof between two valid STHs or (b) an inclusion proof for a certificate to an STH any time after the log's MMD has elapsed from the issuance of the SCT. The log's inability to provide either proof will not be externally cryptographically-verifiable, as it may be indistinguishable from a network error.

7. IANA considerations

TBD

8. Contributors

The authors would like to thank the following contributors for valuable suggestions: Al Cutter, Ben Laurie, Benjamin Kaduk, Karen Seo, Magnus Ahltorp, Yan Zhu.

9. ChangeLog

9.1. Changes between -01 and -02

- STH Pollination defined.
- Trusted Auditor Relationship defined.
- Overview section rewritten.
- Data flow picture added.
- Section on privacy considerations expanded.

9.2. Changes between -00 and -01

- Add the SCT feedback mechanism: Clients send SCTs to originating web server which shares them with auditors.
- Stop assuming that clients see STHs.
- Don’t use HTTP headers but instead .well-known URL’s - avoid that battle.
- Stop referring to trans-gossip and trans-gossip-transport-https - too complicated.
- Remove all protocols but HTTPS in order to simplify - let’s come back and add more later.
- Add more reasoning about privacy.
- Do specify data formats.

10. References

10.1. Normative References

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

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

10.2. Informative References

[THREAT-ANALYSIS]
Kent, S., "Threat Analysis for Certificate Transparency", 2015, <https://datatracker.ietf.org/doc/draft-ietf-trans-threat-analysis/>.

Authors' Addresses

Linus Nordberg
NORDUnet

Email: linus@nordu.net

Daniel Kahn Gillmor
ACLU

Email: dkg@fifthhorseman.net

Tom Ritter

Email: tom@ritter.vg

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: November 6, 2015

D. Zhang

D. Gillmor
CMRG

D. He
Huawei

B. Sarikaya
Huawei USA

May 5, 2015

CT for Binary Codes
draft-zhang-trans-ct-binary-codes-02

Abstract

This document proposes a solution to use CT for publishing softwares/ binary codes and their signatures.

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].

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 November 6, 2015.

Copyright Notice

Table of Contents

1. Introduction

Digital signatures have been widely used in software distributions to demonstrate the authenticity of softwares. Through verifying signatures, end users can could ensure that the softwares they got are provided by legal organizations and never modified during the delivery. If an end user does not have a direct trust relationship with the software provider, an authentication chain need to be generated from the key used for generating the signature to a trust anchor that the user trusts. That is why many signature mechanisms for software distribution are based on PKI.

However, signatures cannot prevent a software developer from distributing softwares with customized backdoors/drawbacks. In some circumstances, it may be hard for a user to detect the differences between the software it got and the software provided to other users.

This draft describe a mechanism to extend Certificate Transparency specified in [I-D.ietf-trans-rfc6962-bis] to support logging binary codes. A software provider can publish its softwares (or digests of

binary codes in the cases the softwares are non-trivial) to one or more logs. Therefore, a user can easily detect the customized backdoors through monitoring the log entries.

In this mechanism, after a section of binary codes is published to a log, the log will return a ticket (called Signed Certificate Timestamp (SCT) in this case) to the software provider. The software without the ticket will not be trusted even it is associated with a proper signature. This approach will force providers to publish their binary codes to logs.

2. Cryptographic Components of Certificate Transparency

The introduction of cryptographic components of CT is in Section 2 of [I-D.ietf-trans-rfc6962-bis]. When applying CT for binary codes, a log is a single, ever-growing, append-only Merkle Tree of Delegation Signer (DS) Resource Records (RRs).

3. Motivation Scenarios

3.1. Dealing with Customized Backdoors in Firmwares

The disclosed documents have raised the concerns of people on the vulnerability of the network devices to the passive attacks performed by NSA or other organizations. Some vendors have met problems in their foreign markets because their products are suspected to have customized backdoors for adversaries to perform attacks. It is desired for vendors to publish the design details of the products and provide sufficient functions for clients to check whether certain hardware or software of a device has been improperly modified. There are various techniques that could be used for this purpose. One way is to force a vendor to publish the binary codes of its firmwares. Therefore, customers can easily detect whether the vendor is releasing the same firmware to everyone. In addition, the binary transparency, customer will have more confidence on the quality of firmwares. Since the same codes are used by different customers all over the world, the drawbacks in firmwares will be easier to be detected.

3.2. Another Scenarios

Ben suggested to us CT for publishing FreeBSD. Can you contribute some ideas here?

4. Log Format and Operation

4.1. Log Entries

A developer of a software can submit the signed software (or the digest of the software to save space) to any preferred logs before distributing it. In order to enable attribution of each logged RR to its issuer, the log SHALL publish a list of certificates to construct an authentication chain connecting the trust anchor and the certificate used to sign the software.

Logs MUST verify the authentication chain and make sure it leads back to a trusted certificate, using the chain of certificates provided by the submitter. Logs MUST refuse to publish a signed software without a valid chain. If the software and the signature are accepted and an SCT issued, the accepting log MUST store the entire chain used for verification, including the signed software itself and including the trusted zone signing key used to verify the chain, and MUST present this chain for auditing upon request.

To comply with the certificate entries specified in [I-D.ietf-trans-rfc6962-bis], each software entry in a log MUST include the following components:

```
 enum { x509_entry(0), precert_entry(1), BIN_entry(TBD1), (65535) } LogEntryT
ype;

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

 opaque BINARY<1..2^24-1>;
 struct {
    BINARY signed_software;
   ASN.1Cert certificate_chain<0..2^24-1>;
 } BINARY_Chain_Entry;
```

"entry_type" is the type of this entry. the type value of a binary log entry is TBD1.

"signed_software" include the binary codes, the signature, and any other additional information used to describe the software and the signer publishing the software. The current version of the document

does not specify how to structure such information. (CMS[RFC5652] can be used.) This work will be left for future work.

"certificate_chain" include the certificates constructing a chain from the certificate of signer to a certificate trusted by the log. The first certificate MUST be the certificate of signer. 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.

4.2. Structure of the Signed Certificate Timestamp

This work reuses the structure of Signed Certificate Timestamp (SCT) specified in Section 3.3 of [I-D.ietf-trans-rfc6962-bis] but make necessary extensions.

```
 enum { certificate_timestamp(0), tree_hash(1),BINARY_timestamp(TBD2), (255)
}
   SignatureType;

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

   struct {
       opaque key_id[32];
   } LogID;

  opaque digestcodes<0..2^24-1>;
  struct {
     opaque issuer_key_hash[32];
     digestcodes binary_digest;
   } Binary_Codes;

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

"key_id" and "issuer_key_hash" are defined in Section 3.3 of [I-D.ietf-trans-rfc6962-bis].

binary_digest is the SHA-256 hash of binary codes. (Open question: In the future version of the this document, do we need to support other digest algorithms? If so, maybe we should provide an digest identifier field here.)

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

"sct_version", "timestamp", "entry_type and extensions" are are identical to what is defined in Section 3.3 of [I-D.ietf-trans-rfc6962-bis]..

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

5. Log Client Messages

In Section 4 of [I-D.ietf-trans-rfc6962-bis], a set of messages is defined for clients to query and verfiy the correctness of the log entries they are interested in. In this memo, two new messages are defined for CT to support binary transparency.

5.1. Add Binary and Certificate Chain to Log

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

Inputs:

software: the binary code, the signature, and the information used to describe the software and the signer publishing the software

chain: An array of base64-encoded certificates. The first element is the certificate used to sign the binary codes; 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, which is the current NTP Time [RFC5905], measured since the epoch (January 1, 1970, 00:00), ignoring leap seconds, in milliseconds.

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.

6. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

7. Security Considerations

8. Acknowledgements

9. Normative References

[I-D.ietf-trans-rfc6962-bis] Laurie, B., Langley, A., Kasper, E., Messeri, E., and R. Stradling, "Certificate Transparency", draft-ietf-trans-rfc6962-bis-07 (work in progress), March 2015.

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

[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.

Authors' Addresses

Dacheng Zhang

Email: dacheng.zhang@gmail.com

Daniel Kahn Gillmor
CMRG

Email: dkg@fifthhorseman.net

Danping He
Huawei

Email: ana.hedanping@huawei.com

Behcet Sarikaya
Huawei USA
5340 Legacy Dr. Building 3
Plano, TX 75024

Email: sarikaya@ieee.org