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

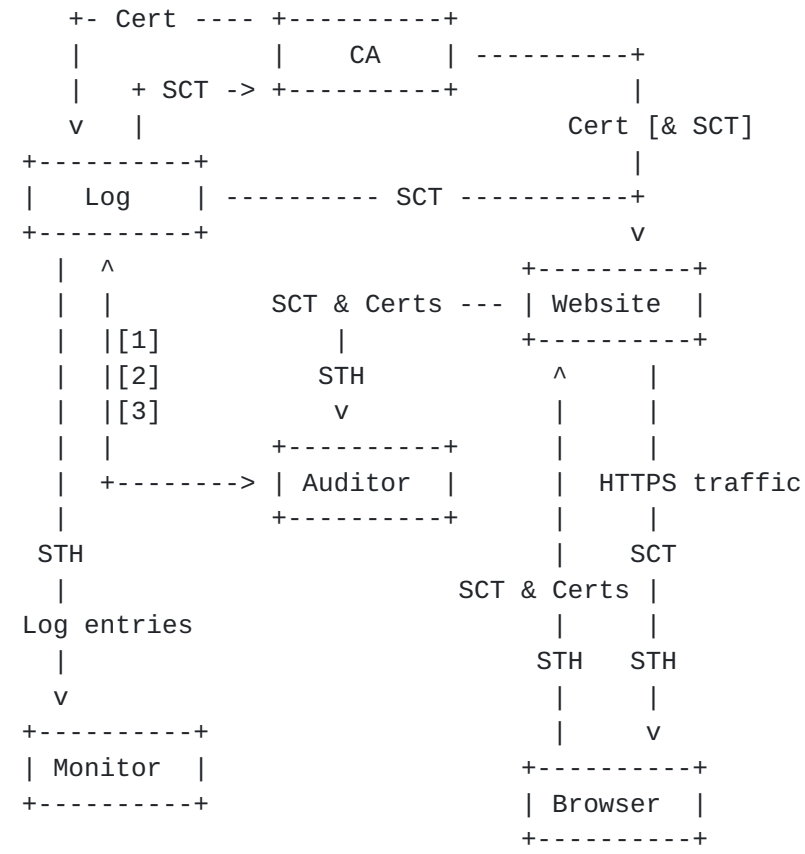
There are three separate gossip streams:

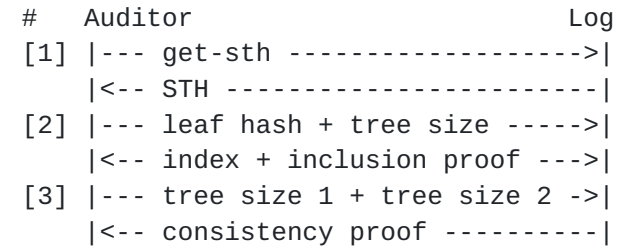
- SCT Feedback - transporting SCTs and certificate chains from HTTPS clients to CT auditors via HTTPS servers.
- STH Pollination - HTTPS clients and CT auditors using HTTPS servers as STH pools for exchanging STHs.
- Trusted Auditor Stream - HTTPS clients communicating directly with trusted CT auditors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