Key Lookup

- Goal:
  - Get the public key for some principal
  - Authenticate the mapping between principal and public key
- So far:
  - We just use a trusted centralized server that provides these public keys
    - Doesn't cover the second at all
  - We covered TOFU, which is trust on first use
    - Covers the second on second use
  - Just give them in-person
  - On other platforms: i.e. post on Twitter

- Solution we will cover today:
  - Trusted directory (server) with a transparency log
  - CA's have started using this
  - Messaging platforms have started using this
- Trusted directory is extremely usable
  - However, it could be malicious and give out bad public keys
  - Then intercepts packets, sends them to actual server with the real key -> MITM attack
  - When the actual server checks on their public key mapping, the malicious server could just send them the actual key

Transparency Log

- Provides a way to audit the directory
- Directory has to maintain a log of all changes it makes
  - Log is append-only and can be audited by anyone to check for inconsistencies between what the directory is claiming and the log
  - Clients can check the log for their own mappings to see if anything is suspicious in the log
- What we need to do:
  - Ensure there is a unique log
  - A client needs to know if the public key they fetched is actually logged in this unique log
    - They need to know that everyone else also sees this fake key
  - A client needs to be able to audit their own key
    - A basic log would require the entire log to be audited (bad and slow)
    - How often they audit is how fast they want to prevent attacks
- CONIKS goes through this fast

Merkle Tree

- Way to succintly talk about a large amount of data like a log
- Say we have nodes `a, b, c, d`
  - Parent of `a` and `b` is `H(a) || H(b)` at node e
  - Parent of `c` and `d` is `H(c) || H(d)` at node f
  - Parent of `e` and `f` is `H(e) || H(f)`
- If `H` is a 256-bit hash function, then each node will have 512 bits of hash information
  - If this is collision resistant, then a single hash value at the top should well represent the entire tree

Proving Something In the Key

- How can we prove efficiently that something is in the tree?

    - For the log, how can we prove that the key we just received is in the log?
- Take the linear log and turn it into a Merkle tree
    - To check if a node is in the tree, we just need to check the path from the root to that node and check all of the hash values to check if they are valid
    - The size of the proof we have to send is just $O(\log n)$ where n is the length of the log

Uniqueness of Log

- When the log service wants to update the tree, they send it to multiple auditors
    - Sign it with the log service's signing key
    - Each of these auditors will gossip with each other and talk with each other to make sure that everything they received is the same
- Clients can also gossip with the auditors / etc. (actually in CONIKS all clients are auditors in a way) so they will all tell each other if there is a discrepancy

Auditing Log Appends

- How can we verify log appends?
    - On each append by a a client, verify that the new hash value returned is correct
    - We need to compute this locally and check it matches with what the server said
- When we add nodes to a Merkle tree, we can pretty efficiently compute with the new nodes and $O(\log n)$ time what the new tree will be, so we can check what the new server says now

CONIKS

- Certificate authorities just do this globally and publish a tree that anyone can audit and anyone can lookup
    - Also the auditors have to spend a lot of bandwidth
    - Fine for servers and certificates because they have a good amount of bandwidth
        - Cell phones on SIGNAL on the other hand do not have this bandwidth
- On the other hand, CONIKS tries to provide privacy as well and make audit efficient
    - Arrange the tree so that we could only appear in one place
    - We don't really have a log, we just have a tree
    - We use the hash of the username as the path to the node
        - i.e. if hash of alice is 101, then we take a right -> left -> right in the tree to get the node for alice
    - There are some extra tricks to make this more private to make sure
- Problem now:
    - We don't have this append-only log, so we need some other way to guarantee that we do not have forking histories
    - We do something akin to blockchain where when we sign the tree, we sign the current tree hash appended to the previous tree hash
        - If someone forks then, they have to maintain that fork forever
        - They cannot reconverge these forks
            - These reconvergent forks cannot both have the correct signature with the previous tree or not