Revocation Transparency

Ben Laurie (benl@google.com)
Emilia Kasper (ekasper@google.com)

Introduction

Like Certificate Transparency, only for revocation. Unlike CT, RT needs a recent status for the certificate, so the general idea is that client will somehow obtain and check a recent proof from the log - for example the server could retrieve it from the log regularly and serve that along with the certificate, or clients could be sent the entire revocation list.

As in CT, consistency with claimed changes from one version to the next must be efficiently demonstrable (strictly, what you want is an efficient update mechanism so monitors can only pay attention to changes instead of having to download and process the whole list every 5 minutes - I claim these are equivalent). Unlike CT, deletions are allowed as well as additions. It must also be possible to efficiently prove that an entry is or is not in any particular version of the log.

This document does not attempt to answer all questions about revocation, such as who can revoke and under what circumstances. It provides a mechanism for transparency: i.e. the ability to know, efficiently, that the list of revoked certificates you see is the same as the list everyone else sees, and that every revocation (or unrevocation) is available for scrutiny by any interested party.

There are currently two possible mechanisms, one based on a Sparse Merkle Tree, the other on more traditional Merkle trees where the leaf nodes contain pairs from a sorted list of all revoked certificates.

Sparse Merkle Tree[1]

The idea is that we store the status of every possible certificate in a Merkle tree of uncomputable size. For example, we could hash certificates with SHA-256 and store a 1 or 0 (for revoked or not revoked) in the corresponding leaf. The idea is, of course, general, and can store sparse values for any list of long numbers.

Normally a tree this size would not be computable, because there are 2^256 leaves. However, the observation is that if non-zero values are sparse, then almost all the leaves are 0[2]. This means that almost all level 2 nodes have the value H(0 || 0), almost all level 3 nodes have the value H(H(0 || 0) || H(0 || 0)), and so on. Only nodes that lead to a non-zero leaf value actually need to be computed individually - the rest have the same value and we can compute them efficiently.

Thus, a certificate can be shown to be unrevoked by showing the path to the root in the Merkle tree in the usual way. Of course, this path is 256 hashes long, but almost all of those hashes will have default values, on average, and so do not need to be shown explicitly.

Computing a tree from scratch can be done efficiently. For example, here is code in Python:

```
hStarEmptyCache = ['0']

def HStarEmpty(n):
```

[1] We have not been able to find any reference to this mechanism in the literature and believe it is novel.
[2] In general any repeating value (or even pattern of values) can be used and the construction is still efficient.

```
    if len(hStarEmptyCache) <= n:
        t = HStarEmpty(n - 1)
        t = Hash(t + t)
        assert len(hStarEmptyCache) == n
        hStarEmptyCache.append(t)
    return hStarEmptyCache[n]

def HStar2b(n, l, lo, hi, offset):
    t = hi - lo
    if n == 0:
        if t == 0:
            return '0'
        assert t == 1
        return '1'
    if t == 0:
        return HStarEmpty(n)
    split = (1 << (n - 1)) + offset
    i = bisect_left(l, split, lo, hi)
    return Hash(HStar2b(n - 1, l, lo, i, offset) +
                HStar2b(n - 1, l, i, hi, split))

def HStar2(n, l):
    l.sort()
    return HStar2b(n, l, 0, len(l), 0)
```

Where the argument `n` is the depth of the tree (i.e. 256 in our example) and `l` is a list of the indexes of nodes that take the value 1. Note that we actually use the characters ‘0’ and ‘1’ not the bit values, for no particular reason.

As in CT, we need the ability to also show consistency from one version of the tree to the next. Again, this can be done relatively easily - for each addition or deletion from the list we need to show the path in the tree from the corresponding leaf to the root, which should yield the existing root hash when the value 0 or 1 is used and the new root hash when the value 1 or 0 is used, respectively.

Again, most of this path will consist of default values, which do not need to be shown.

Intuition says that a typical path will contain O(log N) non-default hash values, where N is the total number of non-zero leaves.

Sorted List

The second mechanism takes the list of all revoked certificates and sorts it into order. A tree is constructed where each leaf is a pair of consecutive entries from this list. Note that the leaves do **not** need to be in order. Non-revocation is shown by showing the pair in the tree that bracket the unrevoked certificate.

Consistency is shown as follows. For an insertion into the list, say w, find the pair (x, y) that bracket w. Replace it with (x, w) and add a new leaf containing (w, y). Show the paths for the replaced node and the new node - this proves that the new tree contains all the elements of the old tree, in the same order, and so long as $x < w < y$, the new tree corresponds to a sorted list.

For a deletion, say of w as added above, find the two pairs (x, w) and (w, y). Replace the first with (x, y) and the second with a special value (null). Again, show consistency by showing the two paths.

Note that (null) leaves can be backfilled as new entries are added.

This mechanism is probably less efficient than the sparse tree as each change must show two paths, each of size log N. However, it has better worst case behaviour. Note that inducing a bad case is hard, because it requires a partial hash collision.

Consistency with Claimed Changes Is Not Enough

Suppose an evil log wants to fool you into thinking a revoked cert is, in fact, not revoked, without raising an alarm with anyone else. This is easy to achieve: the log creates a new version of the tree with the cert unrevoked, and then changes it back to revoked. The final state is consistent with everyone else's view, but it need only show you these two changes - and in between, you will believe the cert is valid, whilst the rest of the world continues to believe it is invalid.

This is easy to solve, using the existing append-only log used by CT: each change of root hash in the RT tree must be logged in a CT-like tree. Then the change to unrevoked and back to revoked becomes part of the public log and must be shown to everyone.

This additional mechanism is only needed if deletions or changes of non-default leaf nodes are allowed.