CONIKS: Bringing Key Transparency to End Users

1 Introduction

Public keys must be distributed securely even in the presence of attackers. This is known as the Public Key Infrastructure problem or PKI. This problem arises in applications such as secure messaging and web surfing/https where a man in the middle can intercept public keys and return erroneous public keys.

Secure communication today depends on placing trust in a few centralized certificate authorities via SSL/TLS protocols depicted below for completeness. However, an attacker can take advantage of the following weaknesses.

1. Attack on Provider: The hypothetical disgruntled employee can intervene in key transfer at the provider level.
2. Provider Coercion: An outside entity, say the NSA, can coerce the email provider into behaving as an attacker.
3. Certificate Authorities are Vulnerable: Hackers can obtain fraudulent certificates by attacking certificate authorities directly posing as a secure provider.

These problems engender the need for end to end encryption so that no provider can decrypt a message without the correct private key. This then becomes the problem of managing keys. Ideally key management would be decentralized. Mutual endorsements allow Alice and Bob to exchange public keys offline or through a secure channel. Unfortunately, keys can be lost and they can be communicated incorrectly (bits flipping). This problem can be partially mitigated by the so called "web of trust" where "key signing parties" are held to distribute public keys reliably. This is properly decentralized as clients reason about encryption, but studies show that it is nonintuitive/error prone, users don’t understand encryption, and it leaks private information.

So one would hope to have the pros (reliability and ease of use) of centralized key management yet dodge the fraught security issues. In this context, it is natural to use "consistency" to define online identities. Users expect that online identities do not change and can be separate from real world identities.

1. Consistency is the property that Alice’s key today is most likely going to be Alice’s key yesterday unless given an explicitly signed key change by Alice.
2. Non Equivocation is the property that the key seen by Alice is the key seen by bob.

Enforcing both consistency and non-equivocation is the central idea behind CONIKS, an alternative Public Key Infrastructure embedded in Google’s project Trillian. Coniks is an End-User Key Management Service with consistent name-to-key bindings; verifiable key directories despite untrusted identity providers; consistency is verified by clients in-band.

First let us consider the strawman solution to performing non equivocation checks. The solution requires all pairs communication for $O(N^2)$ downloads per client. Our goal will be to reduce overhead yet preserve non-equivocation.
2 Goals

Coniks Design

1. Divide time into epochs

2. Providers generate snapshots of directory (publish signed merkle root)

3. Providers distribute snapshots to other providers to build publically verifiable history.

4. Non-Repudiation: Snapshots are digitally signed by the provider and key changes are signed by the clients for which the key is being changed.

CONIKS uses standard Merkle tree hashing. Therefore, instead of passing around the entire namespace, only the root needs to be distributed. See figure 1.

Consistency and non-equivocation are checked in the following way. Clients check their own bindings for validity. Providers verify one another’s STR history. Clients query random providers to ensure non-equivocation.

2.1 Security

Non-equivocation: In a fork attack, an identity provider may equivocate by presenting diverging views of the name-to-key bindings in its namespace to different clients. Because CONIKS providers issue signed, chained “snapshots” of each version of the key directory, any equivocation to two distinct clients must be maintained forever. Furthermore, cross talk with a random subset of providers will practically ensure non-equivocation with high probability.

2.2 Privacy:

CONIKS servers do not need to make any information about their bindings public in order to allow consistency verification. An adversary who has obtained consistency proof for a set of usernames, cannot reverse engineer any information about which other users exist in the namespace or which keys are bound to them. This is because being able to access specific branches of the Merkle tree, does not allow the adversary to reverse engineer any more of the tree.
2.3 Deployability:

All overhead should scale at most logarithmically in the number of total users. This will be accomplished with Merkle trees.

3 Data Structures

**Overview:** CONIKS identity providers manage a directory of verifiable bindings of usernames to public keys constructed as a Merkle prefix tree of all registered bindings in the provider’s namespace.

At regular time intervals (epochs) the identity provider generates a non-repudiable (signed by provider’s signing key) "snapshot" (publishes Merkle root) of the directory. Hence the name "signed tree root" or STR. Clients use STRs to check the consistency of key bindings efficiently to avoid the need to access the entire contents of the key directory. Each STR includes the hash of the previous STR forming a linear history of the directory.

**Proofs of Inclusion:** Providers present proof of key inclusion by providing a complete authentication path between leaf node and root. Similarly, key absence is proven by providing the authentication path of the key’s neighbors.

**Hash Chain:** At each epoch, the provider signs the root of the directory tree using their directory signing key. The STR is added to a chain of STR’s indexed by epoch. This ensures that if an identity provider ever equivocates by creating a fork in its history, the provider must maintain these forked hash chains for the rest of time i.e maintain fork consistency. Otherwise, clients will detect the equivocation when presented with an STR belonging to a different branch of the hash chain. See figure 2.

4 Operations

4.1 Registration:

To register a client key binding with a provider, the client sends a registration request to the provider to bind the public key to the client name. If this name is not already taken in the provider’s namespace it returns a temporary binding for this key. The client then needs to wait for the next epoch and ensure that the provider has kept its promise of inserting the binding into its key directory.

4.2 Lookup:

Since CONIKS clients only check directory roots for consistency, they need to ensure that public keys retrieved from the provider are contained in the most recently validated directory.

When Alice wants to send a secure message to Bob, she first requests Bob’s public key from the provider. To allow Alice to check whether the recipient’s binding is included in the STR for the current epoch, the identity provider returns the full authentication path from the recipient’s binding in the Merkle prefix tree along with the current STR. Finally, Alice recomputes the root of the tree using the authentication path and checks that this root is consistent with the presented STR. See figure 3.
4.3 Monitoring:

A client is monitoring to see if the key bindings of its users have not been maliciously changed. The client begins by performing a key lookup for each user to make sure the provider responds with the matching public key. This is done by checking that a user’s key is consistent between epochs. If the keys have not changed, or the client detects an authorized key change, the user need not be notified. Else, whistleblow. See figure 4.

4.4 Auditing:

Clients need to verify that any provider is maintaining a linear STR history. Comparing each observed STR with every single other client would be a significant performance burden. Providers distribute their most recent STR to other identity providers in the system at the beginning of every epoch acting as auditors for one another. The auditing protocol checks whether an identity provider is maintaining a linear STR history. Whenever auditors observe a new STR from any provider, they check for the provider’s signature. Then they check whether the hash of the previous epoch’s STR matches what the auditor saw previously. If they do not match, the provider has generated a fork in its STR history.

Thus each auditor has an independently verified STR of all the other providers. Thus if Alice wishes to speak to Bob, she queries some number of providers at random to compare their STR histories with the one offered by the provider for Bob’s public key. If they all match, then there is no equivocation with high probability. See figures 5 6.
Figure 5: Steps taken when verifying if a provider's STR history is linear in the auditing protocol.

Figure 6: Steps taken when comparing STRs in the auditing protocol.