Certificate Transparency in Chrome: Monitoring CT logs consistency

PUBLIC

Created: 2015-05-01
Last updated: 2017-03-02
Status: **IMPLEMENTED**
Authors: rsleevi, eranm

Overview

Certificate Transparency (CT) is an open effort to publicly log certificates issued by CAs, allowing third parties to audit and notice misissuance/compromise. Basic support for CT already exists in Chrome (in the form of verifying Signed Certificate Timestamps).

The next phase is auditing CT logs by checking for certificate inclusion. Chrome clients will be provided with fresh, verified Signed Tree Heads to check inclusion against and will fetch inclusion proofs over a DNS-based protocol. As the implication of failure to check inclusion is not yet clear, this document focuses on the components necessary to experiment with inclusion checking.

Terminology

Log: Public repository of certificates, conforming to RFC 6962 (CT).

Signed Certificate Timestamp (SCT): A log's cryptographically-signed promise to include (and make publicly available) a given certificate, after a given time.

Signed Tree Head (STH): A cryptographically-signed checksum of the entire Merkle tree a log represents, which also includes the tree size.

Consistency Proof: A cryptographic way to verify that a Merkle tree is a newer version of another (older) Merkle tree (details).

Inclusion Proof: A cryptographic way to verify that a given certificate + SCT combination is included in a Merkle tree published by a log.

Maximum Merge Delay (MMD): The longest time period allowed for a CT log between the issuance of an SCT and the creation of a log entry for that certificate + SCT combination.

Objective

There are two goals for this phase:

- Facilitating checking of certificate inclusion proofs (based on observed SCTs).
- Estimate the effectiveness of log auditing (by measuring freshness of STHs and success rate of inclusion checking observed SCTs).

The results will mostly be invisible to the end-user; Chrome will report back the metrics on freshness of data to Google and metrics on cases where inclusion checks would fail. Gathered data will assist in choosing the path for dealing with SCTs for which inclusion cannot be proven.

To start with, the aim is to get answers to the following questions:

- How likely is it for a client to observe SCTs that are newer than the STH it knows about, so they have to be queued for inclusion checking?
- How often do inclusion checks succeed?

Later on, measuring the duration SCTs would be pending inclusion checking would help us estimate potential load on CT logs when SCTs for new certificates trigger inclusion checking.

Design Highlights

The intended behaviour is for Chrome clients to receive a collection of fresh STHs via Chrome's component updater, which will then be used for inclusion checking of SCTs. See the **Security Considerations** section as to why consistency proofs are not necessary.

We do not intend clients to actively fetch STHs by themselves from logs or Google-operated mirrors at this phase.

A client that observes a new SCT will check for inclusion of the associated log entry in the CT log, if it has an STH that covers this SCT (that is, an STH that was issued after the MMD has passed from when the SCT was issued). If not, the SCT will be queued for inclusion checking until the next STH is provided.

Since any communication over HTTPS could result in more SCTs to validate (for example, https://ct.googleapis.com/ provides SCTs in the TLS handshake), care should be taken to make SCT inclusion proof checking asynchronous and independent.

Profile independence:

To avoid leaking state (SCTs and the certificates they relate to) between profiles, SCTs will be audited independently in each profile (coupled to CTVerifier instances). That implies that the same SCT, if observed on connections in different profiles, will be audited multiple times.

Memory consumption and persistence considerations:

- The queue of SCTs pending inclusion checked will be limited to 1,000 entries (per CT log), taking up to ~32K of memory: The hash of the LogEntry to be audited will be calculated as the SCT is observed, saving the need to keep the SCT and certificate in the queue (that will change when the auditing result will matter not just for collecting data). Additional new SCTs will be ignored.
- For SCTs which have been successfully checked for inclusion, an array of leaf hashes will be kept in-memory to avoid re-checking inclusion status. The size of this array will be limited to 1,000 entries, taking up to 32K (as each leaf hash is 32 bytes).
- Upon notification of memory pressure, all queues/caches will be emptied.

QUESTION: Should we cache failures? Not doing so means failures will be re-tried, doing so means failures will be persisted until browser restart.

Bandwidth and resource consumption:

As many SCTs may be observed in a short time period and as Chrome can establish multiple sockets, we'd like to avoid a large number of DNS queries being sent at once. To reduce this load, we'll maintain a queue of inclusion proofs to fetch, so that we're never fetching more than N concurrently. We'll start with N=1 initially, to minimize the chance that concurrent fetching contributes to proof fetching failure rates.

Furthermore, the same SCT may be observed multiple times before its inclusion check is completed, which may lead to multiple proof requests being sent. To reduce this traffic, the queue of SCTs pending inclusion will be de-duplicated to avoid having the same SCT in the queue multiple times. This means metrics will be measured for the SCT, which gives an estimate about the reliability of fetching in general, because we don't do a unique verification for every connection, it means that more data would be needed to estimate the impact of

making inclusion proof fetching critical. That is, if two connections share the same SCT and are coalesced into one inclusion proof fetch, and that fetch fails, two connections could be impacted, but we'll only record that a single inclusion proof check failed. For judging the inclusion proof checking mechanism, and since inclusion checking is asynchronous and cannot be attributed to a particular connection, this is acceptable.

Design Details

The functionality required to implement this behaviour can be split into several modules:

- STH Component Updater (**already implemented**)
  - Implements the ComponentInstallerTraits interface, receives new STHs (that are provided once a day) and parses them.
- New SCT listener (**already implemented**)
  - The recommended pattern is an interface under net/cert/ and an implementation under components. This is to break any circular dependencies between the //net stack, as SCTs are handled by certificate verification, but getting SCT status may require making additional requests (as URLRequests sit above certificate verification)
- Tree State Tracker (**under review**)
  - Active component that drives fetching and verifying inclusion proofs.
  - Receives notifications of new STHs received via the component updater.
  - Will implement the SCT Listener interface to be notified of new SCTs.
  - Checks for inclusion if possible, queues SCTs for inclusion checking if the STH containing them is not available yet. **In this phase** will store a limited number of SCTs and their inclusion status only in memory.
- Log DNS Client (**already implemented**)
  - Uses a DnsClient to asynchronously fetch inclusion proofs from CT logs over DNS.
  - Sends queries to a specially-designed DNS resolver, conforming to the CT over DNS draft RFC. All of the logs trusted by Chrome have such a resolver, provided by Google. The disqualified logs currently do not have.
  - Each trusted log will have a domain name that can be used to perform CT DNS queries. This domain name will be added to the CTLogInfo stored in Chrome for each log.
  - Rate-limits the number of concurrent DNS requests made.
- Inclusion checking
  - The CTLogVerifier will be extended to be able to verify a MerkleAuditProof.
- Tree state persistence: Stores SCT verification status to disk, to minimize number of repeated requests clients make (optional at this phase).

Wiring to existing code

Two new interfaces will be added (under net/cert): STHObserver, STHReporter. STHObserver for receiving call back of a new STH being observed, STHReporter for registering/unregistering STHObservers. A new class for delegating notifications, STHDistributor, will implement both.

The STHSetComponentInstaller will be registered together with all other components in chrome_browser_main.cc. It will have a pointer to an STHDistributor which will be shared with the IOThread so that TreeStateTracker instances could register for notification of new STHs (rationale).

There will be a TreeStateTracker instance per-profile, created in IOThread::Init and ProfileIOData::Init, where it will be registered to receive notifications of new SCTs (as it will implement the CTVerifier::Observer interface) from the CTVerifier that exists in each profile/IO Thread.

As the CTVerifier will outlive the TreeStateTracker (as it's used by the URLRequestContext), the link between them will be severed by resetting the observer on the CTVerifier to null in IOThread::CleanUp / ProfileIOData::DestroyResourceContext, such that the

SCTInclusionChecker will not receive notifications of new SCTs during profile tear-down.

New component - components/certificate_transparency/

A new component is needed for several reasons:

- There is no reason to put this code under net/, as the results of providing fresh STHs and ultimately inclusion checking will not be used by consumers of net/ code (unlike SCT validation, whose results are exposed via SSLInfo).
- There is a dependency on net/ code to provide observed SCTs and usage of the net/ DNS code to fetch inclusion proofs.
- Inclusion checking will not be done initially on Android, to conserve network traffic and battery life, until we have data on resource consumption of the feature.

The following modules will be added in the new component:

- TreeStateTracker: A delegate class that will implement STHObserver and CTVerifier::Observer for receiving new STHs (via the component updater) and observed SCTs and will delegate them to the per-log SingleTreeTracker instance.
- SingleTreeTracker: Receives notification of fresh STHs from the TreeStateTracker checks their signature. Also receives notifications of observed SCTs for the log it's tracking and check their inclusion state or queue them for inclusion check when a fresh STH becomes available. Once a fresh STH is available, checks inclusion of pending SCTs. Holds references to CTLogVerifier instances for STH validation. May also hold a NetLog instance to report individual events (STH verification, etc).
- LogDnsClient: Will provide an asynchronous interface for obtaining an entry's leaf index and an entry's inclusion proof, using net::DnsClient.

There will be one instance of the TreeStateTracker on the IOThread which will be used by all profiles. It will notify SingleTreeTracker instances of new STHs. The reasons for this approach can be found in a net-dev thread.

Log DNS client

Each CT log recognized by Chrome will have a DNS end-point associated with it (even logs that ceased operation), to detect backdating of SCTs.

The Log DNS client will rate-limit by limiting the number of concurrent DNS requests and rejecting (synchronously) requests when its queue is full.

There will be a single LogDNSClient per profile, which will be owned by the TreeStateTracker. The callbacks provided to the LogDNSClient will have weak pointers to SingleTreeTracker instances, which will all share the LogDNSClient instance and will be cleaned up by the TreeStateTracker before the LogDNSClient is.

UI modifications

None. Reports of successful/failed STH validation and inclusion checks will be logged through the NetLog.

Metrics reporting

Clients will report, via UMA:

- The age of the STHs currently held (as buckets - less than 24h, 24h-48h, 48h-96h, older than 96h).
- Failure to verify STH signature.
- How often can SCTs be checked for inclusion immediately (if needed, can be refined to log the number of SCTs pending verification, i.e. for which a newer STH than the currently-held one is available).
- How often inclusion proof checks are successful.

Scope

Apart from asynchronous notification of observed SCTs (to SCTInclusionChecker instances acting as listeners) there are no changes to any flow of networking components.

Risk

The risk of negatively impacting user experience is minimal as STH validation and inclusion proof fetching happens independently of any other flow, particularly certificate validation.

A different risk is the amount of traffic generated by Chrome clients who fetch inclusion proofs - this will be handled by gradual deployment of the feature using Finch (see Release Plan below) and measurement of the number of times clients fetch inclusion proofs. Clients will also keep a local cache of log entries and their inclusion status to limit the number of requests made.

Privacy implications

All clients will receive the same STHs from Google and will report back metrics that are not related to specific certificates / domains.

Clients will ask for inclusion proofs over DNS, which is meant to be proxied via the clients' DNS servers.

Analysis of privacy implications for using a DNS-based protocol are documented here.

Security considerations

Two scenarios have to be addressed to protect clients against log misbehaviour:

1. Ensuring all clients have the same view of the tree published by a log.
2. Verifying that for each SCT issued, the appropriate entry is in the log (i.e. the certificate + timestamp).

Generally, to ensure consistent view, clients have to check consistency between STHs and should share the STHs observed with other clients. However, Chrome clients do not need to do either as all clients will receive the same STHs from Google via the component updater (the updates are signed).

The threat of Google's STH distribution mechanism being compromised (by an attacker who was also able to compromise a log) is not mitigated by clients checking consistency proofs as the attacker could equally change Chrome's behaviour (in the same way an attacker could prevent gossiping of such STHs).

Verification of inclusion of issued SCTs (and the certificates they were issued for) will be done by each Chrome client. Issuance of an SCT without inclusion of the right entry (in time) should be rare enough to be considered a security incident and reported in the same manner that invalid certificates are reported. Before that can happen we will measure how often clients fail to obtain proofs of inclusion (and for how long) so as to not falsely report such SCTs and only make a decision on reporting SCTs once we have strong confidence that these are security incidents.

Possible Performance Impact

Note that the process of validating STHs and fetching inclusion proofs is asynchronous and is independent from any certificate validation or any other networking operation. All operations would be done in the background at a low priority.

CPU Consumption: The additional operations are verifying ECDSA signatures for each new STH. This is negligible given it is performed roughly once every 24 hours.

Network overhead: Each client would receive a component update containing ~10 STHs.

Assuming ~240 bytes per STH, that would amount to ~2.4k received once (on average) a day.

Each client will also send a leaf index and inclusion proof request over DNS for each SCT observed. These requests are ~100 and ~60 bytes in size, respectively. The leaf index response is slightly larger than the request (by 2-10 bytes). The inclusion proof response size can range from only 33 bytes larger than the request to as much as ~1k larger, depending on the size of the log that the request is sent to.

Release Plan

Since, at this phase, all client operations are passive validation or reporting metrics, the component updater will be deployed to all clients. Later on, when inclusion proofs are fetched over DNS, a Finch experiment will be used to control gradual roll-out.

Future steps

Potentially fetching STHs independently if we witness some clients always miss component updater updates / have to wait a long time for them.

Figuring out actions clients should take if they cannot obtain inclusion proofs for a period of time from a particular log or all logs.