Dename

Andres Erbsen Daniel Ziegler

April 28, 2014

Abstract

We build a public key distribution mechanism suitable for universal adoption using simple and widely understood mechanisms.

Many applications rely on some form of directory service for connecting human-meaningful user identifiers (names) with application data associated with that user. When trying to provide security, the lack of a sufficiently trusted directory can easily become bottleneck: compromising any one certificate authority of an attacker's choice breaks anything that relies certificate-authority-based public key infrastructure [@EllisonSchneierPKI][@SchneierVerisignHacked]; a breach of the Kerberos domain controller would result in a total compromise of the security domain. For this reason, security-critical applications try to work around the need for a dircetory service; for example, `ssh`, OpenPGP, OTR and Pond have users manually communicate the critical bits of authenticating information to each other. This approach is tedious[@arsTechnicaGGreenwaldPGP] and prone to human error, especially if the users in question are not online at the same time[@Johnny2008]. Recently, better ways to maintain a user directory have been discovered, such as [@SwartzSquareZoooko], [@CertificateTransparancy] and NameCoin. However, all of those rely on economic feedback loops for security, and the cost is passed on to the users. We present `dename` – an efficient distributed user directory service that works under the assumption that any one of the servers is secure.

1 Overview

In essence, `dename` works by having a group of predetermined but independently administered servers maintain identical copies of the user directory and collectively vouch for the correctness of the directory's contents. Any one server's honesty is sufficient for a unanimous result to be correct. We have the servers require that all updates to a user's profile are digitally signed by that user, thus preventing any other party (including malicious servers) from modifying it. The discussion of the operation of this system is organized as follows: first, we describe how the servers communicate with each other to apply changes to the directory while ensuring that they end up with identical results. In general, this is the problem of replicating a state machine in the presence of malicious faults, but our solution is significantly simpler than the previous because it requires all parties to participate in order to make progress. Second, we describe the procedure of looking up users' profiles. We start with a trivial but inefficient protocol and end up storing the directory in a Merkle-hashed radix tree and serving its branches. We argue that if a

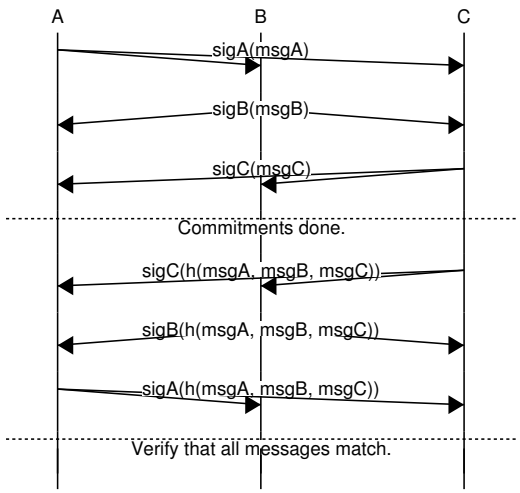

Figure 1: Verified broadcast

lookup succeeds then the result must have been accepted by all servers. Third, we tackle the issue of freshness, that is, we provide a system for ensuring that the result represents the most up-to-date state of the system. Fourth, we show how independent verifiers can be added to this system in the spirit of Certificate Transparency[@CertificateTransparancy] and VerSum[@VerSum]. Starting from the Merkle tree data structure described previously this addition is relatively straightforward and, as a side effect, enables efficient coherent caching of lookup results.

2 Maintaining consensus

Changes to the user directory happen in discrete rounds: at regular time intervals (currently every 3 seconds) the servers propose changes and 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 the algorithm for handling them is deterministic. Additionally, we describe some malicious behavior servers could engage in that would not directly violate the security claim but is nevertheless undesirable and modify 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 just the equality of the sets of announcements received by different servers is important (not 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[hbt].

If two honest servers transition to a new state as a result of a round, they transition to the same state:

$$h(\text{inputs of A}) = h(\text{inputs of B})$$
$$\text{inputs of A} = \text{inputs of B}$$
$$\text{apply(state, inputs of A)} = \text{apply(state, inputs of B)}$$

In the description above, all messages are assumed to be authenticated. If one server was 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,

name	pubkey	profile	last change
alice	pk_a	`22:`$pk_{\texttt{ssh}}$`,443:`pk_{x509}	2014-04-10
bob	pk_b	`25:bob@example.com`	2013-09-12

Table 1: `dename` directory schema

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.

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

1. All proposed profiles must contain a public key that can be used to verify digital signatures. To enforce this, all requests that are not signed with that key are denied.
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 it are only accepted if they are also signed with the current key in addition to the new one.

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

4. If the profile a name maps to has not been modified in the last T_e rounds, the profile is automatically deleted.

This requires users to regularly confirm that they still use that profile by requesting a nil change to it. The possibility of a profile expiring complicates the situation because somebody else may later claim the name, but the old profile still fits the criteria of being accepted by all servers – this is the main motivation for freshness assertions (section 4). It is, of course, possible to have names not expire, but doing so would seriously hamper the usability of the system when the space due to more and more names pointing to profiles with lost keys.

The described rules of changing the directory are sensitive to the order in which changes are processed: if two servers propose two valid requests to modify the same name in different ways, it is crucial to ensure that all servers choose to apply them in the same order because applying one of them may make the other invalid. We use a standard protocol akin to [@XXXsharedRandomness] to establish shared randomness between servers and use it to pick a random permutation of the list of servers that determines the order in which the requests they introduced are handled.

However, a malicious server could observe the announcements other servers make and deliberately introduce requests that conflict with a particular user's requests. To prevent this, the requests are hashed before they are broadcast using the verified broadcast protocol and actual requests are only revealed after every server has announced the hash of their proposed changes. Therefore, all changes a server proposes must be independent of the ones proposed by other servers because it only gets to observe the other proposals after broadcasting its own. To spread out network load, the current implementation actually pushes encrypted requests to other servers before having received hashes from them and reveals the encryption key to reveal the requests. The final protocol

is displayed in figure 2.

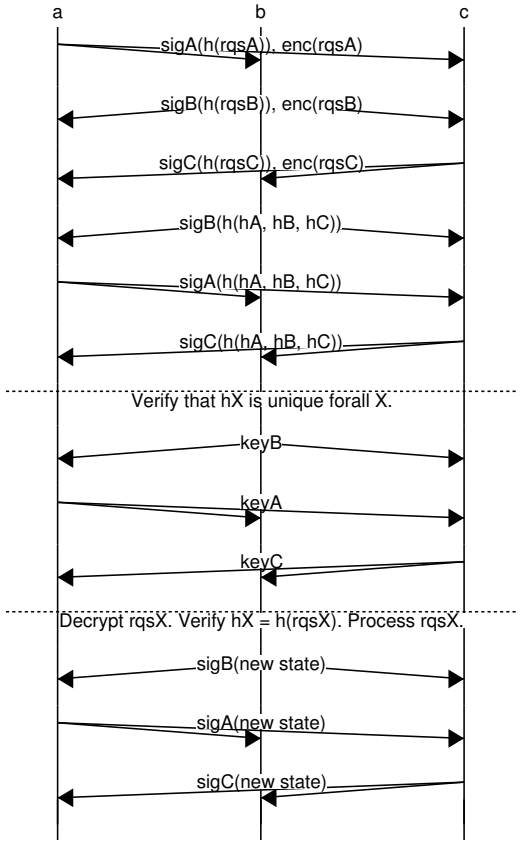

Figure 2: `dename` consensus protocol

3 Lookups

Simplistically, looking up a profile could be implemented by having the client download the entire directory from each server and consider it correct if all copies are equal. This is impractical if there are millions of users. As an improvement on this, the client could instead download the hash of the directory from all servers and the whole directory from one server. If the hashes the servers reported are all equal to the hash of the downloaded directory, the directory must be correct. This scheme is slightly better, but still insufficient.

What we need is mechanism to prove that a single name-profile pair is a part of a larger directory with the given hash without downloading the whole directory. Assume that the directory is implemented as a binary prefix tree with profiles in the leaves. Now, every node in the tree is augmented with a hash of its children. If the hash function is collision resistant, each node uniquely determines the state of all names (and the respective profiles) that start with the prefix this node corresponds to. The root hash summarizes the whole directory.

To prove that a name-profile pair is a part of a directory with a known root hash, a server supplies the client with the list of hashes stored in the siblings of all the nodes along the path from the leaf to the root – the *Merkle hash path*. To verify the proof, the client first hashes together the name and the profile. This hash is then hashed together with the server-provided hash stored in the sibling of the leaf, resulting in the hash of their parent. This is repeated recursively, hashing in the rest of the siblings all the way up to the root. If the resulting root hash matches, the client knows the name indeed maps to the given profile in the dictionary with the known root hash. As all servers vouched for the whole dictionary and we are assuming that at least one of them is honest, the profile must have been registered adhering to the requirements of this system.

TODO: picture of Merkle tree, like in `http://comjnl.oxfordjournals.org/content/27/3/218.full.pdf`

As an optimization, the servers can sign the root hash

after each round and send the signature to all other servers. This way, a client only has to talk to one server to do a lookup but can still be assured that all servers agree about the result after verifying the signatures. The current implementation also uses the hash of a name instead of the name itself in the prefix tree. This serves to keep the tree balanced and simplify the implementation. As we assume hash collisions do not happen, it does not change any other properties of the system.

3.1 Name absence proofs

If a requested name is not in the tree, the server can prove its absence by returning the name-identity pairs right before and right after the missing name, as defined by the lexicographical order of the dictionary keys, along with the associated Merkle hash paths. The client has to verify that both hash paths result give the correct root hash and that there are indeed no nodes in between: up to their common ancestor, the former only has left siblings and the latter only has right siblings. *We have not implemented this mechanism.*

4 Freshness

The protocol as described guarantees that if a client looks up a profile for a name, this assignment must have been approved by all servers at some point in time. However, nothing so far prevents it from being superseded by a later change to the same profile. In this section we describe two mechanisms for ensuring that the lookup result is *fresh* (not superseded). The more efficient one requires the client and the servers to have a reasonably accurate clock; without that there is an option to get a confirmation of freshness from each server individually.

Each server will regularly (every Δt seconds) sign a *freshness assertion* with the contents "As of time t, the most recent root hash is H". The most recent assertion from each server will be distributed together with the root hash. Before accepting a root hash as valid, the client will verify the signatures on the freshness assertions and check that the timestamps are withing $\Delta t + \epsilon$ of its current time, where ϵ accounts for network latency and uncertainty of the current time value seen by the client.

Requiring all timestamps to match can be an availability problem: if any of the servers is down, all lookups will fail. However, this requirement can be easily relaxed: for any $f \geq 0$ of the client's choice, it can check that at most f of the timestamps are outside the allowed range and thus continue operating even when f servers are down.

Note that unlike in Spanner[@spanner], the time uncertainty can be quite large for the system to operate correctly. Even though future lookups by a machine that has its clock within ϵ of all servers' clocks are only guaranteed to observe that happened more than $\Delta t + 2\epsilon$ ago, we do not see it as a problem because we expect security-critical changes to profiles to be rare and therefore waiting after them to be acceptable. Proof of the $\Delta t + 2\epsilon$ bound: it will take at most Δt for the new mapping to be timestamped by the servers, the client will accept any mapping bearing a timstamp less than ϵ before its own observed time (because its clock may be at most ϵ ahead), but its clock may also be at most ϵ behind of the server time, in which case it may end up accepting a mapping that was timestamped 2ϵ ago.

In case a reasonably accurate clock source is not available, a client can still look up the current profile for an username by contacting a set of servers such that at least on of them can be assumed to be honest and requiring an unanimous answer.

5 Verifiers

We view having a fixed set of servers as a necessary evil: it is inherently a central point of compromise, but the only alternative we know is to have the evolution of the directory state determined by the entities that score highest by some arbitrary metric, such as the hashing power they control as in BitCoin and NameCoin. To mitigate this weakness, we provide an additional accountability mechanism: everyone can observe how the central servers change the state of the directory, detect deviations from the rules and, in case of invalid changes being applied, have proof of wrongdoing on the servers' part. We describe a *verifier* design that is significantly simpler (and therefore more likely to be implemented correctly) than the servers themselves. We also show how to leverage the Merkle tree structure already used for lookups to audit the changes made during an interval of time without ever having to download the whole directory.

5.1 The simple offline verifier

The purpose of the simple verifier design is to check that the core servers have been enforcing the semantics of the directory. The design we describe here does not aim to provide optimal throughput or responsiveness, instead we focus on keeping the implementation as simple as possible with the hope that it can therefore be widely audited and gain public confidence.

The program takes as input a range of rounds starting with the very first one (in the beginning of which the directory was empty) and for each round the ordered sequence of changes considered by the core servers. It processes the change requests in order, validating each one against the current state of the directory and then updating the directory to reflect this change. At the end of each round, it prints out the current hash of the Merkle tree. To verify that some sequence of observed root hashes corresponds to a consistent history of valid changes to the directory, one would download a copy of the purported changes applied to the directory from any one server and use the simple verifier to compute the root hashes for all past rounds. If all root observed hashes are present in the output of the simple verifier in the right order, the server must have adhered to the semantics of the directory.

We implemented this verifier using 40 lines of readable python code and 20 lines of `protobuf`[@protobuf] format specification. No custom libraries were used; an in-memory implementation of the Merkle-tree is included in these 40 lines.

5.2 Incremental verification

The verification system described in the previous paragraph may be simple, but it will become more and more costly to use as the total number of handled requests increases. We wish to provide a mechanism through which independent parties can participate in the verification of new changes made to the directory without having to pay the up-front cost of downloading all past changes. Naively omitting

the old changes from the inputs of the simple verifier would not yield a solution: it would have no way of determining whether a name has been already registered or not. Instead, the core servers will supply the verifiers with Merkle-tree proofs about the relevant directory state in addition to the requested changes. Specifically, each request to transfer a name will be annotated with the old profile, its Merkle path and all siblings used to calculate the hashes for the new Merkle path. The verifier will then use the lookup procedure to verify the old mapping and calculate the new root hash using the server-provided values instead of storing a local copy of the whole tree. *We have not implemented this mechanism.*

5.3 Coherent caching

A continuously running incremental verifier will observe all changes to the directory. Therefore, if the verifier ever learns a name and the corresponding profile, it will also receive all future updates to that name. Servicing a lookup for a name requires having the branch of the Merkle tree that corresponds to that name, not the whole tree, so a verifier can serve the contents of its possibly incomplete but up-to-date directory. Furthermore, when presented with a lookup request for a name that it does not have a profile for, it can just look up the name from a server that has a more complete version of the directory. A server operating this way is effectively a cache and can be useful to reduce lookup latency in a local network and reduce the load on the core servers. Unlike with DNS and Namecoin, a client using a cache achieves the same security guarantees as a client that interacts with any one core server.

6 Implementation details

We implemented the `dename` server and client libraries in less than 4000 lines of `go` using `postgresql` for storage. The current implementation is a compromise between performance and understandability. For example, independent tasks are done in parallel and in-process state is kept to eliminate redundant database accesses and server signature verifications, but client signatures are verified twice in some scenarios, batch signature verification is not used at all and some invariants are enforced using expensive database byte array indexes even though doing it manually is possible and has shown better performance. Nonetheless, a laptop with a `Core 2 Duo L9400` cpu and a `Corsair Force GT` SSD drive can handle 300 registrations per second, being just slightly disk-bound. This number may not seem high when compared to non-cryptographic databases, but 800 million registrations per month is unlikely to become a limiting factor in any realistic deployment scenario. Our implementation also detects and reports various kinds of deviations from the specified protocol by other servers even if ignoring them would be completely harmless – this is intended to assist with debugging and validation of possible other implementations.

6.1 The consenus protocol

The protocol we use to maintain verifiable consensus in a group of peers, some of which may be malicious, is not specific to `dename` and can potentially be of interest for other projects. We preserve this separation in the implementation: roughly a quarter of the codebase is made up by a reusable consensus library.

The library waits for the application to submit operations to be handled and calls an application-specified state transition function with the inputs chosen for a single round whenever one is processed. Similarly, network communication is implemented by the application and exposed to the library using a simple `send(peer, data)` interface. However, persistence is currently handled by the consensus library itself because the crash recovery procedure involves fairly complicated queries to the consensus-specific state.

We sought to preserve the semantic separation between rounds in the implementation by making each round be managed by its own thread and control structures, but in order to simplify the crash recovery procedure, we added an explicit dependency between adjacent rounds. If a server has not seen all other servers' signatures for the shared state at the end of round i, it is not allowed to push (encrypted) requests to rounds after $i + 1$. Therefore, if a server has published its final signature for round $i - 1$ but not i, it must have finished processing rounds $\leq i - 2$ and does not need to do anything for rounds $\geq i + 2$: there can be at most 3 rounds in progress at the same time. The code also makes use of similar constraints implied by this rule and the explicit requirements of the protocol to bound the window during which each type of message can arrive from another correct server during one round.

When a server crashes, it loses its in-memory state. To have a complete overview of the exact behavior of other servers, our implementation stores all received messages on disk. This requirement could be loosened in a performance-oriented implementation, but it is absolutely critical to synchronize the following pieces of information to disk before acting upon them:

- The symmetric key used for pushed requests
- Whether a commitment has been made
- Any requests pushed or committed to
- The acknowledged commitments' signatures

Currently, the in-memory data structures are reconstructed after a crash by reinterpreting the stored messages and using the saved encryption key instead of generating a new one. This approach is very robust and allows for relatively straightforward code, but it depends on having a full log of all messages received during the last 3 rounds. Synchronously writing these messages to disk is the performance bottleneck of the current implementation.

6.2 Persistent Merkle radix tree

TODO

6.3 Cryptography

No exotic cryptographic primitives are required for the operation of `dename`, but because the choice of specific algorithms dictates performance and log size, will describe our choices and the reasoning behind them. For all algorithms, we required a security level of 128 bits and existing adoption in real-world systems.

- `ed25519` for digital signatures. Fast signature verification and small signature and public key size are essential for the performance of `dename`. Unlike other common digital signature schemes, `ed25519` supports even faster batch verification.
- `sha256` for collision-resistant hashing and entropy extraction. Widely used, fast enough.

- `salsa20poly1305` encryption for concealing messages from servers during the commitment phase of a round. Any authenticated encryption scheme would work, chosen for simplicity.
- `salsa20` keystream for pseudo-random number generation to break ties between requested changes. Chosen for simplicity.

6.4 Integration with existing systems

To show that it is practical to replace manual public key distribution with `dename`, we integrated a `dename` client with the Pond[@Pond] asynchronous messaging system and `ssh`. Modifying Pond to work with `dename` required changing 50 lines of logic code and 200 lines of user interface declarations; the two `ssh` wrapper scripts are 2 lines each.

Pond requires each pair of users to establish a shared secret before they can use Pond to communicate with each other. The Pond User Guide gives a detailed explanation of several acceptable ways that may be used to establish a shared secret, but despite the presence of instructions, using Pond requires effort; it is designed for users with a strong commitment to privacy. We believe that exchanging public keys between contacts is the limiting factor of Pond's usability. Our variant of Pond includes the necessary user interface for associating a Pond account with a `dename` profile and adding contacts using their `dename` names instead of shared secrets, thus enabling an user experience similar to email.

We can leverage `dename` to improve the usability of `ssh` in two separate ways. As with Pond, we can use `dename` to look up users' public keys. As `ssh` reads the keys that an user is allowed to log in with from a file, we can simply get the newly authorized user's public key from `dename` and append it to that file. A less obvious but arguably more useful enhancement is to verify ssh *host* keys against the maintainer's `dename` profile. When connecting to a machine the first time, `ssh` usually presents the user with the server's public key hash and asks them to check its authenticity. Instead, if we know the server maintainer's `dename` name, we can preemptively look up the corresponding profile and get the host key from there, without having to prompt the user at all. As in our experience many users tend to neglect the `ssh` host key validation step, this modification will not only increase convenience but also improve security.