                            Gossiping in CT
                        draft-ietf-trans-gossip-05

Abstract

   The logs in Certificate Transparency are untrusted in the sense that
   the users of the system don't have to trust that they behave
   correctly since the behavior of a log can be verified to be correct.

   This document tries to solve the problem with logs presenting a
   "split view" of their operations or failing to incorporate a
   submission within MMD. It describes three gossiping mechanisms for
   Certificate Transparency: SCT Feedback, STH Pollination and Trusted
   Auditor Relationship.

publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction

   The purpose of the protocols in this document, collectively referred
   to as CT Gossip, is to detect certain misbehavior by CT logs. In
   particular, CT Gossip aims to detect logs that are providing
   inconsistent views to different log clients, and logs failing to
   include submitted certificates within the time period stipulated by
   MMD.

   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 expose
   any additional information about the operation of any of the other
   participants. Privacy of consumers of log information (in
   particular, of web browsers and other TLS clients) should not be
   undermined by gossip.

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

2. Defining the problem

When a log provides different views of the log to different clients
this is described as a partitioning attack. Each client would be
able to verify the append-only nature of the log but, in the extreme
case, each client might see a unique view of the log.

The CT logs are public, append-only and untrusted and thus have to be
audited for consistency, i.e., they should never rewrite history.
Additionally, auditors and other log clients need to exchange
information about logs in order to be able to detect a partitioning
attack (as described above).

Gossiping about log behavior helps address the problem of detecting
malicious or compromised logs with respect to a partitioning attack.
We want some side of the partitioned tree, and ideally all sides, to
see at least one other side.

Disseminating information about a log poses a potential threat to the
privacy of end users. Some data of interest (e.g., SCTs) is linkable
to specific log entries and thereby to specific websites, which makes
sharing them with others a privacy concern. Gossiping about this
data has to take privacy considerations into account in order not to
expose 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 (see
Section 8.2).

3. Overview

This document presents three gossiping mechanisms: SCT Feedback, STH
Pollination, and a Trusted Auditor Relationship.

SCT Feedback enables HTTPS clients to share Signed Certificate
Timestamps (SCTs) (Section 4.8 of [RFC-6962-BIS-27]) 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 to share
Signed Tree Heads (STHs) (Section 4.10 of [RFC-6962-BIS-27]) with

other connecting clients in the hope that STHs will find their way to
CT auditors.

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

Despite the privacy risks with sharing SCTs there is no loss in
privacy if a client sends SCTs for a given site to the site
corresponding to the SCT. This is because the site's cookies could
already indicate that the client had accessed 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 that could be shared with auditors. Auditors can use this
information to detect a misbehaving log that fails to include a
certificate within the time period stipulated by its MMD log
parameter.

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 log clients. This safety in numbers can be achieved by only
allowing gossiping of STHs issued in a certain window of time, while
also refusing to gossip about STHs from logs with too high an STH
issuance frequency (see Section 8.2).

4. Terminology

This document relies on terminology and data structures defined in
[RFC-6962-BIS-27], including MMD, STH, SCT, Version, LogID, SCT
timestamp, CtExtensions, SCT signature, Merkle Tree Hash.

This document relies on terminology defined in
[draft-ietf-trans-threat-analysis-12], including Auditing.

4.1. Pre-Loaded vs Locally Added Anchors

Through the document, we refer to both Trust Anchors (Certificate
Authorities) and Logs. Both Logs and Trust Anchors may be locally
added by an administrator. Unless otherwise clarified, in both cases
we refer to the set of Trust Anchors and Logs that come pre-loaded
and pre-trusted in a piece of client software.

5. Who gossips with whom

   o HTTPS clients and servers (SCT Feedback and STH Pollination)

   o HTTPS servers and CT auditors (SCT Feedback and STH Pollination)

o CT auditors (Trusted Auditor Relationship)

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

o HTTPS clients and CT auditors (Trusted Auditor Relationship)

6. What to gossip about and how

There are three separate gossip streams:

o SCT Feedback - transporting SCTs and certificate chains from HTTPS
   clients to CT auditors via HTTPS servers.

o STH Pollination - HTTPS clients and CT auditors using HTTPS
   servers as STH pools for exchanging STHs.

o Trusted Auditor Stream - HTTPS clients communicating directly with
   trusted CT auditors sharing SCTs, certificate chains and STHs.

It is worthwhile to note that when an HTTPS client or CT auditor
interacts with a log, they may equivalently interact with a log
mirror or cache that replicates the log.

7. Data flow

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.

```
    +- Cert ---- +----------+
    |            |    CA    | ----------+
    |    + SCT -> +----------+          |
    v    |                         Cert [& SCT]
 +----------+                           |
 |   Log    | ---------- SCT -----------+
 +----------+                           v
   |  ^                         +----------+
   |  |       SCTs & Certs ---  | Website  |
   |  |[1]              |       +----------+
   |  |[2]           STHs          ^    |
   |  |[3]             v           |    HTTPS traffic
   |  |            +----------+    |    |
   |  +--------> | Auditor  |    |    SCT & Cert
   |              +----------+    |    |
  STH                             |    STH & Inclusion proof
   |                              |    |
 Log entries          SCTs & Certs |    |
   |                           |    |
   v                          STHs  |
 +----------+                   |    v
 | Monitor  |             +----------+
 +----------+             | Browser  |
                          +----------+
```
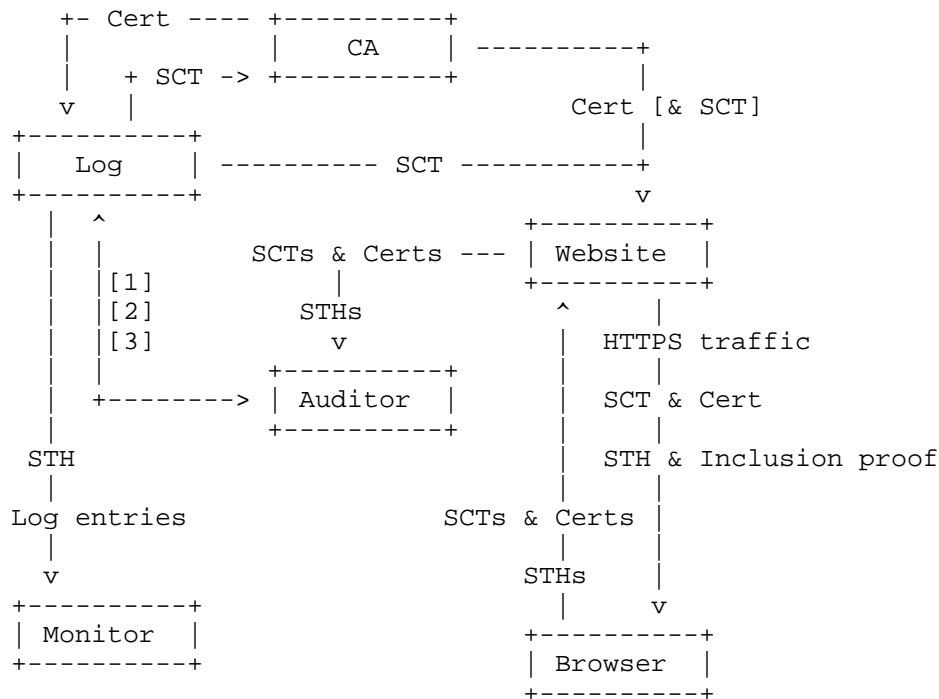
```
 #    Auditor                           Log
[1] |--- get-sth ------------------->|
    |<-- STH ------------------------|
[2] |--- leaf hash + tree size ----->|
    |<-- index + inclusion proof --->|
[3] |--- tree size 1 + tree size 2 ->|
    |<-- consistency proof ----------|
```

8. Gossip Mechanisms

8.1. SCT Feedback

   The goal of SCT Feedback is for clients to share SCTs and certificate
   chains with CT auditors while still preserving the privacy of the end
   user. The sharing of SCTs contribute to the overall goal of
   detecting misbehaving logs by providing auditors with SCTs from many
   vantage points, making it more likely to catch a violation of a log's
   MMD or a log presenting inconsistent views. The sharing of
   certificate chains is beneficial to HTTPS server operators interested
   in direct feedback from clients for detecting bogus certificates
   issued in their name and therefore incentivizes server operators to
   take part in SCT Feedback.

   SCT Feedback is the most privacy-preserving gossip mechanism, as it
   does not directly expose any links between an end user and the sites
   they've visited to any third party.

   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 (associated with that server), as described in
   Section 8.1.2. Note that clients will send the same SCTs and chains
   to a server multiple times with the assumption that any man-in-the-
   middle attack eventually will cease, and an honest server will
   eventually receive collected malicious SCTs and certificate chains.

   HTTPS servers store SCTs and certificate chains received from
   clients, as described in Section 8.1.3. They later share them with
   CT auditors by either posting them to auditors or making them
   available via a well-known URL. This is described in Section 8.1.4.

8.1.1. SCT Feedback data format

   The data shared between HTTPS clients and servers, as well as between
   HTTPS servers and CT auditors, is a JSON array [RFC7159]. Each item
   in the array is a JSON object containing at least the first of the
   following members:

   o "x509_chain" : An array of PEM-encoded X.509 certificates. The
      first element is the end-entity certificate, the second certifies
      the first and so on. The "x509_chain" member is mandatory to
      include.

   o "sct_data_v1" : An array of base64 encoded
      "SignedCertificateTimestampList"s as defined in [RFC6962] section
      3.3. The "sct_data_v1" member is optional.

   o "sct_data_v2" : An array of base64 encoded "TransItem" structures
      of type "x509_sct_v2" or "precert_sct_v2" as defined in
      [RFC-6962-BIS-27] section 4.8. The "sct_data_v2" member is
      optional.

   We will refer to this object as 'sct_feedback'.

   The x509_chain element always contains a full chain from a leaf
   certificate to a self-signed trust anchor.

   See Section 8.1.2 for details on what the sct_data element contains
   as well as more details about the x509_chain element.

8.1.2. HTTPS client to server

   When an HTTPS client connects to an HTTPS server, the client receives
   a set of SCTs as part of the TLS handshake. SCTs are included in the
   TLS handshake using one or more of the three mechanisms described in
   [RFC-6962-BIS-27] section 6 - in the server certificate, in a TLS
   extension, or in an OCSP extension. The client MUST discard SCTs
   that are not signed by a log known to the client and SHOULD store the
   remaining SCTs together with a locally constructed certificate chain
   which is trusted (i.e., terminated in a pre-loaded or locally
   installed Trust Anchor) in an sct_feedback object or equivalent data
   structure for later use in SCT Feedback.

   The SCTs stored on the client MUST be keyed by the exact domain name
   the client contacted. They MUST NOT be sent to the well-known URI of
   any domain not matching the original domain (e.g., if the original
   domain is sub.example.com they must not be sent to
   sub.sub.example.com or to example.com.) In particular, they MUST NOT
   be sent to the well-known URI of any Subject Alternate Names
   specified in the certificate. In the case of certificates that
   validate multiple domain names, after visiting a second domain name
   specified in the certificate, the same SCT is expected to be stored
   once under each domain name's key. If Connection Reuse as defined in
   [RFC7540] is available, reusing an existing connection to
   sub.example.com to send data to sub.sub.example.com is permitted.

   Not following these constraints would increase the risk for two types
   of privacy breaches. First, the HTTPS server receiving the SCT would
   learn about other sites visited by the HTTPS client. Second,
   auditors receiving SCTs from the HTTPS server would learn information
   about other HTTPS servers visited by its clients.

   If the client later again connects to the same HTTPS server, it again
   receives a set of SCTs and calculates a certificate chain, and again
   creates an sct_feedback or similar object. If this object does not
   exactly match an existing object in the store, then the client MUST
   add this new object to the store, associated with the exact domain
   name contacted, as described above. An exact comparison is needed to
   ensure that attacks involving alternate chains are detected. An
   example of such an attack is described in
   [dual-ca-compromise-attack]. However, at least one optimization is
   safe and MAY be performed: If the certificate chain exactly matches
   an existing certificate chain, the client MAY store the union of the
   SCTs from the two objects in the first (existing) object.

   If the client does connect to the same HTTPS server a subsequent
   time, it MUST send to the server sct_feedback objects in the store
   that are associated with that domain name. However, it is not

necessary to send an sct_feedback object constructed from the current
TLS session, and if the client does so, it MUST NOT be marked as sent
in any internal tracking done by the client.

Refer to Section 11.3 for recommendations for implementation.

Because SCTs can be used as a tracking mechanism (see
Section 10.5.2), they deserve special treatment when they are
received from (and provided to) domains that are loaded as
subresources from an origin domain. Such domains are commonly called
'third party domains'. An HTTPS client SHOULD store SCT Feedback
using a 'double-keying' approach, which isolates third party domains
by the first party domain. This is described in [double-keying].
Gossip would be performed normally for third party domains only when
the user revisits the first party domain. In lieu of 'double-
keying', an HTTPS client MAY treat SCT Feedback in the same manner it
treats other security mechanisms that can enable tracking (such as
HSTS and HPKP.)

SCT Feedback is only performed when a user connects to a site via
intentional web browsing or normal third party resource inclusion.
It MUST NOT be performed automatically as part of some sort of
background process.

Finally, if the HTTPS client has configuration options for not
sending cookies to third parties, SCTs of third parties MUST be
treated as cookies with respect to this setting. This prevents third
party tracking through the use of SCTs/certificates, which would
bypass the cookie policy. For domains that are only loaded as third
party domains, the client may never perform SCT Feedback; however the
client may perform STH Pollination after fetching an inclusion proof,
as specified in Section 8.2.

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

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

The data sent in the POST is defined in Section 8.1.1. This data
SHOULD be sent in an already-established TLS session. This makes it
hard for an attacker to disrupt SCT Feedback without also disturbing
ordinary secure browsing (https://). This is discussed more in
Section 11.1.1.

The HTTPS server SHOULD respond with an HTTP 200 response code and an
empty body if it was able to process the request. An HTTPS client
which receives any other response SHOULD consider it an error.

Some clients have trust anchors or logs that are locally added (e.g.,
by an administrator or by the user themselves). These additions are
potentially privacy-sensitive because they can carry information
about the specific configuration, computer, or user.

Certificates validated by locally added trust anchors will commonly
have no SCTs associated with them, so in this case no action is
needed with respect to CT Gossip. SCTs issued by locally added logs
MUST NOT be reported via SCT Feedback.

If a certificate is validated by SCTs that are issued by publicly
trusted logs, but chains to a local trust anchor, the client MAY
perform SCT Feedback for this SCT and certificate chain bundle. If
it does so, the client MUST include the full chain of certificates
chaining to the local trust anchor in the x509_chain array.
Performing SCT Feedback in this scenario may be advantageous for the
broader internet and CT ecosystem, but may also disclose information
about the client. If the client elects to omit SCT Feedback, it can
choose to perform STH Pollination after fetching an inclusion proof,
as specified in Section 8.2.

We require the client to send the full chain (or nothing at all) for
two reasons. Firstly, it simplifies the operation on the server if
there are not two code paths. Secondly, omitting the chain does not
actually preserve user privacy. The Issuer field in the certificate
describes the signing certificate. And if the certificate is being
submitted at all, it means the certificate is logged, and has SCTs.
This means that the Issuer can be queried and obtained from the log,
so omitting the signing certificate from the client's submission does
not actually help user privacy.

8.1.3. HTTPS server operation

HTTPS servers can be configured (or omit configuration), resulting
in, broadly, two modes of operation. In the simpler mode, the server
will only track leaf certificates and SCTs applicable to those leaf
certificates. In the more complex mode, the server will confirm the
client's chain validation and store the certificate chain. The
latter mode requires more configuration, but is necessary to prevent
denial of service (DoS) attacks on the server's storage space.

In the simple mode of operation, upon receiving a submission at the
sct-feedback well-known URL, an HTTPS server will perform a set of
operations, checking on each sct_feedback object before storing it:

o (1) the HTTPS server MAY modify the sct_feedback object, and
   discard all items in the x509_chain array except the first item
   (which is the end-entity certificate)

   o (2) if a bit-wise compare of the sct_feedback object matches one
      already in the store, this sct_feedback object SHOULD be discarded

   o (3) if the leaf cert is not for a domain for which the server is
      authoritative, the SCT MUST be discarded

   o (4) if an SCT in the sct_data array can't be verified to be a
      valid SCT for the accompanying leaf cert, and issued by a known
      log, the individual SCT SHOULD be discarded

   The modification in step number 1 is necessary to prevent a malicious
   client from exhausting the server's storage space. A client can
   generate their own issuing certificate authorities, and create an
   arbitrary number of chains that terminate in an end-entity
   certificate with an existing SCT. By discarding all but the end-
   entity certificate, we prevent a simple HTTPS server from storing
   this data. Note that operation in this mode will not prevent the
   attack described in [dual-ca-compromise-attack]. Skipping this step
   requires additional configuration as described below.

   The check in step 2 is for detecting duplicates and minimizing
   processing and storage by the server. As on the client, an exact
   comparison is needed to ensure that attacks involving alternate
   chains are detected. Again, at least one optimization is safe and
   MAY be performed. If the certificate chain exactly matches an
   existing certificate chain, the server MAY store the union of the
   SCTs from the two objects in the first (existing) object. If the
   validity check on any of the SCTs fails, the server SHOULD NOT store
   the union of the SCTs.

   The check in step 3 is to help malfunctioning clients from exposing
   which sites they visit. It additionally helps prevent DoS attacks on
   the server.

   The check in step 4 is to prevent DoS attacks where an adversary
   fills up the store prior to attacking a client (thus preventing the
   client's feedback from being recorded), or an attack where an
   adversary simply attempts to fill up server's storage space.

   The above describes the simpler mode of operation. In the more
   advanced server mode, the server will detect the attack described in
   [dual-ca-compromise-attack]. In this configuration the server will
   not modify the sct_feedback object prior to performing checks 2, 3,
   and 4. Instead, to prevent a malicious client from filling the
   server's data store, the HTTPS server SHOULD perform an additional
   check in the more advanced mode:

   o (5) if the x509_chain consists of an invalid certificate chain, or
      the culminating trust anchor is not recognized by the server, the
      server SHOULD modify the sct_feedback object, discarding all items
      in the x509_chain array except the first item

   The HTTPS server MAY choose to omit checks 4 or 5. This will place
   the server at risk of having its data store filled up by invalid
   data, but can also allow a server to identify interesting certificate
   or certificate chains that omit valid SCTs, or do not chain to a
   trusted root. This information may enable an HTTPS server operator
   to detect attacks or unusual behavior of Certificate Authorities even
   outside the Certificate Transparency ecosystem.

8.1.4. HTTPS server to auditors

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

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

   or by HTTPS POSTing them to a set 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 8.1.1 with the following difference: The
   x509_chain element may contain only he end-entity certificate, as
   described below.

   HTTPS servers SHOULD share all sct_feedback objects they see that
   pass the checks in Section 8.1.3. If this is an infeasible amount of
   data, the server MAY choose to expire submissions according to an
   undefined policy. Suggestions for such a policy can be found in
   Section 11.3.

   HTTPS servers MUST NOT share any other data that they may learn from
   the submission of SCT Feedback by HTTPS clients, like the HTTPS
   client IP address or the time of submission.

   As described above, HTTPS servers can be configured (or omit
   configuration), resulting in two modes of operation. In one mode,
   the x509_chain array will contain a full certificate chain. This
   chain may terminate in a trust anchor the auditor may recognize, or
   it may not. (One scenario where this could occur is if the client
   submitted a chain terminating in a locally added trust anchor, and
   the server kept this chain.) In the other mode, the x509_chain array

will consist of only a single element, which is the end-entity
certificate.

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

https://<auditor>/ct-gossip/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. For example, if a
poll to example.com occurs directly after a client submits an SCT for
example.com, an adversary observing the auditor can trivially
conclude the activity of the client.

8.2. STH pollination

The goal of sharing Signed Tree Heads (STHs) through pollination is
to share STHs between HTTPS clients and CT auditors while still
preserving the privacy of the end user. The sharing of STHs
contribute to the overall goal of detecting misbehaving logs by
providing CT auditors with STHs from many vantage points, making it
possible to detect logs that are presenting inconsistent views.

HTTPS servers supporting the protocol act as STH pools. HTTPS
clients and CT auditors in the possession of STHs can pollinate STH
pools by sending STHs to them, and retrieving new STHs to send to
other STH pools. CT auditors can improve the value of their auditing
by retrieving STHs from pools.

HTTPS clients send STHs to HTTPS servers by POSTing them to the well-
known URL:

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

The data sent in the POST is defined in Section 8.2.4. This data
SHOULD be 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://). This is discussed more in
Section 11.1.1.

On a successful connection to an HTTPS server implementing STH
Pollination, the response code will be 200, and the response body is
application/json, containing zero or more STHs in the same format, as
described in Section 8.2.4.

An HTTPS client may acquire STHs by several methods:

o in replies to pollination POSTs;

o asking logs that it recognizes for the current STH, either
   directly (v2/get-sth) or indirectly (for example over DNS)

o resolving an SCT and certificate to an STH via an inclusion proof

o resolving one STH to another via a consistency proof

HTTPS clients (that have STHs) and CT auditors SHOULD pollinate STH
pools with STHs. Which STHs to send and how often pollination should
happen is regarded as undefined policy with the exception of privacy
concerns explained below. Suggestions for the policy can be found in
Section 11.3.

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.

o HTTPS clients silently ignore STHs from logs with an STH issuance
   frequency of more than one STH per hour. Logs use the STH
   Frequency Count log parameter to express this ([RFC-6962-BIS-27]
   section 4.1).

o 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. Clients MUST
ignore STHs older than 14 days. We consider STHs within this
validity window not to be personally identifiable data, and STHs
outside this window to be personally identifiable.

When multiplied by the number of logs from which a client accepts
STHs, 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.
This concern is discussed upon in Section 10.5.5.

A log may cease operation, in which case there will soon be no STH
within the validity window. Clients SHOULD perform all three methods
of gossip about a log that has ceased operation since it is possible
the log was still compromised and gossip can detect that. STH
Pollination is the one mechanism where a client must know about a log
shutdown. A client which does not know about a log shutdown MUST NOT
attempt any heuristic to detect a shutdown. Instead the client MUST

be informed about the shutdown from a verifiable source (e.g., a
software update), and be provided the final STH issued by the log.
The client SHOULD resolve SCTs and STHs to this final STH. If an SCT
or STH cannot be resolved to the final STH, clients SHOULD follow the
requirements and recommendations set forth in Section 11.1.2.

8.2.1. HTTPS Clients and Proof Fetching

   There are two types of proofs a client may retrieve; inclusion proofs
   and consistency proofs.

   An HTTPS client will retrieve SCTs together with certificate chains
   from an HTTPS server. Using the timestamp in the SCT together with
   the end-entity certificate and the issuer key hash, it can obtain an
   inclusion proof to an STH in order to verify the promise made by the
   SCT.

   An HTTPS client will have STHs from performing STH Pollination, and
   may obtain a consistency proof to a more recent STH.

   An HTTPS client may also receive an SCT bundled with an inclusion
   proof to a historical STH via an unspecified future mechanism.
   Because this historical STH is considered personally identifiable
   information per above, the client needs to obtain a consistency proof
   to a more recent STH.

   A client SHOULD attempt proof fetching. A client MAY do network
   probing to determine if proof fetching may succeed, and if it learns
   that it does not, SHOULD periodically re-probe (especially after
   network change, if it is aware of these events.) If it does succeed,
   queued events can be processed.

   A client MUST NOT perform proof fetching for any SCTs or STHs issued
   by a locally added log. A client MAY fetch an inclusion proof for an
   SCT (issued by a pre-loaded log) that validates a certificate
   chaining to a locally added trust anchor.

   If a client requested either proof directly from a log or auditor, it
   would reveal the client's browsing habits to a third party. To
   mitigate this risk, an HTTPS client MUST retrieve the proof in a
   manner that disguises the client.

   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). See [draft-ct-over-dns] for an example of how
   this can be done.

Anonymity networks such as Tor also present a mechanism for a client to anonymously retrieve a proof from an auditor or log.

Even when using a privacy-preserving layer between the client and the log, certain observations may be made about an anonymous client or general user behavior depending on how proofs are fetched. For example, if a client fetched all outstanding proofs at once, a log would know that SCTs or STHs received around the same time are more likely to come from a particular client. This could potentially go so far as correlation of activity at different times to a single client. In aggregate the data could reveal what sites are commonly visited together. HTTPS clients SHOULD use a strategy of proof fetching that attempts to obfuscate these patterns. A suggestion of such a policy can be found in Section 11.2.

Resolving either SCTs and STHs may result in errors. These errors may be routine downtime or other transient errors, or they may be indicative of an attack. Clients SHOULD follow the requirements and recommendations set forth in Section 11.1.2 when handling these errors in order to give the CT ecosystem the greatest chance of detecting and responding to a compromise.

8.2.2. STH Pollination without Proof Fetching

An HTTPS client MAY participate in STH Pollination without fetching proofs. In this situation, the client receives STHs from a server, applies the same validation logic to them (signed by a known log, within the validity window) and will later pass them to another HTTPS server.

When operating in this fashion, the HTTPS client is promoting gossip for Certificate Transparency, but derives no direct benefit itself. In comparison, a client which resolves SCTs or historical STHs to recent STHs and pollinates them is assured that if it was attacked, there is a probability that the ecosystem will detect and respond to the attack (by distrusting the log).

8.2.3. Auditor Action

CT auditors 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.

8.2.4. STH Pollination data format

   The data sent from HTTPS clients and CT auditors to HTTPS servers is
   a JSON object [RFC7159] with one or both of the following two
   members:

   o "v1" : array of 0 or more objects each containing an STH as
      returned from ct/v1/get-sth, see [RFC6962] section 4.3

   o "v2" : array of 0 or more objects each containing an STH as
      returned from ct/v2/get-sth, see [RFC-6962-BIS-27] section 5.2

   Note that all STHs MUST be fresh as defined in Section 8.2.

8.3. Trusted Auditor Stream

   HTTPS clients MAY send SCTs and cert chains, as well as STHs,
   directly to auditors. If sent, this data MAY include data that
   reflects locally added logs or trust anchors. Note that there are
   privacy implications in doing so, these are outlined in
   Section 10.5.1 and Section 10.5.6.

   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 to which an employee is connected 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
   proofs from that third party could be considered reasonable from a
   privacy perspective. The HTTPS client may also do its own auditing
   and might additionally share SCTs and STHs with the trusted party to
   contribute to herd immunity. Here, the ordinary [RFC-6962-BIS-27]
   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 which sites their users
   visit. As mentioned in Section 8.2.1, in order for HTTPS clients to
   be able to retrieve proofs in a privacy preserving manner, logs could
   expose a DNS interface in addition to the ordinary HTTPS interface.
   A specification of such a protocol can be found in
   [draft-ct-over-dns].

8.3.1. Trusted Auditor data format

   Trusted Auditors expose a REST API at the fixed URI:

   https://<auditor>/ct-gossip/v1/trusted-auditor

   Submissions are made by sending an HTTPS POST request, with the body
   of the POST in a JSON object. Upon successful receipt the Trusted
   Auditor returns 200 OK.

   The JSON object consists of two top-level keys: 'sct_feedback' and
   'sths'. The 'sct_feedback' value is an array of JSON objects as
   defined in Section 8.1.1. The 'sths' value is an array of STHs as
   defined in Section 8.2.4.

   Example:

   {
     'sct_feedback' :
       [
         {
           'x509_chain' :
             [
               '----BEGIN CERTIFICATE---\n
                AAA...',
               '----BEGIN CERTIFICATE---\n
                AAA...',
                ...
             ],
           'sct_data' :
             [
               'AAA...',
               'AAA...',
                ...
             ]
         }, ...
       ],
     'sths' :
       [
         'AAA...',
         'AAA...',
          ...
       ]
   }

9. 3-Method Ecosystem

   The use of three distinct methods for auditing logs may seem
   excessive, but each represents a needed component in the CT
   ecosystem. To understand why, the drawbacks of each component must
   be outlined. In this discussion we assume that an attacker knows
   which mechanisms an HTTPS client and HTTPS server implement.

9.1. SCT Feedback

   SCT Feedback requires the cooperation of HTTPS clients and more
   importantly HTTPS servers. Although SCT Feedback does require a
   significant amount of server-side logic to respond to the
   corresponding APIs, this functionality does not require
   customization, so it may be pre-provided and work out of the box.
   However, to take full advantage of the system, an HTTPS server would
   wish to perform some configuration to optimize its operation:

   o Minimize its disk commitment by maintaining a list of known SCTs
      and certificate chains (or hashes thereof)

   o Maximize its chance of detecting a misissued certificate by
      configuring a trust store of CAs

   o Establish a "push" mechanism for POSTing SCTs to CT auditors

   These configuration needs, and the simple fact that it would require
   some deployment of software, means that some percentage of HTTPS
   servers will not deploy SCT Feedback.

   If SCT Feedback was the only mechanism in the ecosystem, any server
   that did not implement the feature would open itself and its users to
   attack without any possibility of detection.

   A webserver not deploying SCT Feedback (or an alternative method
   providing equivalent functionality) may never learn that it was a
   target of an attack by a malicious log, as described in Section 10.1,
   although the presence of an attack by the log could be learned
   through STH Pollination. Additionally, users who wish to have the
   strongest measure of privacy protection (by disabling STH Pollination
   Proof Fetching and forgoing a Trusted Auditor) could be attacked
   without risk of detection.

9.2. STH Pollination

   STH Pollination requires the cooperation of HTTPS clients, HTTPS
   servers, and logs.

For a client to fully participate in STH Pollination, and have this
mechanism detect attacks against it, the client must have a way to
safely perform Proof Fetching in a privacy preserving manner. (The
client may pollinate STHs it receives without performing Proof
Fetching, but we do not consider this option in this section.)

HTTPS servers must deploy software (although, as in the case with SCT
Feedback this logic can be pre-provided) and commit some configurable
amount of disk space to the endeavor.

Logs (or a third party mirroring the logs) must provide access to
clients to query proofs in a privacy preserving manner, most likely
through DNS.

Unlike SCT Feedback, the STH Pollination mechanism is not hampered if
only a minority of HTTPS servers deploy it. However, it makes an
assumption that an HTTPS client performs Proof Fetching (such as the
DNS mechanism discussed). Unfortunately, any manner that is
anonymous for some (such as clients which use shared DNS services
such as a large ISP), may not be anonymous for others.

For instance, DNS requests expose a considerable amount of sensitive
information (including what data is already present in the cache) in
plaintext over the network. For this reason, some percentage of
HTTPS clients may choose to not enable the Proof Fetching component
of STH Pollination. (Although they can still request and send STHs
among participating HTTPS servers, even when this affords them no
direct benefit.)

If STH Pollination was the only mechanism deployed, users that
disable it would be able to be attacked without risk of detection.

If STH Pollination (or an alternative method providing equivalent
functionality) was not deployed, HTTPS clients visiting HTTPS Servers
which did not deploy SCT Feedback could be attacked without risk of
detection.

9.3. Trusted Auditor Relationship

The Trusted Auditor Relationship is expected to be the rarest gossip
mechanism, as an HTTPS client is providing an unadulterated report of
its browsing history to a third party. While there are valid and
common reasons for doing so, there is no appropriate way to enter
into this relationship without retrieving informed consent from the
user.

However, the Trusted Auditor Relationship mechanism still provides
value to a class of HTTPS clients. For example, web crawlers have no

concept of a "user" and no expectation of privacy. Organizations already performing network auditing for anomalies or attacks can run their own Trusted Auditor for the same purpose with marginal increase in privacy concerns.

The ability to change one's Trusted Auditor is a form of Trust Agility that allows a user to choose who to trust, and be able to revise that decision later without consequence. A Trusted Auditor connection can be made more confidential than DNS (through the use of TLS), and can even be made (somewhat) anonymous through the use of anonymity services such as Tor. (Note that this does ignore the de-anonymization possibilities available from viewing a user's browsing history.)

If the Trusted Auditor relationship was the only mechanism deployed, users who do not enable it (the majority) would be able to be attacked without risk of detection.

If the Trusted Auditor relationship was not deployed, crawlers and organizations would build it themselves for their own needs. By standardizing it, users who wish to opt-in (for instance those unwilling to participate fully in STH Pollination) can have an interoperable standard they can use to choose and change their trusted auditor.

9.4. Interaction

Assuming no other log consistency measures exist, clients who perform only a subset of the mechanisms described in this document are exposed to the following vulnerabilities:

HTTPS clients can be attacked without risk of detection if they do not participate in any of the three mechanisms.

HTTPS clients are afforded the greatest chance of detecting an attack when they either participate in both SCT Feedback and STH Pollination with Proof Fetching or if they have a Trusted Auditor relationship. (Participating in SCT Feedback is the only way specified in this document to prevent a malicious log from refusing to ever resolve an SCT to an STH, as put forward in Section 10.1). Additionally, participating in SCT Feedback enables an HTTPS client to assist in detecting the exact target of an attack.

HTTPS servers that omit SCT Feedback enable malicious logs to carry out attacks without risk of detection. If these servers are targeted specifically, even if the attack is detected, without SCT Feedback they may never learn that they were specifically targeted. HTTPS servers without SCT Feedback do gain some measure of herd immunity,

but only because their clients participate in STH Pollination (with
Proof Fetching) or have a Trusted Auditor Relationship.

When HTTPS servers omit SCT feedback, it allows their users to be
attacked without detection by a malicious log; the vulnerable users
are those who do not have a Trusted Auditor relationship.

10. Security considerations

10.1. Attacks by actively malicious logs

One of the most powerful attacks possible in the CT ecosystem is a
trusted log that has actively decided to be malicious. It can carry
out an attack in at least two ways:

In the first attack, the log can present a split view of the log for
all time. This attack can be detected by CT auditors, but a naive
auditor implementation may fail to do so. The simplest, least
efficient way to detect the attack is to mirror the entire log and
assert inclusion of every peice of data. If an auditor does not
mirror the log, one way to detect this attack is to resolve each view
of the log to the most recent STHs available and then force the log
to present a consistency proof. (Which it cannot.) We highly
recommend auditors plan for this attack scenario and ensure it will
be detected.

In the second attack, the log can sign an SCT, and refuse to ever
include the certificate that the SCT refers to in the tree.
(Alternately, it can include it in a branch of the tree and issue an
STH, but then abandon that branch.) Whenever someone requests an
inclusion proof for that SCT (or a consistency proof from that STH),
the log would respond with an error, and a client may simply regard
the response as a transient error. This attack can be detected using
SCT Feedback, or an Auditor of Last Resort, as presented in
Section 11.1.2.

Both of these attack variants can be detected by CT auditors who have
obtained an STH of an 'abnormal' view of the log. However, they may
not be able to link the STH to any particular SCT or Certificate.
This means that while the log misbehavior was successfully detected,
the target of the attack was not identified. To assertively identify
the target(s) of the attack, SCT Feedback is necessary.

10.2. Dual-CA Compromise

[dual-ca-compromise-attack] describes an attack possible by an
adversary who compromises two Certificate Authorities and a Log. This
attack is difficult to defend against in the CT ecosystem, and

[dual-ca-compromise-attack] describes a few approaches to doing so. We note that Gossip is not intended to defend against this attack, but can in certain modes.

Defending against the Dual-CA Compromise attack requires SCT Feedback, and explicitly requires the server to save full certificate chains (described in Section 8.1.3 as the 'complex' configuration.) After CT auditors receive the full certificate chains from servers, they MAY compare the chain built by clients to the chain supplied by the log. If the chains differ significantly, the auditor SHOULD raise a concern. A method of determining if chains differ significantly is by asserting that one chain is not a subset of the other and that the roots of the chains are different.

10.3. Censorship/Blocking considerations

We assume a network attacker who is able to fully control the client's internet connection for some period of time, including selectively blocking requests to certain hosts and truncating TLS connections based on information observed or guessed about client behavior. In order to successfully detect log misbehavior, the gossip mechanisms must still work even in these conditions.

There are several gossip connections that can be blocked:

1. Clients sending SCTs to servers in SCT Feedback

2. Servers sending SCTs to auditors in SCT Feedback (server push mechanism)

3. Servers making SCTs available to auditors (auditor pull mechanism)

4. Clients fetching proofs in STH Pollination

5. Clients sending STHs to servers in STH Pollination

6. Servers sending STHs to clients in STH Pollination

7. Clients sending SCTs to Trusted Auditors

If a party cannot connect to another party, it can be assured that the connection did not succeed. While it may not have been maliciously blocked, it knows the transaction did not succeed. Mechanisms which result in a positive affirmation from the recipient that the transaction succeeded allow confirmation that a connection was not blocked. In this situation, the party can factor this into strategies suggested in Section 11.3 and in Section 11.1.2.

The connections that allow positive affirmation are 1, 2, 4, 5, and
7.

More insidious is blocking the connections that do not allow positive
confirmation: 3 and 6. An attacker may truncate or drop a response
from a server to a client, such that the server believes it has
shared data with the recipient, when it has not. However, in both
scenarios (3 and 6), the server cannot distinguish the client as a
cooperating member of the CT ecosystem or as an attacker performing a
Sybil attack, aiming to flush the server's data store. Therefore the
fact that these connections can be undetectably blocked does not
actually alter the threat model of servers responding to these
requests. The choice of algorithm to release data is crucial to
protect against these attacks; strategies are suggested in
Section 11.3.

Handling censorship and network blocking (which is indistinguishable
from network error) is relegated to the implementation policy chosen
by clients. Suggestions for client behavior are specified in
Section 11.1.

10.4. Flushing Attacks

A flushing attack is an attempt by an adversary to flush a particular
piece of data from a pool. In the CT Gossip ecosystem, an attacker
may have performed an attack and left evidence of a compromised log
on a client or server. They would be interested in flushing that
data, i.e. tricking the target into gossiping or pollinating the
incriminating evidence with only attacker-controlled clients or
servers with the hope they trick the target into deleting it.

Flushing attacks may be defended against differently depending on the
entity (HTTPS client or HTTPS server) and record (STHs or SCTs with
Certificate Chains).

10.4.1. STHs

For both HTTPS clients and HTTPS servers, STHs within the validity
window SHOULD NOT be deleted. An attacker cannot flush an item from
the cache if it is never removed so flushing attacks are completely
mitigated.

The required disk space for all STHs within the validity window is
336 STHs per log that is trusted. If 20 logs are trusted, and each
STH takes 1 Kilobytes, this is 6.56 Megabytes.

Note that it is important that implementors do not calculate the
exact size of cache expected - if an attack does occur, a small

number of additional, fraudulent STHs will enter into the cache.
These STHs will be in addition to the expected set, and will be
evidence of the attack. Flooding the cache will not work, as an
attacker would have to include fraudulent STHs in the flood.

If an HTTPS client or HTTPS server is operating in a constrained
environment and cannot devote enough storage space to hold all STHs
within the validity window it is recommended to use the below
Deletion Algorithm in section Section 11.3.2 to make it more
difficult for the attacker to perform a flushing attack.

10.4.2. SCTs & Certificate Chains on HTTPS Servers

An HTTPS server will only accept SCTs and Certificate Chains for
domains it is authoritative for. Therefore the storage space needed
is bound by the number of logs it accepts, multiplied by the number
of domains it is authoritative for, multiplied by the number of
certificates issued for those domains.

Imagine a server authoritative for 10,000 domains, and each domain
has 3 certificate chains, and 10 SCTs. A certificate chain is 5
Kilobytes in size and an SCT 1 Kilobyte. This yields 732 Megabytes.

This data can be large, but it is calculable. Web properties with
more certificates and domains are more likely to be able to handle
the increased storage need, while small web properties will not seen
an undue burden. Therefore HTTPS servers SHOULD NOT delete SCTs or
Certificate Chains. This completely mitigates flushing attacks.

Again, note that it is important that implementors do not calculate
the exact size of cache expected - if an attack does occur, the new
SCT(s) and Certificate Chain(s) will enter into the cache. This data
will be in addition to the expected set, and will be evidence of the
attack.

If an HTTPS server is operating in a constrained environment and
cannot devote enough storage space to hold all SCTs and Certificate
Chains it is authoritative for it is recommended to configure the SCT
Feedback mechanism to allow only certain certificates that are known
to be valid. These chains and SCTs can then be discarded without
being stored or subsequently provided to any clients or auditors. If
the allowlist is not sufficient, the below Deletion Algorithm in
Section 11.3.2 is recommended to make it more difficult for the
attacker to perform a flushing attack.

10.4.3. SCTs & Certificate Chains on HTTPS Clients

   HTTPS clients will accumulate SCTs and Certificate Chains without
   bound. It is expected they will choose a particular cache size and
   delete entries when the cache size meets its limit. This does not
   mitigate flushing attacks, and such an attack is documented in
   [gossip-mixing].

   The below Deletion Algorithm Section 11.3.2 is recommended to make it
   more difficult for the attacker to perform a flushing attack.

10.5. Privacy considerations

   CT Gossip deals with HTTPS clients which are trying to share
   indicators that correspond to their browsing history. 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-anonymization of clients and
   relationship-mapping or clustering of servers or of clients.

   There are, however, certain clients that do not require privacy
   protection. Examples of these clients are web crawlers or robots.
   But even in this case, the method by which these clients crawl the
   web may in fact be considered sensitive information. In general, it
   is better to err on the side of safety, and not assume a client is
   okay with giving up its privacy.

10.5.1. 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 NOT transmit that
   information in any other than the following two channels: to the
   server associated with the SCT itself (via a TLS connection with a
   certificate identifying the Domain Name of the web site with a Host
   header specifying the domain name); or to a Trusted Auditor, if one
   exists.

10.5.2. 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 and certificate chains 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. Suggestions for mixing SCTs are presented in
Section 11.3.

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 considered acceptable.

The fingerprinting attack described above would be mitigated by a
requirement that logs must use a deterministic signature scheme when
signing SCTs ([RFC-6962-BIS-27] section 2.2). A log signing using
RSA is not required to use a deterministic signature scheme.

Since logs are allowed to issue a new SCT for a certificate already
present in the log, mandating deterministic signatures does not stop
this fingerprinting attack altogether. It does make the attack
harder to pull off without being detected though.

There is another similar fingerprinting attack where an HTTPS server
tracks a client by using a unique certificate or a variation of cert
chains. The risk for this attack is accepted on the same grounds as
the unique SCT attack described above.

10.5.3. Privacy for HTTPS clients performing STH Proof Fetching

An HTTPS client performing Proof Fetching SHOULD NOT request proofs
from a CT log that it doesn't accept 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.

The time between two polls for new STH's SHOULD NOT be significantly
shorter than the MMD of the polled log divided by its STH Frequency
Count ([RFC-6962-BIS-27] section 4.1).

The actual mechanism by which Proof Fetching is done carries
considerable privacy concerns. Although out of scope for the
document, DNS is a mechanism currently discussed. DNS exposes data
in plaintext over the network (including what sites the user is
visiting and what sites they have previously visited) and may not be
suitable for some.

10.5.4. Privacy in STH Pollination

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

   o that the client gossips;

   o that the client has been using CT at least until the time that the
      timestamp and the tree size indicate;

   o that the client is talking, possibly indirectly, to the log
      indicated by the tree hash;

   o which software and software version is being used.

   There is a possible fingerprinting attack where a log issues a unique
   STH for a targeted HTTPS client. This is similar to the
   fingerprinting attack described in Section 10.5.2, but can operate
   cross-origin. If a log (or HTTPS server cooperating with a log)
   provides a unique STH to a client, the targeted client will be the
   only client pollinating that STH cross-origin.

   It is mitigated partially because the log is limited in the number of
   STHs it can issue. It must 'save' one of its STHs each MMD to
   perform the attack. A log violating its STH Frequency Count
   ([RFC-6962-BIS-27] section 4.1) can be identified as non-compliant by
   CT auditors following the procedure described in [RFC-6962-BIS-27]
   section 8.3.

10.5.5. Privacy in STH Interaction

   An HTTPS client may pollinate any STH within the last 14 days. An
   HTTPS client may also pollinate an STH for any log that it knows
   about. When a client pollinates STHs to a server, it will release
   more than one STH at a time. It is unclear if a server may 'prime' a
   client and be able to reliably detect the client at a later time.

   It's clear that a single site can track a user any way they wish, but
   this attack works cross-origin and is therefore more concerning. Two
   independent sites A and B want to collaborate to track a user cross-
   origin. A feeds a client Carol some N specific STHs from the M logs
   Carol trusts, chosen to be older and less common, but still in the
   validity window. Carol visits B and chooses to release some of the
   STHs she has stored, according to some policy.

   Modeling a representation for how common older STHs are in the pools
   of clients, and examining that with a given policy of how to choose
   which of those STHs to send to B, it should be possible to calculate

statistics about how unique Carol looks when talking to B and how
useful/accurate such a tracking mechanism is.

Building such a model is likely impossible without some real world
data, and requires a given implementation of a policy. To combat
this attack, suggestions are provided in Section 11.3 to attempt to
minimize it, but follow-up testing with real world deployment to
improve the policy will be required.

10.5.6. Trusted Auditors for HTTPS Clients

Some HTTPS clients may choose to use a trusted auditor. This trust
relationship exposes a large amount of information about the client
to the auditor. In particular, it will 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 SCTs to a trusted auditor run by the same organization does
not appear to expose any additional information to the trusted third
party.

Clients which wish to contact a CT auditor without associating their
identities with their SCTs 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 exposure to the auditor by
caching and delaying their SCT Feedback reports. This is elaborated
upon in Section 11.3. This strategy is only as effective as the
granularity of the timestamps embedded in the SCTs and STHs.

10.5.7. HTTPS Clients as Auditors

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

o an Auditing HTTPS client potentially exposes its history to the
   logs that they query. Querying the log through a cache or a proxy
   with many other users may avoid this exposure, but may expose
   information to the cache or proxy, in the same way that a non-
   Auditing HTTPS Client exposes information to a Trusted Auditor.

o an effective CT 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.

11. Policy Recommendations

   This section is intended as suggestions to implementors of HTTPS
   Clients, HTTPS servers, and CT auditors. It is not a requirement for
   technique of implementation, so long as the privacy considerations
   established above are obeyed.

11.1. Blocking Recommendations

11.1.1. Frustrating blocking

   When making gossip connections to HTTPS servers or Trusted Auditors,
   it is desirable to minimize the plaintext metadata in the connection
   that can be used to identify the connection as a gossip connection
   and therefore be of interest to block. Additionally, introducing
   some randomness into client behavior may be important. We assume
   that the adversary is able to inspect the behavior of the HTTPS
   client and understand how it makes gossip connections.

   As an example, if a client, after establishing a TLS connection (and
   receiving an SCT, but not making its own HTTP request yet),
   immediately opens a second TLS connection for the purpose of gossip,
   the adversary can reliably block this second connection to block
   gossip without affecting normal browsing. For this reason it is
   recommended to run the gossip protocols over an existing connection
   to the server, making use of connection multiplexing such as HTTP
   Keep-Alive or SPDY.

   Truncation is also a concern. If a client always establishes a TLS
   connection, makes a request, receives a response, and then always
   attempts a gossip communication immediately following the first
   response, truncation will allow an attacker to block gossip reliably.

   For these reasons, we recommend that, if at all possible, clients
   SHOULD send gossip data in an already established TLS session. This
   can be done through the use of HTTP Pipelining, SPDY, or HTTP/2.

11.1.2. Responding to possible blocking

   In some circumstances a client may have a piece of data that they
   have attempted to share (via SCT Feedback or STH Pollination), but

have been unable to do so: with every attempt they receive an error.
These situations are:

1. The client has an SCT and a certificate, and attempts to retrieve
    an inclusion proof - but receives an error on every attempt.

2. The client has an STH, and attempts to resolve it to a newer STH
    via a consistency proof - but receives an error on every attempt.

3. The client has attempted to share an SCT and constructed
    certificate via SCT Feedback - but receives an error on every
    attempt.

4. The client has attempted to share an STH via STH Pollination -
    but receives an error on every attempt.

5. The client has attempted to share a specific piece of data with a
    Trusted Auditor - but receives an error on every attempt.

In the case of 1 or 2, it is conceivable that the reason for the
errors is that the log acted improperly, either through malicious
actions or compromise. A proof may not be able to be fetched because
it does not exist (and only errors or timeouts occur). One such
situation may arise because of an actively malicious log, as
presented in Section 10.1. This data is especially important to
share with the broader internet to detect this situation.

If an SCT has attempted to be resolved to an STH via an inclusion
proof multiple times, and each time has failed, this SCT might very
well be a compromising proof of an attack. However the client MUST
NOT share the data with any other third party (excepting a Trusted
Auditor should one exist).

If an STH has attempted to be resolved to a newer STH via a
consistency proof multiple times, and each time has failed, a client
MAY share the STH with an "Auditor of Last Resort" even if the STH in
question is no longer within the validity window. This auditor may
be pre-configured in the client, but the client SHOULD permit a user
to disable the functionality or change whom data is sent to. The
Auditor of Last Resort itself represents a point of failure and
privacy concerns, so if implemented, it SHOULD connect using public
key pinning and not consider an item delivered until it receives a
confirmation.

In the cases 3, 4, and 5, we assume that the webserver(s) or trusted
auditor in question is either experiencing an operational failure, or
being attacked. In both cases, a client SHOULD retain the data for
later submission (subject to Private Browsing or other history-

clearing actions taken by the user.) This is elaborated upon more in
Section 11.3.

11.2. Proof Fetching Recommendations

   Proof fetching (both inclusion proofs and consistency proofs) SHOULD
   be performed at random time intervals. If proof fetching occurred
   all at once, in a flurry of activity, a log would know that SCTs or
   STHs received around the same time are more likely to come from a
   particular client. While proof fetching is required to be done in a
   manner that attempts to be anonymous from the perspective of the log,
   the correlation of activity to a single client would still reveal
   patterns of user behavior we wish to keep confidential. These
   patterns could be recognizable as a single user, or could reveal what
   sites are commonly visited together in the aggregate.

11.3. Record Distribution Recommendations

   In several components of the CT Gossip ecosystem, the recommendation
   is made that data from multiple sources be ingested, mixed, stored
   for an indeterminate period of time, provided (multiple times) to a
   third party, and eventually deleted. The instances of these
   recommendations in this draft are:

   o When a client receives SCTs during SCT Feedback, it should store
      the SCTs and Certificate Chain for some amount of time, provide
      some of them back to the server at some point, and may eventually
      remove them from its store

   o When a client receives STHs during STH Pollination, it should
      store them for some amount of time, mix them with other STHs,
      release some of them them to various servers at some point,
      resolve some of them to new STHs, and eventually remove them from
      its store

   o When a server receives SCTs during SCT Feedback, it should store
      them for some period of time, provide them to auditors some number
      of times, and may eventually remove them

   o When a server receives STHs during STH Pollination, it should
      store them for some period of time, mix them with other STHs,
      provide some of them to connecting clients, may resolve them to
      new STHs via Proof Fetching, and eventually remove them from its
      store

   o When a Trusted Auditor receives SCTs or historical STHs from
      clients, it should store them for some period of time, mix them

with SCTs received from other clients, and act upon them at some
period of time

Each of these instances have specific requirements for user privacy,
and each have options that may not be invoked. As one example, an
HTTPS client should not mix SCTs from server A with SCTs from server
B and release server B's SCTs to Server A. As another example, an
HTTPS server may choose to resolve STHs to a single more current STH
via proof fetching, but it is under no obligation to do so.

These requirements should be met, but the general problem of
aggregating multiple pieces of data, choosing when and how many to
release, and when to remove them is shared. This problem has
previously been considered in the case of Mix Networks and Remailers,
including papers such as [trickle].

There are several concerns to be addressed in this area, outlined
below.

11.3.1. Mixing Algorithm

When SCTs or STHs are recorded by a participant in CT Gossip and
later used, it is important that they are selected from the datastore
in a non-deterministic fashion.

This is most important for servers, as they can be queried for SCTs
and STHs anonymously. If the server used a predictable ordering
algorithm, an attacker could exploit the predictability to learn
information about a client. One such method would be by observing
the (encrypted) traffic to a server. When a client of interest
connects, the attacker makes a note. They observe more clients
connecting, and predicts at what point the client-of-interest's data
will be disclosed, and ensures that they query the server at that
point.

Although most important for servers, random ordering is still
strongly recommended for clients and Trusted Auditors. The above
attack can still occur for these entities, although the circumstances
are less straightforward. For clients, an attacker could observe
their behavior, note when they receive an STH from a server, and use
javascript to cause a network connection at the correct time to force
a client to disclose the specific STH. Trusted Auditors are stewards
of sensitive client data. If an attacker had the ability to observe
the activities of a Trusted Auditor (perhaps by being a log, or
another auditor), they could perform the same attack - noting the
disclosure of data from a client to the Trusted Auditor, and then
correlating a later disclosure from the Trusted Auditor as coming
from that client.

Random ordering can be ensured by several mechanisms. A datastore
can be shuffled, using a secure shuffling algorithm such as Fisher-
Yates. Alternately, a series of random indexes into the data store
can be selected (if a collision occurs, a new index is selected.) A
cryptographically secure random number generator must be used in
either case. If shuffling is performed, the datastore must be marked
'dirty' upon item insertion, and at least one shuffle operation
occurs on a dirty datastore before data is retrieved from it for use.

11.3.2. The Deletion Algorithm

No entity in CT Gossip is required to delete records at any time,
except to respect user's wishes such as private browsing mode or
clearing history. However, it is likely that over time the
accumulated storage will grow in size and need to be pruned.

While deletion of data will occur, proof fetching can ensure that any
misbehavior from a log will still be detected, even after the direct
evidence from the attack is deleted. Proof fetching ensures that if
a log presents a split view for a client, they must maintain that
split view in perpetuity. An inclusion proof from an SCT to an STH
does not erase the evidence - the new STH is evidence itself. A
consistency proof from that STH to a new one likewise - the new STH
is every bit as incriminating as the first. (Client behavior in the
situation where an SCT or STH cannot be resolved is suggested in
Section 11.1.2.) Because of this property, we recommend that if a
client is performing proof fetching, that they make every effort to
not delete data until it has been successfully resolved to a new STH
via a proof.

When it is time to delete a record, it can be done in a way that
makes it more difficult for a successful flushing attack to to be
performed.

1. When the record cache has reached a certain size that is yet
    under the limit, aggressively perform proof fetching. This
    should resolve records to a small set of STHs that can be
    retained. Once a proof has been fetched, the record is safer to
    delete.

2. If proof fetching has failed, or is disabled, begin by deleting
    SCTs and Certificate Chains that have been successfully reported.
    Deletion from this set of SCTs should be done at random. For a
    client, a submission is not counted as being reported unless it
    is sent over a connection using a different SCT, so the attacker
    is faced with a recursive problem. (For a server, this step does
    not apply.)

3. Attempt to save any submissions that have failed proof fetching
    repeatedly, as these are the most likely to be indicative of an
    attack.

4. Finally, if the above steps have been followed and have not
    succeeded in reducing the size sufficiently, records may be
    deleted at random.

Note that if proof fetching is disabled (which is expected although
not required for servers) - the algorithm collapses down to 'delete
at random'.

The decision to delete records at random is intentional. Introducing
non-determinism in the decision is absolutely necessary to make it
more difficult for an adversary to know with certainty or high
confidence that the record has been successfully flushed from a
target.

11.4. Concrete Recommendations

We present the following pseudocode as a concrete outline of our
policy recommendations.

Both suggestions presented are applicable to both clients and
servers. Servers may not perform proof fetching, in which case large
portions of the pseudocode are not applicable. But it should work in
either case.

Note that we use a function 'rand()' in the pseudocode, this function
is assumed to be a cryptographically secure pseudorandom number
generator. Additionally, when N unique items are needed, they are
chosen at random by drawing a random index repeatedly until the N
unique items from an array have been chosen. Although simple, when
the array is N or near-N items in length this is inefficient. A
secure shuffle algorithm followed by selecting the first N items may
be more efficient, especially when N is large.

11.4.1. STH Pollination

The STH class contains data pertaining specifically to the STH
itself.

```
   class STH
   {
     uint16   proof_attempts
     uint16   proof_failure_count
     uint32   num_reports_to_thirdparty
     datetime timestamp
     byte[]   data
   }
```

The broader STH store itself would contain all the STHs known by an
entity participating in STH Pollination (either client or server).
This simplistic view of the class does not take into account the
complicated locking that would likely be required for a data
structure being accessed by multiple threads. Something to note
about this pseudocode is that it does not remove STHs once they have
been resolved to a newer STH. Doing so might make older STHs within
the validity window rarer and thus enable tracking.

```
   class STHStore
   {
     STH[] sth_list

     // This function is run after receiving a set of STHs from
     // a third party in response to a pollination submission
     def insert(STH[] new_sths) {
       foreach(new in new_sths) {
         if(this.sth_list.contains(new))
           continue
         this.sth_list.insert(new)
       }
     }

     // This function is called to delete the given STH
     // from the data store
     def delete_now(STH s) {
       this.sth_list.remove(s)
     }

     // When it is time to perform STH Pollination, the HTTPS client
     // calls this function to get a selection of STHs to send as
     // feedback
     def get_pollination_selection() {
       if(len(this.sth_list) < MAX_STH_TO_GOSSIP)
         return this.sth_list
       else {
         indexes = set()
         modulus = len(this.sth_list)
         outdated_sths = 0
```

```
        while(len(indexes) + outdated_sths < MAX_STH_TO_GOSSIP) {
          r = randomInt() % modulus
          if(r not in indexes)
            // Ignore STHs that are past the validity window but not
            // yet removed.
            if(now() - this.sth_list[i].timestamp < TWO_WEEKS)
              outdated_sths++;
            else
              indexes.insert(r)
        }

        return_selection = []
        foreach(i in indexes) {
          return_selection.insert(this.sth_list[i])
        }
        return return_selection
      }
    }
  }
```

   We also suggest a function that will be called periodically in the
   background, iterating through the STH store, performing a cleaning
   operation and queuing consistency proofs. This function can live as
   a member functions of the STHStore class.

```
//Just a suggestion:
#define MIN_PROOF_FAILURES_CONSIDERED_SUSPICIOUS 3

def clean_list() {
  foreach(sth in this.sth_list) {

    if(now() - sth.timestamp > TWO_WEEKS) {
      //STH is too old, we must remove it
      if(proof_fetching_enabled
         && auditor_of_last_resort_enabled
         && sth.proof_failure_count
            > MIN_PROOF_FAILURES_CONSIDERED_SUSPICIOUS) {
        queue_for_auditor_of_last_resort(sth,
                                    auditor_of_last_resort_callback)
      } else {
        delete_now(sth)
      }
    }


    else if(proof_fetching_enabled
            && now() - sth.timestamp > LOG_MMD
            && sth.proof_attempts != UINT16_MAX
            // Only fetch a proof is we have never received a proof
            // before. (This also avoids submitting something
            // already in the queue.)
            && sth.proof_attempts == sth.proof_failure_count) {
      sth.proof_attempts++
      queue_consistency_proof(sth, consistency_proof_callback)
    }
  }
}
```

   These functions also exist in the STHStore class.

```
// This function is called after successfully pollinating STHs
// to a third party. It is passed the STHs sent to the third
// party, which is the output of get_gossip_selection(), as well
// as the STHs received in the response.
def successful_thirdparty_submission_callback(STH[] submitted_sth_list,
                                              STH[] new_sths)
{
  foreach(sth in submitted_sth_list) {
    sth.num_reports_to_thirdparty++
  }

  this.insert(new_sths);
}


// Attempt auditor of last resort submissions until it succeeds
def auditor_of_last_resort_callback(original_sth, error) {
  if(!error) {
    delete_now(original_sth)
  }
}


def consistency_proof_callback(consistency_proof, original_sth, error) {
  if(!error) {
    insert(consistency_proof.current_sth)
  } else {
    original_sth.proof_failure_count++
  }
}
```

11.4.2. SCT Feedback

   The SCT class contains data pertaining specifically to an SCT itself.

```
   class SCT
   {
     uint16 proof_failure_count
     bool   has_been_resolved_to_sth
     bool   proof_outstanding
     byte[] data
   }
```

   The SCT bundle will contain the trusted certificate chain the HTTPS
   client built (chaining to a trusted root certificate.) It also
   contains the list of associated SCTs, the exact domain it is
   applicable to, and metadata pertaining to how often it has been
   reported to the third party.

```
   class SCTBundle
   {
     X509[] certificate_chain
     SCT[]  sct_list
     string domain
     uint32 num_reports_to_thirdparty

     def equals(sct_bundle) {
       if(sct_bundle.domain != this.domain)
         return false
       if(sct_bundle.certificate_chain != this.certificate_chain)
         return false
       if(sct_bundle.sct_list != this.sct_list)
         return false

       return true
     }
     def approx_equals(sct_bundle) {
       if(sct_bundle.domain != this.domain)
         return false
       if(sct_bundle.certificate_chain != this.certificate_chain)
         return false

       return true
     }

     def insert_scts(sct[] sct_list) {
       this.sct_list.union(sct_list)
       this.num_reports_to_thirdparty = 0
     }

     def has_been_fully_resolved_to_sths() {
       foreach(s in this.sct_list) {
         if(!s.has_been_resolved_to_sth && !s.proof_outstanding)
           return false
       }
       return true
     }

     def max_proof_failures() {
       uint max = 0
       foreach(sct in this.sct_list) {
         if(sct.proof_failure_count > max)
           max = sct.proof_failure_count
       }
       return max
     }
   }
```

For each domain, we store a SCTDomainEntry that holds the SCTBundles
seen for that domain, as well as encapsulating some logic relating to
SCT Feedback for that particular domain. In particular, this data
structure also contains the logic that handles domains not supporting
SCT Feedback. Its behavior is:

1. When a user visits a domain, SCT Feedback is attempted for it.
    If it fails, it will retry after a month (configurable). If it
    succeeds, excellent. SCT Feedback data is still collected and
    stored even if SCT Feedback failed.

2. After 3 month-long waits between failures, the domain will be
    marked as failing long-term. No SCT Feedback data will be stored
    beyond meta-data, but SCT Feedback will still be attempted after
    month-long waits

3. If at any point in time, SCT Feedback succeeds, all failure
    counters are reset

4. If a domain succeeds, but then begins failing, it must fail more
    than 90% of the time (configurable) and then the process begins
    at (2).

    If a domain is visited infrequently (say, once every 7 months) then
    it will be evicted from the cache and start all over again (according
    to the suggestion values in the below pseudocode).

```
//Suggestions:
// After concluding a domain doesn't support feedback, we try again
// after WAIT_BETWEEN_SCT_FEEDBACK_ATTEMPTS amount of time to see if
// they added support
#define WAIT_BETWEEN_SCT_FEEDBACK_ATTEMPTS                   1 month


// If we've waited MIN_SCT_FEEDBACK_ATTEMPTS_BEFORE_OMITTING_STORAGE
// multiplied by WAIT_BETWEEN_SCT_FEEDBACK_ATTEMPTS amount of time, we
// still attempt SCT Feedback, but no longer bother storing any data
// until the domain supports SCT Feedback
#define MIN_SCT_FEEDBACK_ATTEMPTS_BEFORE_OMITTING_STORAGE    3


// If this percentage of SCT Feedback attempts previously succeeded,
// we consider the domain as supporting feedback and is just having
// transient errors
#define MIN_RATIO_FOR_SCT_FEEDBACK_TO_BE_WORKING             .10


class SCTDomainEntry
{
  // This is the primary key of the object, the exact domain name it
  // is valid for
```

```
  string    domain

  // This is the last time the domain was contacted. For client
  // operations it is updated whenever the client makes any request
  // (not just feedback) to the domain. For server operations, it is
  // updated whenever any client contacts the domain. Responsibility
  // for updating lies OUTSIDE of the class
  public datetime last_contact_for_domain

  // This is the last time SCT Feedback was attempted for the domain.
  // It is updated whenever feedback is attempted - responsibility for
  // updating lies OUTSIDE of the class
  // This is not used when this algorithm runs on servers
  public datetime last_sct_feedback_attempt

  // This is the number of times we have waited an
  // WAIT_BETWEEN_SCT_FEEDBACK_ATTEMPTS amount of time, and still failed
  // e.g., 10 months of failures
  // This is not used when this algorithm runs on servers
  private uint16   num_feedback_loop_failures

  // This is whether or not SCT Feedback has failed enough times that we
  // should not bother storing data for it anymore. It is a small
  // function used for illustrative purposes.
  // This is not used when this algorithm runs on servers
  private bool     sct_feedback_failing_longterm()
    { num_feedback_loop_failures >=
        MIN_SCT_FEEDBACK_ATTEMPTS_BEFORE_OMITTING_STORAGE }

  // This is the number of SCT Feedback submissions attempted.
  // Responsibility for incrementing lies OUTSIDE of the class
  // (And watch for integer overflows)
  // This is not used when this algorithm runs on servers
  public uint16   num_submissions_attempted

  // This is the number of successful SCT Feedback submissions. This
  // variable is updated by the class.
  // This is not used when this algorithm runs on servers
  private uint16   num_submissions_succeeded

  // This contains all the bundles of SCT data we have observed for
  // this domain
  SCTBundle[] observed_records


  // This function can be called to determine if we should attempt
  // SCT Feedback for this domain.
  def should_attempt_feedback() {
```

```
   // Servers always perform feedback!
   if(operator_is_server)
     return true

   // If we have not tried in a month, try again
   if(now() - last_sct_feedback_attempt >
       WAIT_BETWEEN_SCT_FEEDBACK_ATTEMPTS)
     return true

   // If we have tried recently, and it seems to be working, go for it!
   if((num_submissions_succeeded / num_submissions_attempted) >
      MIN_RATIO_FOR_SCT_FEEDBACK_TO_BE_WORKING)
     return true

   // Otherwise don't try
   return false
 }

// For Clients, this function is called after a successful
// connection to an HTTPS server, with a single SCTBundle
// constructed from that connection's certificate chain and SCTs.
// For Servers, this is called after receiving SCT Feedback with
// all the bundles sent in the feedback.
def insert(SCTBundle[] bundles) {
  // Do not store data for long-failing domains
  if(sct_feedback_failing_longterm()) {
    return
  }

  foreach(b in bundles) {
    if(operator_is_server) {
      if(!passes_validity_checks(b))
        return
    }

    bool have_inserted = false
    foreach(e in this.observed_records) {
      if(e.equals(b))
        return
      else if(e.approx_equals(b)) {
        have_inserted = true
        e.insert_scts(b.sct_list)
      }
    }
    if(!have_inserted)
      this.observed_records.insert(b)
  }
  SCTStoreManager.update_cache_percentage()
```

```
  }

  // When it is time to perform SCT Feedback, the HTTPS client
  // calls this function to get a selection of SCTBundles to send
  // as feedback
  def get_gossip_selection() {
    if(len(observed_records) > MAX_SCT_RECORDS_TO_GOSSIP) {
      indexes = set()
      modulus = len(observed_records)
      while(len(indexes) < MAX_SCT_RECORDS_TO_GOSSIP) {
        r = randomInt() % modulus
        if(r not in indexes)
          indexes.insert(r)
      }

      return_selection = []
      foreach(i in indexes) {
        return_selection.insert(this.observed_records[i])
      }

      return return_selection
    }
    else
      return this.observed_records
  }

  def passes_validity_checks(SCTBundle b) {
    // This function performs the validity checks specified in
    // {{feedback-srvop}}
  }
}
```

The SCTDomainEntry is responsible for handling the outcome of a
submission report for that domain using its member function:

```
// This function is called after providing SCT Feedback
// to a server. It is passed the feedback sent to the other party, which
// is the output of get_gossip_selection(), and also the SCTBundle
// representing the connection the data was sent on.
// (When this code runs on the server, connectionBundle is NULL)
// If the Feedback was not sent successfully, error is True
def after_submit_to_thirdparty(error, SCTBundle[] submittedBundles,
                               SCTBundle connectionBundle)
{
  // Server operation in this instance is exceedingly simple
  if(operator_is_server) {
    if(error)
      return
```

```
    foreach(bundle in submittedBundles)
      bundle.num_reports_to_thirdparty++
    return
  }

  // Client behavior is much more complicated
  if(error) {
    if(sct_feedback_failing_longterm()) {
      num_feedback_loop_failures++
    }
    else if((num_submissions_succeeded / num_submissions_attempted)
            > MIN_RATIO_FOR_SCT_FEEDBACK_TO_BE_WORKING) {
      // Do nothing. num_submissions_succeeded will not be incremented
      // After enough of these failures, the ratio will fall beyond
      // acceptable
    } else {
      // The domain has begun its three-month grace period. We will
      // attempt submissions once a month
      num_feedback_loop_failures++
    }
    return
  }
  // We succeeded, so reset all of our failure states
  // Note, there is a race condition here if clear_old_data() is called
  // while this callback is outstanding.
  num_feedback_loop_failures    = 0
  if(num_submissions_succeeded != UINT16_MAX )
    num_submissions_succeeded++


  foreach(bundle in submittedBundles)
  {
    // Compare Certificate Chains, if they do not match, it counts as a
    // submission.
    if(!connectionBundle.approx_equals(bundle))
      bundle.num_reports_to_thirdparty++
    else {
      // This check ensures that a SCT Bundle is not considered reported
      // if it is submitted over a connection with the same SCTs. This
      // satisfies the constraint in Paragraph 5 of {{feedback-clisrv}}
      // Consider three submission scenarios:
      // Submitted SCTs       Connection SCTs    Considered Submitted
      // A, B                 A, B               No - no new information
      // A                    A, B               Yes - B is a new SCT
      // A, B                 A                  No - no new information
      if(connectionBundle.sct_list is NOT a subset of bundle.sct_list)
        bundle.num_reports_to_thirdparty++
    }
```

```
   }
}
```

   Instances of the SCTDomainEntry class are stored as part of a larger
   class that manages the entire SCT Cache, storing them in a hashmap
   keyed by domain. This class also tracks the current size of the
   cache, and will trigger cache eviction.

```
//Suggestions:
#define CACHE_PRESSURE_SAFE                 .50
#define CACHE_PRESSURE_IMMINENT             .70
#define CACHE_PRESSURE_ALMOST_FULL          .85
#define CACHE_PRESSURE_FULL                 .95
#define WAIT_BETWEEN_IMMINENT_CACHE_EVICTION  5 minutes

class SCTStoreManager
{
  hashmap<String, SCTDomainEntry> all_sct_entries
  uint32                          current_cache_size
  datetime                        imminent_cache_pressure_check_performed

  float current_cache_percentage() {
    return current_cache_size / MAX_CACHE_SIZE;
  }

  static def update_cache_percentage() {
    // This function calculates the current size of the cache
    // and updates current_cache_size
    /* ... perform calculations ... */
    current_cache_size = /* new calculated value */

    // Perform locking to prevent multiple of these functions being
    // called concurrently or unnecessarily
    if(current_cache_percentage() > CACHE_PRESSURE_FULL) {
        cache_is_full()
    }

    else if(current_cache_percentage() > CACHE_PRESSURE_ALMOST_FULL) {
      cache_pressure_almost_full()
    }

    else if(current_cache_percentage() > CACHE_PRESSURE_IMMINENT) {
      // Do not repeatedly perform the imminent cache pressure operation
      if(now() - imminent_cache_pressure_check_performed >
         WAIT_BETWEEN_IMMINENT_CACHE_EVICTION) {
        cache_pressure_is_imminent()
      }
    }
  }
}
```

   The SCTStoreManager contains a function that will be called
   periodically in the background, iterating through all SCTDomainEntry
   objects and performing maintenance tasks. It removes data for
   domains we have not contacted in a long time. This function is not

   intended to clear data if the cache is getting full, separate
   functions are used for that.

```
// Suggestions:
#define TIME_UNTIL_OLD_SUBMITTED_SCTDATA_ERASED     3 months
#define TIME_UNTIL_OLD_UNSUBMITTED_SCTDATA_ERASED   6 months

def clear_old_data()
{
  foreach(domainEntry in all_sct_stores)
  {
    // Queue proof fetches
    if(proof_fetching_enabled) {
      foreach(sctBundle in domainEntry.observed_records) {
        if(!sctBundle.has_been_fully_resolved_to_sths()) {
          foreach(s in bundle.sct_list) {
            if(!s.has_been_resolved_to_sth && !s.proof_outstanding) {
              sct.proof_outstanding = True
              queue_inclusion_proof(sct, inclusion_proof_callback)
            }
          }
        }
      }
    }

    // Do not store data for domains who are not supporting SCT
    if(!operator_is_server
        && domainEntry.sct_feedback_failing_longterm())
    {
      // Note that reseting these variables every single time is
      // necessary to avoid a bug
      all_sct_stores[domainEntry].num_submissions_attempted     = 0
      all_sct_stores[domainEntry].num_submissions_succeeded     = 0
      delete all_sct_stores[domainEntry].observed_records
      all_sct_stores[domainEntry].observed_records              = NULL
    }

    // This check removes successfully submitted data for
    // old domains we have not dealt with in a long time
    if(domainEntry.num_submissions_succeeded > 0
        && now() - domainEntry.last_contact_for_domain
            > TIME_UNTIL_OLD_SUBMITTED_SCTDATA_ERASED)
    {
      all_sct_stores.remove(domainEntry)
    }

    // This check removes unsuccessfully submitted data for
    // old domains we have not dealt with in a very long time
```

```
      if(now() - domainEntry.last_contact_for_domain
         > TIME_UNTIL_OLD_UNSUBMITTED_SCTDATA_ERASED)
      {
        all_sct_stores.remove(domainEntry)
      }

 SCTStoreManager.update_cache_percentage()
 }
```

   Inclusion Proof Fetching is handled fairly independently

```
// This function is a callback invoked after an inclusion proof
// has been retrieved. It can exist on the SCT class or independently,
// so long as it can modify the SCT class' members
def inclusion_proof_callback(inclusion_proof, original_sct, error)
{
  // Unlike the STH code, this counter must be incremented on the
  // callback as there is a race condition on using this counter in the
  // cache_* functions.
  original_sct.proof_attempts++
  original_sct.proof_outstanding = False
  if(!error) {
    original_sct.has_been_resolved_to_sth = True
    insert_to_sth_datastore(inclusion_proof.new_sth)
  } else {
    original_sct.proof_failure_count++
  }
}
```

   If the cache is getting full, these three member functions of the
   SCTStoreManager class will be used.

```
   // -----------------------------------------------------------------
   // This function is called when the cache is not yet full, but is
   // nearing it. It prioritizes deleting data that should be safe
   // to delete (because it has been shared with the site or resolved
   // to an STH)
   def cache_pressure_is_imminent()
   {
     bundlesToDelete = []
     foreach(domainEntry in all_sct_stores) {
       foreach(sctBundle in domainEntry.observed_records) {

         if(proof_fetching_enabled) {
           // First, queue proofs for anything not already queued.
           if(!sctBundle.has_been_fully_resolved_to_sths()) {
             foreach(sct in bundle.sct_list) {
               if(!sct.has_been_resolved_to_sth
```

```
                && !sct.proof_outstanding) {
                sct.proof_outstanding = True
                queue_inclusion_proof(sct, inclusion_proof_callback)
              }
            }
          }

          // Second, consider deleting entries that have been fully
          // resolved.
          else {
            bundlesToDelete.append( Struct(domainEntry, sctBundle) )
          }
        }

        // Third, consider deleting entries that have been successfully
        // reported
        if(sctBundle.num_reports_to_thirdparty > 0) {
          bundlesToDelete.append( Struct(domainEntry, sctBundle) )
        }
      }
    }

    // Third, delete the eligible entries at random until the cache is
    // at a safe level
    uint recalculateIndex                 = 0
    #define RECALCULATE_EVERY_N_OPERATIONS 50

    while(bundlesToDelete.length > 0 &&
          current_cache_percentage() > CACHE_PRESSURE_SAFE) {
      uint rndIndex = rand() % bundlesToDelete.length
      bundlesToDelete[rndIndex].domainEntry.observed_records.remove(
          bundlesToDelete[rndIndex].sctBundle)
      bundlesToDelete.removeAt(rndIndex)

      recalculateIndex++
      if(recalculateIndex % RECALCULATE_EVERY_N_OPERATIONS == 0) {
        update_cache_percentage()
      }
    }

    // Finally, tell the proof fetching engine to go faster
    if(proof_fetching_enabled) {
      // This function would speed up proof fetching until an
      // arbitrary time has passed. Perhaps until it has fetched
      // proofs for the number of items currently in its queue? Or
      // a percentage of them?
      proof_fetch_faster_please()
    }
```

```
      update_cache_percentage();
    }

    // -----------------------------------------------------------------
    // This function is called when the cache is almost full. It will
    // evict entries at random, while attempting to save entries that
    // appear to have proof fetching failures
    def cache_pressure_almost_full()
    {
      uint recalculateIndex                  = 0
      uint savedRecords                      = 0
      #define RECALCULATE_EVERY_N_OPERATIONS 50

      while(all_sct_stores.length > savedRecords &&
            current_cache_percentage() > CACHE_PRESSURE_SAFE) {
        uint rndIndex1 = rand() % all_sct_stores.length
        uint rndIndex2 = rand() %
            all_sct_stores[rndIndex1].observed_records.length

        if(proof_fetching_enabled) {
          if(all_sct_stores[rndIndex1].observed_records[
              rndIndex2].max_proof_failures() >
           MIN_PROOF_FAILURES_CONSIDERED_SUSPICIOUS) {
            savedRecords++
            continue
          }
        }

        // If proof fetching is not enabled we need some other logic
        else {
          if(sctBundle.num_reports_to_thirdparty == 0) {
            savedRecords++
            continue
          }
        }

        all_sct_stores[rndIndex1].observed_records.removeAt(rndIndex2)
        if(all_sct_stores[rndIndex1].observed_records.length == 0) {
          all_sct_stores.removeAt(rndIndex1)
        }

        recalculateIndex++
        if(recalculateIndex % RECALCULATE_EVERY_N_OPERATIONS == 0) {
          update_cache_percentage()
        }
      }

      update_cache_percentage();
```

```
     }

   // -----------------------------------------------------------------
   // This function is called when the cache is full, and will evict
   // cache entries at random
   def cache_is_full()
   {
     uint recalculateIndex               = 0
     #define RECALCULATE_EVERY_N_OPERATIONS 50

     while(all_sct_stores.length > 0 &&
           current_cache_percentage() > CACHE_PRESSURE_SAFE) {
       uint rndIndex1 = rand() % all_sct_stores.length
       uint rndIndex2 = rand() %
           all_sct_stores[rndIndex1].observed_records.length

       all_sct_stores[rndIndex1].observed_records.removeAt(rndIndex2)
       if(all_sct_stores[rndIndex1].observed_records.length == 0) {
         all_sct_stores.removeAt(rndIndex1)
       }


       recalculateIndex++
       if(recalculateIndex % RECALCULATE_EVERY_N_OPERATIONS == 0) {
         update_cache_percentage()
       }
     }

     update_cache_percentage();
   }
```

12. IANA considerations

   There are no IANA considerations.

13. Contributors

   The authors would like to thank the following contributors for
   valuable suggestions: Al Cutter, Andrew Ayer, Ben Laurie, Benjamin
   Kaduk, Graham Edgecombe, Josef Gustafsson, Karen Seo, Magnus Ahltorp,
   Steven Kent, Yan Zhu.

14. ChangeLog

14.1. Changes between ietf-04 and ietf-05

   o STH and SCT data formats changed to support CT v1 and v2.

   o Address ED review comments.

14.2. Changes between ietf-03 and ietf-04

   o No changes.

14.3. Changes between ietf-02 and ietf-03

   o TBD's resolved.

   o References added.

   o Pseduocode changed to work for both clients and servers.

14.4. Changes between ietf-01 and ietf-02

   o Requiring full certificate chain in SCT Feedback.

   o Clarifications on what clients store for and send in SCT Feedback
      added.

   o SCT Feedback server operation updated to protect against DoS
      attacks on servers.

   o Pre-Loaded vs Locally Added Anchors explained.

   o Base for well-known URL's changed.

   o Remove all mentions of monitors - gossip deals with auditors.

   o New sections added: Trusted Auditor protocol, attacks by actively
      malicious log, the Dual-CA compromise attack, policy
      recommendations,

14.5. Changes between ietf-00 and ietf-01

   o Improve language and readability based on feedback from Stephen
      Kent.

   o STH Pollination Proof Fetching defined and indicated as optional.

   o 3-Method Ecosystem section added.

   o Cases with Logs ceasing operation handled.

o Text on tracking via STH Interaction added.

o Section with some early recommendations for mixing added.

o Section detailing blocking connections, frustrating it, and the
   implications added.

14.6. Changes between -01 and -02

o STH Pollination defined.

o Trusted Auditor Relationship defined.

o Overview section rewritten.

o Data flow picture added.

o Section on privacy considerations expanded.

14.7. Changes between -00 and -01

o Add the SCT feedback mechanism: Clients send SCTs to originating
   web server which shares them with auditors.

o Stop assuming that clients see STHs.

o Don't use HTTP headers but instead .well-known URL's - avoid that
   battle.

o Stop referring to trans-gossip and trans-gossip-transport-https -
   too complicated.

o Remove all protocols but HTTPS in order to simplify - let's come
   back and add more later.

o Add more reasoning about privacy.

o Do specify data formats.

15. References

15.1. Normative References

   [RFC-6962-BIS-27]
             Laurie, B., Langley, A., Kasper, E., Messeri, E., and R.
             Stradling, "Certificate Transparency", October 2017,
             <https://datatracker.ietf.org/doc/draft-ietf-trans-
             rfc6962-bis/>.

   [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>.

   [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data
              Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March
              2014, <https://www.rfc-editor.org/info/rfc7159>.

   [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext
              Transfer Protocol Version 2 (HTTP/2)", RFC 7540,
              DOI 10.17487/RFC7540, May 2015, <https://www.rfc-
              editor.org/info/rfc7540>.

15.2. Informative References

   [double-keying]
              Perry, M., Clark, E., and S. Murdoch, "Cross-Origin
              Identifier Unlinkability", May 2015,
              <https://www.torproject.org/projects/torbrowser/
              design/#identifier-linkability>.

   [draft-ct-over-dns]
              Laurie, B., Phaneuf, P., and A. Eijdenberg, "Certificate
              Transparency over DNS", February 2016,
              <https://github.com/google/certificate-transparency-
              rfcs/blob/master/dns/draft-ct-over-dns.md>.

   [draft-ietf-trans-threat-analysis-12]
              Kent, S., "Attack and Threat Model for Certificate
              Transparency", October 2017,
              <https://datatracker.ietf.org/doc/draft-ietf-trans-threat-
              analysis/>.

   [dual-ca-compromise-attack]
              Gillmor, D., "can CT defend against dual CA compromise?",
              n.d., <https://www.ietf.org/mail-
              archive/web/trans/current/msg01984.html>.

   [gossip-mixing]
              Ritter, T., "A Bit on Certificate Transparency Gossip",
              June 2016, <https://ritter.vg/blog-
              a_bit_on_certificate_transparency_gossip.html>.

   [trickle] Serjantov, A., Dingledine, R., and . Paul Syverson, "From
              a Trickle to a Flood: Active Attacks on Several Mix
              Types", October 2002,
              <http://freehaven.net/doc/batching-taxonomy/taxonomy.pdf>.

Authors' Addresses

   Linus Nordberg
   NORDUnet

   Email: linus@nordu.net

   Daniel Kahn Gillmor
   ACLU

   Email: dkg@fifthhorseman.net

   Tom Ritter

   Email: tom@ritter.vg

                   Certificate Transparency Version 2.0
                     draft-ietf-trans-rfc6962-bis-42

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.

   [RFC Editor: please update 'RFCXXXX' to refer to this document, once
   its RFC number is known, through the document. Also, the OID
   assigned below must also appear in the appendix as indicated.]

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 4 March 2022.

Copyright Notice

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)). A typical definition of "public" can
   be found in [CABBR].

   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.
   A log may also limit the length of the chain it is willing to accept;
   such chains must also end with an acceptable trust anchor. 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. While mechanisms are being
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", "NOT RECOMMENDED", "MAY", and
"OPTIONAL" in this document are to be interpreted as described in BCP
14 [RFC2119] [RFC8174] when, and only when, they appear in all
capitals, as shown here.

1.2. Data Structures

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

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.

1.3. Major Differences from CT 1.0

This document revises and obsoletes the 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.

* 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

A full description of Merkle Hash Tree is beyond the scope of this document. Briefly, it is a binary tree where each non-leaf node is a hash of its children. For CT, the number of children is at most two. Additional information can be found in the Introduction and Reference section of [RFC8391].

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). This document establishes a registry of acceptable hash algorithms (see Section 10.2.1). 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:

* || denotes concatenation

* : denotes concatenation of lists

* 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 "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, where "LSB" means
    the least significant bit. 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 (the sub-items of 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" from the "inclusion_proof_v2" structure
    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, using the appropriate step
   below:

   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 "consistency_path" is an empty array, stop and fail the proof
    verification.

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

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

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

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

6. 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.

   7. 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:

```
            hash
           /    \
          /      \
         /        \
        /          \
       /            \
      k              l
     / \            / \
    /   \          /   \
   /     \        /     \
  g       h      i       j
 / \     / \    / \      |
 a b     c d    e f      d6
 | |     | |    | |
 d0 d1   d2 d3  d4 d5
```
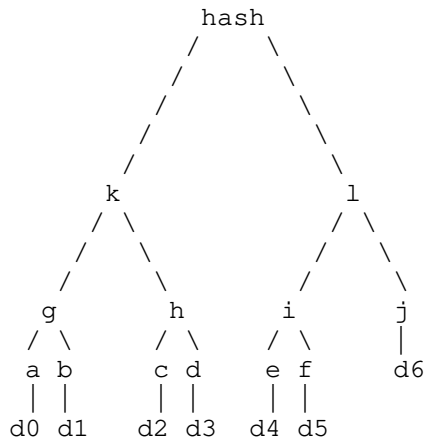
   The inclusion proof for d0 is [b, h, l].

   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:

```
        hash0               hash1=k
        / \                 / \
       /   \               /   \
      /     \             /     \
     g       c           g       h
    / \      |          / \     / \
   a   b     d2        a   b   c   d
   |   |               |   |   |   |
   d0  d1              d0  d1  d2  d3

            hash2                       hash
            / \                         / \
           /   \                       /   \
          /     \                     /     \
         /       \                   /       \
        /         \                 /         \
       k           i               k           l
      / \         / \             / \         / \
     /   \       e   f           /   \       /   \
    /     \      |   |          /     \     /     \
   g       h     d4  d5        g       h   i       j
  / \     / \                 / \     / \ / \      |
 a   b   c   d               a   b   c  d e   f     d6
 |   |   |   |               |   |   |  | |   |
 d0  d1  d2  d3              d0  d1  d2 d3 d4 d5
```
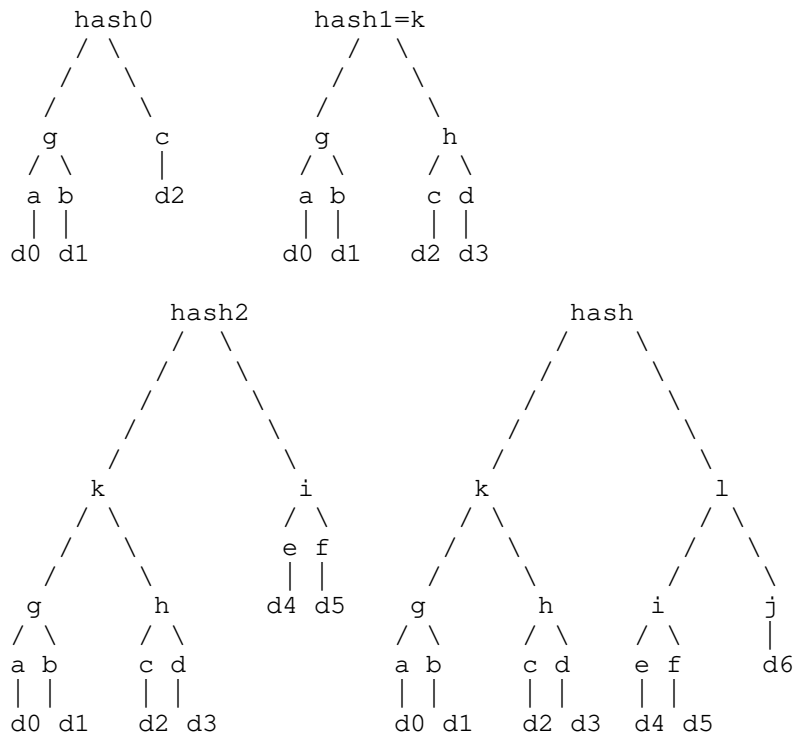
The consistency proof between hash0 and hash is PROOF(3, D[7]) = [c,
d, g, l]. c, g are used to verify hash0, and d, l are additionally
used to show hash is consistent with hash0.

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

When signing data structures, a log MUST use one of the signature
algorithms from the IANA CT Signature Algorithms registry, described
in Section 10.2.2.

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 as described in [X690].

   * "SignedData.version" MUST be v3(3).

   * "SignedData.digestAlgorithms" MUST be the same as 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 the IANA CT Hash Algorithms Registry, described in Section 10.2.1.

- "signedAttrs" MUST be present and MUST contain two attributes:

  o A content-type attribute whose value is the same as "SignedData.encapContentInfo.eContentType".

  o 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 intends to issue the corresponding certificate (see Section 3.2.1).

- "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.

3.2.1. Binding Intent to Issue

   Under normal circumstances, there will be a short delay between
   precertificate submission and issuance of the corresponding
   certificate. Longer delays are to be expected occasionally (e.g.,
   due to log server downtime), and in some cases the CA might not
   actually issue the corresponding certificate. Nevertheless, a
   precertificate's "signature" indicates the CA's binding intent to
   issue the corresponding certificate, which means that:

   * Misissuance of a precertificate is considered equivalent to
      misissuance of the corresponding certificate. The CA should
      expect to be held to account, even if the corresponding
      certificate has not actually been issued.

   * Upon observing a precertificate, a client can reasonably presume
      that the corresponding certificate has been issued. A client may
      wish to obtain status information (e.g., by using the Online
      Certificate Status Protocol [RFC6960] or by checking a Certificate
      Revocation List [RFC5280]) about a certificate that is presumed to
      exist, especially if there is evidence or suspicion that the
      corresponding precertificate was misissued.

   * TLS clients may have policies that require CAs to be able to
      revoke, and to provide certificate status services for, each
      certificate that is presumed to exist based on the existence of a
      corresponding precertificate.

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, as
   discussed in Section 11.3. 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 immutable parameters, which are used by clients to communicate with the log and to verify log artifacts. Except for the Final Signed Tree Head (STH), each of these parameters MUST be established before the log operator begins to operate the log.

Base URL: The prefix used to construct URLs ([RFC3986]) for client messages (see Section 5). The base URL MUST be an "https" URL, MAY contain a port, MAY contain a path with any number of path segments, but MUST NOT contain a query string, fragment, or trailing "/". Example: https://ct.example.org/blue

Hash Algorithm: The hash algorithm used for the Merkle Tree (see Section 10.2.1).

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. This document deliberately does not specify any limits on the value, to allow for experimentation.

Version: The version of the protocol supported by the log (currently

   1 or 2).

   Maximum Chain Length: The longest certificate 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. This value MUST be
      provided in the form of a TransItem of type "signed_tree_head_v2".
      If a log is still accepting entries, this value should not be
      provided.

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

4.2. Evaluating Submissions

   A log determines whether to accept or reject a submission by
   evaluating it against the minimum acceptance criteria (see
   Section 4.2.1) and against the log's discretionary acceptance
   criteria (see Section 4.2.2).

   If the acceptance criteria are met, the log SHOULD accept the
   submission. (A log may decide, for example, to temporarily reject
   acceptable submissions to protect itself against denial-of-service
   attacks).

   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.

4.2.1. Minimum Acceptance Criteria

   To ensure that logged certificates and precertificates are
   attributable to an accepted trust anchor, to set clear expectations
   for what monitors would find in the log, and to avoid being
   overloaded by invalid submissions, the log MUST reject a submission
   if any of the following conditions are not met:

* The "submission", "type" and "chain" inputs MUST be set as
  described in Section 5.1. The log MUST NOT accommodate misordered
  CA certificates or use any other source of intermediate CA
  certificates to attempt certification path construction.

* Each of the zero or more intermediate CA certificates in the chain
  MUST have one or both of the following features:

  - The Basic Constraints extension with the cA boolean asserted.

  - The Key Usage extension with the keyCertSign bit asserted.

* Each certificate in the chain MUST fall within the limits imposed
  by the zero or more Basic Constraints pathLenConstraint values
  found higher up the chain.

* Precertificate submissions MUST conform to all of the requirements
  in Section 3.2.

4.2.2. Discretionary Acceptance Criteria

If the minimum acceptance criteria are met but the submission is not
fully valid according to [RFC5280] verification rules (e.g., the
certificate or precertificate has expired, is not yet valid, has been
revoked, exhibits ASN.1 DER encoding errors but the log can still
parse it, etc), then the acceptability of the submission is left to
the log's discretion. It is useful for logs to accept such
submissions in order to accommodate quirks of CA certificate-issuing
software and to facilitate monitoring of CA compliance with
applicable policies and technical standards. However, it is
impractical for this document to enumerate, and for logs to consider,
all of the ways that a submission might fail to comply with
[RFC5280].

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).

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 provide this chain for auditing upon
request (see Section 5.6) so that the CA cannot avoid 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 10.2.5). One way to get an OID arc, from which OIDs can be
allocated, is to request a Private Enterprise Number from IANA, by
completing the registration form (https://pen.iana.org/pen/
PenApplication.page). The only advantage of the registry is that the
DER encoding can be small. (Recall that OID allocations do not
require a central registration, although logs will most likely want
to make themselves known to potential clients through out of band
means.) 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 full DER encoding (including the
tag and length) of the OID can be reproduced simply by prepending an
OBJECT IDENTIFIER tag (0x06) to the opaque vector length and
contents.

The OID used to identify a log is limited such that the DER encoding
of its value, excluding the tag and length, MUST be no longer than
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 {
        x509_entry_v2(0x0100), precert_entry_v2(0x0101),
        x509_sct_v2(0x0102), precert_sct_v2(0x0103),
        signed_tree_head_v2(0x0104), consistency_proof_v2(0x0105),
        inclusion_proof_v2(0x0106),

        /* Reserved Code Points */
        reserved_rfc6962(0x0000..0x00FF),
        reserved_experimentaluse(0xE000..0xEFFF),
        reserved_privateuse(0xF000..0xFFFF),
        (0xFFFF)
    } 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 10.2.3
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 SCTs (Section 4.8) and STHs (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 10.2.4).
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" is byte-for-byte identical to 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<1..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. All SCT extensions are similar to non-critical X.509v3 extensions (i.e., the "mustUnderstand" field is not set), and a recipient SHOULD ignore any extension it does not understand. 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, using the format defined in
Section 4.7.

"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<1..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<0..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 as described in Section 2.1.4.

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<0..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 as described in
   Section 2.1.3.

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. In order to avoid that, the
   following actions SHOULD be taken:

   * Make it known to clients and monitors that the log will be frozen.
      This is not part of the API, so it will have to be done via a
      relevant out-of-band mechanism.

   * 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 according to section 4 of
   [RFC4648] as specified in the individual messages.

   Clients are configured with a log's base URL, which is one of the
   log's parameters. Clients 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 [RFC8820]. 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, to allow for experimentation. Any fields that are
not understood SHOULD be ignored.

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 "<Base URL>/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. |
+-----------+--------------------------------+
```

Table 1

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 field in order to request a minimum time for
the client to wait before retrying the request. In the absence of
this header field, this document does not specify a minimum.

Clients SHOULD treat any 4xx error as a problem with the request and
not attempt to resubmit without some modification to the request.
The full status code MAY provide additional details.

This document deliberately does not provide more specific guidance on
the use of HTTP status codes.

5.1. Submit Entry to Log

   POST <Base URL>/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 JSON strings, each of which
        is a base64 encoded CA certificate. 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.

                               Table 2

   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 is not able to 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. For example, if "type" was 2 (indicating "precert_sct_v2") then all three "TransItem"s could be embedded in the certificate.

5.2. Retrieve Latest STH

   GET <Base URL>/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 STHs

   GET <Base URL>/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".

Table 3

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 <Base URL>/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. A v2 STH must exist for the "tree_size". Because of skew, the front-end may not know the requested tree head. 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 tree head 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 certificate (as
        specified by the "hash" parameter) 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. |
    +----------------+--------------------------------------+

                                Table 4

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

5.5. Retrieve Merkle Inclusion Proof, STH and Consistency Proof by Leaf
      Hash

    GET <Base URL>/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. A v2 STH
    must exist for the "tree_size".

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

```
+===================+====================================+
| Case              | Response                           |
+===================+====================================+
| latest STH <      | Return latest STH                  |
| requested tree head|                                   |
+-------------------+------------------------------------+
| latest STH >      | Return latest STH and a consistency|
| requested tree head| proof between it and the requested|
|                   | tree head (see Section 5.3)        |
+-------------------+------------------------------------+
| index of requested| Return "inclusion"                 |
| hash < latest STH |                                    |
+-------------------+------------------------------------+
```

Table 5

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 certificate (as
      specified by the "hash" parameter) in the selected 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 tree head 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 <Base URL>/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 equivalent to 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". Note that limit on the number of entries
is not immutable and therefore the restriction may be changed or
lifted at any time and is not listed with the other Log Parameters in
Section 4.1.

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".                              |
+---------------+-------------------------------------+
```

Table 6

5.7. Retrieve Accepted Trust Anchors

GET <Base URL>/ct/v2/get-anchors

No inputs.

Outputs: certificates: An array of JSON strings, each of which is a
     base64 encoded CA certificate that is 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.

   This data is not signed and the protocol depends on the security
   guarantees of TLS to ensure correctness.

6. TLS Servers

   CT-using TLS servers MUST use at least one of the mechanisms
   described below to present one or more SCTs from one or more logs to
   each TLS client during full TLS handshakes, when requested by the
   client, where each SCT corresponds to the server certificate. (Of
   course, a server can only send a TLS extension if the client has
   specified it first.) Servers SHOULD also present corresponding
   inclusion proofs and STHs.

   A server can provide SCTs using a TLS 1.3 extension (Section 4.2 of
   [RFC8446]) with type "transparency_info" (see Section 6.5). 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.

   The server may also use an Online Certificate Status Protocol (OCSP)
   [RFC6960] response extension (see Section 7.1.1), providing the OCSP
   response as part of the TLS handshake. Providing a response during a
   TLS handshake is popularly known as "OCSP stapling." For TLS 1.3,
   the information is encoded as an extension in the "status_request"
   extension data; see Section 4.4.2.1 of [RFC8446]. For TLS 1.2
   ([RFC5246]), the information is encoded in the "CertificateStatus"
   message; see Section 8 of [RFC6066]. Using stapling also allows SCTs
   and inclusion proofs to be updated on the fly.

   CT information can also be encoded as an extension in the X.509v3
   certificate (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 may subsequently consider the certificate to be non-compliant
   and in need of re-issuance or the use of one of the other two methods
   for delivering CT information.

6.1. TLS Client Authentication

   This specification includes no description of how a TLS server can
   use CT for TLS client certificates. While this may be useful, it is
   not documented here for the following reasons:

   * The greater security exposure is for clients to end up interacting
      with an illegitimate server.

   * In general, TLS client certificates are not expected to be
      submitted to CT logs, particularly those intended for general
      public use.

A future version could include such information.

6.2. 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. Note that client discovery, trust, and distrust of
  logs is expected to be handled out-of-band and is out of scope of
  this document.

* 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.

6.3. 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).

6.4. Presenting SCTs, inclusions proofs and STHs

In each "TransItemList" that is sent during a TLS handshake, the TLS
server MUST include a "TransItem" structure of type "x509_sct_v2" or
"precert_sct_v2".

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".

6.5. 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 6.4), 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". If the list is empty then the server SHOULD
   omit the "extension_data" element, but MAY send it with an empty
   array.

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.

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 6.5) 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 order to make use of a received SCT, the TLS client MUST first validate it as follows:

* Compute 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.

* Verify the SCT's "signature" against the computed signature input 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.

If the TLS client does not have the corresponding log's parameters, it cannot attempt to validate the SCT. When evaluating compliance (see Section 8.1.6), the TLS client will consider only those SCTs that it was able to validate.

Note that SCT validation is not a substitute for the normal validation of the server certificate and its chain.

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 6.4).

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.

8.2. 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. This document does not specify the polling frequency,
       to allow for experimentation.

   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. Additions to existing registries

This sub-section defines additions to existing registries.

10.1.1. New Entry to the TLS ExtensionType Registry

IANA is asked to add the following entry to the "TLS ExtensionType
Values" registry defined in [RFC8446], with an assigned Value:

+=======+===================+============+=============+===========+
| Value | Extension Name | TLS 1.3 | Recommended | Reference |
+=======+===================+============+=============+===========+
| TBD | transparency_info | CH, CR, CT | Y | RFCXXXX |
+-------+-------------------+------------+-------------+-----------+

Table 7

10.1.2. URN Sub-namespace for TRANS (urn:ietf:params:trans)

IANA is requested to add a new entry in the "IETF URN Sub-namespace
for Registered Protocol Parameter Identifiers" registry, following
the template in [RFC3553]:

Registry name: trans

Specification: RFCXXXX

Repository: https://www.iana.org/assignments/trans

Index value: No transformation needed.

10.2. New CT-Related registries

IANA is requested to add a new protocol registry, "Public Notary
Transparency", to the list that appears at https://www.iana.org/
assignments/

The rest of this section defines sub-registries to be created within the new Public Notary Transparency registry.

10.2.1. Hash Algorithms

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

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

Table 8

The Designated Expert(s) 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.

10.2.2. Signature Algorithms

IANA is asked to establish a registry of signature algorithm values, named "Signature Algorithms".

The following notes should be added:

* This is a subset of the TLS SignatureScheme Registry, limited to those algorithms that are appropriate for CT. A major advantage of this is leveraging the expertise of the TLS working group and its Designated Expert(s).

* The value "0x0403" appears twice. While this may be confusing, it is okay because the verification process is the same for both algorithms, and the choice of which to use when generating a signature is purely internal to the log server.

The registry should initially consist of:

SignatureScheme Value	Signature Algorithm	Reference / Assignment Policy
0x0000 - 0x0402	Unassigned	Specification Required
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]
0x0404 - 0x0806	Unassigned	Specification Required
ed25519(0x0807)	Ed25519 (PureEdDSA with the edwards25519 curve)	[RFC8032]
0x0808 - 0xFDFF	Unassigned	Expert Review
0xFE00 - 0xFEFF	Reserved	Experimental Use
0xFF00 - 0xFFFF	Reserved	Private Use

Table 9

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

10.2.3. VersionedTransTypes

IANA is asked to establish a registry of "VersionedTransType" values, named "VersionedTransTypes".

The following note should be added:

*   The range 0x0000..0x00FF is reserved so that v1 SCTs are
    distinguishable from v2 SCTs and other "TransItem" structures.

The registry should initially consist of:

Value	Type and Version	Reference / Assignment Policy
=======	==================	===============================
0x0000 – 0x00FF	Reserved	[RFC6962]
0x0100	x509_entry_v2	RFCXXXX
0x0101	precert_entry_v2	RFCXXXX
0x0102	x509_sct_v2	RFCXXXX
0x0103	precert_sct_v2	RFCXXXX
0x0104	signed_tree_head_v2	RFCXXXX
0x0105	consistency_proof_v2	RFCXXXX
0x0106	inclusion_proof_v2	RFCXXXX
0x0107 – 0xDFFF	Unassigned	Specification Required
0xE000 – 0xEFFF	Reserved	Experimental Use
0xF000 – 0xFFFF	Reserved	Private Use

Table 10

The Designated Expert(s) should review the public specification to
ensure that it is detailed enough to ensure implementation
interoperability.

10.2.4. Log Artifact Extension Registry

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

```
+==============+===========+=====+==============================+
| ExtensionType | Status    | Use | Reference / Assignment Policy |
+==============+===========+=====+==============================+
| 0x0000 -     | Unassigned | n/a | Specification Required       |
| 0xDFFF       |           |     |                              |
+--------------+-----------+-----+------------------------------+
| 0xE000 -     | Reserved  | n/a | Experimental Use             |
| 0xEFFF       |           |     |                              |
+--------------+-----------+-----+------------------------------+
| 0xF000 -     | Reserved  | n/a | Private Use                  |
| 0xFFFF       |           |     |                              |
+--------------+-----------+-----+------------------------------+
```

                              Table 11

   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.

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

10.2.5. Log IDs Registry

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

```
+=================+==============+==============+===================+
| Log ID          | Log Base URL | Log Operator | Reference /       |
|                 |              |              | Assignment Policy |
+=================+==============+==============+===================+
| 1.3.101.8192 -  | Unassigned   | Unassigned   | First Come First  |
| 1.3.101.16383   |              |              | Served            |
+-----------------+--------------+--------------+-------------------+
| 1.3.101.80.0 -  | Unassigned   | Unassigned   | First Come First  |
| 1.3.101.80.*    |              |              | Served            |
+-----------------+--------------+--------------+-------------------+
```

                              Table 12

All OIDs in the range from 1.3.101.8192 to 1.3.101.16383 have been
set aside for Log IDs. 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 also been set aside for Log IDs. 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 MUST be accompanied
by:

* the Log's Base URL (see Section 4.1).

* the Log Operator's contact details.

IANA is asked to reject any request to update a Log ID or Log Base
URL in this registry, because these fields are immutable (see
Section 4.1).

IANA is asked to accept requests from log operators to update their
contact details in this registry.

Since log operators can choose to not use this registry (see
Section 4.4), it is not expected to be a global directory of all
logs.

10.2.6. Error Types Registry

IANA is requested to create a new registry for errors, the "Error
Types" registry.

Requirements for this registry are Specification Required.

This registry should have the following three fields:

Field Name	Type	Reference
identifier	string	RFCXXXX
meaning	string	RFCXXXX
reference	string	RFCXXXX

Table 13

The initial values are as follows, taken from the text above:

Identifier	Meaning	Reference
============	=========	===========
malformed	The request could not be parsed.	RFCXXXX
badSubmission	"submission" is neither a valid certificate nor a valid precertificate	RFCXXXX
badType	"type" is neither 1 nor 2	RFCXXXX
badChain	The first element of "chain" is not the certifier of the "submission", or the second element does not certify the first, etc.	RFCXXXX
badCertificate	One or more certificates in the "chain" are not valid (e.g., not properly encoded)	RFCXXXX
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	RFCXXXX
shutdown	The log is no longer accepting submissions	RFCXXXX
firstUnknown	"first" is before the latest known STH but is not from an existing STH.	RFCXXXX
secondUnknown	"second" is before the latest known STH but is not from an existing STH.	RFCXXXX
secondBeforeFirst	"second" is smaller than "first".	RFCXXXX
hashUnknown	"hash" is not the hash of a known leaf (may be caused by skew or by a known certificate not yet merged).	RFCXXXX
treeSizeUnknown	"hash" is before the latest	RFCXXXX

	known STH but is not from an existing STH.		
startUnknown	"start" is greater than the number of entries in the Merkle tree.	RFCXXXX	
endBeforeStart	"start" cannot be greater than "end".	RFCXXXX	

Table 14

10.3. OID Assignment

IANA is asked to assign one object identifier from the "SMI Security for PKIX Module Identifier" registry to identify the ASN.1 module in Appendix B of this document with an assigned Decimal value.

Decimal	Description	References
TBD	id-mod-public-notary-v2	RFCXXXX

Table 15

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.

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.

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 multiple clients comparing their instances of the STHs. This technique, known as "gossip," is an active area of research and not defined here. 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.

Clients that 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.

11.4. 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 6.2).

11.5. Leakage of DNS Information

Malicious monitors can use logs to learn about the existence of domain names that might not otherwise be easy to discover. Some subdomain labels may reveal information about the service and software for which the subdomain is used, which in turn might facilitate targeted attacks.

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, Rich Salz, 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", 1 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, 24 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>.

   [RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An
              IETF URN Sub-namespace for Registered Protocol
              Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June
              2003, <https://www.rfc-editor.org/info/rfc3553>.

   [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
              Resource Identifier (URI): Generic Syntax", STD 66,
              RFC 3986, DOI 10.17487/RFC3986, January 2005,
              <https://www.rfc-editor.org/info/rfc3986>.

   [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>.

   [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
              (TLS) Protocol Version 1.2", RFC 5246,
              DOI 10.17487/RFC5246, August 2008,
              <https://www.rfc-editor.org/info/rfc5246>.

   [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>.

   [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>.

   [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>.

   [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>.

   [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>.

   [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>.

   [RFC8174]   Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC
               2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174,
               May 2017, <https://www.rfc-editor.org/info/rfc8174>.

   [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>.

   [RFC8391]   Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A.
               Mohaisen, "XMSS: eXtended Merkle Signature Scheme",
               RFC 8391, DOI 10.17487/RFC8391, May 2018,
               <https://www.rfc-editor.org/info/rfc8391>.

   [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>.

    [X690]   ITU-T, "Information technology - ASN.1 encoding Rules:
              Specification of Basic Encoding Rules (BER), Canonical
              Encoding Rules (CER) and Distinguished Encoding Rules
              (DER)", ISO/IEC 8825-1:2002, November 2015.

13.2. Informative References

    [CABBR]   CA/Browser Forum, "Baseline Requirements for the Issuance
              and Management of Publicly-Trusted Certificates", 2020,
              <https://cabforum.org/wp-content/uploads/CA-Browser-Forum-
              BR-1.7.3.pdf>.

    [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>.

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

    [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>.

    [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>.

    [RFC8820]   Nottingham, M., "URI Design and Ownership", BCP 190,
                RFC 8820, DOI 10.17487/RFC8820, June 2020,
                <https://www.rfc-editor.org/info/rfc8820>.

Appendix A. Supporting v1 and v2 simultaneously (Informative)

   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.

Appendix B. An ASN.1 Module (Informative)

   The following ASN.1 module may be useful to implementors.

   CertificateTransparencyV2Module-2021
    -- { id-mod-public-notary-v2 from above, in
          iso(1) identified-organization(3) ...
       form }
   DEFINITIONS IMPLICIT TAGS ::= BEGIN

```
-- EXPORTS ALL --

IMPORTS
  EXTENSION
  FROM PKIX-CommonTypes-2009 -- RFC 5912
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkixCommon-02(57) }

  CONTENT-TYPE
  FROM CryptographicMessageSyntax-2010  -- RFC 6268
    { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)
      pkcs-9(9) smime(16) modules(0) id-mod-cms-2009(58) }

  TBSCertificate
  FROM PKIX1Explicit-2009 -- RFC 5912
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkix1-explicit-02(51) }
;

--
-- Section 3.2.  Precertificates
--

ct-tbsCertificate CONTENT-TYPE ::= {
  TYPE TBSCertificate
  IDENTIFIED BY id-ct-tbsCertificate }

id-ct-tbsCertificate OBJECT IDENTIFIER ::= { 1 3 101 78 }

--
-- Section 7.1.  Transparency Information X.509v3 Extension
--

ext-transparencyInfo EXTENSION ::= {
   SYNTAX TransparencyInformationSyntax
   IDENTIFIED BY id-ce-transparencyInfo
   CRITICALITY { FALSE } }

id-ce-transparencyInfo OBJECT IDENTIFIER ::= { 1 3 101 75 }

TransparencyInformationSyntax ::= OCTET STRING

--
-- Section 7.1.1.  OCSP Response Extension
--
```

```
ext-ocsp-transparencyInfo EXTENSION ::= {
    SYNTAX TransparencyInformationSyntax
    IDENTIFIED BY id-pkix-ocsp-transparencyInfo
    CRITICALITY { FALSE } }

id-pkix-ocsp-transparencyInfo OBJECT IDENTIFIER ::=
    id-ce-transparencyInfo

--
-- Section 8.1.2.  Reconstructing the TBSCertificate
--

ext-embeddedSCT-CTv1 EXTENSION ::= {
    SYNTAX SignedCertificateTimestampList
    IDENTIFIED BY id-ce-embeddedSCT-CTv1
    CRITICALITY { FALSE } }

id-ce-embeddedSCT-CTv1 OBJECT IDENTIFIER ::= {
    1 3 6 1 4 1 11129 2 4 2 }

SignedCertificateTimestampList ::= OCTET STRING

END
```

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

                Attack and Threat Model for Certificate Transparency
                     draft-ietf-trans-threat-analysis-16

Abstract

   This document defines an attack model and discusses threats based on
   the system design presented in [I-D.ietf-trans-rfc6962-bis]. It
   analyzes potential vulnerabilities associated with that design, and
   considers compromises of system elements and malicious behavior by
   such elements. It does not consider implementation vulnerabilities,
   including ones that might enable denial of service attacks against
   these elements.

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 April 7, 2019.

the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

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 (aka
   erroneous) 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 often are application-specific. For example, certificates used
   in the 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 context.

   There are three classes of beneficiaries of CT: certificate Subjects,
   CAs, and relying parties (RPs). In the initial focus context of CT,
   Subjects are web sites and RPs are users of browsers employing HTTPS
   to access these web sites. (In some contexts human users may not be
   the final arbiters of what certificates are accepted, e.g., an
   organization may The CAs that benefit are issuers of certificates
   used to authenticate 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.

Certificate Revocations Lists (CRLs) [RFC5280] and the Online Certificate Status Protocol (OCSP) data [RFC6960] are the primary certificate revocation mechanisms established by IETF standards. Browsers may make use of proprietary mechanisms to effect revocation status checking, in lieu or in addition to the mechanisms noted above. If a certificate contains an Authority Information Access (AIA) extension [RFC5280], it directs a relying party to an OCSP server to which a request can be directed. A browser may also request OCSP responses from a TLS server with which it is communicating [RFC6066][RFC6961].

RFC 5280 does not require inclusion of an AIA extension in certificates, so a browser cannot assume that this extension will be present. The Certification Authority browser Forum (CABF) Baseline Requirements and Extended Validation Guidelines do mandate inclusion of this extension in EE certificates (in conjunction with their certificate policies). (See cabforum.org [1] for the most recent versions of these policies.)

As noted above, browser vendors may employ proprietary means of conveying certificate revocation status information to their products, e.g., via a blacklist that enumerates revoked certificates (EE or CA). Such capabilities enable a browser vendor to cause browsers to reject any certificates on the blacklist. This approach also can be employed to remedy 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/erroneous certificate is logged.

A relying party (e.g., browser user) benefits from CT if it rejects a bogus certificate, i.e., treats it as invalid. (Note that [I-D.ietf-trans-rfc6962-bis] notes that anyone can elect to monitor logs for mis-issuance, indicating that there is a potentially larger, unspecified set of potential beneficiaries.) 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, even in the
absence of an SCT. (An RP also is protected if a browser vendor
"blacklists" a certificate or places a CA on a "bad-CA-list", causing
all certs issued by the CA to be treated as invalid.) An RP also may
benefit from CT if the RP validates an SCT associated with a
certificate (see 8.1.3 in [I-D.ietf-trans-rfc6962-bis]), and rejects
the certificate if the Signed certificate Timestamp (SCT)
[I-D.ietf-trans-rfc6962-bis] is invalid. If an RP acquires and
verifies an inclusion proof for a certificate that claims to have
been logged has a valid log entry (8.1.4 in
[I-D.ietf-trans-rfc6962-bis]), the RP probably would have a higher
degree of confidence that the certificate is not bogus. 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 potential privacy violation.
Thus this attack model does not assume that all RPs will check log
entries.

A CA benefits from CT when it (acting as a Monitor for its clients)
detects a (mis-issued) certificate that represents the same Subject
name as a legitimate certificate issued by the CA.

Note that all RPs may benefit from CT even if they do nothing with
SCTs. If Monitors inform Subjects of potential 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.

Also note that one proposal [I-D.ietf-trans-gossip] for distributing
Audit information (to detect misbehaving logs) calls for a browser to
send SCTs it receives to the corresponding website when visited by
the browser. If a website acquires an inclusion proof from a log for
each (unique) SCT it receives in this fashion, this would cause a
bogus SCT to be discovered, and, presumably, trigger a revocation
request.

Logging [I-D.ietf-trans-rfc6962-bis] is the central element of CT.
Logging enables a Monitor to detect a bogus certificate based on
reference information provided by the certificate Subject. (Monitors
also perform an Audit function.) Logging of certificates thus helps
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 detection and subsequent revocation of bogus certificates in some circumstances.

Auditing is a function employed by CT to detect misbehavior by logs and to deter mis-issuance that is abetted by misbehaving logs. Auditing detects several types of log misbehavior, including failures to adhere to the advertised Maximum Merge Delay (MMD) and Signed Tree Head (STH) frequency count [I-D.ietf-trans-rfc6962-bis] violating the append-only property, and providing inconsistent views of the log to different log clients. The first three of these are relatively easy for an individual auditor to detect, but the last form of misbehavior requires communication among multiple log clients. Monitors ought not trust logs that are detected misbehaving. Thus the Audit function does not detect mis-issuance per se. The CT design identifies audit functions designed to detect several types of misbehavior. However, mechanisms to detect some forms of log misbehavior are not yet standardized.

Figure 1a (below) illustrates the data exchanges among the major elements of the CT system, in the contest of CA submission of certificate (or pre-certificates) to Logs. It is based on the log specification [I-D.ietf-trans-rfc6962-bis] 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.

Figure 1b (later) illustrates data exchanges in the context where a Subject submit a certificate to a Log.

```
+----+         +---------+         +---------+
| CA |---[1]--->|   Log   |<---[8]---| Monitor |
|    |         |         |         |         |
|    |<--[2]----|         |----[9]-->|         |
|    |         |         |         |         |
|    |---[3]--->|         |<--[10]---|         |
|    |         |         |         |         |----------+
|    |<--[4]----|         |---[11]-->|         |        |
|    |         |         |         +---------+        |
|    |         |         |                            |
|    |         |         |         +---------+        |
|    |         |         |<--[8]----|  Self-  |        |
|    |         |         |         | Monitor |        |
|    |         |         |---[9]--->|(Subject)|        |
|    |         |         |         |         |        |
|    |         |         |<--[10]---|         |   [12] |
|    |         |         |         |         |        |
|    |         |         |---[11]-->|         |        |
|    |         +---------+         +---------+        |
|    |                                                 |
|    |         +---------+         +---------+        |
|    |---[5]--->| Website |---[7]--->| Browser |        |
|    |         |(Subject)|         +---------+        |
|    |<--[6]--->|         |         |<------------------------+
+----+         +---------+
```
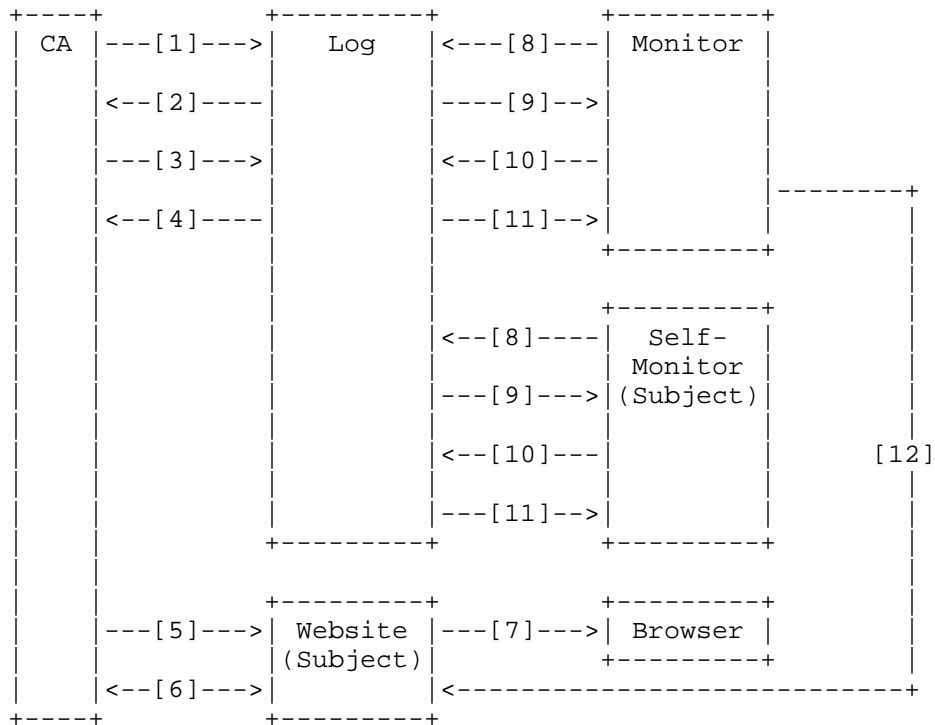
```
       [ 1] Retrieve accepted root certs
       [ 2] accepted root certs
       [ 3] Add chain to Log/add PreCertChain to Log
       [ 4] SCT
       [ 5] send cert + SCTs (or cert with embedded SCTs)
       [ 6] Revocation request/response (in response to detected
            mis-issuance)
       [ 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 1a: Data Exchanges Among CT System Element (CA submission)

```
+----+          +---------+          +---------+
| W  |---[1]--->|   Log   |<---[8]---| Monitor |
| e  |          |         |          |  (3rd   |
| b  |<--[2]----|         |----[9]-->|  party) |
| s  |          |         |          |         |
| i  |---[3]--->|         |<--[10]---|         |
| t  |          |         |          |         |--------+
| e  |<--[4]----|         |---[11]-->|         |        |
|    |          +---------+          +---------+        |
| S  |                                                  |
| u  |                                                [12]
| b  |                                                  |
| j  |                          +---------+             |
| e  |                          |         |             |
| c  |-------------[7]------------------->| Browser |<-------+
| t  |                          |         |
+----+                          +---------+
```
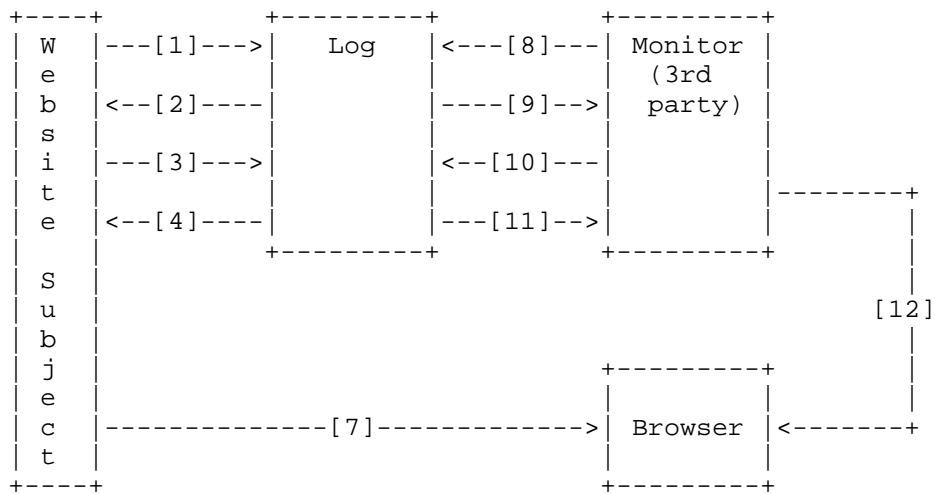
         [1] Retrieve accepted root certs
         [2] accepted root certs
         [3] Add chain to Log
         [4] SCT + STH
         [7] cert + SCTs
         [8] Retrieve entries from Log
         [9] returned entries from Log
         [10] Retrieve latest STH
         [11] returned STH
         [12] bogus/erroneous cert notification

        Figure 1b: Data Exchanges Among CT System Elements (Subject
                              submission)

   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 context of
   browsers and TLS-enabled web servers. 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:

```
    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 - CT-supporting (or not)
```

The next section of the document briefly discusses threats.
Subsequent sections examine each of the cases described above. As
noted earlier, the focus here is on the context, although most of the
analysis is applicable to other PKI contexts.

1.1. 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. Threats

In the context of this document, a threat is defined, traditionally,
as a motivated, capable adversary. An adversary who is not motivated
to attack a system is not a threat. An adversary who is motivated
but not "capable" also is not a threat. Threats change over time;
new classes of adversaries may arise, new motivations may come into
play, and the capabilities of adversaries may change. Nonetheless,
it is useful to document perceived threats against a system to
provide a context for understanding attacks (even though some attacks
may be the result of errors, not threats). Even if the assumptions
about adversaries prove to be incorrect, documenting the assumptions
is valuable.

As noted above, the goals of CT appear to be to deter, detect, and
facilitate remediation of attacks that result in certificate mis-
issuance in the context of browsers and TLS-enabled web servers.
(Note that errors by a CA are viewed as attacks, in the context of
this document.) Such attacks can enable an attacker to spoof the
identity of TLS-enabled web sites. Spoofing enables an adversary to
perform many types of attacks, e.g., delivery of malware to a client,
reporting bogus information, or acquiring information that a client
would not communicate if the client were aware of the spoofing. Such
information may include personal identification and authentication
information and electronic payment authorization information.
Because of the nature of the information that may be divulged (or
misinformation or malware that may be delivered), the principal

adversaries in the CT context are perceived to be (cyber) criminals and nation states. Both adversaries are motivated to acquire personal identification and authentication information. Criminals are also motivated to acquire electronic payment authorization information.

To make use of bogus web site certificates, an adversary must be able to direct a TLS client to a spoofed web site, so that it can present the bogus certificate during a TLS handshake. An adversary may achieve this in various ways, e.g., by manipulation of the DNS response sent to a TLS client or via a man-in-the-middle attack.

The elements of CT may themselves be targets of attacks, as described below. A criminal organization might compromise a CA and cause it to issue bogus certificates, or it may exert influence over a CA (or CA staff) to do so, e.g., through extortion or physical threat. A CA may be the victim of social engineering, causing it to issue a certificate to an inappropriate Subject. (Even though the CA is not intentionally malicious in this case, the action is equivalent to a malicious CA, hence the use of the term "bogus" here.) A nation state may operate or influence a CA that is part of the large set of "root CAs" in browsers. A CA, acting in this fashion, is termed a "malicious" CA. A nation state also might compromise a CA in another country, to effect issuance of bogus certificates. In this case the (non-malicious) CA, upon detecting the compromise (perhaps because of CT) is expected to work with Subjects to remedy the mis-issuance.

A log also might be compromised by a suitably sophisticated criminal organization or by a nation state. Compromising a log would enable a compromised or rogue CA to acquire SCTs, but log entries would be suppressed, either for all log clients or for targeted clients (e.g., to selected Monitors or Auditors).

Finally, note that a browser trust store may include a CA that is use to issue certificates to enable monitoring of encrypted browser sessions, for example. Additional certificates may be locally installed to enable an organization to acts as its own trust anchor. CT mechanisms may or may not be applied address locally-managed certificates of this sort.

3. Semantic mis-issuance

3.1. Non-malicious 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 or a technical attack. 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 a technical attack, a CA may have no record of a bogus certificate.

3.1.1. Certificate logged

3.1.1.1. 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 this mis-issuance is the responsibility of a Monitor.

3.1.1.1.1. 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 5.1, 5.2 and 5.3.)

3.1.1.1.2. 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 5.1, 5.2 and 5.3.)

3.1.1.2. Misbehaving 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. A misbehaving log may will suppress a bogus certificate log entry, or it may create an entry for the certificate but report it selectively. (A misbehaving log also could create and report entries for bogus certificates that have not been issued by the indicated CA (hereafter called "fake"). Fake bogus certificates could cause the Monitors to

report non-existent semantic problems to a Subject who would, in
turn, report them to the indicated issuing CA. This might cause the
CA to 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
suggests that the log may be misbehaving.

3.1.1.2.1. Self-monitoring Subject & Benign third party Monitor

If a misbehaving 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 misbehaving will cease to be relied upon
by Monitors, non-malicious CAs, and by browser vendors. This
assumption forms the basis for the perceived deterrent. It is not
clear if mechanisms to detect this sort of log misbehavior will be
viable.

Similarly, when a misbehaving log suppresses 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 may rely upon a
distributed Auditing mechanism, e.g., [I-D.ietf-trans-gossip], to
detect log misbehavior, as a deterrent. (See Section 5.6 below.)
However, a Monitor (third-party or self) must participate in the
Audit mechanism in order to become aware of log misbehavior.

If the misbehaving log has logged the bogus certificate when 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 relevant
STHs 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
encounters the bogus certificate (and SCT) will detect this when it
checks with the log for inclusion proofs and STH (see Section 3.1.2.)

3.1.1.3. Misbehaving third party Monitor

   A third party Monitor that misbehaves 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
   misbehaving 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.

3.1.2. Certificate not logged

   If the CA (or Subject) does not submit a pre-certificate to a log,
   whether a log is benign or misbehaving 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 misbehaving 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.

3.2. Malicious 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.

3.2.1. Certificate logged

3.2.1.1. 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.

3.2.1.1.1. 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 may request revocation. The CA may
   refuse to revoke, or may substantially delay revoking, the bogus

certificate. For example, 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 may not suffice to remedy the problem. (See
Sections 4 and 6.)

A malicious CA might revoke a bogus certificate to avoid having
browser vendors take punitive action against the CA and/or to
persuade them to not enter the bogus certificate on a vendor-
maintained blacklist. However, the CA might provide a "good" OCSP
response (from a server it operates) to a targeted browser instance
as a way to circumvent the remediation nominally offered by
revocation. No component of CT is tasked with detecting this sort of
misbehavior by a CA. (The misbehavior is analogous to a log offering
split views to different clients, as discussed later. The Audit
element of CT is tasked with detecting this sort of attack.)

3.2.1.1.2. 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
3.2.1.1.1, the CA may or may not revoke the certificate and it might
revoke the certificate but provide "good" OCSP responses to a
targeted browser instance.

3.2.1.2. Misbehaving log

A bogus (pre-)certificate may have been submitted to one or more logs
that are misbehaving, e.g., conspiring with an attacker. These logs
presumably issue SCTs, but will hide the log entries from some or all
Monitors.

3.2.1.2.1. 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 3.1.1.2.2.) If a Monitor learns
of misbehaving log operation, it alerts the Subjects that it is
protecting. The Monitor also may avoid relying upon such a for
future entries. However, unless a distributed Audit mechanism, or
equivalent, proves effective in detecting such misbehavior, CT cannot

be relied upon to detect this form of mis-issuance. (See Section 5.6 below.)

3.2.1.3. Misbehaving third party Monitor

If the third party Monitor that is "protecting" the targeted Subject is misbehaving, 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. The CT architecture does not include any measures to detect misbehavior by third-party monitors.

3.2.2. 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. (Note that an entity other than the issuing CA might submit a certificate issued by this CA to a log, if it encountered the certificate. In a narrowly-focused attack, such logging would not occur, i.e., only the target of the attack would see the certificate.)

When a CA does not submit a certificate to a log, whether a log is benign or misbehaving does not matter. Also, since there is no log entry, there is no difference in behavior between a benign and a misbehaving 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.

3.2.2.1. CT-aware browser

If careful browsers reject certificates without SCTs, CAs may be "encouraged" to log certificates (see section 5.4). However, the CT architecture does not require a browser to reject a certificate lacking a matching SCT (or equivalent evidence of logging) in all cases. This is a matter of local policy. Section 8.1.6 of [I-D.ietf-trans-rfc6962-bis] says: "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." As a result, this attack model does not assume that browsers will reject a certificate that is not accompanied by an SCT in all circumstances. Certificates have to be logged to enable detection of possible mis-issuance by Monitors, and to trigger possible subsequent revocation. The effectiveness of CT in protecting an RP is diminished in circumstances where local policy

does not mandate SCT or inclusion proof checking by the RP's
software.

3.3. Undetected Compromise of CAs or Logs

Sections 3.1 and 3.2 examined attacks in the context of non-malicious
and malicious CAs, and benign and misbehaving logs. Another class of
attacks might occur in the context of a non-malicious CA and/or a
benign log. Specifically these CT elements might be compromised and
the compromise might go undetected. Compromise of CAs and logs was
noted in Section 2, as was coercion of a CA. As noted there, a
compromised CA is almost equivalent to a malicious CA, and thus the
discussions in Section 3.2 are applicable. Section 3.4 explores the
undetected compromise of a CA in the context of attacks designed to
issue a bogus certificate that might avoid revocation (because the
certificate would appear on distinct certificate paths).

The section focuses on undetected compromise of CAs. Such
compromises warrant some additional discussion, since some relying
parties may see signed objects issued by the legitimate (non-
malicious) CA, others may see signed objects from its compromised
counterpart, and some may see objects from both. In the case of a
compromised CA or log the adversary may have access to the private
key used by a CA to sign certificates, or used by a log to sign SCTs
and STHs. (An attacker might not have access to a CA or log private
key per se. The attacker may be able to cause a CA to issue bogus
certificates, or a log to generate bogus objects, and not have a
record of them. The DigiNotar [2] case is an example of this sort of
attack on a CA.) Until such time that the compromise is detected,
there will be no effort by a CA to have its certificate revoked or by
a log to shut down the log.

3.3.1. Compromised CA, Benign Log

In the case of a compromised (non-malicious) CA, an attacker may have
acquired the CA's private key, or it may be able to cause the CA to
sign certificates using that key, even though the attacker does not
know the key per se. In other cases the goal is to cause the CA to
issues a bogus certificate (that the CA would not knowingly issue).
If this certificate is submitted to a (benign) log, then it is
subject to detection by a Monitor, as discussed in 3.1.1.1. If the
bogus certificate is submitted to a misbehaving log, then an SCT can
be generated, but there will be no entry for it, as discussed in
3.1.1.2. If the bogus certificate is not logged, then there will be
no SCT, and the implications are as described in 3.1.2.

This sort of attack may be most effective if the CA that is the
victim of the attack has issued a certificate for the targeted

Subject. In this case the bogus certificate will then have the same certification path as the legitimate certificate, which may help hide the bogus certificate (depending on details of Monitor behavior). However, means of remedying the attack are independent of this aspect, i.e., revocation can be effected irrespective of whether the targeted Subject received its certificate from the compromised CA.

A compromised (non-malicious) CA may be able to revoke the bogus certificate if it is detected by a Monitor, and the targeted Subject has been notified. It can do so only when the serial number of the bogus certificate is made known to this CA and assuming that the bogus certificate was not issued with an Authority Information Access (AIA) or CRL Distribution Point (CRL DP) extension that enables only the malicious twin to revoke the certificate. (The AIA extension in the bogus certificate could be used to direct relying parties to an OCSP server controlled by the malicious twin. The CRL DP extension could be used to direct relying parties to a CRL controlled by the malicious twin.) If the bogus certificate contains either extension, the compromised CA cannot effectively revoke it. However, the presence of either of these extensions provides some evidence that an entity other than the compromised CA issued the certificate in question. (If the extensions differ from those in other certificates issued by the compromised CA, that is suspicious.)

If the serial number of the bogus certificate is the same as for a valid, not-expired certificate issued by the CA (to the target or to another Subject), then revocation poses a problem. This is because revocation of the bogus certificate will also invalidate a legitimate certificate. This problem may cause the compromised CA to delay revocation, thus allowing the bogus certificate to remain a danger for a longer time.

The compromised CA may not realize that the bogus certificate was issued by a malicious twin; one occurrence of this sort might be regarded as an error, and not cause the CA to transition to a new key pair. (This assumes that the bogus certificate does not contain an AIA or CRL DP extension that wrests control of revocation from the compromised CA.)

Also note that the malicious twin of the compromised CA may be capable of issuing its own CRL or OCSP responses, without changing any AIA/CRL DP data present in the targeted certificate. The revocation status data from the evil twin will appear as valid as those of the compromised CA. If the attacker has the ability to control the sources of revocation status data available to a targeted user (browser instance), then the user may not become aware of the attack.

A bogus certificate issued by the malicious CA will not match the SCT for the legitimate certificate, since they are not identical, e.g., at a minimum the private keys do not match. Thus a CT-aware browser that rejects certificates without SCTs (see 3.2.2.1) will reject a bogus certificate created under these circumstances if it is not logged. If the bogus certificate is logged it is subject to detection by Monitors. Because the CA is presumed to be malicious the CA may delay revocation or try to suppress revocation status (see Section 3.5) even when confronted with evidence of issuance of the bogus certificate. In this case, even browsers that require an SCT will still accept the bogus certificate until they become aware of its revocation status.

3.3.2. Benign CA, Compromised Log

A benign CA does not issue bogus certificates, except as a result of an accident or attack. So, in normal operation, it is not clear what behavior by a compromised log would yield an attack. If a bogus certificate is issued by a benign CA (under these circumstances) is submitted to a compromised (non-malicious) log, then both an SCT and a log entry will be created. Again, it is not clear what additional adverse actions the compromised log would perform to further an attack on CT.

It is worth noting that if a benign CA was attacked and thus issued one or more bogus certificates, then a malicious log might provide split views of its log to help conceal the bogus certificate from targeted users. Specifically, the log would show an accurate set of log entries (and STHs) to most clients, but would maintain a separate log view for targeted users. This sort of attack motivates the need for Audit capabilities based on "gossiping" [I-D.ietf-trans-gossip]. However, even if such mechanisms are employed, they might be thwarted if a user is unable to exchange log information with trustworthy partners.

3.3.3. Compromised CA, Compromised Log

As noted in 3.4, an evil twin CA may issue a bogus certificate that contains the same Subject name as a legitimate certificate issued by the compromised CA. Alternatively, the bogus certificate may contain a different name but reuse a serial number from a valid, not revoked certificate issued by that CA.

An attacker who compromises a log might act in one of two ways. It might use the private key of the log only to generate SCTs for a malicious CA or the evil twin of a compromised CA. If a browser checks the signature on an SCT but does not acquire an inclusion proof, then this could be an effective attack strategy.

Alternatively, the attacker might not only generate SCTs, but also
pose as the compromised log, at least with regard to requests from
targeted users. In the latter case, this "evil twin" log could
respond to STH requests from targeted users, making it appear that
the compromised log was offering a split view (thus acting as a
malicious log). To detect this attack an Auditor may need to employ
a mechanism that is able to acquire CT data from diverse sources,
e.g., [I-D.ietf-trans-gossip].

An evil twin CA might submit a bogus certificate to the evil twin of
a compromised log. (The same adversary may be controlling both.)
The operator of the evil twin log can use the purloined private key
to generate SCTs for certificates that have not been logged by its
legitimate counterpart. These SCTs will appear valid relative to the
public key associated with the legitimate log. However, an STH
issued by the legitimate log will not correspond to a tree
(maintained by the compromised log) containing these SCTs. Thus
checking the SCTs issued by the evil twin log against STHs from the
compromised log will identify this discrepancy. As noted above, if
an attacker uses the key to generate log entries and respond to log
queries, the effect is analogous to a malicious log.)

An Auditor checking for log consistency and with access to bogus
SCTs, might conclude that the compromised log is acting maliciously,
and is presenting a split view to its clients. In this fashion the
compromised log may be shunned and forced to shut down. However, if
an attacker targets a set of TLS clients that do not have access to
the legitimate log, they may not be able to detect this
inconsistency. In this case CT might need to rely on a distributed
gossiping audit mechanism to detect the compromise (see Section 5.6).

3.4. Attacks Based on Exploiting Multiple Certificate Chains

Section 3.2 examined attacks in which a malicious CA issued a bogus
certificate and either tried to prevent the Subject from detecting
the bogus certificate, or reported the bogus certificate as valid, to
at least some relying parties, even if the Subject requested
revocation. These attacks are limited in that if the bogus
certificate is not submitted to a log, then it may not be accepted by
CT-aware browsers, and submitting the bogus certificate to a log
increases the chances that the CA's malicious behavior will be
detected.

In general, if a CA is discovered to be acting maliciously, its
certificates will no longer be accepted, either because its parent
will revoke its CA certificate, its CA certificate will be added to
browsers' blacklists, or both. However, a malicious CA may be able
to obtain an SCT for each bogus certificate that it issues and

continue to have those certificates accepted by relying parties even
after its malicious behavior has been detected. It can do this by
creating more than one path validation chain for the certificates, as
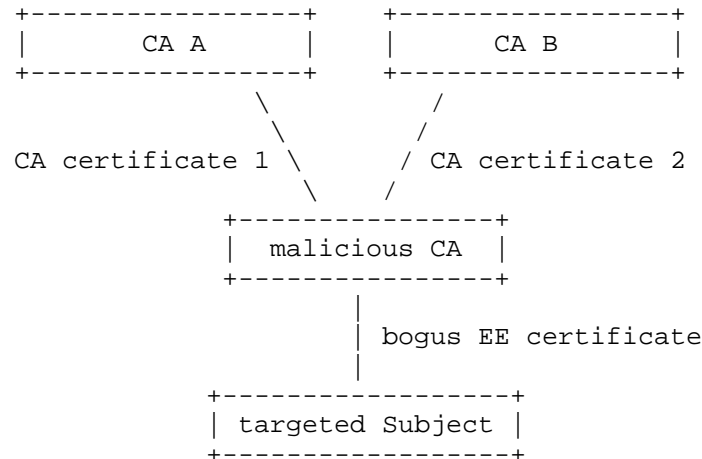shown in Figure 2.

```
     +-----------------+   +-----------------+
     |     CA A        |   |     CA B        |
     +-----------------+   +-----------------+
                 \             /
                  \           /
     CA certificate 1 \     / CA certificate 2
                    \     /
              +---------------+
              |  malicious CA  |
              +---------------+
                      |
                      | bogus EE certificate
                      |
            +-----------------+
            | targeted Subject |
            +-----------------+
```

     Figure 2: Multiple Certificate Chains for a Bogus Certificate

In Figure 2, the malicious CA has been issued CA certificates by two
different parent CAs. The parent CAs may be two different trust
anchors, or one or both of them may be an intermediate CA (i.e., it
is subordinate to some trust anchor). If both parent CAs are
intermediate CAs, they may be subordinate to the same trust anchor or
to different trust anchors. The malicious CA may have obtained
certificates from the two parents by applying to them for the
certificates, or by compromising the parent CAs and creating the
certificates without the knowledge of the CAs. If the malicious CA
applied for its certificates from these CAs, it may have presented
false information as input to the CA's normal issuance procedures,
with the result that the CAs do not realize that a certificate with
the same subject name and public key has been issued by another CA.

Because there are two certificate path validation chains, the
malicious CA could provide the chain that includes CA A when
submitting a bogus certificate to one or more logs, but an attacker
(colluding with the malicious CA) could provide the chain that
includes CA B to targeted browsers. If the CA's malicious behavior
is detected, then CA A and browser vendors may be alerted (e.g., via
the CT Monitor function) and revoke/blacklist CA certificate 1.
However, CA certificate 2 does not appear in any logs, and CA A is
unaware that CA B has issued a certificate to the malicious CA. Thus
those who detected the malicious behavior may not discover the second

chain and so may not alert CA B or browser vendors of the need to
revoke/blacklist CA certificate 2. In this case, targeted browsers
would continue to accept the bogus certificates issued by the
malicious CA, since the certificate chain they are provided is valid
and because the SCT issued for the bogus certificate it the same
irrespective of which certificate chain is presented.

This sort of attack might be thwarted if all intermediate (i.e., CA)
certificates had to be logged. In that case CA certificate 2 might
be rejected by CT-aware browsers.

This type of attack also might be thwarted if a browser vendor
blacklists a malicious CA using the CA's public key (not by its
serial number and the name of the parent CA or by a hash of the
certificate). This approach to revocation would cause CA certificate
2 to be rejected as well as CA certificate 1. However none of these
mechanisms are part of the CT specification
[I-D.ietf-trans-rfc6962-bis] nor general IETF PKI standards (e.g.,
[RFC5280]).

3.5. Attacks Related to Distribution of Revocation Status

A bogus certificate that has been revoked may still appear valid to a
browser under certain circumstances. In part this is because the
revocation information seen by a relying party is partly under the
control of the CA and/or the certificate subject. As a result,
different relying parties may be presented with different revocation
information. This is true irrespective of whether revocation is
effected via use of a CRL or OCSP. (This analysis does not consider
proprietary browser revocation status mechanisms.) Additionally, an
attacker can steer a browser to specific revocation status data via
various means, preventing a targeted browser from acquiring accurate
revocation status information for a bogus certificate.

The bogus certificate might contain an AIA extension pointing to an
OCSP server controlled by the malicious CA (or the attacker). As
noted in Section 3.2.1.1.1, the malicious CA could send a "good" OCSP
response to a targeted browser instance, even if other parties are
provided with a "revoked" response. A TLS server can supply an OCSP
response to a browser as part of the TLS handshake [RFC6961], if
requested by the browser. A TLS server posing as the entity named in
the bogus certificate also could acquire a "good" OCSP response from
the malicious CA to effect the attack. If the browser relies upon a
trusted, third-party OCSP responder, one not part of the collusion,
would these OCSP-based attacks fail.

The bogus certificate could contain a CRL distribution point
extension instead of an AIA extension. In that case a site supplying

CRLs for the malicious CA could supply different CRLs to different requestors, in an attempt to hide the revocation status of the bogus certificate from targeted browser instances. This is analogous to a split-view attack effected by a CT log. However, as noted in Section 3.2.1.1 and 3.2.1.1.1, no element of CT is responsible for detecting inconsistent reporting of certificate revocation status data. (Monitoring in the CT context tracks log entries made by CAs or Subjects. Auditing is designed to detect misbehavior by logs, not by CAs per se.)

The failure of a bogus certificate to be detected as revoked (by a browser) is not the fault of CT. In the class of attacks described above, CT achieves its goal of detecting the bogus certificate when that certificate is logged and a Monitor observes the log entry. Detection is intended to trigger revocation, to effect remediation, the details of which are outside the scope of CT. However the SCT mechanism is intended to assure a relying party that certificate has been logged, is susceptible to being detected as bogus by a Monitor, and presumably will be revoked if detected as such. In the context of these attacks, because of the way revocation may be implemented, the assurance provided by the SCT may not have the anticipated effect.

4. Syntactic mis-issuance

4.1. Non-malicious CA context

This section analyzes the scenario in which the CA has no intent to issue a syntactically incorrect certificate, but it may do so in error. (Remember that errors are considered form of attack in this document, see Section 2). As noted in Section 1, we refer to a syntactically incorrect certificate as erroneous. A certificate is erroneous if it violates a criteria to which the issuing CA claims to adhere. This might be a general profile such as [RFC5280], or a narrower profile such as those established by the CABF [1] for domain validated (DV) or extended validation (EV) certificates. If the Subject is a web site that expected to receive an EV certificate, but the certificate issued to it carries the DV policy OID, or no policy OID, relying parties may reject the certificate, causing harm to the business of the Subject. Conversely, if a CA issues a certificate to a web site and erroneously includes the EV policy OID, relying parties may place more trust in the certificate than is warranted.

4.1.1. Certificate logged

4.1.1.1. 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 could
   be noted in an SCT extension returned to the submitter.

   If the (pre-)certificate is submitted by the non-malicious issuing
   CA, 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 could help a
   CA detect and 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.

   This analysis suggests that 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.)

4.1.1.2. Misbehaving 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 misbehavior 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 that 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.

4.1.2. Certificate not logged

   If a CA does not submit a certificate to a log, there can be no
   syntactic checking by the log. (Note that a Monitor might choose to
   perform such checks, instead of a log, although this capability is
   not addressed in [I-D.ietf-trans-rfc6962-bis].) Detection of
   syntactic errors will depend on a Subject performing the requisite
   checks when it receives its certificate from a CA. A Monitor that
   performs syntactic checks on behalf of a Subject also could detect
   such problems, but the CT architecture does not require Monitors to
   perform such checks.

4.1.2.1. 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, or the
   lack of an SCT supplied to the Subject via an out-of-band channel. A
   Subject ought to notify the CA if the Subject expected that its
   certificate was to be logged. (A Subject would expect its
   certificate to be logged if there is an agreement between the Subject
   and the CA to do so, or because the CA advertises that it logs all of
   the certificates that it issues.) 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.

4.1.3. Situations Independent of Certificate Logging

4.1.3.1. Self-monitoring Subject and Benign third party Monitor

   If a Subject or benign third party 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.

4.1.3.2. CT-enabled 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 5.) Unfortunately, experience suggests that
   many browsers do not always perform very good syntactic checks on
   certificates. For example, a browser may fail to verify that a
   certificate used in a certificate path is properly marked as a CA
   certificate. Also, it would be problematic for a browser to check a
   certificate against a specific version of a profile if the profile
   changes and the policy OID remains constant. Thus it seems unlikely
   that browsers will be a reliable way to detect erroneous certificates
   in all circumstances. 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. Additionally, if a
   browser directly contacts a CA when an erroneous certificate is
   detected, this is a potential privacy violation, i.e., the CA learns
   that the browser user is visiting the web site in question. These
   observations argue for syntactic checking to be performed by other
   elements of the CT system, e.g., logs and/or Monitors.

4.2. Malicious 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.

   Note that irrespective of whether syntactic checks are performed by a
   log, a malicious CA can acquire an embedded SCT, or post-issuance
   will acquire a standalone SCT for an erroneous certificate. 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 the action(s) needed to remedy the problem.

4.2.1. Certificate logged

4.2.1.1. Benign log

Because the CA is presumed to be malicious, the CA might cause the log to not perform checks (if the log offered this option). Because logs are not required to perform syntax checks, there probably would have to be a way for a CA to request checking, the CA might indicate that it did not desire such checks to be performed. Or the CA might submit a (pre-)certificate to a log that is known to not perform any syntactic checks, and thus avoid syntactic checking.

4.2.1.2. Misbehaving log or third party Monitor

A misbehaving log or third party Monitor will either not perform syntactic checks or not report any problems that it discovers. (See 4.1.1.2 for further problems). Also, as noted above, the CT architecture includes no explicit provisions for detecting a misbehaving third-party Monitor.

4.2.1.3. CT-enabled browser

As noted above (4.1.3.2), most browsers do not perform thorough syntax checks on certificates. Such browsers might benefit from having syntax checks performed by a log and reported in the SCT, although the pervasive nature of syntactically-defective certificates may limit the utility of such checks. (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 the browser from syntax checks perform by logs or Monitors.

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 attack noted in 4.2.1.1 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 attack strategies noted above could be thwarted.

4.2.2. Certificate is not logged

   Since certificates are not logged in this scenario, a third-party
   Monitor cannot detect the issuance of an erroneous certificate based
   on examination of log entries. However, if a Subject informs a
   Monitor of the syntactic criteria applicable to the certificate it is
   supplying, the Monitor can perform syntactic checks on behalf of the
   Subject. Thus there is no difference between a benign or a
   malicious/conspiring log or a benign or conspiring/malicious Monitor.
   (Also note that 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.

5. Issues Applicable to Sections 3 and 4

5.1. 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. If
   third-party Monitors advertise which logs they track, Subjects may be
   able to use this information to select an appropriate Monitor (or set
   thereof). Also, it is not clear whether every third-party Monitor
   must offer to track every Subject that requests protection. If a
   Subject acts as its own Monitor, this problem is solved for that
   Subject.

5.2. 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, or to determine what set of logs suffice to ensure an
   adequate level of coverage.

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

   A CA being presented with evidence of a bogus or erroneous
   certificate, supported by 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. 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).

5.4. 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. [I-D.ietf-trans-rfc6962-bis] does not
describe a strategy for incremental deployment, however it calls for
local policy controls that might be used to facilitate incremental
deployment (see 3.2.2.1 earlier). For example a browser might
establish a date after which all certificates issued MUST contain an
SCT or be accompanied by an SCT during TLS session establishment. A
strategy like this would allow certificates issued before that date
to be "grandfathered". This approach would allow a malicious CA to
backdate a certificate to avoid logging it, exploiting a window of
vulnerability. Note that 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 [RFC5280].

5.5. 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
bad-CA-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. If a browser
vendor operates it's own Monitor, there is no need for a standard way
to convey this information. However, there are no standard ways to
convey Monitor information to a browser, e.g., to reject individual
bogus or erroneous certificates based on information provided by 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
facilitate remediation of this form of attack, even though it
provides a basis for detecting such attacks.

5.6. Auditing - detecting misbehaving 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 may not be visible to
   Monitors. If browsers reject certificates that contain SCTs from
   conspiring logs (e.g., based on information from an auditor) CT
   should be able to detect and deter use of such logs by (benign) CAs.

   Section 8.3 of [I-D.ietf-trans-rfc6962-bis] specifies that auditing
   is performed by Monitors and/or browsers. If a Monitor performs the
   function, then it needs a way to communicate the results of audit
   infractions to CAs and browsers. If a browser vendor operates a
   Monitor it could use its audit information to cause browsers to
   reject certificates with SCTs from suspect logs. However, there is
   no standard mechanism defined to allow a self-monitoring Subject to
   convey this information to browsers directly.

   If auditing is performed by browsers directly there may be user
   privacy concerns due to direct interaction with logs, as noted in
   Section 8.1.4 of [I-D.ietf-trans-rfc6962-bis]. Also, unless browsers
   have ways to share audit information with other browsers, local
   detection of a misbehaving log does not necessarily benefit a larger
   community. At the time of this writing, one mechanism has been
   defined (via an RFC) for use with CT to achieve the necessary
   communication: [I-D.ietf-trans-gossip].

   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 should not rely on a CA or RA database for its
reference information or use certificate discovery protocols; this
information should 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.

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.

6. Security Considerations

An attack and 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. It then analyses the ways in which the CT
architecture addresses these attacks.

7. IANA Considerations

None.

8. Acknowledgments

The author would like to thank David Mandelberg and Karen Seo for
their assistance in reviewing and preparing this document, and other
members of the TRANS working group for reviewing it. Most of the
text of Section 3.4 was provided by David Cooper, motivated by
observations from Daniel Kahn Gilmor. Thanks also go to Daiming Li
for her editorial assistance.

9. References

9.1. Normative References

   [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>.

9.2. Informative References

   [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-rfc6962-bis]
              Laurie, B., Langley, A., Kasper, E., Messeri, E., and R.
              Stradling, "Certificate Transparency Version 2.0", draft-
              ietf-trans-rfc6962-bis-28 (work in progress), March 2018.

   [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>.

   [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>.

   [RFC6961] Pettersen, Y., "The Transport Layer Security (TLS)
              Multiple Certificate Status Request Extension", RFC 6961,
              DOI 10.17487/RFC6961, June 2013,
              <https://www.rfc-editor.org/info/rfc6961>.

9.3. URIs

   [1] https://cabforum.org

   [2] https://en.wikipedia.org/wiki/DigiNotar

Author's Address

   Stephen Kent
   Independent

   Email: kent@alum.mit.edu

                Certificate Transparency: Domain Label Redaction
                       draft-strad-trans-redaction-01

Abstract

   This document defines mechanisms to allow DNS domain name labels that
   are considered to be private to not appear in public Certificate
   Transparency (CT) logs, while still retaining most of the security
   benefits that accrue from using Certificate Transparency.

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 July 21, 2017.

Table of Contents

1. Introduction

   Some domain owners regard certain DNS domain name labels within their
   registered domain space as private and security sensitive. Even
   though these domains are often only accessible within the domain
   owner's private network, it's common for them to be secured using
   publicly trusted Transport Layer Security (TLS) server certificates.

   Certificate Transparency v1 [RFC6962] and v2
   [I-D.ietf-trans-rfc6962-bis] describe protocols for publicly logging
   the existence of TLS server certificates as they are issued or
   observed. Since each TLS server certificate lists the domain names
   that it is intended to secure, private domain name labels within
   registered domain space could end up appearing in CT logs, especially
   as TLS clients develop policies that mandate CT compliance. This
   seems like an unfortunate and potentially unnecessary privacy leak,
   because it's the registered domain names in each certificate that are
   of primary interest when using CT to look for suspect certificates.

   TODO: Highlight better the differences between registered domains and
   subdomains, referencing the relevant DNS RFCs.

2. 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].

3. Redaction Mechanisms

   We propose three mechanisms, in increasing order of implementation
   complexity, to allow certain DNS domain name labels to not appear in
   public CT logs:

   o Using wildcard certificates (Section 3.1) is the simplest option,
      but it only covers certain use cases.

   o Logging a name-constrained intermediate CA certificate in place of
      the end-entity certificate (Section 3.2) covers more, but not all,
      use cases.

   o Therefore, we define a domain label redaction mechanism
      (Section 3.3) that covers all use cases, at the cost of
      considerably increased implementation complexity.

   We anticipate that TLS clients may develop policies that impose
   additional compliancy requirements on the use of the Section 3.2 and
   Section 3.3 mechanisms.

   To ensure effective redaction, CAs and domain owners should note the
   privacy considerations (Section 5).

   TODO(eranm): Do we need to further expand (either here or in the
   following subsections) on when each of the mechanisms is/isn't
   suitable?

   TODO: Previously, these mechanisms were defined in earlier revisions
   of CTv2 [I-D.ietf-trans-rfc6962-bis], and nothing was said about
   compatibility with CTv1. But now, given that these mechanisms have
   been decoupled from [I-D.ietf-trans-rfc6962-bis], and given that at
   least one major TLS client has announced a policy of mandatory CT
   compliance that will almost certainly take effect before CTv2 is
   widely deployed, we should consider making some or all of these
   mechnanisms compatible with both CTv1 and CTv2.

3.1. Using Wildcard Certificates

   A certificate containing a DNS-ID [RFC6125] of "*.example.com" could
   be used to secure the domain "topsecret.example.com", without
   revealing the label "topsecret" publicly.

Since TLS clients only match the wildcard character to the complete leftmost label of the DNS domain name (see Section 6.4.3 of [RFC6125]), a different mechanism is needed when any label other than the leftmost label in a DNS-ID is considered private (e.g., "top.secret.example.com"). Also, wildcard certificates are prohibited in some cases, such as Extended Validation Certificates [EV.Certificate.Guidelines].

3.2. Using a Name-Constrained Intermediate CA

An intermediate CA certificate or intermediate CA precertificate that contains the Name Constraints [RFC5280] extension MAY be logged in place of end-entity certificates issued by that intermediate CA, as long as all of the following conditions are met:

o there MUST be a non-critical extension (OID 1.3.101.76, whose extnValue OCTET STRING contains ASN.1 NULL data (0x05 0x00)). This extension is an explicit indication that it is acceptable to not log certificates issued by this intermediate CA.

o there MUST be a Name Constraints extension, in which:

   * permittedSubtrees MUST specify one or more dNSNames.

   * excludedSubtrees MUST specify the entire IPv4 and IPv6 address ranges.

Below is an example Name Constraints extension that meets these conditions:

```
        SEQUENCE {
          OBJECT IDENTIFIER '2 5 29 30'
          BOOLEAN TRUE
          OCTET STRING, encapsulates {
            SEQUENCE {
              [0] {
                SEQUENCE {
                  [2] 'example.com'
                }
              }
              [1] {
                SEQUENCE {
                  [7] 00 00 00 00 00 00 00 00
                }
                SEQUENCE {
                  [7]
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                }
              }
            }
          }
        }
```

3.2.1. Presenting SCTs, Inclusion Proofs and STHs

   Each SCT (and optional corresponding inclusion proof and STH)
   presented by a TLS server MAY correspond to an intermediate CA
   certificate or intermediate CA precertificate (to which the server
   certificate chains) that meets the requirements in Section 3.2. This
   extends section TBD of CT v2 [I-D.ietf-trans-rfc6962-bis], which
   specifies that each SCT always corresponds to the server certificate
   or to a precertificate that corresponds to that certificate.

   Each SCT (and optional corresponding inclusion proof and STH)
   included by a certification authority in a Transparency Information
   X.509v3 extension in the "singleExtensions" of a "SingleResponse" in
   an OCSP response MAY correspond to an intermediate CA certificate or
   intermediate CA precertificate (to which the certificate identified
   by the "certID" of that "SingleResponse" chains) that meets the
   requirements in Section 3.2. This extends section TBD of CT v2
   [I-D.ietf-trans-rfc6962-bis], which specifies that each SCT always
   corresponds to the certificate identified by the "certID" of that
   "SingleResponse" or to a precertificate that corresponds to that
   certificate.

   Each SCT (and optional corresponding inclusion proof and STH)
   included by a certification authority in a Transparency Information

   X.509v3 extension in a certificate MAY correspond to an intermediate
   CA certificate or intermediate CA precertificate (to which the
   certificate chains) that meets the requirements in Section 3.2. This
   extends section TBD of CT v2 [I-D.ietf-trans-rfc6962-bis], which
   specifies that each SCT always corresponds to a precertificate that
   corresponds to that certificate.

   TODO: Refactor this section to avoid repetition.

3.2.2. Matching an SCT to the Correct Certificate

   Before considering any SCT to be invalid, a TLS client MUST attempt
   to validate it against the server certificate and against each of the
   zero or more suitable name-constrained intermediates in the chain.
   These certificates may be evaluated in the order they appear in the
   chain, or indeed, in any order.

   TODO: Shall we specify that there MUST be no more than ONE name-
   constrained intermediate in the chain?

   TODO: Shall we specify that all presented SCTs MUST correspond to the
   same (end-entity or name-constrained intermediate) certificate?

3.3. Redacting Labels in Precertificates

   When creating a precertificate, the CA MAY include a
   redactedSubjectAltName (Section 3.3.1) extension that contains, in a
   redacted form, the same entries that will be included in the
   certificate's subjectAltName extension. When the
   redactedSubjectAltName extension is present in a precertificate, the
   subjectAltName extension MUST be omitted (even though it MUST be
   present in the corresponding certificate).

   Wildcard "*" labels MUST NOT be redacted, but one or more non-
   wildcard labels in each DNS-ID [RFC6125] can each be replaced with a
   redacted label as follows:

      REDACT(label) = prefix || BASE32(index || _label_hash)
        _label_hash = LABELHASH(keyid_len || keyid || label_len || label)

   "label" is the case-sensitive label to be redacted.

   "prefix" is the "?" character (ASCII value 63).

   "index" is the 1 byte index of a hash function in the CT hash
   algorithm registry (section TBD of [I-D.ietf-trans-rfc6962-bis]).
   The value 255 is reserved.

"keyid_len" is the 1 byte length of the "keyid".

"keyid" is the keyIdentifier from the Subject Key Identifier extension (section 4.2.1.2 of [RFC5280]), excluding the ASN.1 OCTET STRING tag and length bytes.

"label_len" is the 1 byte length of the "label".

"||" denotes concatenation.

"BASE32" is the Base 32 Encoding function (section 6 of [RFC4648]). Pad characters MUST NOT be appended to the encoded data.

"LABELHASH" is the hash function identified by "index".

3.3.1. redactedSubjectAltName Certificate Extension

The redactedSubjectAltName extension is a non-critical extension (OID 1.3.101.77) that is identical in structure to the subjectAltName extension, except that DNS-IDs MAY contain redacted labels (Section 3.3).

When used, the redactedSubjectAltName extension MUST be present in both the precertificate and the corresponding certificate.

This extension informs TLS clients of the DNS-ID labels that were redacted and the degree of redaction, while minimizing the complexity of TBSCertificate reconstruction (Section 3.3.3). Hashing the redacted labels allows the legitimate domain owner to identify whether or not each redacted label correlates to a label they know of.

TODO: Consider the pros and cons of this 'un'redaction feature. If the cons outweigh the pros, switch to using Andrew Ayer's alternative proposal of hashing a random salt and including that salt in an extension in the certificate (and not including the salt in the precertificate).

Only DNS-ID labels can be redacted using this mechanism. However, CAs can use the Section 3.2 mechanism to allow DNS domain name labels in other subjectAltName entries to not appear in logs.

TODO: Should we support redaction of SRV-IDs and URI-IDs using this mechanism?

3.3.2. Verifying the redactedSubjectAltName extension

   If the redactedSubjectAltName extension is present, TLS clients MUST
   check that the subjectAltName extension is present, that the
   subjectAltName extension contains the same number of entries as the
   redactedSubjectAltName extension, and that each entry in the
   subjectAltName extension has a matching entry at the same position in
   the redactedSubjectAltName extension. Two entries are matching if
   either:

   o The two entries are identical; or

   o Both entries are DNS-IDs, have the same number of labels, and each
      label in the subjectAltName entry has a matching label at the same
      position in the redactedSubjectAltName entry. Two labels are
      matching if either:

      * The two labels are identical; or,

      * Neither label is "*" and the label from the
         redactedSubjectAltName entry is equal to REDACT(label from
         subjectAltName entry) (Section 3.3).

   If any of these checks fail, the certificate MUST NOT be considered
   compliant.

3.3.3. Reconstructing the TBSCertificate

   Section TBD of [I-D.ietf-trans-rfc6962-bis] describes how TLS clients
   can reconstruct the TBSCertificate component of a precertificate from
   a certificate, so that associated SCTs may be verified.

   If the redactedSubjectAltName extension (Section 3.3.1) is present in
   the certificate, TLS clients MUST also:

   o Verify the redactedSubjectAltName extension against the
      subjectAltName extension according to Section 3.3.2.

   o Once verified, remove the subjectAltName extension from the
      TBSCertificate.

4. Security Considerations

4.1. Avoiding Overly Redacted Domain Names

   Redaction of domain name labels (Section 3.3) carries the same risks
   as the use of wildcards (e.g., section 7.2 of [RFC6125]). If the
   entirety of the domain space below the unredacted part of a domain

name is not registered by a single domain owner (e.g.,
REDACT(label).com, REDACT(label).co.uk and other [Public.Suffix.List]
entries), then the domain name may be considered by clients to be
overly redacted.

CAs should take care to avoid overly redacting domain names in
precertificates. It is expected that monitors will treat
precertificates that contain overly redacted domain names as
potentially misissued. TLS clients MAY consider a certificate to be
non-compliant if the reconstructed TBSCertificate (Section 3.3.3)
contains any overly redacted domain names.

TODO(eranm): Describe how the CT ecosystem would be harmed if the use
of redaction becomes too widespread.

5. Privacy Considerations

5.1. Ensuring Effective Redaction

Although the mechanisms described in this document remove the need
for private labels to appear in CT logs, they do not guarantee that
this will never happen. For example, anyone who encounters a
certificate could choose to submit it to one or more logs, thereby
rendering the redaction futile.

Domain owners are advised to take the following steps to minimize the
likelihood that their private labels will become known outside their
closed communities:

o Avoid registering private labels in public DNS.

o Avoid using private labels that are predictable (e.g., "www",
   labels consisting only of numerical digits, etc). If a label has
   insufficient entropy then redaction will only provide a thin layer
   of obfuscation, because it will be feasible to recover the label
   via a brute-force attack.

o Avoid using publicly trusted certificates to secure private domain
   space.

o Avoid enabling unrestricted access for DNS zone transfer (AXFR)
   requests (see section 5 of [RFC5936]).

CAs are advised to carefully consider each request to redact a label
using the Section 3.3 mechanism. When a CA believes that redacting a
particular label would be futile, we advise rejecting the redaction
request. TLS clients may have policies that forbid redaction, so

   label redaction should only be used when it's absolutely necessary
   and likely to be effective.

6. Acknowledgements

   The authors would like to thank Andrew Ayer and TBD 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.

7. References

7.1. Normative References

   [I-D.ietf-trans-rfc6962-bis]
               Laurie, B., Langley, A., Kasper, E., Messeri, E., and R.
               Stradling, "Certificate Transparency Version 2.0", draft-
               ietf-trans-rfc6962-bis-24 (work in progress), December
               2016.

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119,
               DOI 10.17487/RFC2119, March 1997,
               <http://www.rfc-editor.org/info/rfc2119>.

   [RFC4648]   Josefsson, S., "The Base16, Base32, and Base64 Data
               Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006,
               <http://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,
               <http://www.rfc-editor.org/info/rfc5280>.

   [RFC5936]   Lewis, E. and A. Hoenes, Ed., "DNS Zone Transfer Protocol
               (AXFR)", RFC 5936, DOI 10.17487/RFC5936, June 2010,
               <http://www.rfc-editor.org/info/rfc5936>.

   [RFC6125]   Saint-Andre, P. and J. Hodges, "Representation and
               Verification of Domain-Based Application Service Identity
               within Internet Public Key Infrastructure Using X.509
               (PKIX) Certificates in the Context of Transport Layer
               Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March
               2011, <http://www.rfc-editor.org/info/rfc6125>.

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

7.2. Informative References

   [EV.Certificate.Guidelines]
              CA/Browser Forum, "Guidelines For The Issuance And
              Management Of Extended Validation Certificates", 2007,
              <https://cabforum.org/wp-content/uploads/
              EV_Certificate_Guidelines.pdf>.

   [Public.Suffix.List]
              Mozilla Foundation, "Public Suffix List", 2016,
              <https://publicsuffix.org>.

Authors' Addresses

   Rob Stradling
   Comodo CA, Ltd.

   Email: rob.stradling@comodo.com

   Eran Messeri
   Google UK Ltd.

   Email: eranm@google.com

                            CT for Binary Codes
                    draft-zhang-trans-ct-binary-codes-04

Abstract

   This document proposes a solution extending the Certificate
   Transparency protocol [I-D.ietf-trans-rfc6962-bis] for transparently
   logging the software binary codes (BC)or its digest with their
   signature, to enable anyone to monitor and audit the software
   provider activity and notice the distribution of suspect software as
   well as to audit the BC logs themselves. The solution is called
   "Binary Transparency" in this document.

Status of This Memo

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

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

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

   This Internet-Draft will expire on September 7, 2017.

Copyright Notice

publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction

   Digital signatures have been widely used in software distributions to
   prove the authenticity of software. Through verifying signature, an
   end user can ensure that the gotten software is developed by a legal
   provider (e.g., Microsoft) and is not tampered during the
   distribution. If an end user does not have a direct trust
   relationship with the software provider, an certificate chain to a
   trust anchor that the user trusts should be provided. That is why
   many signature mechanisms for software distribution are based on
   public key infrastructure (PKI). However, signature mechanisms
   cannot prevent software provider from distributing software either
   with customized backdoors/drawbacks, or they do not own the right to
   distribute. Besides, 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 describes the Binary Transparency mechanism which extends
   the Certificate Transparency (CT) protocol specified in [I-D.ietf-
   trans-rfc6962-bis] to support logging binary codes. A software
   provider can submit its software Binary Codes (BC) (or digests of
   codes in order to e.g., save space or avoid violating license
   restrictions) with associated signature to one or more CT logs.
   Therefore, a user can easily detect the existence of software BC with
   customized backdoors, by comparing with the according CT log entries.
   The software provider can monitor the logs all the time to detect
   whether there are tempered copies of its software in the log, or its
   software is submitted into the log by other software providers
   without authority. In summary, the end users should be informed when
   all the above situations happen, how to achieve it is beyond the
   scope of this document.

   With this mechanism, when a section of binary codes and associated
   signature has been submitted to a log, if the provided certificate
   chain ends with a trust anchor that is accepted by the log, the log
   will accept it and return the Signed Binary Timestamp (SBT) to the
   software provider as the evidence of its acceptance provided to the
   users later. Thus, the users should only trust the software
   accompanied by SBT, even if it is associated with a proper signature.
   This approach then forces the software providers to submit their
   binary codes to logs before distributing them.

   Binary Transparency is an extension to Certificate Transparency,
   which comply with most of the specification in [I-D.ietf-trans-
   rfc6962-bis]. This document only focuses on the extension part of
   Binary Transparency mechanisms.

1.1. Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119 [RFC2119].

2. Cryptographic Components of Binary Transparency

   When applying CT for binary codes, a log is a single, ever-growing,
   append-only binary Merkle Hash Tree of software BC, with associated
   signature and certificate chain, complying with the Merkle Hash Tree
   specification in Section 2 of [I-D.ietf-trans-rfc6962-bis].

3. Motivation Scenarios

   The documents disclosed by Edward Snowden have raised the concerns of
   people on the vulnerability of the network devices to the passive
   attacks performed by NSA or other organizations. Meanwhile, the

network device vendors are also concerned 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 facilities
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
submit the binary codes of its firmwares to the public CT logs.
Therefore, anyone can verify the correctness of each log entry and
monitor when new software BCs are added to it. Specially, customers
can easily detect whether the vendor is releasing the same firmware
to everyone. In addition, under the assistance of the Binary
Transparency, customer will have more confidence on the quality of
firmware. Since the same codes are used by different customers all
over the world, the drawbacks in firmware will be easier to be
detected.

There are similar requirements to detect the customized backdoors or
misdistribution in the software market. Besides the software itself,
a user may also concern whether there are customized backdoors in the
patches. The Binary Transparency can help address such concerns in
the same way. In addition, this mechanism can also show some
advantages in the scenarios where the signer is not aware that their
keys have been compromised. If their update system is required to
use a CT log, they have the chance to find out about their
compromise.

4. Log Format and Operation Extensions

The software provider can submit the software and the associated
signature to any preferred CT logs before distributing it. In some
cases, the software provider may select only to submit the signed
digest of the software because of the license restriction or the
space restriction of log entry. In order to verify the attribution
of each log entry, a log SHALL publish a set of certificates that it
trusts to benefit an software provider to construct an certificate
chain connecting a trust anchor and the certificate containing the
key used to sign the software.

A log needs to verify the certificate chain provided by the software
provider, and MUST refuse to accept the signed software/digest if the
chain cannot lead back to a trusted anchor. If the software/digest
and the signature are accepted by a log and an SBT is issued, the log
MUST store the entire chain and MUST present this chain for auditing
upon request.

Complying with the log format definition in [I-D.ietf-trans-
rfc6962-bis], some definitions remain the same: "Log ID", "Merkle

Tree Head", "Signed Tree Head", "Merkle Consistency Proofs", "Merkle
Inclusion Proofs", "Shutting down a log"... The other required log
format extension for Binary Transparency are specified in the
following sections:

4.1. Log Entries

Each software entry in a log MUST include a "BinaryChainEntryV2"
structure as below:

```
enum { binary(TBD1), binary_digest(TBD2) } BIN_Signed_Type;

opaque BINARY<1..2^24-1>;
opaque ASN.1Cert<1..2^24-1>;
struct {
    BIN_Signed_Type bin_signed_type;
    BINARY signed_software;
    ASN.1Cert certificate_chain<1..2^24-1>;
} BinaryChainEntryV2;
```

"bin_signed_type" indicates whether the signature is generated based
on the software or its digest.

"signed_software" consists a ContentInfo structure specified in
CMS[RFC5652]. Specifically, this field includes the binary codes/
digest, the signature, and any other additional information used to
describe the software and the issuer publishing the software. The
software SHOULD be encapsulated and signed following the ways
specified in CMS[RFC5652] . If signed_type is TBD1, the software
binary code is encapsulated in this field. If signed_type is TBD2,
the SHA-256 digest of software binary code is encapsulated in this
field.

"certificate_chain" includes the certificates constructing a chain
from the certificate of software provider to a certificate trusted by
the log. The first certificate MUST be the certificate of software
provider. 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. If the certificate chain is
provided in the "signed_software" field structure, this field is set
to empty.

4.2. TransItem Structure

The extended "TransItem" structure is defined as below:

```
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), x509_sct_with_proof_v2(8),
      precert_sct_with_proof_v2(9), BIN_entry_v2(TBD3),
      BIN_sbt_v2(TBD4), BIN_sbt_with_proof_v2(TBD5),
      (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;
            case x509_sct_with_proof_v2: SCTWithProofDataV2;
            case precert_sct_with_proof_v2: SCTWithProofDataV2;
            case BIN_entry_v2: TimestampedBinaryEntryDataV2;
            case BIN_sbt_v2: SignedBinaryTimestampDataV2;
            case BIN_sbt_with_proof_v2: SBTWithProofDataV2;
      } data;
} TransItem;
```

"versioned_type " is the type of the encapsulated data structure of
TransItem. Three new values are added to it -- BIN_entry_v2(TBD3),
BIN_sbt_v2(TBD4), BIN_sbt_with_proof_v2(TBD5).

For "data" structure, a new type structure of
TimestampedBinaryEntryDataV2 is added.

4.3. Merkle Tree Leaves

Each Merkle Tree leaf is defined as the hash value of a "TransItem"
structure of according type. Here, a new type ("BIN_entry_v2") of
"TransItem" structure is created, which encapsulates a new
"TimestampedBinaryEntryDataV2" structure defined as below:

```
    opaque TBSCertificate<1..2^24-1>;
    struct {
            uint64 timestamp;
            opaque issuer_key_hash<32..2^8-1>;
            BIN_Signed_Type bin_signed_type;
            TBSSignedSoftware tbs_signed_software;
            SbtExtension sbt_extensions<0..2^16-1>;
        } TimestampedBinaryEntryDataV2;
```

"timestamp" is the NTP Time [RFC5905] at which the software binary
code was accepted by the log, measured in milliseconds since the
epoch (January 1, 1970, 00:00 UTC), ignoring leap seconds. 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 software
provider that signed the software, calculated over the DER encoding
of the key represented as SubjectPublicKeyInfo [RFC5280]. This is
needed to bind the software provider to the software binary code,
making it impossible for the corresponding SBT to be valid for any
other software whose TBSSignedSoftware matches "tbs_signed_software".
The length of the "issuer_key_hash" MUST match HASH_SIZE.

"bin_signed_type" indicates whether the signature is generated based
on the software or its digest.

"tbs_signed_software" is the DER encoded TBSSignedSoftware from the
"signed_software" in the case of a "BinaryChainEntryV2".

4.4. Structure of the Signed Binary Timestamp

   An SBT is a "TransItem" structure of type "bin_sbt_v2", which
   encapsulates a "SignedBinaryTimestampDataV2" structure:

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

  struct {
        SbtExtensionType sbt_extension_type;
        opaque sbt_extension_data<0..2^16-1>;
      } SbtExtension;

  struct {
        LogID log_id;
        uint64 timestamp;
        SbtExtension sbt_extensions<0..2^16-1>;
        digitally-signed struct {
            TransItem timestamped_entry;
        } signature;
      } SignedBinaryTimestampDataV2;
```

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

"timestamp" is equal to the timestamp from the
"TimestampedBinaryEntryDataV2" structure encapsulated in the
"timestamped_entry".

"sbt_extension_type" identifies a single extension from the IANA
registry in Section 6. At the time of writing, no extensions are
specified.

The interpretation of the "sbt_extension_data" field is determined
solely by the value of the "sbt_extension_type" field. Each document
that registers a new "sbt_extension_type" must describe how to
interpret the corresponding "sbt_extension_data".

"sbt_extensions" is a vector of 0 or more SBT extensions. This
vector MUST NOT include more than one extension with the same
"sbt_extension_type". The extensions in the vector MUST be ordered
by the value of the "sbt_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.

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

"timestamped_entry" is a "TransItem" structure that MUST be of type
"BIN_entry_v2".

5. Log Client Messages

   In Section 5 of [I-D.ietf-trans-rfc6962-bis], a set of messages is
   defined for clients to query and verify the correctness of the log
   entries they are interested in. In this document, a new message is
   defined and an existing message is extended for CT to support Binary
   Transparency.

5.1. Add Binary Code and Certificate Chain to Log

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

   Inputs:
    bin_signed_type: indicates whether the input parameter "software"
                     is constructed by the binary code or its digest.
    software: the binary code (or digest), the signature, and the
              information used to describe the software and the software
              provider publishing the software, which are encapsulated
              following the way specified in CMS[RFC5652] . The submitter
              desires a SBT for this element.
    chain: An array of base64-encoded certificates. The first element is
              the certificate used to sign the binary code (or digest); the
              second certifies the first and so on to the last, which either is,
              or is certified by, an accepted trust anchor.If the certificate
              chain information has been included in the "software" field, this
              field could be empty.

   Outputs:
     sbt: A base64 encoded "TransItem" of type "BIN_sbt_v2", signed by this
           log, that corresponds to the submitted software.

   Error codes:
     Be identical with the according part in Section 5.1 (Add Chain to Log) of
     [I-D.ietf-trans-rfc6962-bis].

5.2. 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
      leaf_input:  The base64 encoded "TransItem" structure of type
                    "x509_entry_v2" or "precert_entry_v2" or "BIN_entry_v2"
                    (see Section 4.3).
      log_entry:  The base64 encoded log entry (see Section 4.1).  In the
                   case of an "x509_entry_v2" entry, this is the whole
                   "X509ChainEntry"; and in the case of a "precert_entry_v2",
                   this is the whole "PrecertChainEntryV2"; and in the case of a
                   "BIN_entry_v2", this is the whole "BinaryChainEntryV2".
      sct:  The base64 encoded "TransItem" of type "x509_sct_v2" or "precert_sct
_v2"
            or "BIN_sbt_v2"corresponding to this log entry.
      sth:  A base64 encoded "TransItem" of type "signed_tree_head_v2", signed
            by this log.
```

   More details are identical with Section 5.7 of [I-D.ietf-trans-
   rfc6962-bis].

5.3. Summary

   In summary, the above extensions of Binary Transparency enable the
   software providers, the end users, and anyone to monitor and audit
   the CT logs to mitigate the possible attacks induced by tampered
   software, or software misdistribution.

   This section gives a brief introduction to all the other aspects of
   Binary Transparency mechanisms for the reason of completeness, since
   they comply with the basic CT protocol specification. For more
   details please refer to the corresponding sections of [I-D.ietf-
   trans-rfc6962-bis].

   Software providers act the same as TLS servers in CT protocol. They
   present one or more SBTs from one or more logs to each end user while
   distributing the software, where each SBT corresponds to the
   software. Software providers SHOULD also present corresponding
   inclusion proofs and STHs. In which way the software providers
   present this information is beyond the scope of this document.

   The end users of software acts the same as Clients of logs described
   in CT protocol. They can perform various different functions, such
   as: get log metadata, exchange STHs they see, receive and validate
   SBTs, Validate inclusion proofs.

   Binary Transparency also provides monitoring and auditing functions
   with the same algorithms defined for CT protocol.

   Binary Transparency supports the same algorithm agility feature for
   signature algorithm and hash algorithm as CT protocol.

6. Acknowledgements

7. IANA Considerations

   To be added.

8. Security Considerations

   To be added.

9. References

9.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <http://www.rfc-editor.org/info/rfc2119>.

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

   [RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch,
              "Network Time Protocol Version 4: Protocol and Algorithms
              Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010,
              <http://www.rfc-editor.org/info/rfc5905>.

9.2. Informative References

   [I-D.ietf-trans-rfc6962-bis]
              Laurie, B., Langley, A., Kasper, E., Messeri, E., and R.
              Stradling, "Certificate Transparency Version 2.0", draft-
              ietf-trans-rfc6962-bis-24 (work in progress), December
              2016.

Authors' Addresses

   Liang Xia (editor)
   Huawei

   Email: frank.xialiang@huawei.com

Dacheng Zhang
Huawei

Email: dacheng.zhang@huawei.com

Daniel Kahn Gillmor
CMRG

Email: dkg@fifthhorseman.net

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

Email: sarikaya@ieee.org

Network Working Group   D. Zhang
Internet-Draft
Intended status: Experimental   D. Gillmor
Expires: January 6, 2016   CMRG
                                                                D. He
                                                               Huawei
                                                          B. Sarikaya
                                                           Huawei USA
                                                              N. Kong
                                                         July 5, 2015

       Certificate Transparency for Domain Name System Security Extensions
                       draft-zhang-trans-ct-dnssec-03

Abstract

   In draft-ietf-trans-rfc6962-bis, a solution (Certificate
   Transparency) is proposed for publicly logging the existence of
   Transport Layer Security (TLS) certificates using Merkle Hash Trees.
   This document proposes a mechanism to extend Certificate Transparency
   for DNSSEC which publicly logs the DS RRs to notice the issuance of
   suspect key signing keys.

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

Copyright Notice

Table of Contents

1. Introduction

   [I-D.ietf-trans-rfc6962-bis] specifies a Certificate Transparency
   (CT) mechanism to disclosing TLS certificates into public logs. This
   mechanism benefits the public to monitor the operations in issuing
   certificates to improper subscribers. The logs do not prevent mis-
   issuing behavior directly, but the provided public audibility can
   increase the possibility in detecting the improper behaviors of
   issuers. The logs are constructed with Merkle Hash Trees to ensure
   the append-only property, and thus enable anyone to verify the
   correctness of each log record. Note that CT is a common mechanism
   although [I-D.ietf-trans-rfc6962-bis] only specifies how to use it to
   publish TLS server certificates issued by public certificate
   authorities (CAs).

   This document discusses the use of CT in addressing the improper
   issuance issues in DNSSEC. DNSSEC establishes chains of public keys
   for clients to assess the validity of DNS resource records. In order
   to prove the validity of keys used for signing DNS data, DNSSEC uses
   DNS public key (DNSKEY) RRsets and Delegation Signer (DS) RRsets to
   form authentication chains for the signed data, with each link in the
   chains vouching for the next by signing the next. If an
   authentication chain can be eventually connected to the a trusted DNS
   key or DS RR, the client then ensures the key for signing the data is
   legitimate. Unlike PKIX, SDNSEC inherently has strong naming
   constraints. The owner of a zone can only be allowed to sign the RRs
   in his zone. Any attempt in signing the RRs in other zones will be
   easily detected by clients. However, the owner of a zone is
   dependent on its parent delegation via the DS record to vouch for its
   DNSKEY. The zone itself is responsible for publishing DS records for
   the child zones that dependant on it. Misbehavior or compromise of
   the parent zone directly affects the core DNS security of the child
   zone. A detailed example is provided in Section 3.

   In order to benefit the detection of improper issuance/delegation of
   DNSSEC keys, this document describes an extension to CT to support
   logging DSs . The CT logs are publicly auditable, making it possible
   for anyone to verify the correctness of the log entries and monitor
   the new DS RR's appended to the log. The logs do not prevent the
   parent from issuing DS records that the child disagrees with, but
   they ensure that interested parties can detect such operations. For
   instance, For example, a zone owner that has been compromised or
   compelled by a third party can hijack a child zone to return
   different DNS data that is indistinguishable from DNSSEC validated
   data from the child zone by using its own DNSKEY to sign DNS data on
   behalf of the child zone. It could deliver this modified DNS data to
   only selected regions or individuals, making this attack very
   difficult to detect by the legitimate child zone.

In DNSSEC, it is assumed that the keys used for signing RRs or other keys will be properly maintained. This work follows this assumption and the compromise of key signing keys are out of scope of this work. This work assumes the existence of inside attacker. That is, a legal owner of a zone may try to attack or circumvent other zones. However, because the naming constraint feature of DNSSEC, a zone owner in principle can only use its keys to perform attacks on its child zones.

This work reuses most of the messages and data structures specified in [I-D.ietf-trans-rfc6962-bis] and makes necessary extensions for supporting DS RRs. Only the extensions to [I-D.ietf-trans-rfc6962-bis] are presented in this document.

2. Cryptographic Components of Certificate Transparency

   The introduce of cryptographic components of CT is in Section 2 of [I-D.ietf-trans-rfc6962-bis]. When applying CT for NDSSEC, a log is a single, ever-growing, append-only Merkle Tree of DS RRs.

3. Motivation Scenario

   Assume a zone (foo.bar.example) and its parent zone (bar.example) are owned by different organizations. Follows are the steps of an example attack that the owner of the parent zone could perform on the child zone.

   1. Set up a fake foo.bar.example DNS server

   2. The owner of parent zone generates a new KSK X1 and ZSK X2 for the fake foo.bar.example DNS server, because it does not know the private key of the KSK of foo.bar.example. The fake server uses the KSK to sign the ZSK and uses the ZSK to sign the fake resource records

   3. The owner of parent zone generates a DS record for the KSK record generated in step 2 in order to generate the certificate chain for the records in the fake server.

   4. The owner of bar.example signs the DS RR with its zone signing key and publishes it

   5. Change the IP address of the DNS server of foo.bar.example in the associated RRs to the IP address of the fake DNS server

   The owner of foo.bar.example may try to periodically access the DNS server of bar.example and monitor the RRs on it . However, there could be still a time window between two assessments which can be

taken advantage of by the owner of bar.example to perform a hijacking
attack and remove the bogus RRs before the owner of foo.bar.example
detects the attack.

In some cases, the parent can even achieve its objectives without
publishing the DS RR containing the invalid KSK, which makes the
attacks more difficult to detect.

If the owner of bar.example is forced to publish his operations on
the public CT logs, the attack introduced above will be detected
eventually. Through checking the log, it is easy detect the improper
issuance of RRs of his parent zone.

4. Log Format and Operation

As illustrated in Section 3, a zone owner may need to publish
multiple RRs in order to hijack the queries to its child zone and re-
direct them to another illegal DNS server. However, it is not
necessary to publish all those associated RRs to the log. In fact,
by publishing the DS RR which is critical in constructing the
authentication chain across two zones will be sufficient for helping
the public to detect the improper issuance behavior. In this
solution, when a zone owner generates a DS RR and delegates a new
public key to a child zone, it MUST publish the DS RR at least one CT
log in order to allow the public to monitor its behavior. Identical
to what is specified in [I-D.ietf-trans-rfc6962-bis], each CT log
needs to return a SCT to the zone owner immediately. The SCT will be
encapsulated in a SCT RR and published within a DS RR.

The SCT is the log's promise to incorporate the RR in the Merkle Tree
within a fixed amount of time known as the Maximum Merge Delay (MMD).
If the log has previously seen the certificate, it MAY return the
same SCT as it returned before. DNS servers MUST provide an SCT
within a SCT RR. DNSSEC clients will not honor a DS RR that does not
have a valid SCT. Therefore it is expected that a zone owner will
usually deliver the DS RRs for audit purposes.

4.1. Log Entries

Before publishing a DS RR, a zone owner MUST submit it to one or more
preferred logs. In order to enable attribution of each logged RR to
its issuer, the log SHALL publish a list of acceptable public keys
(or hashes of public keys) of root zone or islands of security. Each
submitted DS RR MUST be accompanied by all additional RRs (DNSKEY
RRs, DS RRs, and RRSIG RRs) which construct an authentication chain
to an accepted root public key.

Logs MUST verify that the authentication chain and make sure it leads
back to a trusted public key, using the chain of intermediate DNSKEY
RRs and DS RRs provided by the submitter. Logs MUST refuse to
publish a DS RR without a valid chain to a trusted key. If a DS RR
is accepted and an SCT issued, the accepting log MUST store the
entire chain used for verification, including the DS RR itself and
including the trusted 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 DS RR entry in a log MUST include
the following components:

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

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

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

  struct {
     DNSSECRR DSRR;
     DNSSECRR DNSSEC_key_chain<0..2^24-1>
  } DSRR_Chain_Entry;
```

"entry_type" is the type of this entry. the type value of a DSRR
LogEntry is TBD1.

"DSRR" is the DS RR submitted for auditing.

"DNSSEC_key_chain" is a chain of additional DNSSEC RRs required to
verify the DS RR.A typical authentication chain is as follow: Trusted
DNSSKEY ->[DS->(DNSKEY)*->DNSKEY]*-> Submitted DS RR, where "*"
denotes zero or more sub-chains. (DNSKEY)* indicates that DNSSEC
permits additional layers of DNSKEY RRs including the keys for
signing other keys within a zone. Each DNSKEY/DS RR in the chain is
authenticated by a RRSIG RR. In practice, a RRSIG RR is normally
used to sign a DS/DNSKEY RRset. Therefore, not only the DS/DNSKEY RR
on the authentication chain but also other records in the RRset
SHOULD be provided to the log the verification purpose. Otherwise,
the log may have to consult DNS again in order to verify the

   authentication chains. Logs SHOULD limit the length of chain they
   will accept.

4.2. Structure of the Signed Certificate Timestamp

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

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

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

      struct {
          opaque key_id[32];
      } LogID;

     struct {
        opaque issuer_key_hash[32];
        C14N_DSRR dsrr;
      } DSRR;

      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].

   dsrr is the submitted DS RR in a canonical form. The
   canconicalization of a DS RR is described in Section 6.2 of
   [RFC4304].

```
    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 BIN_entry: BinaryDigest;
                case BINDI_entry: BinaryDigest
            } signed_entry;
            CtExtensions extensions;
        };
    } SignedCertificateTimestamp;
```

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

"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].

"signed_entry" is the is DSRR (in the case of a DSRR_entry), as
described above.

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

4.3. Merkle Tree

This specification extends the structure of the Merkle Tree input in
Section 3.5 of [I-D.ietf-trans-rfc6962-bis] and enable it to
encapsulate DS RR:

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

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

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

The fields in the input are introduced in Section 3.5 of
[I-D.ietf-trans-rfc6962-bis].

Open question[dacheng]: We should include the RRs constucting the
authenticaiton chain in the input, right?

5. Including the Signed Certificate Timestamp into DNS Security
    Extensions

    In section 3.5 of [I-D.ietf-trans-rfc6962-bis]

5.1. SCT RR

    The SCT associated with a DS RR is stored within a STC RR. A DNS
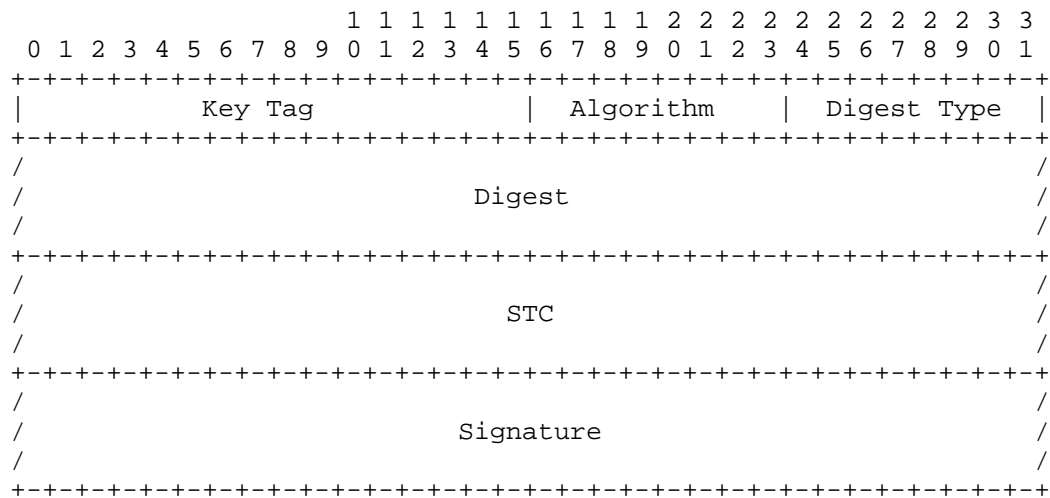   server MAY provide multiple SCT RRs for one DS RR.

    The type number for the SCT RR is TBD3.

    The SCT resource record is class independent.

    The life period of SCT RR should not be set in a way that the RR will
    not be expired before the associated DS RR.

    The RDATA portion of an SCT RR is as shown below.

```
                       1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |             Key Tag             |  Algorithm  | Digest Type  |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    /                                                               /
    /                           Digest                              /
    /                                                               /
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    /                                                               /
    /                            STC                                /
    /                                                               /
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    /                                                               /
    /                         Signature                             /
    /                                                               /
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

5.1.1. The Key Tag Field

   The Key Tag field lists the key tag of the DNSKEY RR referred to by
   the SCT record, in network byte order. Appendix B of [RFC4034]
   describes how to compute a Key Tag.

5.1.2. The Algorithm Field

   The Algorithm field lists the algorithm number of the DNSKEY RR
   referred to by the SCT record. Appendix A.1 of [RFC4034] lists the
   algorithm number types.

5.1.3. The Digest Type Field

   The Digest Type field identifies the algorithm used to construct the
   digest used to identify the DS RR that the SCT RR refers to.
   Appendix A.2 of [RFC4034] lists the possible digest algorithm types.

5.1.4. The Digest Field

   The method of calculating digest is identical to what is specified in
   Section 5.1.4 of [RFC2065].[RFC4034]

5.1.5. The SCT Field

   This field contains the SCT got from the log, encoded in BASE64.

5.1.6. The Signature Field

   This field contains the SCT signature associated with the SCT. The
   Signature field is represented as a Base64 encoding of the signature.

5.2. Operations

   After introducing the SCT RR, the verification procedures of DNS data
   specified in DNSSEC[RFC4305] do not change a lot. However, the
   correctness of CTS needs to be assessed during checking the validity
   of a DS RR.

   A DS RR needs to be associated with a CTS RR which contains a valid
   CTS and signed with a proper public key. Otherwise, the DS RR will
   not be used to construct the authentication chain. The signatures of
   DS RR and its CTS RR should be stored in different RRSIG RR
   respectively. In addition, a DNS server will sends CTS RRs and the
   associated RRSIG RRs to a resolver only when it indicates the support
   of CT in the request.

6. 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 DNSSEC.

6.1. Add DNSSEC RR Chain to Log

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

   Inputs:

      chain: An array of base64-encoded DNS RR. The first element is
         the submitted DS RR; the second chains to the first and so on to
         the last, which is a trurst DNSKey RR.

   Outputs:

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

      id: The log ID, base64 encoded.

timestamp: The SCT timestamp, in decimal.

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

signature: The SCT signature, base64 encoded.

6.2. Retrieve Accepted Root DNSKEY RRs

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

No inputs.

Outputs:

RRs: An array of base64-encoded DNSKEY RRs that are acceptable to
   the log.

7. IANA Considerations

This document specified a new LogEntryType value TBD1 to identify DS
RR entry, a new SCT Type value TBD2, and a type number for the SCT
DNS RR TBD3.

8. Security Considerations

8.1. Logging Other Types of RRs

This solution only tries to describes a solution to disclose keys for
DNSSEC in logs for the public to audit. However, it may be valuable
to also log the RRs specified in [RFC1035]. For instance, assume
there is an attacker which has compromised the zone authentication
key and is able to perform the MITM attack between a resolver and the
DNS server of the zone. It is possible for an attacker to transfer a
forged RR which is signed with the compromised key. The current
solution cannot benefit the detection of this attack in this
scenario. However, if the RR is also required to be uploaded to
public logs, the condition is changed. If the attacker does not
publish the RR to a log, it cannot get the SCT. When the attacker
tries to publish the RR to the log, the owner of the zone may detect
the problem even if the attacker can provide keys to convince the log
to accept the RR.

8.2. Scalability Concerns

   The log MAY limit accepting entries where the TTL is too short or the
   RRSIG times are too far in the future or the past, to avoid spamming
   the log. It should probably also put a maximum on the number of
   child zones to avoid getting spammed.

9. Acknowledgements

10. 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.

   [RFC1035] Mockapetris, P., "Domain names - implementation and
             specification", STD 13, RFC 1035, November 1987.

   [RFC2065] Eastlake, D. and C. Kaufman, "Domain Name System Security
             Extensions", RFC 2065, January 1997.

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

   [RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S.
             Rose, "Resource Records for the DNS Security Extensions",
             RFC 4034, March 2005.

   [RFC4304] Kent, S., "Extended Sequence Number (ESN) Addendum to
             IPsec Domain of Interpretation (DOI) for Internet Security
             Association and Key Management Protocol (ISAKMP)", RFC
             4304, December 2005.

   [RFC4305] Eastlake, D., "Cryptographic Algorithm Implementation
             Requirements for Encapsulating Security Payload (ESP) and
             Authentication Header (AH)", RFC 4305, December 2005.

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

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

   Ning Kong

   Email: nkong@cnnic.cn