TRANS
Internet-Draft
Intended status: Experimental
Expires: September 22, 2016

L. Nordberg
NORDUnet
D. Gillmor
ACLU
T. Ritter

March 21, 2016

Gossiping in CT
draft-ietf-trans-gossip-02

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 behaviour 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. It describes three gossiping mechanisms for Certificate Transparency: SCT Feedback, STH Pollination and Trusted Auditor Relationship.

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 22, 2016.

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.

[TODO: enumerate the interfaces used for detecting misbehaviour?]

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 behaviour 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 both sides, to see the 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

SCT Feedback enables HTTPS clients to share Signed Certificate Timestamps (SCTs) (Section 3.3 of [RFC-6962-BIS-09]) with CT auditors in a privacy-preserving manner by sending SCTs to originating HTTPS servers, who in turn share them with CT auditors.

In STH Pollination, HTTPS clients use HTTPS servers as pools to share Signed Tree Heads (STHs) (Section 3.6 of [RFC-6962-BIS-09]) 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 logs would already indicate that the client is accessing that site. In this way a site can accumulate records of SCTs that have been issued by various logs for that site, providing a consolidated repository of SCTs that could be shared with auditors. Auditors can use this information to detect logs that misbehave by not including certificates within the time period stipulated by the MMD metadata.

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-09], including STH, SCT, Version, LogID, SCT timestamp, CtExtensions, SCT signature, Merkle Tree Hash.

This document relies on terminology defined in [draft-ietf-trans-threat-analysis-03], 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

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

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

- HTTPS clients and CT auditors (Trusted Auditor Relationship)

6. What to gossip about and how

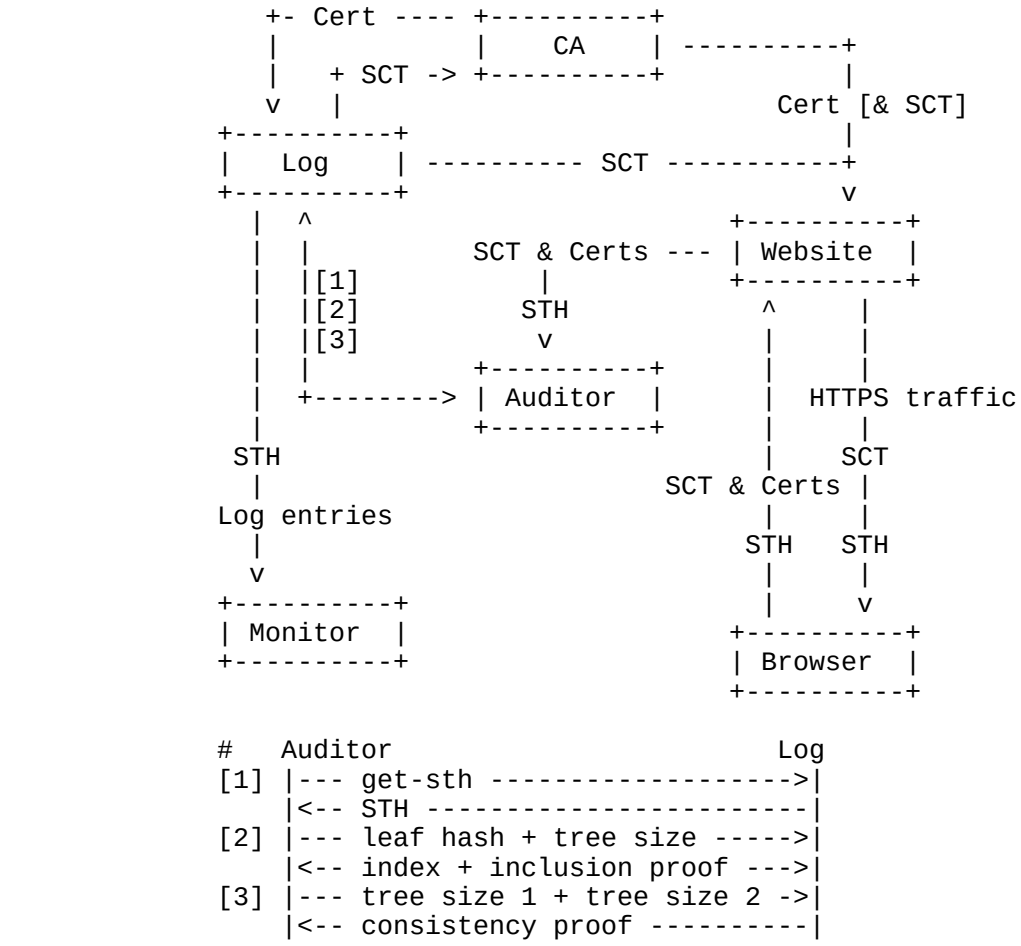
There are three separate gossip streams:

- SCT Feedback - transporting SCTs and certificate chains from HTTPS clients to CT auditors via HTTPS servers.
- STH Pollination - HTTPS clients and CT auditors using HTTPS servers as STH pools for exchanging STHs.
- 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 interact 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.



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 incentivises 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 visisted 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 with the following content:

- x509_chain: An array of base64-encoded X.509 certificates. The first element is the end-entity certificate, the second certifies the first and so on.

- sct_data: An array of objects consisting of the base64 representation of the binary SCT data as defined in [RFC-6962-BIS-09] Section 3.3.

We will refer to this object as 'sct_feedback'.

The x509_chain element always contains at least one element. It also always contains a full chain from a leaf certificate to a self-signed trust anchor.

[TBD: Be strict about what sct_data may contain or is this sufficiently implied by previous sections?]

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-09] section 3.4 - 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 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.) They MUST NOT be sent to any Subject Alternate Names specified in the certificate. In the case of certificates that validate multiple domain names, the same SCT is expected to be stored multiple times.

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 [TODO double-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. It is not necessary to send an sct_feedback object constructed from the current TLS session.

The client MUST NOT send the same set of SCTs to the same server more often than TBD.

[TODO: expand on rate/resource limiting motivation]

Refer to Section 11.3 for recommendations about strategies.

Because SCTs can be used as a tracking mechanism (see Section 10.4.2), 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 XXX. 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.)

[XXX is currently https://www.torproject.org/projects/torbrowser/design/#identifier-linkability How should it be references? Do we need to copy this out into another document? An appendix?]

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.

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.

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 perfom 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. Perfoming 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 still 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 parent 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 recieving 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:

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)

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

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

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 Section 10.2. 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. It should do this after completing the validity check on 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.

[TBD: Thinking about building this, how does the SCT Feedback app know which sites it's authoritative for?]

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 more advanced server configuration will detect the [TODO double-CA-compromise] attack. In this configuration the server will not modify the sct_feedback object prior to performing checks 2, 3, and 4.

To prevent a malicious client from filling the server's data store, the HTTPS Server SHOULD perform an additional check:

1. 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.

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 terminiating 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

[TBD: Should that be .well-known? Depends on whether auditors will operate in their own URL name space or not.]

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.

HTPS 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.

The response contains zero or more STHs in the same format, described in Section 8.2.4.

An HTTPS client may acquire STHs by several methods:

- in replies to pollination POSTs;
- asking logs that it recognises for the current STH, either directly (v2/get-sth) or indirectly (for example over DNS)
- resolving an SCT and certificate to an STH via an inclusion proof
- 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 may 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.

- 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 metadata to express this ([RFC-6962-BIS-09] sections 3.6 and 5.1).

- 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 10.4.5.

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 who 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). The client SHOULD be provided the final STH issued by the log and 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 from an HTTPS server, and must obtain an inclusion proof to an STH in order to verify the promise made by the SCT.

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 must obtain a consistency proof to a more recent STH.

A client SHOULD perform proof fetching. 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.

[TBD: Linus doesn't like this because we're mandating behavior that is not necessarily safe. Is it unsafe? Not sure.]

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.)

[TODO: Add a reference to Google's DNS mechanism more proper than http://www.certificate-transparency.org/august-2015-newsletter]

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 recieved 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 an 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 who 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 the following content:

- sths - an array of 0 or more fresh SignedTreeHead’s as defined in [RFC-6962-BIS-09] Section 3.6.1.

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 10.4.1 and Section 10.4.6.

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-09] 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. An informal writeup of such a protocol can be found at XXX.

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:

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

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

It is worthwhile to note that an attacker may be able to prevent detection of an attack on a webserver (in all cases) if SCT Feedback is not implemented. This attack is detailed in Section 10.1).

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.

If SCT Feedback is not deployed by a webserver, malicious logs will be able to attack all users of the webserver (who do not have a Trusted Auditor relationship) with impunity. 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) 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 who 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 was not deployed, HTTPS Clients visiting HTTPS Servers who 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

The interactions of the mechanisms is thus outlined:

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 required 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 two ways:

In the first attack, the log can present a split view of the log for all time. The only way to detect this attack is to resolve each view of the log to the two most recent STHs and then force the log to present a consistency proof. (Which it cannot.) This attack can be detected by CT auditors participating in STH Pollination, as long as they are explicitly built to handle the situation of a log continuously presenting a split view.

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.

10.2. Dual-CA Compromise

XXX describes an attack possible by an adversary who compromises two Certificate Authorites and a Log. This attack is difficult to defend against in the CT ecosystem, and XXX 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 must compare the chain built by clients to the chain supplied by the log. If the chains differ significantly, the auditor can raise a concern.

[What does 'differ significantly' mean? We should provide guidance. I _think_ the correct algorithm to raise a concern is:

If one chain is not a subset of the other AND If the root certificates of the chains are different THEN It's suspicious.

Justification: - Cross-Signatures could result in a different org being treated as the 'root', but in this case, one chain would be a subset of the other. - Intermediate swapping (e.g. different signature algorithms) could result in different chains, but the root would be the same.

(Hitting both those cases at once would cause a false positive though.)

What did I miss?]

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 scenatios (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. 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-anonymisation 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.

10.4.1. Privacy and SCTs

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

10.4.2. 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-09] Section 2.1.4). 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 unqiue 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. [XXX any mitigations possible here?]

10.4.3. Privacy for HTTPS clients performing STH Proof Fetching

An HTTPS client performing Proof Fetching should only request proofs from a CT log that it accepts SCTs from. An HTTPS client MAY [TBD SHOULD?] regularly request an STH from all logs it is willing to accept, even if it has seen no SCTs from that log.

[TBD how regularly? This has operational implications for log operators]

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) an may not be suitable for some.

10.4.4. Privacy in STH Pollination

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

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

- that the client is talking, possibly indirectly, to the log indicated by the tree hash;
- which software and software version is being used.

There is a possible fingerprinting attack where a log issues a unique STH for a targeted HTTPS client. This is similar to the fingerprinting attack described in Section 10.4.2, 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.

10.4.5. 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.

10.4.6. 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 who 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.

10.4.7. 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:

- 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.

- 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 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-Alives 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 cirsumstances 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 recieve an error. These situations are:

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

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

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

5. The client has attempted to share a specific piece of data with a Trusted Auditor - but recieves 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, a client SHOULD make every effort to send this SCT via SCT Feedback. 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, so if implemented, it should connect using public key pinning and not considered an item delivered until it recieves 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 occured all at once, in a flurry of activity, a log would know that SCTs or STHs recieved 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.

[TBD: What other recommendations do we want to make here? We can talk more about the inadequecies of DNS... The first paragraph is 80% identical between here and above]

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:

- 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

- 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

- 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

- 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

- 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 "From a Trickle to a Flood: Active Attacks on Several Mix Types", [Y], and [Z].

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 recieve 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 cryptographyically 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. 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.

Servers are most vulnerable to flushing attacks, as they release records to anonymous connections. An attacker can perform a Sybil attack - connecting to the server hundreds or thousands of times in an attempt to trigger repeated release of a record, and then deletion. For this reason, servers must be especially aggressive about retaining data for a longer period of time.

Clients are vulnerable to flushing attacks targetting STHs, as these can be given to any cooperating server and an attacker can generally induce connections to random servers using javascript. It would be more difficult to perform a flushing attack against SCTs, as the target server must be authenticated (and an attacker impersonating an authentic server presents a recursive problem for the attacker). Nonetheless, flushing SCTs should not be ruled impossible. A Trusted Auditor may also be vulnerable to flushing attacks if it does not perform auditing operations itself.

Flushing attacks are defended against using non-determinism and dummy messages. The goal is to ensure that an adversary does not know for certain if the data in question has been released or not, and if it has been deleted or not.

[TBD: At present, we do not have any support for dummy messages. Do we want to define a dummy message that clients and servers alike know to ignore? Will HTTP Compression leak the presence of >1 dummy messages?

Is it sufficient to define a dummy message as _anything_ with an invalid siganture? This would negatively impact SCT Feedback servers that log all things just in case they're interesting.]

11.3.3. The Deletion Algorithm

No entity in CT Gossip is required to delete SCTs or STHs at any time, except to respect user's wishes such as private browsing mode or clearing history. However, requiring infinite storage space is not a desirable characteristic in a protocol, so deletion is expected.

While deletion of SCTs and STHs 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 an SCT or STH until it has been successfully resolved to a new STH via a proof.

When it is time to delete a record, it is important that the decision to do so not be done deterministicly. Introducing non-determinism in the decision is absolutely necessary to prevent an adversary from knowing with certainty that the record has been successfully flushed from a target. Therefore, we speak of making a record 'eligible for deletion' and then being processed by the 'deletion algorithm'. Making a record eligible for deletion simply means that it will have the deletion algorithm run. The deletion algorithm will use a probability based system and a secure random number generator to determine if the record will be deleted.

Although the deletion algorithm is specifically designed to be non-deterministic, if the record has been resolved via proof to a new STH the record may be safely deleted, as long as the new STH is retained.

The actual deletion algorithm may be [STATISTICS HERE]. [Something as simple as 'Pick an integer securely between 1 and 10. If it's greater than 7, delete the record.' Or something more complicated.]

[TODO Enumerating the problems of different types of mixes vs Cottrell Mix]

11.3.3.1. Experimental Algorithms

More complex algorithms could be inserted at any step. Three examples are illustrated:

SCTs are not eligible to be submitted to an Auditor of Last Resort. Therefore, it is more important that they be resolved to STHs and reported via SCT feedback. If fetching an inclusion proof regularly fails for a particular SCT, one can require it be reported more times than normal via SCT Feedback before becoming eligible for deletion.

Before an item is made eligible for deletion by a client, the client could aim to make it difficult for a point-in-time attacker to flush the pool by not making an item eligible for deletion until the client has moved networks (as seen by either the local IP address, or a report-back providing the client with its observed public IP address). The HTTPS client could also require reporting over a timespan, e.g. it must be reported at least N time, M weeks apart. This strategy could be employed always, or only when the client has disabled proof fetching and the Auditor of Last Resort, as those two mechanisms (when used together) will enable a client to report most attacks.

11.3.3.2. Concrete Recommendations

The recommendations for behavior are: - If proof fetching is enabled, do not delete an SCT until it has had a proof resolving it to an STH. - If proof fetching continually fails for an SCT, do not make the item eligible for deletion of the SCT until it has been released, multiple times, via SCT Feedback. - If proof fetching continually fails for an STH, do not make the item eligible for deletion until it has been queued for release to an Auditor of Last Resort. - Do not dequeue entries to an Auditor of Last Resort if reporting fails. Instead keep the items queued until they have been successfully sent. - Use a probability based system, with a cryptographically secure random number generator, to determine if an item should be deleted. - Select items from the datastores by selecting random indexes into the datastore. Use a cryptographically secure random number generator.

[TBD: More?]

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

11.3.3.2.1. STH Data Structures

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

```
class STH
{
  uint32   proof_attempts
  uint32   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. One thing to note about this pseudocode is that it aggressively removes STHs once they have been resolved to a newer STH (if proof fetching is configured). The only STHs in the store are ones that have never been resolved to a newer STH, either because proof fetching does not occur, has failed, or because the STH is considered too new to request a proof for. It seems less likely that servers will perform proof fetching. Therefore it would be recommended that the various constants in use be increased considerably to ensure STHs are pollinated more aggressively.

```
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 possibly delete the given STH
  //  from the data store
  def delete_maybe(STH s) {
    //Perform statistical test and see if I should delete this bundle
  }
```

```
    //  This function is called to (certainly) 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)
        while(len(indexes) < MAX_STH_TO_GOSSIP) {
          r = randomInt() % modulus
          if(r not in indexes
             && now() - this.sth_list[i].timestamp < ONE_WEEK)
            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 can 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.

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

       if(now() - sth.timestamp > ONE_WEEK) {
         //STH is too old, we must remove it
         if(proof_fetching_enabled
            && auditor_of_last_resort_enabled
            && (sth.proof_failure_count / sth.proof_attempts)
               > MIN_PROOF_FAILURE_RATIO_CONSIDERED_SUSPICIOUS) {
           queue_sth_for_auditor_of_last_resort(sth)
           delete_maybe(sth)
         } else {
           delete_now(sth)
         }
       }

       else if(proof_fetching_enabled
               && now() - sth.timestamp > TWO_DAYS
               && now() - sth.timestamp > LOG_MMD) {
         sth.proof_attempts++
         queue_consistency_proof(sth, consistency_proof_callback)
       }
     }
   }
```

11.3.3.2.2. STH Deletion Procedure

The STH Deletion Procedure is run after successfully submitting a list of STHs to a third party during pollination. The following pseudocode would be included in the STHStore class, and called with the result of get_pollination_selection(), after the STHs have been (successfully) sent to the third party.

```
   //  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()
   def after_submit_to_thirdparty(STH[] sth_list)
   {
     foreach(sth in sth_list)
     {
       sth.num_reports_to_thirdparty++

       if(proof_fetching_enabled) {
         if(now() - sth.timestamp > LOG_MMD) {
           sth.proof_attempts++
           queue_consistency_proof(sth, consistency_proof_callback)
         }

         if(auditor_of_last_resort_enabled
            && sth.proof_failure_count >
               MIN_PROOF_ATTEMPTS_CONSIDERED_SUSPICIOUS
            && (sth.proof_failure_count / sth.proof_attempts) >
               MIN_PROOF_FAILURE_RATIO_CONSIDERED_SUSPICIOUS) {
            queue_sth_for_auditor_of_last_resort(sth)
         }
       }
       else { //proof fetching not enabled
         if(sth.num_reports_to_thirdparty
            > MIN_STH_REPORTS_TO_THIRDPARTY) {
           delete_maybe(sth)
         }
       }
     }
   }

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

11.3.3.2.3. SCT Data Structures

TBD TBD This section is not well abstracted to be used for both servers and clients. TKTK

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

```
class SCT
{
  uint32 proof_attempts
  uint32 proof_failure_count
  bool   has_been_resolved_to_sth
  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)
        return false
    }
    return true
  }

  def max_proof_failure_count() {
    uint32 max = 0
    foreach(s in this.sct_list) {
      if(s.proof_failure_count > max)
        max = proof_failure_count
    }
    return max
  }
}
```

We suppose a large data structure is used, such as a hashmap, indexed by the domain name. For each domain, the structure will contain a data structure that holds the SCTBundles seen for that domain, as well as encapsulating some logic relating to SCT Feedback for that particular domain.

```
class SCTStore
{
  string   domain
  datetime last_contact_for_domain
  uint32   num_submissions_attempted
  uint32   num_submissions_succeeded
  SCTBundle[] observed_records

  //  This function is called after recieving an SCTBundle.
  //  For Clients, this is after a successful connection to a
  //  HTTPS Server, calling this function with an SCTBundle
  //  constructed from that certificate chain and SCTs
  //  For Servers, this is after receiving SCT Feedback
  def insert(SCTBundle b) {
    if(operator_is_server) {
      if(!passes_validity_checks(b))
        return
    }
    foreach(e in this.observed_records) {
      if(e.equals(b))
        return
      else if(e.approx_equals(b)) {
        e.insert_scts(b.sct_list)
        return
      }
    }
    this.observed_records.insert(b)
  }

  //  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) > MAC_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 delete_maybe(SCTBundle b) {
      //Perform statistical test and see if I should delete this bundle
    }

    def delete_now(SCTBundle b) {
      this.observed_records.remove(b)
    }

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

We also suggest a function that can be called periodically in the background, iterating through all SCTStore objects in the large hashmap (here called 'all_sct_stores') and removing old data.

```
def clear_old_data()
{
  foreach(storeEntry in all_sct_stores)
  {
    if(storeEntry.num_submissions_succeeded == 0
       && storeEntry.num_submissions_attempted
          > MIN_SCT_ATTEMPTS_FOR_DOMAIN_TO_BE_IGNORED)
    {
      all_sct_stores.remove(storeEntry)
    }
    else if(storeEntry.num_submissions_succeeded > 0
            && now() - storeEntry.last_contact_for_domain
               > TIME_UNTIL_OLD_SCTDATA_ERASED)
    {
      all_sct_stores.remove(storeEntry)
    }
  }
}
```

11.3.3.2.4. SCT Deletion Procedure

The SCT Deletion procedure is more complicated than the respective STH procedure. This is because servers may elect not to participate in SCT Feedback, and this must be accounted for by being more conservative in sending SCT reports to them.

The following pseudocode would be included in the SCTStore class, and called with the result of get_gossip_selection() after the SCT Feedback has been sent (successfully) to the server. We also note that the first experimental algorithm from above is included in the pseudocode as an illustration.

```
//  This function is called after successfully providing SCT Feedback
//  to a server. It is passed the feedback sent to the server, which
//  is the output of get_gossip_selection()
def after_submit_to_thirdparty(SCTBundle[] submittedBundles)
{
  foreach(bundle in submittedBundles)
  {
    bundle.num_reports_to_thirdparty++

    if(proof_fetching_enabled) {
      if(!bundle.has_been_fully_resolved_to_sths()) {
        foreach(s in bundle.sct_list) {
          if(!s.has_been_resolved_to_sth) {
            s.proof_attempts++
            queue_inclusion_proof(sct, inclusion_proof_callback)
          }
        }
      }
      else {
        if(run_ct_gossip_experiment_one) {
          if(bundle.num_reports_to_thirdparty
             > MIN_SCT_REPORTS_TO_THIRDPARTY
             && bundle.num_reports_to_thirdparty * 1.5
                > bundle.max_proof_failure_count()) {
            maybe_delete(bundle)
          }
        }
        else { //Do not run experiment
          if(bundle.num_reports_to_thirdparty
             > MIN_SCT_REPORTS_TO_THIRDPARTY) {
            maybe_delete(bundle)
          }
        }
      }
    }
```

```
      else {//proof fetching not enabled
        if(bundle.num_reports_to_thirdparty
           > (MIN_SCT_REPORTS_TO_THIRDPARTY
              * NO_PROOF_FETCHING_REPORT_INCREASE_FACTOR)) {
          maybe_delete(bundle)
        }
      }
    }
  }

  // This function is a callback invoked after an inclusion proof
  // has been retrieved
  def inclusion_proof_callback(inclusion_proof, original_sct, error)
  {
    if(!error) {
      original_sct.has_been_resolved_to_sth = True
      insert_to_sth_datastore(inclusion_proof.new_sth)
    } else {
      original_sct.proof_failure_count++
    }
  }
```

12. IANA considerations

[TBD]

13. Contributors

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

14. ChangeLog

14.1. Changes between ietf-01 and ietf-02

- Requiring full certificate chain in SCT Feedback.
- Clarifications on what clients store for and send in SCT Feedback added.
- SCT Feedback server operation updated to protect against DoS attacks on servers.
- Pre-Loaded vs Locally Added Anchors explained.
- Base for well-known URL’s changed.

- Remove all mentions of monitors - gossip deals with adutitors.
- New sections added: Trusted Auditor protocol, attacks by actively malicious log, the Dual-CA compromise attack, policy recommendations,

14.2. Changes between ietf-00 and ietf-01

- Improve langugage and readability based on feedback from Stephen Kent.
- STH Pollination Proof Fetching defined and indicated as optional.
- 3-Method Ecosystem section added.
- Cases with Logs ceasing operation handled.
- Text on tracking via STH Interaction added.
- Section with some early recommendations for mixing added.
- Section detailing blocking connections, frustrating it, and the implications added.

14.3. Changes between -01 and -02

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

14.4. Changes between -00 and -01

- Add the SCT feedback mechanism: Clients send SCTs to originating web server which shares them with auditors.
- Stop assuming that clients see STHs.
- Don’t use HTTP headers but instead .well-known URL’s - avoid that battle.
- Stop referring to trans-gossip and trans-gossip-transport-https - too complicated.

- Remove all protocols but HTTPS in order to simplify - let's come back and add more later.
- Add more reasoning about privacy.
- Do specify data formats.

15. References

15.1. Normative References

[RFC-6962-BIS-09] Laurie, B., Langley, A., Kasper, E., Messeri, E., and R. Stradling, "Certificate Transparency", October 2015, <https://datatracker.ietf.org/doc/draft-ietf-trans-rfc6962-bis/>.

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

15.2. Informative References

[draft-ietf-trans-threat-analysis-03] Kent, S., "Attack Model and Threat for Certificate Transparency", October 2015, <https://datatracker.ietf.org/doc/draft-ietf-trans-threat-analysis/>.

Authors' Addresses

Linus Nordberg
NORDUnet

Email: linus@nordu.net

Daniel Kahn Gillmor
ACLU

Email: dkg@fifthhorseman.net

Tom Ritter

Email: tom@ritter.vg

Public Notary Transparency Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 22, 2016

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

March 21, 2016

Certificate Transparency
draft-ietf-trans-rfc6962-bis-13

Abstract

This document describes a protocol for publicly logging the existence of Transport Layer Security (TLS) certificates as they are issued or observed, in a manner that allows anyone to audit certification authority (CA) activity and notice the issuance of suspect certificates as well as to audit the certificate logs themselves. The intent is that eventually clients would refuse to honor certificates that do not appear in a log, effectively forcing CAs to add all issued certificates to the logs.

Logs are network services that implement the protocol operations for submissions and queries that are defined in this document.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 22, 2016.

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 need to be trusted because they are publicly auditable. Anyone may verify the correctness of each log and monitor when new certificates are added to it. The logs do not themselves prevent misissue, but they ensure that interested parties (particularly those named in certificates) can detect such misissuance. Note that this is a general mechanism, but in this document, we only describe its use for public TLS server certificates issued by public certification authorities (CAs).

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

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

Similarly, those who have seen signed timestamps from a particular log can later demand a proof of inclusion from that log. If the log

is unable to provide this (or, indeed, if the corresponding certificate is absent from monitors' copies of that log), that is evidence of the incorrect operation of the log. The checking operation is asynchronous to allow 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 technically achieved using Merkle Trees, which can be used to show that any particular instance of the log is a superset of any particular previous instance. Likewise, Merkle Trees avoid the need to blindly trust logs: if a log attempts to show different things to different people, this can be efficiently detected by comparing tree roots and consistency proofs. Similarly, other misbehaviors of any log (e.g., issuing signed timestamps for certificates they then don't log) can be efficiently detected and proved to the world at large.

1.1. Requirements Language

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

1.2. Data Structures

Data structures are defined according to the conventions laid out in Section 4 of [RFC5246].

2. Cryptographic Components

2.1. Merkle Hash Trees

Logs use a binary Merkle Hash Tree for efficient auditing. The hashing algorithm used by each log is expected to be specified as part of the metadata relating to that log. We have established a registry of acceptable algorithms, see Section 11.2. The hashing algorithm in use is referred to as HASH throughout this document and the size of its output in bytes as HASH_SIZE. The input to the Merkle Tree Hash is a list of data entries; these entries will be hashed to form the leaves of the Merkle Hash Tree. The output is a single HASH_SIZE Merkle Tree Hash. Given an ordered list of n inputs, D[n] = {d(0), d(1), ..., d(n-1)}, the Merkle Tree Hash (MTH) is thus defined as follows:

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

MTH({}) = HASH().

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

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

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

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

where || is concatenation and D[k1:k2] denotes the list {d(k1), d(k1+1),..., d(k2-1)} of length (k2 - k1). (Note that the hash calculations for leaves and nodes differ. This domain separation is required to give second preimage resistance.)

Note that we do not require the length of the input list to be a power of two. The resulting Merkle Tree may thus not be balanced; however, its shape is uniquely determined by the number of leaves. (Note: This Merkle Tree is essentially the same as the history tree [CrosbyWallach] proposal, except our definition handles non-full trees differently.)

2.1.1. Merkle Inclusion Proofs

A Merkle inclusion proof for a leaf in a Merkle Hash Tree is the shortest list of additional nodes in the Merkle Tree required to compute the Merkle Tree Hash for that tree. Each node in the tree is either a leaf node or is computed from the two nodes immediately below it (i.e., towards the leaves). At each step up the tree (towards the root), a node from the inclusion proof is combined with the node computed so far. In other words, the inclusion proof consists of the list of missing nodes required to compute the nodes leading from a leaf to the root of the tree. If the root computed from the inclusion proof matches the true root, then the inclusion proof proves that the leaf exists in the tree.

Given an ordered list of n inputs to the tree, D[n] = {d(0), ..., d(n-1)}, the Merkle inclusion proof PATH(m, D[n]) for the (m+1)th input d(m), 0 <= m < n, is defined as follows:

The proof for the single leaf in a tree with a one-element input list D[1] = {d(0)} is empty:

PATH(0, {d(0)}) = {}

For n > 1, let k be the largest power of two smaller than n. The proof for the (m+1)th element d(m) in a list of n > m elements is then defined recursively as

PATH(m, D[n]) = PATH(m, D[0:k]) : MTH(D[k:n]) for m < k; and

PATH(m, D[n]) = PATH(m - k, D[k:n]) : MTH(D[0:k]) for m >= k,

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

2.1.2. Merkle Consistency Proofs

Merkle consistency proofs prove the append-only property of the tree. A Merkle consistency proof for a Merkle Tree Hash MTH(D[n]) and a previously advertised hash MTH(D[0:m]) of the first m leaves, m <= n, is the list of nodes in the Merkle Tree required to verify that the first m inputs D[0:m] are equal in both trees. Thus, a consistency proof must contain a set of intermediate nodes (i.e., commitments to inputs) sufficient to verify MTH(D[n]), such that (a subset of) the same nodes can be used to verify MTH(D[0:m]). We define an algorithm that outputs the (unique) minimal consistency proof.

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

PROOF(m, D[n]) = SUBPROOF(m, D[n], true)

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

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

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

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

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

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

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

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

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

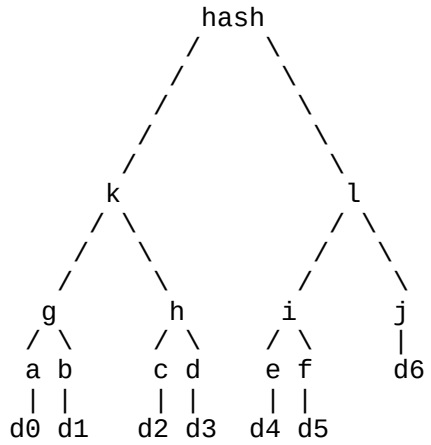
SUBPROOF(m, D[n], b) = SUBPROOF(m - k, D[k:n], false) : MTH(D[0:k])

Here, : is a concatenation of lists, and D[k1:k2] denotes the length (k2 - k1) list {d(k1), d(k1+1),..., d(k2-1)} as before.

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

2.1.3. Example

The binary Merkle Tree with 7 leaves:



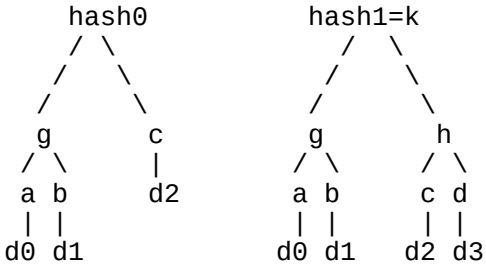
The inclusion proof for d0 is [b, h, l].

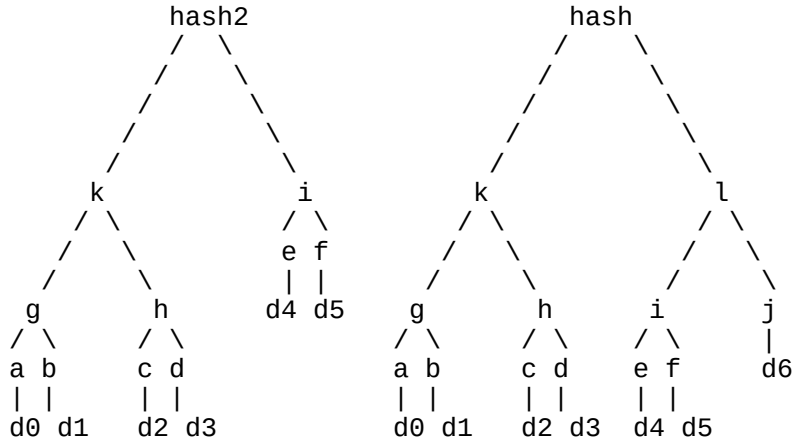
The inclusion proof for d3 is [c, g, l].

The inclusion proof for d4 is [f, j, k].

The inclusion proof for d6 is [i, k].

The same tree, built incrementally in four steps:





The consistency proof between hash0 and hash is PROOF(3, D[7]) = [c, d, g, l]. c, g are used to verify hash0, and d, l are additionally used to show hash is consistent with hash0.

The consistency proof between hash1 and hash is PROOF(4, D[7]) = [l]. hash can be verified using hash1=k and l.

The consistency proof between hash2 and hash is PROOF(6, D[7]) = [i, j, k]. k, i are used to verify hash2, and j is additionally used to show hash is consistent with hash2.

2.1.4. Signatures

Various data structures are signed. A log MUST use one of the signature algorithms defined in the Section 11.3 section.

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. 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 5.6). The submitter SHOULD validate the returned SCT as described in Section 9.2 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 6.1) to a log. Since certificates may not be accepted by TLS clients unless logged, it is expected that certificate owners or their CAs will usually submit them.

3.2. Precertificates

Alternatively, (root as well as intermediate) CAs may preannounce a certificate prior to issuance by submitting a precertificate (Section 6.2) 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. Examples of situations where the returned SCT is not incorporated into the issued certificate would be when a CA sends the precertificate to multiple logs, but only incorporates the SCTs that are returned first, or the CA is using domain name redaction and intends to use another mechanism to publish SCTs (such as an OCSP response (Section 8.1.1) or the TLS extension (Section 7.2)).

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

- It MUST be DER encoded.
- "SignedData.encapContentInfo.eContentType" MUST be the OID 1.3.101.78.

- "SignedData.encapContentInfo.eContent" MUST contain a TBSCertificate [RFC5280], which MAY redact certain domain name labels that will be present in the issued certificate (see Section 4.2) and MUST NOT contain any SCTs, but which will be otherwise identical to the TBSCertificate in the issued certificate.

- "SignedData.signerInfos" MUST contain a signature from the same (root or intermediate) CA that will ultimately issue the certificate. This signature indicates the CA's intent to issue the certificate. This intent is considered binding (i.e. misissuance of the precertificate is considered equivalent to misissuance of the certificate). (Note that, because of the structure of CMS, the signature on the CMS object will not be a valid X.509v3 signature and so cannot be used to construct a certificate from the precertificate).

- "SignedData.certificates" SHOULD be omitted.

4. Private Domain Name Labels

Some regard some DNS domain name labels within their registered domain space as private and security sensitive. Even though these domains are often only accessible within the domain owner's private network, it's common for them to be secured using publicly trusted TLS server certificates. We define a mechanism to allow these private labels to not appear in public logs.

4.1. Wildcard Certificates

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

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

4.2. Redacting Domain Name Labels in Precertificates

When creating a precertificate, the CA MAY substitute one or more labels in each DNS-ID with a corresponding number of "?" labels. Every label to the left of a "?" label MUST also be redacted. For example, if a certificate contains a DNS-ID of

"top.secret.example.com", then the corresponding precertificate could contain "?.?.example.com" instead, but not "top.?.example.com" instead.

Wildcard "*" labels MUST NOT be redacted. However, if the complete leftmost label of a DNS-ID is "*", it is considered redacted for the purposes of determining if the label to the right may be redacted. For example, if a certificate contains a DNS-ID of "*.top.secret.example.com", then the corresponding precertificate could contain "*.?.?.example.com" instead, but not "?.?.?.example.com" instead.

When a precertificate contains one or more redacted labels, a non-critical extension (OID 1.3.101.77, whose extnValue OCTET STRING contains an ASN.1 SEQUENCE OF INTEGERs) MUST be added to the corresponding certificate: the first INTEGER indicates the total number of "?" labels in the precertificate's first DNS-ID; the second INTEGER does the same for the precertificate's second DNS-ID; etc. There MUST NOT be more INTEGERs than there are DNS-IDs. If there are fewer INTEGERs than there are DNS-IDs, the shortfall is made up by implicitly repeating the last INTEGER. Each INTEGER MUST have a value of zero or more. The purpose of this extension is to enable TLS clients to reconstruct the TBSCertificate component of the precertificate from the certificate, as described in Section 9.2.2.

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

4.3. Using a Name-Constrained Intermediate CA

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

- there MUST be a non-critical extension (OID 1.3.101.76, 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.
- permittedSubtrees MUST specify one or more dNSNames.
- excludedSubtrees MUST specify the entire IPv4 and IPv6 address ranges.

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

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

5. Log Format and Operation

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

When it receives a valid submission, the log MUST return an SCT that corresponds to the submitted certificate or precertificate. If the log has previously seen this valid submission, it SHOULD return the same SCT as it returned before (to reduce the ability to track clients as described in Section 12.5). Note that if a certificate was previously logged as a precertificate, then the precertificate's SCT of type "precert_sct" would not be appropriate; instead, a fresh SCT of type "x509_sct" should be generated.

An SCT is the log's promise to incorporate the submitted entry in its Merkle Tree no later than a fixed amount of time, known as the Maximum Merge Delay (MMD), after the issuance of the SCT. Periodically, the log MUST append all its new entries to its Merkle Tree and sign the root of the tree.

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

5.1. Accepting Submissions

Logs MUST verify that each submitted certificate or precertificate has a valid signature chain to an accepted trust anchor, using the chain of intermediate CA certificates provided by the submitter. Logs MUST accept certificates and precertificates that are fully valid according to RFC 5280 [RFC5280] verification rules and are submitted with such a chain. Logs MAY accept certificates and precertificates that have expired, are not yet valid, have been revoked, or are otherwise not fully valid according to RFC 5280 verification rules in order to accommodate quirks of CA certificate-issuing software. However, logs MUST reject submissions without a valid signature chain to an accepted trust anchor. Logs MUST also reject precertificates that do not conform to the requirements in Section 3.2.

Logs SHOULD limit the length of chain they will accept. The maximum chain length is specified in the log's metadata.

The log SHALL allow retrieval of its list of accepted trust anchors (see Section 6.8), 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.

5.2. Log Entries

If a submission is accepted and an SCT issued, the accepting log MUST store the entire chain used for verification. This chain MUST include the certificate or precertificate itself, the zero or more intermediate CA certificates provided by the submitter, and the trust anchor used to verify the chain (even if it was omitted from the submission). The log MUST present this chain for auditing upon request (see Section 6.7). This chain is required to prevent a CA from avoiding blame by logging a partial or empty chain.

Each certificate entry in a log MUST include a "X509ChainEntry" structure, and each precertificate entry MUST include a "PrecertChainEntryV2" structure:

```
opaque ASN.1Cert<1..2^24-1>;

struct {
    ASN.1Cert leaf_certificate;
    ASN.1Cert certificate_chain<0..2^24-1>;
} X509ChainEntry;

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

struct {
    CMSPrecert pre_certificate;
    ASN.1Cert precertificate_chain<1..2^24-1>;
} PrecertChainEntryV2;
```

"leaf_certificate" is a submitted certificate that has been accepted by the log.

"certificate_chain" is a vector of 0 or more additional certificates required to verify "leaf_certificate". The first certificate MUST certify "leaf_certificate". Each following certificate MUST directly certify the one preceding it. The final certificate MUST be a trust anchor accepted by the log. If "leaf_certificate" is an accepted trust anchor, then this vector is empty.

"pre_certificate" is a submitted precertificate that has been accepted by the log.

"precertificate_chain" is a vector of 1 or more additional certificates required to verify "pre_certificate". The first certificate MUST certify "pre_certificate". Each following certificate MUST directly certify the one preceding it. The final certificate MUST be a trust anchor accepted by the log.

5.3. Log ID

Each log is uniquely identified by an OID. A log's operator MUST either allocate the OID themselves or request an OID from one of the two Log ID Registries (see Section 11.6.1 and Section 11.6.2). The OID is specified in the log's metadata. Various data structures include the DER encoding of this OID, excluding the ASN.1 tag and length bytes, in an opaque vector:

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

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

5.4. The TransItem Structure

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

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

enum {
    x509_entry(0), precert_entry(1), x509_sct(2), precert_sct(3),
    tree_head(4), signed_tree_head(5), consistency_proof(6),
    inclusion_proof(7), (65535)
} TransType;

struct {
    Version version;
    TransType type;
    select (type) {
        case x509_entry: TimestampedCertificateEntryDataV2;
        case precert_entry: TimestampedCertificateEntryDataV2;
        case x509_sct: SignedCertificateTimestampDataV2;
        case precert_sct: SignedCertificateTimestampDataV2;
        case tree_head: TreeHeadDataV2;
        case signed_tree_head: SignedTreeHeadDataV2;
        case consistency_proof: ConsistencyProofDataV2;
        case inclusion_proof: InclusionProofDataV2;
    } data;
} TransItem;
```

"version" is the earliest version of this protocol to which the encapsulated data structure conforms. This document is v2. Note that v1 [RFC6962] did not define "TransItem", but this document provides guidelines (see Appendix A) on how v2 implementations can co-exist with v1 implementations. Note also that, since each "TransItem" object is individually versioned, the version should be increased only if changes to it are made that are not backwards-compatible. The addition of encapsulated data structures can be done by adding "TransType" values without increasing the version.

"type" is the type of the encapsulated data structure. (Note that "TransType" combines the v1 type enumerations "LogEntryType",

"SignatureType" and "MerkleLeafType"). Future revisions of this protocol may add new "TransType" values.

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

5.5. Merkle Tree Leaves

The leaves of a log's Merkle Tree correspond to the log's entries (see Section 5.2). Each leaf is the leaf hash (Section 2.1) of a "TransItem" structure of type "x509_entry" or "precert_entry", which in this version (v2) encapsulates a "TimestampedCertificateEntryDataV2" structure. Note that leaf hashes are calculated as HASH(0x00 || TransItem), where the hashing algorithm is specified in the log's metadata.

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

struct {
    uint64 timestamp;
    opaque issuer_key_hash[HASH_SIZE];
    TBSCertificate tbs_certificate;
    SctExtension sct_extensions<0..2^16-1>;
} TimestampedCertificateEntryDataV2;
```

"timestamp" is the NTP Time [RFC5905] at which the certificate or precertificate was accepted by the log, measured in milliseconds since the epoch (January 1, 1970, 00:00), 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 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".

"tbs_certificate" is the DER encoded TBSCertificate from either the "leaf_certificate" (in the case of an "X509ChainEntry") or the "pre_certificate" (in the case of a "PrecertChainEntryV2"). (Note that a precertificate's TBSCertificate can be reconstructed from the corresponding certificate as described in Section 9.2.2).

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

5.6. Signed Certificate Timestamp (SCT)

An SCT is a "TransItem" structure of type "x509_sct" or "precert_sct", which in this version (v2) encapsulates a "SignedCertificateTimestampDataV2" structure:

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

struct {
    SctExtensionType sct_extension_type;
    opaque sct_extension_data<0..2^16-1>;
} SctExtension;

struct {
    LogID log_id;
    uint64 timestamp;
    SctExtension sct_extensions<0..2^16-1>;
    digitally-signed struct {
        TransItem timestamped_entry;
    } signature;
} SignedCertificateTimestampDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as described in Section 5.3.

"timestamp" is equal to the timestamp from the "TimestampedCertificateEntryDataV2" structure encapsulated in the "timestamped_entry".

"sct_extension_type" identifies a single extension from the IANA registry in Section 11.4. At the time of writing, no extensions are specified.

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

"sct_extensions" is a vector of 0 or more SCT extensions. This vector MUST NOT include more than one extension with the same "sct_extension_type". The extensions in the vector MUST be ordered by the value of the "sct_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 "x509_entry" or "precert_entry" (see Section 5.5) and MUST have an empty "item_extensions" vector.

5.7. Merkle Tree Head

The log stores information about its Merkle Tree in a "TransItem" structure of type "tree_head", which in this version (v2) encapsulates a "TreeHeadDataV2" structure:

```
opaque NodeHash[HASH_SIZE];

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

"timestamp" is the current NTP Time [RFC5905], measured in milliseconds since the epoch (January 1, 1970, 00:00), ignoring leap seconds.

"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" matches the STH extensions of the corresponding STH.

5.8. Signed Tree Head (STH)

Periodically each log SHOULD sign its current tree head information (see Section 5.7) to produce an STH. When a client requests a log's latest STH (see Section 6.3), the log MUST return an STH that is no older than the log's MMD. However, STHs could be used to mark individual clients (by producing a new one for each query), so logs MUST NOT produce them more frequently than is declared in their metadata. In general, there is no need to produce a new STH unless there are new entries in the log; however, in the unlikely event that it receives no new submissions during an MMD period, the log SHALL sign the same Merkle Tree Hash with a fresh timestamp.

An STH is a "TransItem" structure of type "signed_tree_head", which in this version (v2) encapsulates a "SignedTreeHeadDataV2" structure:

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

struct {
    SthExtensionType sth_extension_type;
    opaque sth_extension_data<0..2^16-1>;
} SthExtension;

struct {
    LogID log_id;
    uint64 timestamp;
    uint64 tree_size;
    NodeHash root_hash;
    SthExtension sth_extensions<0..2^16-1>;
    digitally-signed struct {
        TransItem merkle_tree_head;
    } signature;
} SignedTreeHeadDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as described in Section 5.3.

"timestamp" is equal to the timestamp from the "TreeHeadDataV2" structure encapsulated in "merkle_tree_head". This timestamp MUST be at least as recent as the most recent SCT timestamp in the tree. Each subsequent timestamp MUST be more recent than the timestamp of the previous update.

"tree_size" is equal to the tree size from the "TreeHeadDataV2" structure encapsulated in "merkle_tree_head".

"root_hash" is equal to the root hash from the "TreeHeadDataV2" structure encapsulated in "merkle_tree_head".

"sth_extension_type" identifies a single extension from the IANA registry in Section 11.5. At the time of writing, no extensions are specified.

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

"sth_extensions" is a vector of 0 or more STH extensions. This vector MUST NOT include more than one extension with the same "sth_extension_type". The extensions in the vector MUST be ordered by the value of the "sth_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.

"merkle_tree_head" is a "TransItem" structure that MUST be of type "tree_head" (see Section 5.7) and MUST have an empty "item_extensions" vector.

5.9. Merkle Consistency Proofs

To prepare a Merkle Consistency Proof for distribution to clients, the log produces a "TransItem" structure of type "consistency_proof", which in this version (v2) encapsulates a "ConsistencyProofDataV2" structure:

```
struct {
    LogID log_id;
    uint64 tree_size_1;
    uint64 tree_size_2;
    NodeHash consistency_path<1..2^8-1>;
} ConsistencyProofDataV2;
```

"log_id" is this log’s unique ID, encoded in an opaque vector as described in Section 5.3.

"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.

5.10. Merkle Inclusion Proofs

To prepare a Merkle Inclusion Proof for distribution to clients, the log produces a "TransItem" structure of type "inclusion_proof", which in this version (v2) encapsulates an "InclusionProofDataV2" structure:

```
struct {
    LogID log_id;
    uint64 tree_size;
    uint64 leaf_index;
    NodeHash inclusion_path<1..2^8-1>;
} InclusionProofDataV2;
```

"log_id" is this log's unique ID, encoded in an opaque vector as described in Section 5.3.

"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.

5.11. Shutting down a log

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

- Make it known to clients and monitors that the log will be frozen.
- Stop accepting new submissions (the error code "shutdown" should be returned for such requests).
- Once MMD from the last accepted submission has passed and all pending submissions are incorporated, issue a final STH and publish it as a part of the log's metadata. 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).

6. Log Client Messages

Messages are sent as HTTPS GET or POST requests. Parameters for POSTs and all responses are encoded as JavaScript Object Notation (JSON) objects [RFC4627]. Parameters for GETs are encoded as order-independent key/value URL parameters, using the "application/x-www-form-urlencoded" format described in the "HTML 4.01 Specification" [HTML401]. Binary data is base64 encoded [RFC4648] as specified in the individual messages.

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

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

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

For example, when a consistency proof between two STHs is requested, the front-end reached may not yet be aware of one or both STHs. In the case where it is unaware of both, it will return the latest STH it is aware of. Where it is aware of the first but not the second, it will return the latest STH it is aware of and a consistency proof from the first STH to the returned STH. The case where it knows the second but not the first should not arise (see the "no gaps" requirement above).

If the log is unable to process a client's request, it MUST return an HTTP response code of 4xx/5xx (see [RFC2616]), and, in place of the responses outlined in the subsections below, the body SHOULD be a JSON structure containing at least the following field:

error_message: A human-readable string describing the error which prevented the log from processing the request.

In the case of a malformed request, the string SHOULD provide sufficient detail for the error to be rectified.

error_code: An error code readable by the client. Some codes are generic and are detailed here. Others are detailed in the individual requests. Error codes are fixed text strings.

not compliant The request is not compliant with this RFC.

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

```
{
    "error_message": "'start' cannot be greater than 'end'",
    "error_code": "not compliant",
}
```

Clients SHOULD treat "500 Internal Server Error" and "503 Service Unavailable" responses as transient failures and MAY retry the same request without modification at a later date. Note that as per [RFC2616], in the case of a 503 response the log MAY include a "Retry-After:" header in order to request a minimum time for the client to wait before retrying the request.

6.1. Add Chain to Log

POST https://<log server>/ct/v2/add-chain

Inputs:

chain: An array of base64 encoded certificates. The first element is the certificate for which the submitter desires an SCT; the second certifies the first and so on to the last, which either is, or is certified by, an accepted trust anchor.

Outputs:

sct: A base64 encoded "TransItem" of type "x509_sct", signed by this log, that corresponds to the submitted certificate.

Error codes:

unknown anchor The last certificate in the chain both is not, and is not certified by, an accepted trust anchor.

bad chain The alleged chain is not actually a chain of certificates.

bad certificate One or more certificates in the chain are not valid (e.g. not properly encoded).

shutdown The log has ceased operation and is not accepting new submissions.

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

If a log detects bad encoding in a chain that otherwise verifies correctly then the log MAY still log the certificate but SHOULD NOT return an SCT. It should instead return the "bad certificate" error. Logging the certificate is useful, because monitors (Section 9.3) can then detect these encoding errors, which may be accepted by some TLS clients.

Note that not all certificate handling software is capable of detecting all encoding errors (e.g. some software will accept BER instead of DER encodings in certificates, or incorrect character encodings, even though these are technically incorrect) .

6.2. Add PreCertChain to Log

POST https://<log server>/ct/v2/add-pre-chain

Inputs:

precertificate: The base64 encoded precertificate.

chain: An array of base64 encoded CA certificates. The first element is the signer of the precertificate; the second certifies the first and so on to the last, which either is, or is certified by, an accepted trust anchor.

Outputs:

sct: A base64 encoded "TransItem" of type "precert_sct", signed by this log, that corresponds to the submitted precertificate.

Errors are the same as in Section 6.1.

6.3. Retrieve Latest Signed Tree Head

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

No inputs.

Outputs:

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

6.4. Retrieve Merkle Consistency Proof between Two Signed Tree Heads

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

Inputs:

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

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

Both tree sizes must be from existing 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", whose "tree_size_1" MUST match the "first" input. If the "sth" output is omitted, then "tree_size_2" MUST match the "second" input.

sth: A base64 encoded "TransItem" of type "signed_tree_head", 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:

first unknown "first" is before the latest known STH but is not from an existing STH.

second unknown "second" is before the latest known STH but is not from an existing STH.

See Section 9.4.2 for an outline of how to use the "consistency" output.

6.5. Retrieve Merkle Inclusion Proof from Log by Leaf Hash

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

Inputs:

hash: A base64 encoded 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 5.5. The "tree_size" must designate an existing v2 STH. Because of skew, the front-end may not know the requested STH. In that case, it will return the latest STH it knows, along with an inclusion proof to that STH. If the front-end knows the requested STH then only "inclusion" is returned.

Outputs:

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

sth: A base64 encoded "TransItem" of type "signed_tree_head", 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:

hash unknown "hash" is not the hash of a known leaf (may be caused by skew or by a known certificate not yet merged).

tree_size unknown "hash" is before the latest known STH but is not from an existing STH.

See Section 9.4.1 for an outline of how to use the "inclusion" output.

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

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

Inputs:

hash: A base64 encoded 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 5.5. The "tree_size" must designate an existing v2 STH.

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

latest STH < requested STH Return latest STH.

latest STH > requested STH Return latest STH and a consistency proof between it and the requested STH (see Section 6.4).

index of requested hash < latest STH Return "inclusion".

Note that more than one case can be true, in which case the returned data is their concatenation. 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" whose "inclusion_path" array of Merkle Tree nodes proves the inclusion of the chosen certificate in the selected STH.

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

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

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

Errors are the same as in Section 6.5.

See Section 9.4.1 for an outline of how to use the "inclusion" output, and see Section 9.4.2 for an outline of how to use the "consistency" output.

6.7. 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" or "precert_entry" (see Section 5.5).

log_entry: The base64 encoded log entry (see Section 5.2). In the case of an "x509_entry" entry, this is the whole "X509ChainEntry"; and in the case of a "precert_entry", this is the whole "PrecertChainEntryV2".

sct: A base64 encoded "TransItem" of type "x509_sct" or "precert_sct" corresponding to this log entry. Note that more than one SCT may have been returned for the same entry - only one of those is returned in this field. It may not be possible to retrieve others.

sth: A base64 encoded "TransItem" of type "signed_tree_head", 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 6.3.

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

Log servers MUST honor requests where 0 <= "start" < "tree_size" and "end" >= "tree_size" by returning a partial response covering only

the valid entries in the specified range. "end" >= "tree_size" could be caused by skew. Note that the following restriction may also apply:

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

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

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

See Section 9.4.3 for an outline of how to use a complete list of "leaf_input" entries to verify the "root_hash".

6.8. Retrieve Accepted Trust Anchors

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

No inputs.

Outputs:

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

max_chain: 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.

7. TLS Servers

TLS servers MUST use at least one of the three mechanisms listed below to present one or more SCTs or inclusion proofs from one or more logs to each TLS client during TLS handshakes, where each SCT or inclusion proof corresponds to the server certificate or to a name-constrained intermediate the server certificate chains to. Three mechanisms are provided because they have different tradeoffs.

- A TLS extension (Section 7.4.1.4 of [RFC5246]) with type "transparency_info" (see Section 7.2). This mechanism allows TLS servers to participate in CT without the cooperation of CAs,

unlike the other two mechanisms. It also allows SCTs and inclusion proofs to be updated on the fly.

- An Online Certificate Status Protocol (OCSP) [RFC6960] response extension (see Section 8.1.1), where the OCSP response is provided in the "CertificateStatus" message, provided that the TLS client included the "status_request" extension in the (extended) "ClientHello" (Section 8 of [RFC6066]). This mechanism, popularly known as OCSP stapling, is already widely (but not universally) implemented. It also allows SCTs and inclusion proofs to be updated on the fly.

- An X509v3 certificate extension (see Section 8.1.2). 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 where the SCTs and inclusion proofs originated won't necessarily be accepted by TLS clients for the full lifetime of the certificate, there is a risk that TLS clients will subsequently consider the certificate to be non-compliant and in need of re-issuance.

Additionally, a TLS server which supports presenting SCTs using an OCSP response MAY provide it when the TLS client included the "status_request_v2" extension ([RFC6961]) in the (extended) "ClientHello", but only in addition to at least one of the three mechanisms listed above.

7.1. Multiple SCTs or inclusion proofs

TLS servers SHOULD send SCTs or inclusion proofs from multiple logs in case one or more logs are not acceptable to the TLS client (for example, if a log has been struck off for misbehavior, has had a key compromise, or is not known to the TLS client). For example:

- If a CA and a log collude, it is possible to temporarily hide misissuance from clients. Including SCTs or inclusion proofs from different logs makes it more difficult to mount this attack.

- If a log misbehaves, a consequence may be that clients cease to trust it. Since the time an SCT or inclusion proof may be in use can be considerable (several years is common in current practice when embedded in a certificate), servers may wish to reduce the probability of their certificates being rejected as a result by including SCTs or inclusion proofs from different logs.

- TLS clients may have policies related to the above risks requiring servers to present multiple SCTs or inclusion proofs. 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.

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).

7.2. TLS Extension

Provided that a TLS client includes the "transparency_info" extension type in the ClientHello, the TLS server MAY include the "transparency_info" extension in the ServerHello with "extension_data" set to a "TransItemList". The TLS server SHOULD ignore any "extension_data" sent by the TLS client. Additionally, the TLS server MUST NOT process or include this extension when a TLS session is resumed, since session resumption uses the original session information.

8. Certification Authorities

8.1. 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 8.1.1) and certificates (see Section 8.1.2). 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".

8.1.1. OCSP Response Extension

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

8.1.2. Certificate Extension

A certification authority MAY include a Transparency Information X.509v3 extension in a certificate. Any included SCTs or inclusion proofs MUST be either for a precertificate that corresponds to this certificate, or for a name-constrained intermediate to which this certificate chains.

8.2. TLS Feature Extension

A certification authority MAY include the transparency_info (Section 7.2) TLS extension identifier in the TLS Feature [RFC7633] certificate extension in root, intermediate and end-entity certificates. When a certificate chain includes such a certificate, this indicates that CT compliance is required.

9. 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 metadata in order to communicate with logs and verify their responses. This metadata is described below, but note that this document does not describe how the metadata is obtained, which is implementation dependent (see, for example, [Chromium.Policy]).

Clients should somehow exchange STHs they see, or make them available for scrutiny, in order to ensure that they all have a consistent view. The exact mechanisms will be in separate documents, but it is expected there will be a variety.

9.1. Metadata

In order to communicate with and verify a log, clients need metadata about the log.

Base URL: The URL to substitute for <log server> in Section 6.

Hash Algorithm The hash algorithm used for the Merkle Tree (see Section 11.2).

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

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

Log ID The OID that uniquely identifies the log.

Maximum Merge Delay The MMD the log has committed to.

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

Maximum Chain Length The longest chain submission the log is willing to accept, if the log chose to limit it.

STH Frequency Count The maximum number of STHs the log may produce in any period equal to the "Maximum Merge Delay" (see Section 5.8).

Final STH If a log has been closed down (i.e. no longer accepts new entries), existing entries may still be valid. In this case, the client should know the final valid STH in the log to ensure no new entries can be added without detection.

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

9.2. TLS Client

9.2.1. Receiving SCTs or inclusion proofs

TLS clients receive SCTs or inclusion proofs alongside or in certificates. TLS clients MUST implement all of the three mechanisms by which TLS servers may present SCTs (see Section 7). TLS clients MAY also accept SCTs via the "status_request_v2" extension ([RFC6961]). TLS clients that support the "transparency_info" TLS extension SHOULD include it in ClientHello messages, with empty "extension_data".

9.2.2. Reconstructing the TBSCertificate

To reconstruct the TBSCertificate component of a precertificate from a certificate, TLS clients should:

- Remove the non-critical extension mentioned in Section 4.2
- Replace leftmost labels of each DNS-ID with "?", based on the INTEGER value corresponding to that DNS-ID in the extension.

A certificate with redacted labels where one of the redacted labels is "*" MUST NOT be considered compliant.

If the SCT checked is for a Precertificate (where the "type" of the "TransItem" is "precert_sct"), then the client SHOULD also remove embedded v1 SCTs, identified by OID 1.3.6.1.4.1.11129.2.4.2 (See Section 3.3. of [RFC6962]), in the process of reconstructing the TBSCertificate. That is to allow embedded v1 and v2 SCTs to co-exist in a certificate (See Appendix A).

9.2.3. Validating SCTs

In addition to normal validation of the server certificate and its chain, TLS clients SHOULD validate each received SCT for which they have the corresponding log’s metadata. To validate an SCT, a TLS client computes the signature input from the SCT data and the corresponding certificate, and then verifies the signature using the corresponding log’s public key. TLS clients MUST NOT consider valid any SCT whose timestamp is in the future.

Before considering any SCT to be invalid, the TLS client MUST attempt to validate it against the server certificate and against each of the zero or more suitable name-constrained intermediates (Section 4.3) in the chain. These certificates may be evaluated in the order they appear in the chain, or, indeed, in any order.

9.2.4. Validating inclusion proofs

TLS clients SHOULD also verify each received inclusion proof (see Section 9.4.1) for which they have the corresponding log's metadata, to audit the log and gain confidence that the certificate is logged.

Before considering any inclusion proof to be invalid, the TLS client MUST attempt to validate it against the server certificate and against each of the zero or more suitable name-constrained intermediates (Section 4.3) in the chain. These certificates may be evaluated in the order they appear in the chain, or, indeed, in any order.

After validating a received SCT, a TLS client MAY request a corresponding inclusion proof (if one is not already available) and then verify it. An inclusion proof can be requested directly from a log using "get-proof-by-hash" (Section 6.5) or "get-all-by-hash" (Section 6.6), but note that this will disclose to the log which TLS server the client has been communicating with.

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 6.3), then verify it by requesting a consistency proof (Section 6.4). Note that if the TLS client uses "get-all-by-hash", then it will already have the new STH.

9.2.5. Evaluating compliance

To be considered compliant, a certificate MUST be accompanied by at least one valid SCT or at least one valid inclusion proof. A certificate not accompanied by any valid SCTs or any valid inclusion proofs MUST NOT be considered compliant by TLS clients.

9.2.6. TLS Feature Extension

If any certificate in a chain includes the transparency_info (Section 7.2) TLS extension identifier in the TLS Feature [RFC7633] certificate extension, then CT compliance (using any of the mechanisms from Section 7) is required.

TLS clients MUST treat certificates which fail this requirement as non-compliant.

9.2.7. Handling of Non-compliance

If a TLS server presents a certificate chain that is non-compliant, there are two possibilities.

- In the case that use of TLS with a valid certificate is mandated by explicit security policy, application protocol specification, or other means, the TLS client MUST refuse the connection.

- If the use of TLS with a valid certificate is optional, the TLS client MAY accept the connection but MUST NOT treat the certificate as valid.

9.3. 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 needs to, at least, inspect every new entry in each log it watches. It may also want to keep copies of entire logs. In order to do this, it should follow these steps for each log:

1. Fetch the current STH (Section 6.3).

2. Verify the STH signature.

3. Fetch all the entries in the tree corresponding to the STH (Section 6.7).

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

5. Fetch the current STH (Section 6.3). Repeat until the STH changes.

6. Verify the STH signature.

7. Fetch all the new entries in the tree corresponding to the STH (Section 6.7). If they remain unavailable for an extended period, then this should be viewed as misbehavior on the part of the log.

8. Either:

    1. Verify that the updated list of all entries generates a tree with the same hash as the new STH.

    Or, if it is not keeping all log entries:

    1. Fetch a consistency proof for the new STH with the previous STH (Section 6.4).

2. Verify the consistency proof.

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

9. Go to Step 5.

9.4. 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.

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 9.3) 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 9.2) 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 6.5). It can also verify that the SCT corresponds to the certificate it arrived with (i.e. the log entry is that certificate, is a precertificate for that certificate or is an appropriate name-constrained intermediate [see Section 4.3]).

The following algorithm outlines may be useful for clients that wish to perform various audit operations.

9.4.1. Verifying an inclusion proof

When a client has received a "TransItem" of type "inclusion_proof" and wishes to verify inclusion of an input "hash" for an STH with a given "tree_size" and "root_hash", the following algorithm may be used to prove the "hash" was included in the "root_hash":

1. Compare "leaf_index" against "tree_size". If "leaf_index" is greater than or equal to "tree_size" 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 "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:

      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.

9.4.2. Verifying consistency between two STHs

When a client has an STH "first_hash" for tree size "first", an STH "second_hash" for tree size "second" where "0 < first < second", and has received a "TransItem" of type "consistency_proof" that they wish to use to verify both hashes, the following algorithm may be used:

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

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

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

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

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

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

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

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

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

Otherwise:

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

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

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

9.4.3. Verifying root hash given entries

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

1. Set "stack" to an empty stack.

2. For each "i" from "0" up to "tree_size - 1":

    1. Push "HASH(0x00 || entries[i])" to "stack".

    2. Set "merge_count" to the lowest value ("0" included) such that "LSB(i >> merge_count)" is not set. In other words, set "merge_count" to the number of consecutive "1"s found starting at the least significant bit of "i".

    3. Repeat "merge_count" times:

        1. Pop "right" from "stack".

        2. Pop "left" from "stack".

        3. Push "HASH(0x01 || left || right)" to "stack".

3. If there is more than one element in the "stack", repeat the same merge procedure (Step 2.3 above) until only a single element remains.

4. The remaining element in "stack" is the Merkle Tree hash for the given "tree_size" and should be compared by equality against the supplied "root_hash".

10. 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 should be frozen as specified in Section 9.1 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.

11. IANA Considerations

11.1. TLS Extension Type

IANA is asked to allocate an RFC 5246 ExtensionType value for the "transparency_info" TLS extension. IANA should update this extension type to point at this document.

11.2. Hash Algorithms

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

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

11.3. Signature Algorithms

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

Index	Signature Algorithm
0	deterministic ECDSA [RFC6979] using the NIST P-256 curve (Section D.1.2.3 of the Digital Signature Standard [DSS]) and HMAC-SHA256
1	RSA signatures (RSASSA-PKCS1-v1_5 with SHA-256, Section 8.2 of [RFC3447]) using a key of at least 2048 bits.

11.4. SCT Extensions

IANA is asked to establish a registry of SCT extensions, initially consisting of:

Type	Extension
65535	reserved

TBD: policy for adding to the registry

11.5. STH Extensions

IANA is asked to establish a registry of STH extensions, initially consisting of:

Type	Extension
65535	reserved

TBD: policy for adding to the registry

11.6. Object Identifiers

This document uses object identifiers (OIDs) to identify Log IDs (see Section 5.3), the precertificate CMS "eContentType" (see Section 3.2), and X.509v3 extensions in certificates (see Section 4.2, Section 4.3 and Section 8.1.2) and OCSP responses (see

Section 8.1.1). The OIDs are defined in an arc that was selected due to its short encoding.

11.6.1. Log ID Registry 1

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

IANA is requested to establish a registry that will allocate Log IDs from this range.

TBD: policy for adding to the registry. Perhaps "Expert Review"?

11.6.2. Log ID Registry 2

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

IANA is requested to establish a registry that will allocate Log IDs from this arc.

TBD: policy for adding to the registry. Perhaps "Expert Review"?

12. 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 misissue and take some action, such as asking a CA to revoke a misissued certificate, or that the log has misbehaved, which will be discovered when the SCT is audited. A signed timestamp is not a guarantee that the certificate is not misissued, since 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.

12.1. Misissued Certificates

Misissued certificates that have not been publicly logged, and thus do not have a valid SCT, are not considered compliant (so TLS clients may decide, for example, to reject them). Misissued certificates that do have an SCT from a log will appear in that public log within the Maximum Merge Delay, assuming the log is operating correctly. Thus, the maximum period of time during which a misissued certificate can be used without being available for audit is the MMD.

12.2. 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.

12.3. Avoiding Overly Redacting Domain Name Labels

Redaction of domain name labels carries the same risks as the use of wildcards (See Section 7.2 of [RFC6125], for example). If the entirety of the domain space below the unredacted part of a domain name is not controlled by a single entity (e.g. "?.com", "?.co.uk" and other public suffixes [Public.Suffix.List]), 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 9.2.2) contains any overly redacted domain names.

12.4. 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 or by presenting different, conflicting views of the Merkle Tree at different times and/or to different parties. Such misbehavior is detectable and the [I-D.ietf-trans-threat-analysis] provides more details on how this can be done.

Violation of the MMD contract is detected by log clients requesting a Merkle inclusion proof (Section 6.5) for each observed SCT. These checks can be asynchronous and need only be done once per each certificate. In order to protect the clients' privacy, these checks need not reveal the exact certificate to the log. Clients can instead request the proof from a trusted auditor (since anyone can

compute the proofs from the log) or request Merkle inclusion proofs for a batch of certificates around the SCT timestamp.

Violation of the append-only property can be detected by clients comparing their instances of the Signed Tree Heads. As soon as two conflicting Signed Tree Heads for the same log are detected, this is cryptographic proof of that log's misbehavior. There are various ways this could be done, for example via gossip (see [I-D.ietf-trans-gossip]) or peer-to-peer communications or by sending STHs to monitors (who could then directly check against their own copy of the relevant log).

12.5. Deterministic Signatures

Logs are required to use deterministic signatures for the following reasons:

- Using non-deterministic ECDSA with a predictable source of randomness means that each signature can potentially expose the secret material of the signing key.

- Clients that gossip STHs or report back SCTs can be tracked or traced if a log was to produce multiple STHs or SCTs with the same timestamp and data but different signatures.

12.6. Multiple SCTs or inclusion proofs

By offering multiple SCTs or inclusion proofs, each from a different log, TLS servers reduce the effectiveness of an attack where a CA and a log collude (see Section 7.1).

12.7. Threat Analysis

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

13. Acknowledgements

The authors would like to thank Erwann Abelea, Robin Alden, Al Cutter, 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, 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.

14. References

14.1. Normative References

[DSS] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS 186-3, June 2009, <http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf>.

[FIPS.180-4] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-4, March 2012, <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.

[HTML401] Raggett, D., Le Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999, <http://www.w3.org/TR/1999/REC-html401-19991224>.

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

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

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

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

[RFC5905] Mills, D., Martin, J., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.

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

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, March 2011.

[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, <http://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, <http://www.rfc-editor.org/info/rfc6961>.

[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, <http://www.rfc-editor.org/info/rfc6979>.

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

14.2. Informative References

[Chromium.Log.Policy]
The Chromium Projects, "Chromium Certificate Transparency Log Policy", 2014, <http://www.chromium.org/Home/chromium-security/certificate-transparency/log-policy>.

[Chromium.Policy]
The Chromium Projects, "Chromium Certificate Transparency", 2014, <http://www.chromium.org/Home/chromium-security/certificate-transparency>.

[CrosbyWallach]
Crosby, S. and D. Wallach, "Efficient Data Structures for Tamper-Evident Logging", Proceedings of the 18th USENIX Security Symposium, Montreal, August 2009, <http://static.usenix.org/event/sec09/tech/full_papers/crosby.pdf>.

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

[I-D.ietf-trans-gossip]
Nordberg, L., Gillmor, D., and T. Ritter, "Gossiping in CT", draft-ietf-trans-gossip-01 (work in progress), October 2015.

[I-D.ietf-trans-threat-analysis]
Kent, S., "Attack Model and Threat for Certificate Transparency", draft-ietf-trans-threat-analysis-03 (work in progress), October 2015.

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

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

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

Appendix A. Supporting v1 and v2 simultaneously

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

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

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

- Create a CMS precertificate as described in Section 3.2 and submit it to v2 logs.
- Embed the obtained v2 SCTs in the TBSCertificate, as described in Section 8.1.2.
- Use that TBSCertificate to create a v1 precertificate, as described in Section 3.1. of [RFC6962] and submit it to v1 logs.
- Embed the v1 SCTs in the TBSCertificate, as described in Section 3.3. of [RFC6962].
- Sign that TBSCertificate (which now contains v1 and v2 SCTs) to issue the final X.509 certificate.

Authors' Addresses

Ben Laurie
Google UK Ltd.

EMail: benl@google.com

Adam Langley
Google Inc.

EMail: agl@google.com

Emilia Kasper
Google Switzerland GmbH

EMail: ekasper@google.com

Eran Messeri
Google UK Ltd.

EMail: eranm@google.com

Rob Stradling
Comodo CA, Ltd.

EMail: rob.stradling@comodo.com

Public Notary Transparency
Internet Draft
Expires: July 2016
Intended Status: Informational

S. Kent
BBN Technologies

January 30, 2016

Attack Model and Threat for Certificate Transparency
draft-ietf-trans-threat-analysis-04

Abstract

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

Table of Contents

1. Introduction

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

A certificate is characterized as syntactically mis-issued (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 typically are application-specific. For example, certificates used in the Web PKI environment might be characterized as domain validation (DV) or extended validation (EV) certificates. Certificates used with applications such as IPsec or S/MIME have different syntactic constraints from those in the Web PKI context.

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

A certificate Subject benefits from CT because CT helps detect certificates that have been mis-issued in the name of that Subject. A Subject learns of a bogus certificate (issued in its name), via the Monitor function of CT. The Monitor function may be provided by the Subject itself, i.e., self-monitoring, or by a third party trusted by the Subject. When a Subject is informed of certificate mis-issuance by a Monitor, the Subject is expected to request/demand revocation of the bogus certificate. Revocation of a bogus certificate is the primary means of remedying mis-issuance.

Certificate Revocations Lists (CRLs) [RFC5280] are the primary means of certificate revocation established by IETF (and ISO) standards. Unfortunately, most browsers do not make use of CRLs to check the revocation status of certificates presented by a TLS Server (Subject). Some browsers make use of Online Certificate Status Protocol(OCSP) data [RFC4366] as a standards-based alternative to CRLs. Also, most browser vendors employ proprietary means of revoking certificates, e.g., via a "blacklist" or a bad-CA-list. Such capabilities enable a browser vendor to cause browsers to reject any certificates on the blacklist or for which the issuing CA is on the bad-CA-list. This approach also remedies mis-issuance. Throughout the remainder of this document references to certificate revocation as a remedy encompass this and analogous forms of browser behavior, if available. Note: there are no IETF standards defining a browser blacklist capability.)

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

A relying party (e.g., browser) benefits from CT if it rejects a bogus certificate, i.e., treats it as invalid. An RP is protected from accepting a bogus certificate if that certificate is revoked, and if the RP checks the revocation status of the certificate. (An RP is also protected if a browser vendor "blacklists" a certificate or "bad-CA-lists" a CA as noted above.) An RP also may benefit from CT if the RP validates an SCT associated with a certificate, and rejects the certificate if the Signed certificate Timestamp (SCT) [TRANS] is invalid. If an RP verified that a certificate that claims to have been logged has a valid log entry, the RP would have a higher degree of confidence that the certificate is genuine. However, checking logs in this fashion imposes a burden on RPs and on logs. Moreover, the existence of a log entry does not ensure that the certificate is not mis-issued. Unless the certificate Subject is monitoring the log(s) in question, a bogus certificate will not be detected by CT mechanisms. Finally, if an RP were to check logs for individual certificates, that would disclose to logs the identity of web sites being visited by the RP, a privacy violation. Thus this attack model assumes that RPs will not check log entries.

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

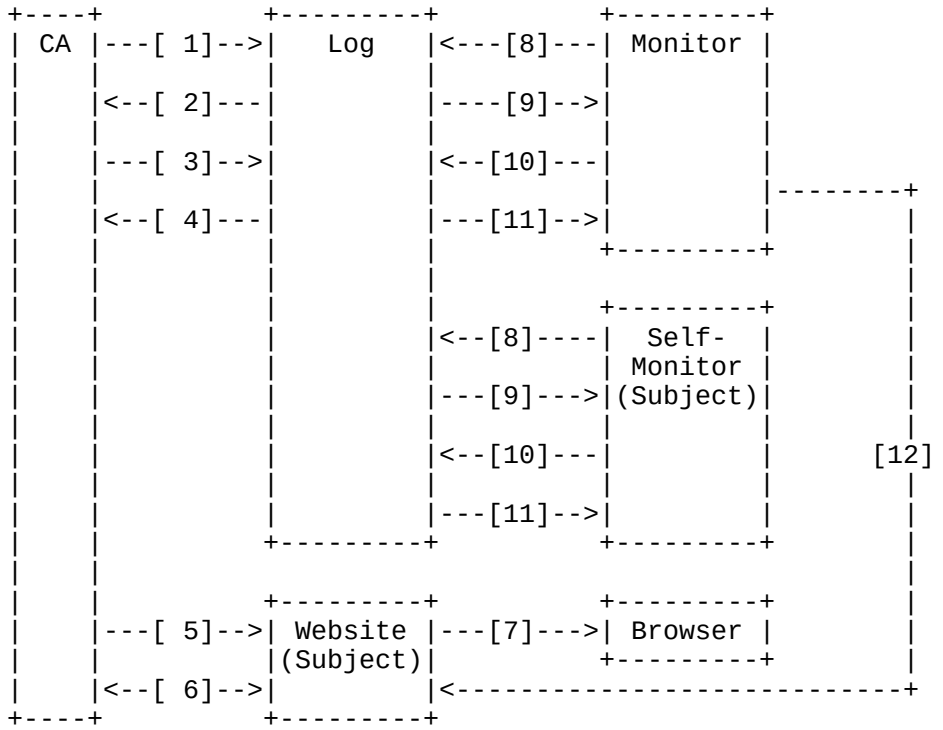
Also note that one proposal for distributing Audit information (to detect misbehaving logs) calls for a browser to send SCTs it receives to the corresponding website when visited by the browser. If a website acquires an inclusion proof from a log for each (unique) SCT it receives in this fashion, this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

Logging [TRANS] is the central element of CT. Logging enables a Monitor to detect a bogus certificate based on reference information provided by the certificate Subject. Logging of certificates is intended to deter mis-issuance, by creating a publicly-accessible record that associates a CA with any certificates that it mis-issues. Logging does not remedy mis-issuance; but it does facilitate remediation by providing the information needed to enable detection and consequently 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 [TRANS] 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. There is no agreed-upon Audit function design for CT at the time of this writing. As a result, the model merely notes where Auditing is needed to detect certain classes of attacks.

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



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

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

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

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

```
Web PKI CA - malicious vs non-malicious
   Certificate - logged vs not logged
         Log - benign vs malicious
         Third party Monitor - benign vs malicious
         Certificate's Subject - self-monitoring (or not)
         Browser - 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 Web PKI context, although most of the analysis is applicable to other PKI contexts.

Conventions used in this document

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

2. Threats

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 if the assumptions about adversaries prove to be incorrect, documenting the assumptions is valuable.

As noted above, the goals of CT are to deter, detect, and facilitate remediation of attacks on the web PKI. Such attacks can enable an attacker to spoof the identity of TLS-enabled web sites. Spoofing enables an adversary to acquire information that a TLS-enabled 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, 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 forged web site certificates, an adversary must be able to direct a TLS client to a spoofed web site, so that it can present the forged certificate during a TLS handshake. An adversary may achieve this in various ways, e.g., by causing a user to click on a link to the website, or by manipulation of the DNS response sent to a TLS client. The former type of attack is well within the perceived capabilities of both classes of adversary. The latter attack may be possible for criminals and is certainly a capability available to a nation state within its borders. Nation states also

may be able to compromise DNS servers outside their own jurisdiction.

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. (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). It seems unlikely that a compromised, non-malicious, log would persist in presenting multiple views of its data, but a malicious log would.

Finally, note that a browser trust store may include a CA that is intended to issue certificates to enable monitoring of encrypted browser sessions. The inclusion of a trust anchor for such a CA is intended to facilitate monitoring encrypted content, via an authorized man-in-the-middle (MITM) attack. CT is not designed to detect and counter this type of locally-authorized interception.

3. Semantic mis-issuance

3.1. Non-malicious Web PKI CA context

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

A CA may have mis-issued a certificate as a result of an error or, in the case of a bogus certificate, because it was the victim of a social engineering attack or an attack such as the one that affected DigiNotar [https://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_any security_incident.aspx]. In the case of an error, the CA should have a record of the erroneous certificate and be prepared to revoke this certificate once it has

discovered and confirmed the error. In the event of an attack, a CA may have no record of a bogus certificate.

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 probably 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"). Unless a Monitor validates the associated certificate chains up to roots that it trusts, these fake bogus certificates could cause the Monitors to report non-existent semantic problems to the Subject who would in turn report them to

the purported issuing CA. This might cause the CA to do needless investigative work or perhaps incorrectly revoke and re-issue the Subject's real certificate. Note that for every certificate submitted to a log, the log MUST verify a complete certificate chain up to one of the roots it accepts. So creating a log entry for a fake bogus certificate marks the log as misbehaving.

3.1.1.2.1. Self-monitoring Subject

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 trusted 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.

3.1.1.2.2. Benign third party Monitor

Because a misbehaving log will suppress a bogus certificate log entry (or report such entries inconsistently) a benign third party Monitor that is protecting the targeted Subject also will not detect a bogus certificate. In this scenario, CT relies on a distributed Auditing mechanism [gossip] to detect log misbehavior, as a deterrent. (See Section 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 information about the log entries and STH between the entities receiving the view that excludes the bogus certificate and entities that receive a view that includes it, i.e., a distributed Audit mechanism.

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

3.1.2.1. 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.

3.1.2.2. CT-enabled browser

If a browser rejects certificates without SCTs (see Section 5.4), CAs may be "encouraged" to log the certificates they issue. This, in

turn, would make it easier for Monitors to detect bogus certificates. However, the CT architecture does not describe how such behavior by browsers can be deployed incrementally throughout the Internet. As a result, this attack model does not assume that browsers will reject a certificate that is not accompanied by an SCT. In the CT architecture certificates have to be logged to enable Monitors to detect mis-issuance, and to trigger subsequent revocation [CTarch]. Thus the effectiveness of CT is diminished in this context.

3.2. Malicious Web PKI CA context

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

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 will 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 does not help remedy the problem. (See Sections 4 and 6.)

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.

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 may or may not 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, so that they no longer acquire SCTs from that log. The Monitor also avoids relying upon such a log in the future. However, unless a distributed Audit mechanism proves effective in detecting such misbehavior, CT cannot be relied upon to detect this form of mis-issuance. (See Section 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.

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 describe how such behavior by browsers can be deployed incrementally throughout the Internet. As a result, this attack model does not assume that browsers will reject a certificate that is not accompanied by an SCT. Since certificates have to be logged to enable detection of mis-issuance by Monitors, and to trigger subsequent revocation, the effectiveness of CT is diminished in this context.

4. Syntactic mis-issuance

4.1. Non-malicious Web PKI CA context

This section analyzes the scenario in which the CA has no intent to issue a syntactically incorrect certificate. As noted in Section 1, we refer to a syntactically incorrect certificate as erroneous.

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 can be noted in an SCT extension returned to the submitter.

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

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

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

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

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.

4.1.1.3. 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.

4.1.1.4. 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 perform thorough syntactic checks on certificates, and so it seems unlikely that browsers will be a reliable way to detect erroneous certificates. Moreover, a protocol used by a browser to notify a Subject and/or CA of an erroneous certificate represents a DoS potential, and thus may not be appropriate. 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.

4.1.2. Certificate not logged

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

4.2. Malicious Web PKI CA context

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

4.2.1. Certificate logged

4.2.1.1. Benign log

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

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

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

3. The CA may check to see which CCIDs a log declares it can check, and chose a registered CCID that is not checked by the log in question.

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

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. Self-monitoring Subject and Benign third party Monitor

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

4.2.1.4. CT-enabled browser

As noted above (4.1.1.4), many browsers fail to perform thorough syntax checks on certificates. Such browsers would benefit from having such checks performed by a log and reported in the SCT. (Remember, in this scenario, the log is benign.) However, if a browser does not discriminate against certificates that do not contain SCTs (or that are not accompanied by an SCT in the TLS

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

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, in the form of a log entry and/or SCT, will need to examine its records to determine if it has knowledge of the certificate in question. It also will likely require the targeted Subject to provide assurances that it is the authorized entity representing the Subject name (subjectAltname) in question. Thus a Subject should not expect immediate revocation of a contested certificate. The time frame in which a CA will respond to a revocation request usually is described in the CPS for the CA. Other certificate fields and extensions may be of interest for forensic purposes, but are not required to effect revocation nor to verify that the certificate to be revoked is bogus or erroneous, based on applicable criteria. The SCT and log entry, because each contains a timestamp from a third party, is probably valuable for forensic purposes (assuming a non-conspiring log operator).

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. 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]. Unless a mechanism is defined that accommodates incremental deployment of this capability, attackers probably will avoid submitting bogus certificates to (benign) logs as a means of evading detection.

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. Such communication has not been specified, i.e., there are no standard ways to configure a browser to reject individual bogus or erroneous certificates based on information provided by an external entity such as a Monitor. Moreover, the same or another malicious CA could issue new bogus or erroneous certificates for the targeted Subject, which would have to be detected and rejected in this (as yet unspecified) fashion. Thus, for now, CT does not seem to provide a way to remedy this form of attack, even though it provides a basis for detecting such attacks.

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 is not visible to Monitors. Only if Monitors and browsers reject certificates that contain SCTs from conspiring logs (based on information from an auditor) will CT be able to detect and deter use of such logs. Thus the means by which a Monitor performing an audit function detects such logs, and informs browsers must be specified for CT to be effective in the context of misbehaving logs.

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

Monitors play a critical role in detecting semantic certificate mis-issuance, for Subjects that have requested monitoring of their certificates. A monitor (including a Subject performing self-monitoring) examines logs for certificates associated with one or more Subjects that are being "protected". A third-party Monitor must

obtain a list of valid certificates for the Subject being monitored, in a secure manner, to use as a reference. It also must be able to identify and track a potentially large number of logs on behalf of its Subjects. This may be a daunting task for Subjects that elect to perform self-monitoring.

Note: A Monitor must not rely on a CA or RA database for its reference information or use certificate discovery protocols; this information must be acquired by the Monitor based on reference certificates provided by a Subject. If a Monitor were to rely on a CA or RA database (for the CA that issued a targeted certificate), the Monitor would not detect mis-issuance due to malfeasance on the part of that CA or the RA, or due to compromise of the CA or the RA. If a CA or RA database is used, it would support detection of mis-issuance by an unauthorized CA. A Monitor must not rely on certificate discovery mechanisms to build the list of valid certificates since such mechanisms might result in bogus or erroneous certificates being added to the list.

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

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.

9. References

9.1. Normative References

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

[CTarch] Kent, S., Mandelberg, D., Seo, K., "Certificate Transparency (CT) System Architecture", draft-kent-trans-architecture-00, (October 2015), work in progress.

9.2. Informative References

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

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

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

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

[RFC5280] Cooper, D., Santession, S., Farrell, S., Boeyen, S., Housley, R., Polk, W., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280, May 2008.

Author’s Addresses

Stephen Kent
BBN Technologies
10 Moulton Street
Cambridge MA 02138
USA

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

Copyright Statement

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: January 6, 2016

D. Zhang

D. Gillmor
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

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 January 6, 2016.

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

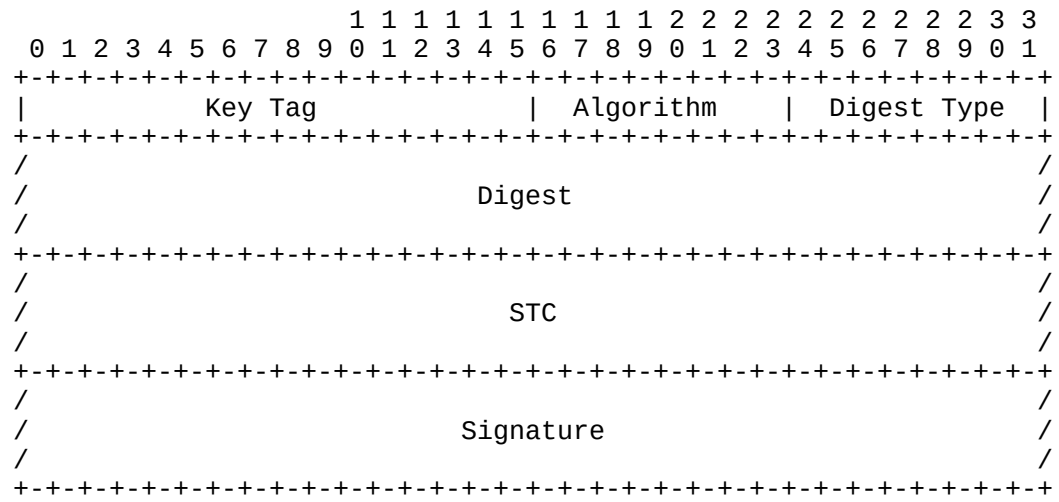
The SCT associated with a DS RR is stored within a STC RR. A DNS server MAY provide multiple SCT RRs for one DS RR.

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.



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 submited 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