

Let's Encrypt: Report on the Implementation and Deployment of Merkle Tree Certificates for Post-Quantum Authentication

Aaron Gable, Josh Aas, et al.

Internet Security Research Group

1 Introduction

Let's Encrypt, a project of the nonprofit Internet Security Research Group (ISRG), is a leading free and open Certificate Authority for the Web PKI. It is our goal to make it easy for websites to enable TLS, to create a safer and more secure Internet for everyone. As part of that goal we have been heavily involved in the development of the next generation of standards that are intended to enable post-quantum authentication in the Web PKI: Merkle Tree Certificates.

Over the past two months our development team has been implementing `draft-ietf-plants-merkle-tree-certs` [1], as well as the constellation of C2SP specs which define the protocol for serving and mirroring the underlying Merkle tree: `tlog-tiles` [2], `tlog-mirror` [3], `mtc-tlog` [4], and their various supporting documents. Over the next couple months we intend to deploy our implementation in our Staging environment [5] for end-to-end testing with the public and for participation in Phase 1 of Chrome's MTC Propagation Plan [6].

In this report and at the Post-Quantum Authentication Workshop we intend to discuss our experiences implementing these protocols, and share our lessons learned for the benefit of both protocol developers and other implementers.

2 Implementation

2.1 Single-Homed Services

One of the key distinctions between classical Certificate Authorities and Merkle Tree Certification Authorities (MTCAs) is that the latter cannot tolerate a fork or a split-view of the underlying data store. While a multi-homed classical CA could experience a network split, continue writing new certificates to the database in each location, and then reconcile the entries after connectivity is restored, such behavior in an MTCA is a catastrophic failure that would result in termination of the log and potentially distrust of the CA.

As such, MTCA operators need to think carefully about how to prevent multi-writer situations, and thereby prevent log forks. A seemingly-simple approach is to run just one copy of the MTCA service. The upside is that there is no second instance which could unexpectedly write to your log storage and cause a fork. The downside is that you're swimming up-

stream against years of conventional wisdom about redundant services and graceful failover.

You also need to ensure that you don't *assume* that there will always be only one instance. All writes to the tree should be protected by a global lock, and should only succeed if the tree is in an expected state when the write begins.

2.2 Linting

It is incumbent upon any Certificate Authority to ensure that the certificates they issue conform not only to the technical requirements of X.509 and RFC 5280 (e.g. correctly encoding a serial number to be a positive integer) but also to the policy requirements of their PKI (e.g. only including certain Extended Key Usages). The primary way this is accomplished today is through the use of "linting" frameworks, such as `zlint` [7], `certlint` [8], and `pkimetal` [9].

However, MTCAs don't, strictly speaking, issue certificates: they issue log entries. By the time a certificate is constructed, the log entry corresponding to that certificate has already been publicly committed to in the issuance log. As such, most linting needs to take place before the log entry has been created. No framework to do so exists today.

Our solution is to approach this from the opposite direction. We first construct a traditional X.509 certificate with a dummy signature and pass that through the same linting system as our classical certificates. Then we derive the `tbsCertificateLogEntry` from that lint certificate, ensuring that what we write to the log is exactly what we linted.

In the future we hope that lint frameworks will have first-party support for checking `tbsCertificateLogEntries` directly, but this mechanism gives us good coverage for all policy lints and most encoding checks.

2.3 Public Key Storage

The Merkle Tree Certificates draft specifies that a `tbsCertificateLogEntry` does not contain the full bytes of the Subject Public Key Info: ML-DSA public keys are very large, so including them verbatim would drastically increase the storage and bandwidth for logs and (more importantly) for their mirrors.

However, a CA needs to retain the full public keys for two purposes. First, to deliver the standalone and landmark-relative certificates to the subscriber, which may not happen

for hours or days after the log entry is created. And second, to be able to scan their corpus of currently-valid certificates for any whose public keys are vulnerable to a newly-discovered private key recovery method, and which therefore need to be revoked as compromised. While the former could reasonably be accomplished with a simple database, cache, or even in-memory storage, the latter requires retention of public keys for the whole lifetime of the certificate, which suggests durable storage.

Because our log implements the tlog-tiles [2] specification, extending our implementation to also store public keys as part of the tiled log storage is straightforward. Alongside our log entry tiles, we maintain a parallel set of “pubkey tiles”, with keys in 1-to-1 correspondence to the entries in the entry layer. Every time we write a new log entry into the entries layer, we simultaneously write the corresponding key into the pubkeys layer. We do not need any additional cryptographic guarantees over the integrity or immutability of the data in this layer, because the `tbsCertificateLogEntries` already contain the hash of their public key.

This system means that we can trivially prune public keys from storage at the same time as pruning expired entries from the log, and that scanning all currently valid public keys should be about as easy as catching up on a tiled log, an operation we already have extensive experience with thanks to our Certificate Transparency log operations. This mechanism also prepares us for storing other kinds of ancillary data, such as validation records for transparency purposes, in the future. The downside is that there is no mechanism for deduplication of the same public key across multiple entries, but we consider the upsides of this approach to outweigh that storage cost.

3 Deployment

Our MTCA implementation is currently only running in our continuous integration test environment; it has not yet been deployed to our staging or production environments. We hope to have at least partially deployed our implementation prior to the Post-Quantum Authentication Workshop, and will update this report with our findings when we do.

References

- [1] *Merkle Tree Certificates*. Internet-Draft draft-ietf-plants-merkle-tree-certs. IETF PLANTS Working Group. URL: <https://datatracker.ietf.org/doc/draft-ietf-plants-merkle-tree-certs/>.
- [2] *tlog-tiles: Tiled Transparency Logs*. C2SP. URL: <https://c2sp.org/tlog-tiles>.
- [3] *tlog-mirror*. C2SP. URL: <https://c2sp.org/tlog-mirror>.
- [4] *mtc-tlog*. C2SP. URL: <https://c2sp.org/mtc-tlog>.
- [5] *Staging Environment*. Let’s Encrypt. URL: <https://letsencrypt.org/docs/staging-environment/>.
- [6] *Cultivating a robust and efficient quantum-safe HTTPS*. Google. URL: <https://blog.google/security/cultivating-a-robust-and-efficient-quantum-safe-https>.
- [7] *zlint: X.509 Certificate Linter*. ZMap Project. URL: <https://github.com/zmap/zlint>.
- [8] *certlint*. URL: <https://github.com/certlint/certlint>.
- [9] *pkimetal: PKI Meta-Linter*. URL: <https://pkimet.al/>.