Certificate Transparency v2

Introduction

The goal is to make it impossible (or at least very difficult) for a Certificate Authority to issue a certificate for a domain without it being visible to the owner of that domain. A secondary goal is to protect users as much as possible from mis-issued certificates.

It is also intended that the solution should be backwards compatible with existing browsers and other clients.

This is achieved by creating a number of cryptographically assured, publicly auditable, append-only logs of certificates. Every certificate will be accompanied by a signature from one or more logs asserting that the certificate has been included in those logs. Browsers, auditors and monitors will collaborate to ensure that the log is honest. Domain owners and other interested parties will monitor the log for mis-issued certificates.

The logs will *not* deal with revocation: that will be accomplished by existing mechanisms.

The Log

Each log is an append-only log of all certificates submitted to it. It is designed so that **monitors** can efficiently ensure that any certificate logged is promptly visible to them and can be checked for legitimacy (for example, by knowing which certificates the domain owner has got from CAs). It is also possible for **auditors** to efficiently check whatever partial information they have about the log is consistent with the append-only nature of the log.

In other words, monitors see the whole of the log, and watch over it on behalf of domain owners and other interested parties. Auditors gather partial information and then verify that all that partial information is consistent with the current state of the log. Inconsistencies indicate dishonesty on the part of the log. For example, an auditor built into a browser would verify that the certificate for each website the browser visited actually appears in the log.

In general, “consistent with the current state of the log” means that the current log provably contains every certificate ever signed by the log.

If the log ever attempts to claim that it has logged a certificate which is not actually in the log, then this will become apparent to auditors and monitors.

Thus, domain owners are assured that only their own legitimate certificates are in circulation, and can take action when certificates are mis-issued. This ultimately protects users as well as domain owners by effectively preventing masquerading as websites that are monitoring the logs.

Detailed Operation

Anyone can submit a certificate to the log. The log will immediately return a signed data structure known as a **Signed Certificate Hash** (SCH) containing

- The certificate and any intermediate CA certificates[1].
- The time the certificate was submitted.

The log promises to incorporate this entry within a certain amount of time[2]. Failure to do so is considered a breach of contract by the log. This time is known as the **Maximum Merge Delay** (MMD)[3].

The log itself consists of an ever-growing Merkle tree of all certificates ever issued. As we show in [ref] it is possible for anyone with a complete copy of the tree to efficiently show that any two versions of the tree are consistent: that is, the later version includes everything in the earlier version, in the same order, and all new entries come after all entries from the earlier version[4]. The size of this proof is logarithmically proportional to the number of entries.

As frequently as possible, but at least as often as the MMD, the log will produce a new version of the tree and will sign the following data, known as a **Signed Tree Head** (STH)

- The root hash of the Merkle tree.
- The number of entries in the tree. This is also, effectively, the sequence number of the STH.
- A timestamp that is equal to the latest timestamp of all SCHs in the tree.

In the unlikely event there are no new entries by the time MMD has expired, the log will insert a dummy certificate in the tree.

Monitors will be able to fetch this new version and a copy of all new certificates in the tree. Since they will have the previous version, they can check for themselves that the two trees are consistent, simply by constructing the consistency proof themselves. Any discrepancy will be a breach of contract by the log. Furthermore, this discrepancy will be provable - the monitor will be able to show the two signed versions of the tree, and that they are not consistent. Because they are both signed by the log, the inconsistent version can only have been produced by the log misbehaving.

It is also possible to show the log has failed to honour the MMD by showing an SCH and an STH whose timestamps differ by more than the MMD, and that the corresponding certificate is not present in the tree[5]. The log must always produce an STH that is more recent than the MMD on request.

Auditors will also be able to request from either the log or a monitor (or anyone with a copy of the log) a proof that any particular SCH is consistent with any particular STH, so long as the MMD has passed since the SCH was issued[6]. This proof consists of a Merkle path from the leaf hash corresponding to the SCH up to the root STH.

---

[1] It is necessary to sign the intermediate CAs because otherwise it would be possible for a CA to issue a certificate, get it logged, revoke it, then serve it instead with a different intermediate CA sharing a private key with the original. This certificate would then have a valid SCH and no revocation.

[2] It is necessary to allow some delay between signing a certificate and logging it because the log must be highly available, in order to promptly issue SCHs, but also needs to put all SCHs into a single log, which requires an ordering agreement across all the signers. This is a known hard computer science problem and cannot be done both quickly and reliably.

[3] If there were not an MMD, then a log could indefinitely delay adding a certificate entry to the tree and so it would be possible to use that certificate to attack users without detection.

[4] This is the same as saying the log is append-only!

[5] This can be shown by simply enumerating the entire tree and showing it is consistent with the STH but does not contain the certificate.

[6] Note that the proof may be available sooner if the log has already incorporated the certificate, but it does not *have* to appear until after the MMD has passed.

However, this will, of course, reveal the particular certificate that was queried and so we must also provide a privacy preserving mechanism for verifying SCHs. We provide two mechanisms.

The first makes the various proofs available via DNS. To get a sense of how this works, say you wanted to see the proof that a certificate with SCH hash 89abcdef is in the current tree (which contains, say, 1300000 entries). To find the location of the SCH in the tree, you would request a TXT record containing its index from

- 89abcdef.hash.<domain>

Say the returned index is 1234567. The Merkle path you want then contains the hash of certificate 1234568, the hash of 1234565 + 1234566, and so on. To get these values, you would request a TXT record containing the hash from

- 1234567.0.1300000.tree.<domain>
- 617283.1.1300000.tree.<domain> (617283 = 1234566 / 2).

and so on. In general, you request <entry number>.<level>.<tree size>.tree.<domain> to get the hash at that position in the particular tree you are interested in[7].

This method does not hide which certificate is being checked, but it does hide who is checking it from the log: most clients are configured to use their ISP's nameservers (or some other caching resolver), so all the log will see is that *some* client of a particular ISP is interested in a particular certificate. The ISP will, of course, know which client, but they also already have access to the IP addresses that client visited.

The second method is for the auditor to request a range of certificates around the one of interest and check all of them, including the one they want to check. In order to do this, the log must also make available a range of timestamps for each chunk of log. That is, for every, say, 256 certificates the log will say what the lowest and highest timestamp in the chunk is[8]. Note that there may be overlap between chunks. The client can thus choose all chunks that may include the certificate it has, since it knows the timestamp from the SCH, and fetch all the corresponding hashes and their proofs. All the log learns is that the client wants to verify one of the certificates it fetched.

Finally, since the proofs can be generated by anyone with a copy of the log, clients can also choose to either keep their own copy, or verify with some trusted third party that keeps a copy (note that because of the signatures on the STH and SCH this third party only need be trusted to preserve privacy, not to be honest about the log contents).

Gossip

All the above allows any particular client to check that their view of the world is consistent with their past views, but the final piece in the puzzle is to show that everyone's views are consistent with each other.

This can be achieved by exchanging STHs and SCHs between different clients through gossip protocols. These can then be checked for consistency using the methods described above. Clients wishing to preserve privacy can verify their own SCHs against STHs fetched from the log or its mirrors and only gossip the latest STH they have seen.

[7] Because the newest part of the tree is often partially filled, these hashes change as new entries are added to the tree. However, older entries (the "left hand side" of the tree) will not change from one version to the next and so can be cached rather than repeatedly queried. This means that even though the tree grows forever, the lengths of proofs transmitted over the wire should remain relatively stable.

[8] We use these summaries for efficiency - the data is available in the STHs already, of course. However, since the client will rely on the summaries for its privacy, the summaries should also be signed and verified.

Sizes

(Assuming SHA-256 as the hash function.)

Data transmitted as part of a TLS handshake: 8 bytes timestamp + signature (<100 bytes for ECDSA, 256+ bytes for RSA)

Signed tree head (STH): 8 bytes timestamp + 32 bytes root hash + 8 bytes tree size + signature

Merkle proof: log2(size of tree) x 32 bytes + STH. For a tree with 1M certificates: 640 bytes + 8 bytes timestamp + signature.

Caching: assume for example a client that caches an intermediate node hash of every subtree of 256 certificates. Cache size for a tree with 1M certificates: 32M bytes/256 + current signature ~ 128 kB. Merkle proof size: 8 x 32 = 256 bytes. (A client will only need to request the rest of the proof + signature if there is a mismatch at the cached node.)