Dename: A Decentralized Egalitarian Name System

1 INTRODUCTION

The purpose of dename is to communicate public keys to each other in a simple manner with minimal trust in third parties. We intend for dename to be usable without training, yet still sufficiently secure for situations previously handled using manual key distribution. Rather than communicating key fingerprints, users simply have to communicate usernames and let the application resolve them to public keys using dename. In his essay[TODO], Zooko Wilcox-O'Hearn postulates what has been called Zooko's Triangle: that no such system can be secure, decentralized, and provide human-meaningful names all at once. However, Aaron Swartz's "square triangle"[TODO] and the very similar NameCoin[TODO] can be thought of as a counterexample. We prefer to think in terms of degrees of decentralization: their design provides human-meaningful names in a manner that is both secure and decentralized (to the extent that hashing power stays decentralized), at the expense of requiring massive amounts of computation for security. By contrast, dename introduces a limited amount of centralization for the sake of efficiency and a clearer security guarantee. A central set of leader servers (which should be operated by different parties who are unlikely to collude) orders update operations, and independently operated verifier servers vet and sign the results. If a client requires signatures from some set of servers, just one of them has to be correct for the security guarantees to hold.

2 OVERVIEW

A deployment of `dename` has a fixed set of servers mapping a single namespace to profiles consisting of public keys and possibly other information. Each server has a signing key and knows the public key of all leaders and some set of verifiers. To modify a name-profile mapping, all leaders have to synchronize with each other. Thus, if one of them is down, no progress can be made. Verifiers differ from leaders in that other servers continue to operate even when a verifier is not available. We assume that the clients obtain the public keys of the core servers and any verifiers whose confirmation they require out of band (for example, together with the software). To register a name or modify a profile associated with an existing name, a client can contact a leader of its choice, but lookups can be served by any server with an up-to-date copy of the state. The servers prevent clients from modifying each others' profiles by requiring the change request to be digitally signed with a key designated in that profile.

A `dename` server exposes the following API:

- `modify(name, newProfile, newSignature, oldSignature)`: Make the name point to the new profile. `newSignature` is a signature on the request with the key contained in the new profile. If an old profile exists, `oldSignature` is a signature on the request with the old key.
- `getRoot()`: Return the hash of the current directory state signed and timestamped by each server known to the server contacted by the client.
- `lookup(name) -> (profile, proof)`: Return the profile that the name points to, along with a proof that the name-profile pair is present in the directory and up to date. A typical client would call `lookup` and `getRoot` together atomically, and check the signatures, timestamps, and the proof.

We envision that `dename`'s first-come-first-served registration policy can be easily incorporated into existing application designs, by simply changing the order of name registration. For example, instead of first registering for an account with an email client, and then creating a mapping for that name, the email client should first register an appropriate name for the user in `dename`, and if that succeeds, create an email account under that username.

[TODO organization of paper]

3 MAINTAINING CONSENSUS

Changes to the user directory happen in discrete rounds: each server proposes a set of changes and all servers apply them in lockstep. We use a verified broadcast primitive (described below) to ensure that all servers receive the same set of requested changes, and we handle the changes in a deterministic order. Additionally, we describe some malicious behavior that servers could engage in which would not directly violate the security claim, but is nevertheless undesirable, and *blind* the protocol to counteract that behavior.

The physical analogy of verified broadcast is a public announcement: everybody learns what the announcer has to say and can be sure that others heard the same thing. In computer networks allowing only point-to-point communication, we can emulate this using a two-phase protocol: first the announcer broadcasts the message, then every server broadcasts an acknowledgment of what they received from the announcer. In `dename`, all n servers announce exactly one set of changes $\Delta_1 \dots \Delta_n$ during each round, so we can group each server's acknowledgments of all messages it received into one message. Furthermore, as only the equality of the sets of announcements received by different servers is important, rather than the actual contents, we can sign a cryptographic hash $h(\Delta_1 \parallel \dots \parallel \Delta_n)$ of all received announcements in an acknowledgment instead of the announcements themselves. The verified broadcast protocol can be seen in Figure 1.

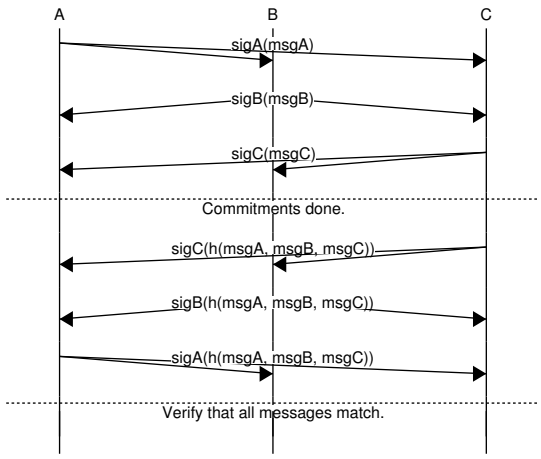

Figure 1: Verified broadcast

In the description above, all messages are assumed to be authenticated. If one server were able to impersonate another, it could fool other servers into thinking that a different set of changes has been announced. We use digital signatures for authentication because

unlike faster symmetric authentication mechanisms, signatures can be used to construct an audit trail in case one of the servers sends out different announcements or acknowledgments during the same round.

4 SEMANTICS

The semantics of what kind of changes are allowed are in some sense a detail, but they are important. For example, if one user were able to edit another's profile without their consent, the directory would be of little use. Our implementation sets the following constraints:

1. All proposed profiles must contain a public key that can be used to verify digital signatures.
2. If a name is not currently in use (does not map to a profile), all requests to make it map to a valid profile are accepted.
3. If a name already maps to a profile, requests to change the mapping are only accepted if they are also signed with the key in the current profile in addition to that in the new one.

These rules ensure that if a user keeps their signing key secret, nobody else can modify their profile. To free up names for which the corresponding secret key has been lost, we also allow expiration:

4. After the expiration time specified in the profile has passed, the name becomes available again. (This is an exception to rule 3.)

Table 1 shows the fields stored by `dename` with example data.

name	pubkey	profile	expiration
alice	pk_a	$pk_{\texttt{ssh}}$, pk_{x509}	2016-01-11
bob	pk_b	`bob@mit.edu`, $pk_{\texttt{gpg}}$	2015-10-15

Table 1: `dename` directory schema