Certificate Transparency

Ziling Yang, Leo McGann

Problem Overview:

Certificate Transparency facilitates identification of mistakenly or maliciously issued certificates. This protects unsuspecting users from malicious websites impersonated as trusted legitimate sites.

Problem Details:

When a client visits a website, they want to ensure that every message they send is to the intended recipient, and every message they receive is from who they say they are. To ensure secure conversation, a symmetric key is used for encryption and authentication between two parties. But being that we may want to visit a website we have never been to before, we need a way to bootstrap ourselves to create this symmetric key. Diffie-Helman is used to generate this symmetric key. However, we face unique problems when trying to authenticate messages when performing Diffie-Helman. We may fall vulnerable to a man-in-the-middle attack where a malicious Mallory sits between the connection between the client and the server and hijacks the creation of the symmetric keys. If the symmetric key creation is compromised, then all security is lost.

As we learned in class, we use public key authentication to ensure all messages sent to and from the client and the server hosting the domain have integrity. So once a client has the public key of the domain they wish to contact, they can execute the Diffie-Helman algorithm without worry. The final issue we face is how we manage these public keys. The server hosting the domain will retrieve its certificate from a certificate authority and forward it to the client. A centralized certificate authority solves this issue. A certificate is a public key coupled with a domain name that lets us know that when making a connection, we know which public key to use for authenticating the messages we receive. To ensure the messages we receive from the certificate authority (ca) are authentic, the CA has its own certificate built into the operating system. This entire process is the backbone of the handshake protocol executed when a user visits a website.

But what if this central certificate authority was compromised. If the CA accidently distributes incorrect certificates because of a data breach then all clients who attempt to access the server who owns the certificate are compromised. A malicious actor would have broken the eu-cma game as they could now sign any message as coming from another actor and the client would incorrectly assert the message as authentic.

Suppose an incorrect certificate was distributed for google.com. If a malicious actor Mallory owned the fake certificate and sat in between the connection during the Diffie-Helman exchange, with the google domain, then Mallory could sign any message as coming from Google with her own secret key, and you would incorrectly authenticate it using the false certificate.

From here Mallory has hijacked the connection with a man-in-the-middle attack. The symmetric key would be established with Mallory, giving her access to all messages sent between you and Google.

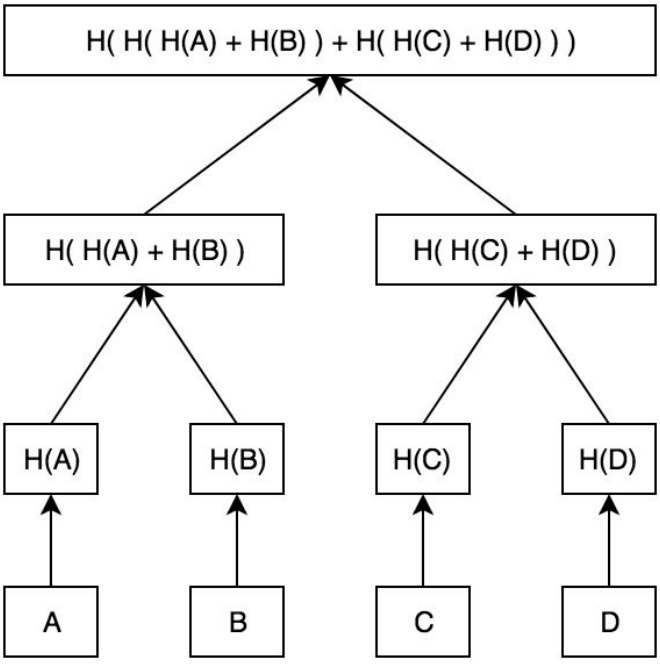
To remedy this and to allow all clients to know if a CA has been tampered with, Google has introduced Certificate Transparency, a framework which adds three new components to the SSLand TLS certificate system: the certificate logs described above, certificate monitors, and certificate auditors. With a log system in place we can restore eu-cma in SSL and TLS connections.

The CA will now use a publicly verifiable append only log of certificates to ensure correctness. The logs are cryptographically secured using a data structure known as a Merkle hash tree. Additionally, logs will be duplicated to allow for fault tolerance among certificate authorities. If one log server is down, an identical copy kept in lock step will be put up.

Solution Overview

Merkle Trees

A Merkle hash tree is a data structure which is used for logging in Certificate Transparency. A merkle tree is a simple binary tree with leaf nodes which leverages cryptographic hashing functions to provide efficient verification that the hash we should be seeing is correct. The output of a merkle tree computation is a root node hash. Any given node's value is defined as the hash of the concatenation of its two leaf nodes. (Figure 1). The bottom-most leaf nodes are simply the hashes of the data we want to have in our tree.



Merging of two trees is done quickly through constructing a new merkle tree with the data we want to add. And using our old tree as one leaf and our new tree as the other, we create a new root node with the hash of the concatenations of the other two roots. A merkle tree has two desirable functions which can be performed as a result of the structure: merkle audit proofs and merkle consistency proofs. To ensure these proofs behave as expected, we must select a hash function that is collision resistant.

Merkle audit proofs let us verify a specific datapoint has been appended to the structure correctly. To verify this, we must recompute the path to the root node. If the parent nodes we compute up to the root are consistent with the tree we are observing, the entry has been added to the log in the correct location. Doing so has us working up the height of the tree and so takes O(log(n)) hash computations.

Merkle consistency proofs let us verify that a dataset represented by some root is the subset of the data represented by another new root. Doing this proof entails reconstructing the subset's root in the new root using the intermediate nodes which should be found in both trees. Doing this verifies that the subset log has been appending in the correct order. Second, we reconstruct the new tree's root and verify it is the same as the root given.

An implementation of the simple Merkle Tree as described above is prone to a second preimage attack. To carry out such an attack, Let us suppose we have a dataset of size four as in figure 1. The input to the intermediate left node is the concatenation H(A)+H(B), and likewise the input to the right intermediate node is H(C)+H(D). If we selected a new dataset with entries H(A)+H(B) and H(C)+H(D), we can construct the same root node. To resolve this, Certificate Transparency's implementation of merkle trees appends a height byte to every node's data before hashing. So in the case of the leaf nodes, we append 0x00.

Logs

Certificate Transparency implements a log for certificates in the form of a merkle tree. The idea is to allow certificate authorities or any other actor to be able to add to the log or to verify the authenticity of the log efficiently and in real time. In the context of Certificate Transparency, consistency proofs show that no entries in the log have been lost, modified, or retroactively appended to, as well as showing that all new entries have been appended to the log correctly. In the context of Certificate Transparency, audit proofs show that a particular certificate is present and correctly placed in the log, and in conjunction with signed certificate timestamps, allow us to know when a certificate which should be present is absent. Because of the need for a pseudorandom hash function, Certificate Transparency uses SHA256. Logs make it possible to prove very quickly and efficiently prove that 1: certificates have been consistently appended to the log, and 2: that a particular certificate has been appended to the log. Doing so leverages the two proof algorithms described above. The consistency and audit proofs guarantee ordering of data by taking advantage of the fact that the hash functions would output different values if the same data was reordered in the tree. Both proofs also show the dataset is the same because only the same dataset can output the same hash value.

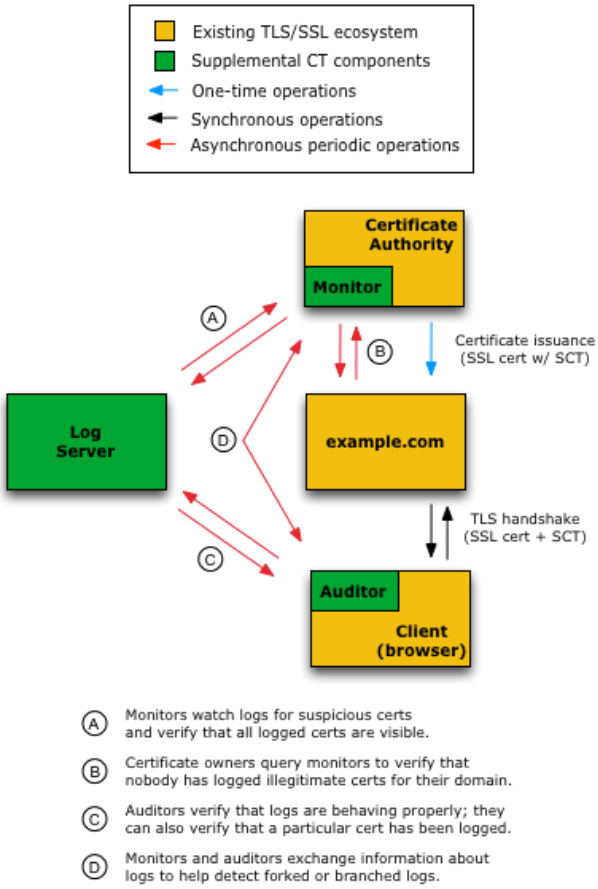
Anybody can submit a certificate to a log. Upon doing so, they will receive a signed certificate timestamp (SCT), a promise to add the certificate to the log in some reasonable time. The server must deliver the SCT along with the certificate for the entire TLS handshake. The

client via an auditor can check if a particular entry which should be present is absent using the SCT. The SCT will assert that an entry should be present and the auditing proof will tell the auditor whether the entry is or is not present. Certificate Transparency provides SCT: X.509v3 Extension as an extension of the TLS protocol to deliver SCTs.

Monitors and Auditors

WIth a log system, anybody can quickly verify the validity of a log. The second component of the Certificate Transparency architecture is an auditor who checks the validity of the log, or sometimes verifies whether one particular entry is correctly placed into the log. The auditor will keep an older verified version on hand and use the consistency and audit proofs to check the validity of the log.

The third component of the Certificate Transparency architecture is a monitor who watches a log to make sure all entries are present and visible. The monitor does this by keeping a copy of the log on hand, and periodically fetching all new entries to the log. Monitors are useful as backups to provide fault tolerance for the system. “If hash function H is collision-resistant and signature scheme SIG is existentially unforgeable under chosen-message attacks, then, in Certificate Transparency (with hash function H and signature scheme SIG), no malicious monitor/auditor can frame an honest logger of not including a promised entry within the maximum merge delay” (Dowling). Certificate Transparency prescribes the configuration in the below figure.

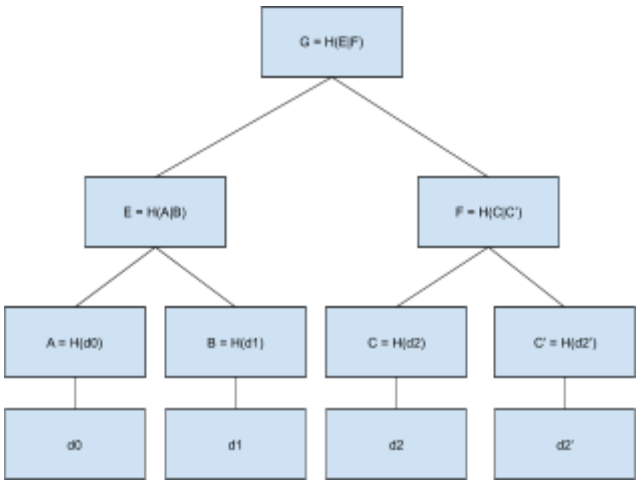


In this configuration, the certificate authority will retrieve certificates from the log server and append the SCT using the X.509v3 Extension to the TLS/SSL protocols. The server can forward the certificate to the browser. The browser can use an auditor to verify the authenticity of the certificate received. Overall, Certificate Transparency's actors are tightly integrated into the TLS/SSL protocols and require some updates to browsers to support. However, the majority of the implementation will fall onto the certificate authorities. This is beneficial as it does not require all domains running on https servers to update.

Description of software:

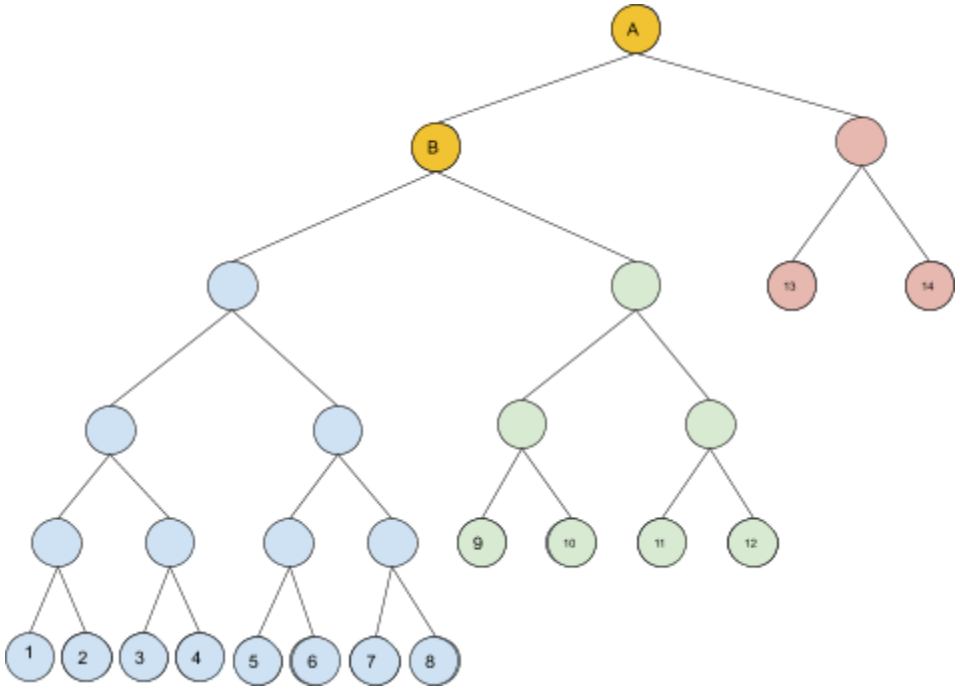
Our software is an implementation of the certificate log in the form of a Merkle tree object and an audit path function that returns a list of positions of the nodes required in the Merkle tree to verify that a certain certificate exists in the log.

The Merkle tree will have certificates as its leaves and their hash as their parents. The parent of two hashes will be the hash of the concatenated value of the two children. For example, the Merkle tree containing 4 certificates will look like the figure on page 2. Our implementation allows for unbalanced Merkle trees, ie. Merkle trees that do not have a power of two number of leaves. However, our implementation only works when there are an even number of leaves. Therefore if a log contains an odd number of certificates, we append the last certificate in the log again to the log. For example, if the log contains three certificates, the third certificate entry will be entered twice with the second entry as the last leaf of the tree. The tree will resemble the figure below:



When the tree is not balanced, the whole Merkle tree can be divided into subtrees of balanced trees with a power of two number of leaves because there will always be an even number of leaves so we can divide the total number of leaves into disjoint groups each containing a power of two number of leaves. For example, when there are 14 leaves in the tree, the tree can be divided into 3 balanced subtrees as indicated by the three different colors in the

figure below. Node B is the hash value of the root of the blue subtree concatenated with the root of the green subtree. The root node of the Merkle tree A is then the hash value of B concatenated with the root node of the red subtree.



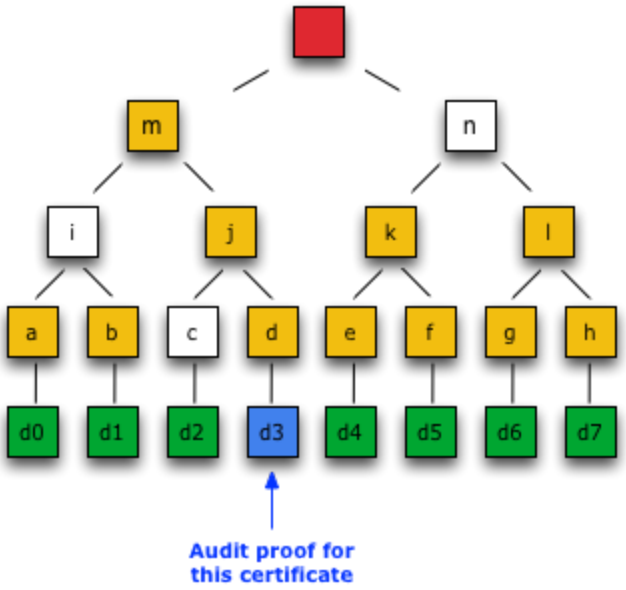
In our implementation, we iteratively partition the list of certificates to be inputted into the tree into decreasing size of sets with a power of two number of certificates, then we construct individual balanced Merkle trees from these sets and combine the individual trees by creating a new root by hashing the concatenation of their roots.

Our Merkle tree object has the following attributes:

1) a dictionary with inputted certificates as keys and the order of when it is inputted as their corresponding values.
2) A dictionary with the layer numbers as key and the hash nodes the layer contains as the corresponding values. The layer that is furthest from the root that contains hash values is numbered 1 and the root belongs to the layer with the largest number.
3) The root value
4) Height of the tree not counting the leaf layer.

Our audit path takes as input the log and a certificate value as input and outputs a list of nodes needed in order to verify the existence of the certificate value in question in the log in the order of the path to the root. For example, if we want to audit d3 in the figure below, nodes, c, i, and n will be in the audit path in that order. So our audit path function will output list [[1, 2], [2,

0], [3, 1]]. The first entry in each list is the layer number starting at one and the second entry is the position of the node needed in the layer starting at index zero.



Let p denote the node of the parent node we can obtain from all the nodes we currently know, then the next node in the audit path will be the sibling of p. In our implementation, we iteratively determine the position of p. The first p will be the location of the hash of the certificate in question. In the example, the first p will be d since that is the only node we can obtain from all currently known nodes. We can determine the first p by determining which subtree the certificate in question is located in since we know the total number of certificates in the log and the order the certificate in question is entered into the log. We can do so by partitioning the certificates into decreasing sizes of sets of power of two. Thus, we can determine the layer and position the first p is located at. Then we enter the sibling of p into the audit path list. If p is a left child, ie. with an odd value of position, then the node needed is the node directly to the right of p, if p is a right child, ie. with an even value of position, then the node needed is the node directly to the left of p. We can iteratively determine the position of the next p by using the following formula: next p = (((p)/2)) mod (length of the next layer). We repeat this process until we reach the layer directly below the root.

Citations

"Certificate Transparency." *Certificate Transparency*, www.certificate-transparency.org/.

Dowling B., Günther F., Herath U., Stebila D. (2016) Secure Logging Schemes and Certificate Transparency. In: Askoxylakis I., Ioannidis S., Katsikas S., Meadows C. (eds) Computer Security – ESORICS 2016. ESORICS 2016. Lecture Notes in Computer Science, vol 9879. Springer, Cham https://eprint.iacr.org/2016/452.pdf

Laurie, B., et al. "Certificate Transparency." *Certificate Transparency*, June 2013, doi:10.17487/rfc6962. https://datatracker.ietf.org/doc/rfc6962/?include_text=1