The Federated Open Key Service (FOKS)

Maxwell Krohn (max@ne43.com)

March 15, 2025

Abstract

This paper presents FOKS (Federated Open Key System), a decentralized key management system designed to provide secure and flexible key distribution across federated networks. The basic problem addressed is that of of two parties sharing end-to-end encrypted data across the internet, where both parties have several devices. They might rotate devices, form mutable teams with other users, or even teams of teams in an arbitrary graph. They need to share secret key material to facilitate symmetric encryption, and this material must rotate whenever devices are replaced, or team membership changes. This is a very natural problem but one that still lacks an adequate solution. Morever, we believe key management should not lock users into a particular, walled provider, but instead, should allow for federation and independent management of server resouces, as we see in HTTP and SMTP. We describe the system architecture, security model, and implementation details of a system that achieves secure, federated key exchange, and enables useful applications like end-to-end encrypted data sharing and git hosting.

1 Introduction

2 Threat Model

In FOKS, we consider a threat model similar to that of the Keybase [1], SEAMLess [2] or CONIKs [4] systems. The high level north star is end-to-end secrecy and integrity. Only the clients at the edges of the system should be able to decrypt important data, and only those clients can make authorized changes to the data. Of course, multiple devices per user and mutable groups complicate the picture.

We assume that clients are trustworthy, and behave properly. If this assumption is violated, say, if a client is compromised by a rootkit, then we cannot offer any guarantees.

Users might sometimes lose their devices. In an ideal world, hardware protections would prevent whoever recovered the device from accessing the device's private key material. In the case of hardware keys (like YubiKeys), or backup-keys written on paper, the user has less protection during comrpomise. Regardless, once the user revokes the lost device, keys should rotate so that data is secure going forward (this property is known post-compromise security). In some cases, past data might be safe from the attacker (this property is known as forward-secrecy). but the specifics depend on the trustworthiness of the server (see below). Similarly, revoked keys on lost devices lose their signing power, and other devices will not accept their signatures going forward.

The threat model is here is similar but not exactly the same as Signal's and WhatsApp's, because our applications feature persistent (rather than ephemeral) data. If a new user joins an existing group, or if a user adds a new device, they should be able to access old data, which might be required to reassemble the shared resource. For instance, when Alice adds Bob to a git project, Bob should see all past commits in the commit history, otherwise the application will break. Thus,

we can't guarantee foward-secrecy, since lack of forward secrecy is needed for the application to function properly.

In FOKS, clients pick their servers. They might select for servers that are generally aligned with. They can run their own servers, or pick from third party hosting providers. Users should assume that servers are generally trustworthy, but might suffer comrpomises from time to time. For instance, servers might be running on cloud infrastructure, and the underlying storage, network, or computation might be compromised. Insiders or state actors might have privileged access to the underlying infrascture.

If servers behave honestly, the FOKS system works securely as expected. If servers behave maliciously, they can deny access to data through a variety of mechanisms: they can go offline, they can withoold data, or they can subtly current server-resident data to confuse clients. In this last case, the system's security design should prevent the clients from leaking secrets or accepting unauthorized changes to data. But as in the other more obvious cases, the clients will lose access to their data.

When servers are behaving honestly, they can provide clients with forward-secrecy. That is, if honest servers throw away data encrypted with old keys, an attackets with access to private keys cannot rcannot ecover past data. This property is an improvement over that offered by the Keybase system, which assumed the worst in the case of a comrpomise. If we assume on the other hand that an attacker who steals a private key operates in cohoots with the server, then we cannot offer any guarantees about forward secrecy.

Servers do not trust each other. If one server becomes corrupted, it has no bearing on the other servers in the system. In other words, we assume attackers can stand up their own servers, since anyone in the system can do so.

3   Design

FOKS is a classic client-server system. At a hight level, the clients manage private keys, and the server manages public keys, encryptions of, shared secret keys, and encrypted data. Users generally trust their servers to be online, available and not to intentionally sabotage agreed-upon protocols.

3.1   System Architecture

Much like HTTP or STMP, FOKS clients communicate with one or more servers, depending on where users have accounts. They can safely ignore the other servers in the system. Most communication is between client and server, and there is little if any server-to-server or client-to-client communication. This property simplifies protocol upgrades and network configuration.

Each client can speak for many users, as users can have accounts on different servers, or several accounts on the same server. By analogy, an email client can server multiple emails accounts for the same user concurrently, say one for work and one for personal use. Or a web browser might have different personae (with different cookies, preeferences, passwords and history) for the same user.

Each of the users can of course have multiple devices, like a desktop, a laptop, a phone, and a YubiKey. Additionally, users can have "backup devices", which can be writted down on paper and stored in a safe place. The system recommends at least two devices to prevent data loss. That is, these devices have private keys that decrypt data, and the loss of the last key prevents decryption of the data. Obviously there is a trade-off here: the more devices, the more likely the user will lose one, or have one stolen; the fewer devices, the more likely the user will lose all devices and therefore access to data. Some optimal middle ground exists, but varies with the users and their behaviors.

3.2   Key Hierarchy

The FOKS key hierarchy sits at the core of the system. It aims to provide users with a sequence of symmetric keys shared across all of their devices, so that they can store data encrypted with the latest key, and can decrypt (and authenticate) data encrypted with older keys when necessary. Similarly, users in a team should share secret keys that users outside their teams cannot see, allowing them to share encrypted data via untrusted FOKS servers.

3.2.1   Device Keys

When a user sits down at a FOKS client to signup or provision a new device for an existing account, she first creates a new key-pair specifically for that device. The private key never leaves the device. She shares the public key with the FOKS server, who eventually selectively shares it with user users. We detail the exact cryptography in Section 3.8.

Hardware keys that support the PIV protocol (like YubiKey version 5 and later) can also be used as device keys. These devices get randomly-generated private keys in the factory, written to one of 20 possible "slots." FOKS users select a slot to use, and the client sends the corresponding public key to the FOKS server. Signing and decryption operations happen on the device against the chose slot.

3.2.2   Per-User Keys (PUKS)

Every user on the FOKS system has one of more per-user keys, or PUKS. A PUKS is a randomly-generated key-pair whose private key is encrypted for each of the device public keys. This way, all current devices can access the current PUK secret key, and perform decryptions or signatures for the current PUK public key. The client makes a new PUK every time the user revokes a device. The system encrypts the old PUK secret keys for the new PUK secret key. This way, a device that has access to the latest PUK can get access easily to all prior PUKs.

Once the PUK sequence is estblished, the system has a convenient way to encrypt a data for all of the user's device — it simply encrypts the data for the user's latest PUK.

3.2.3   Per-Team Keys (PTKs)

Each team has a sequence of per-team-keys, or PTKs, which are analogous to PUKs for users. Upon creation, a team gets a new random PTK. The client performing the creation sends the public part of the PTK to the server. The private part of the PTK is encrypted for each member's latest PUK, and therefore is available on each of the user's devices.

As with PUKs, data that the team shares is encrypted for the team's latest PTK, and all members can decrypt it. As we will see in Section 3.6, teams can join other teams, but the key hierarchy works just the same. When team A joins team B, the secret part of team B's PTK is encrypted for team A's latest PTK, so that all members of team A can decrypt B's PTK, and therefore, all of B's encrypted data.

3.3   Key Roles

FOKS has a notion of a "role" for device keys, PUKs and PTKs. The roles are: owner, admin, and reader, but reader keys have a "visibility level" that varies between -32768 and 32767. There is a total ordering among key roles, so that owner > admin > reader, and between reader keys, $k_1 > k_2$ iff k_1 has a higher visibility level than k_2.

The important property enforced is that we only encrypt PUK k for device key j if $\mathsf{role}(k) \leq \mathsf{role}(j)$, and similarly, we only encrypt PTK k for PUK j if $\mathsf{role}(k) \leq \mathsf{role}(j)$.

The idea here is that the owners of a group get to see all the keys; the admins can see the admin and reader keys; and the readers can see keys at or below their visiblility level. This configuration allows groups to have lower-privileged members, and for users to have lower-privileged devices. At Keybase, a similar but less-flexible property allows "bots" into teams, so that all the members of the teams can interact with the bots, but the members had channels to communicate that the bots aren't privvy to. For now, all user devices are at the owner role, but we plan to relax this requirement in the future.

3.4   Data Structures

We now have some basic motivation as to what the key system ought to achieve. It ought to allow groups of devices, groups of users, or groups of users and teams to share a secret encryption key. From there, they can share data encrypted (and authenticated) with that key. But the question becomes, how are users formulated from devices, and how are teams formulated from users so that only desired members are in the group, especially if the server behave maliciously?

For instance, a malicious server might fool a user into encrypting secret data for an invalid device, or team administrator into encrypted data for an invalid user.

3.4.1   Signature Chains

FOKS uses the same mechanism as Keybase here — the signature chain (or sigchain for short). The sigchain is a series of signed statements that form a cryptographic chain, meaning they can only be replayed in the intended order. Replaying the chain allows a viewer to confirm the chain appears how the author intended and wasn't tampered with, even if the set of signers varies over time. Of course, signers do vary over time as users add and remove devices, or as they add and remove members from teams.

Each user (and team) gets its own sigchain. The sigchain keeps an indellable record of which keys can update the chain, and which PUKs or PTKs are currently active for the user (or team).

Users   The first link in a sigchain is called the "eldest" link. For user sigchains, the first device generates this link, generates the first PUK, and then computes a signature over the following data:

A.1 The hash of the previous link in the chain (nil for the eldest)

A.2 The current sequence number of the sigchain (which is 1 for the eldest link)

A.3 A commitment to the next random tree location (see Section 3.4.5)

A.4 The current Merkle root hash (see Section 3.4.3)

A.5 The user's ID and the server's host ID (see Section 3.4.4)

A.6 The user's new PUK public keys

A.7 The user's new device key

A.8 A "subchain tree location seed commitment" (see Section 3.4.6)

A.9 A cryptographic commitment to the user's username (see Section 3.9.1)

A.10 A cryptographic commitment to the user's device name (picked by the user)

A.11 The role of the new device (currently always owner).

A.12 For Yubikeys, a public "subkey' (see Section 5.3)

The client computes nested signatures first by the new PUKs introduced in Step A.6, and lastly by the user's device key. (Recall that sometimes several PUKs can be introduced at once due to the different possible device roles). The client uploads the whole package as the user's eldest link.

Subsequent links proceed in largely the same way, with a few minor differences. The previous hash (A.1) is the collision-resistent hash of the package uploaded in the previous step. In some cases, like device addition, new PUK public keys (A.6) do not appear. In these cases, no signatures with PUKs are required.

For any link in the chain, a set of devices is authorized to make further updates to the chain. After the first link, the set contains only the first device (sometimes called the "eldest" device). A link can either add a new device, or revoke an existing device, updating the set of authorized devices accordingly. When clients upload new chainlinks, the server enforces valid signatures by authorized devices. When users replay this chain, they perform the same check. This simple mechanism ensures the server can't introduce a bogus device.

Teams A team chain link contains the following fields, many of which are analagous to user chains:

B.1 The hash of the previous link in the chain (nil for the eldest)

B.2 The current sequence number of the sigchain (starting at 1)

B.3 A commitment to the next random tree location

B.4 The current Merkle root hash

B.5 The team's ID and the server's host ID

B.6 The user (or team) ID, host ID, and PUK (or PTK) of the actor making the change

B.7 New PTK public keys

B.8 A set of membership changes

B.9 A "subchain tree location seed commitment"

B.10 A cryptographic commitment to the team's name (optional if not changing)

B.11 The team's "index range" (see 3.6.3)

Since teams can contain both users and other teams, the actor creating or modifying the team can be either a user or a team. In FOKS, a *party* refefes to someone or something that can be in a team, so either a user or a team. In field B.6, the link contains the unique identifier of the party (which is the user or team ID plus the host ID), and also the key making the change. For users, this key is the user's latest PUK at the owner role. For teams, it's the team's latest PTK at the desired source role. That is, consider a teams T where users a and b are owners of T, c is an admin and d is a reader (at visibility level 0). If T creates a new team U with source role of owner, then only users a and b will have access. If T creates the new team with source role of reader, then all users will have access.

FOKS clients and servers enforce these access controls with the key hierachy. In the case of the owners of T creating U, T's owner PTK appears in field B.6 and performs the signature over the chainlink. As T creates U, it makes new PTKs for U. It encrypts the secret keys of these new PTKs for the owner PTK of T. This way, everyone in the owner group of T can now access U's

PTKs. The second example follows similarly, with the readers of T getting access to U's secret PTKs after team creation.

The membership changes field (B.8) contains the following fields for each member being modified:

C.1 The "destination role": the role the member is to have in the team; for removals, this is the role none.

C.2 The member's party and host IDs (a party ID is a user ID or a team ID).

C.3 The source role: the role the member has in its current party.

C.4 The member's public PTK or PUK from its current party.

C.5 The generation number of that PTK or PUK.

C.6 A commitment to a "team removal key" (see Section 3.7.1)

In the case of team creation, member addition, member role upgrade or downgrade, the role in field C.1 is the new role in the target team that the member has after the change is applied. In the case of removal, the role is none.

Note that in field C.4, the public PUK (or PTK) appears directly in the chain, in addition to the ID of that party. Admins and owners are later allowed to make team modifications, and these are the public keys that will sign these modifications. Team readers in particular might lack the permission to load the chains of these users and teams directly, so it's crucial the keys appear directly in the team chain. See Section 3.9.2 for further details about sigchain visibility.

As alluded to above, owners have ultimate control over the team. They can add and remove members, add other owners, downgrade owners to admins, etc. Admins have more limited control; the have similar control over admins and readers, but cannot upgrade admins to owners, introduce new members as owners, remove existing owners or downgrade existing owners. Readers cannot make any team modifications but can of course read team chains, and can access data protected by the reader's PTKs at their level and below.

Teams, unlike users, can include members located on different servers. Above, in item C.2, we include the host ID of the party in the membership change. Remote members cannot be admins or owners, but can be readers. This configuration allows for convenient data sharing across federation boundaries, but simplifies team management relative to an alternative system where remote members can be owners or admins.

Parties making changes to team sign new team chain links much the same way as users sign user chain links. First, all new PTKs sign the chain link, and then the acting party's latest PUK or PTK signs the chain link. This PUK or PTK must be the exact key advertised earlier in the chain in the case of link 2 and above. The eldest link is essentially self-signed.

3.4.2 Sigchain Playback

Whenever a client on the time interacts with another party, or even itself, it begins by playing back that party's sigchain. The client connects to the party's home server, downloads any new links it hasn't seen, and ensures the new links play back cleanly on top of links it previously cached. The rules for playback are largely implied by the discussion above, such as:

- All chains start at 1, and have ascending sequence numbers.
- All links should contain the hash of the previous link.

- All links should have valid signatures by devices, PUKs or PTKs that the chain itself authorized. Eldest links are essentially self-signed.

- The role restrictions described in Section 3.4.1 are obeyed in team chains.

Once playback succeeds, the client has a set of devices keys, PUKs and PTKs that can speak on behalf of the party. It also has rosters of devices of members who share that key. From here, useful application work can take over. For instance, in the case of file sharing (whether via Git or via a KV-store interface), a key derived from the most recent PUK or PTK serves as the symmetric key for authenticated encryption. Whenever the PUK or PTK rotates to a new generation, future encryption should use the new key.

3.4.3 Merkle Tree

Section 2 describes the FOKS threat model, and we assume the server can behave maliciously. Sigchains prevent a server from creating new chainlinks out of whole cloth and interleaving those with legitimate links. The protection simply is that the server does not have the private keys the parties use for signing. We assume those never leave the devices they are made on. By these mechanisms, the server cannot extend a chain or replace an inner link. The server can of course create a brand new chain, and might attempt clients into using this chain rather than the one the users intended. This case of "mistaken identity" is one of *naming*, which we cover in Section 3.4.7.

The server can nonetheless *withhold* links at the end of a chain, or show some clients the full chains, and other various subchains, with the hope of forcing them to "fork" the sigchain into 2 incompatible directions. We need a mechanism to force the server's hand into showing a consistent chain tail to all users who request it. This is where the Merkle tree comes in. FOKS servers expose a Merkle Tree that forces the server to commit to a coherent state of the system, across all parties, preventing selective rollback of sigchains.

FOKS uses a Merkle Tree strategy that is a hybrid of Keybase's [1], CONIK's [4] and SEAMLess's [2], but with a simple novel mechanism introduced to obviate the need for pseudo-random functions. The Merkle tree stores each party's sigchain tail at a leaf in the Merkle tree. The server computes hashes all the way up to the root node. The meta-root contains this tree root hash, a pointer to the previous version of the tree (called the previous "epoch"), a logarithmicly-sized set of pointers to previous roots further back in history, and also the tail of the server's hostchain (see Section 3.4.4). The hash of this meta-root is new a summary of the entire system. The server attempts to publish roots immediately after every sigchain update, but sometimes batches them for efficiency. In general, roots should be produced no more than 15 seconds after a sigchain update to keep clients responsive (since sometimes they need to wait for updates).

For the party p, at sequence number i, the leaf node is a key-value pair, where the key is $H(p, i, t, r_i)$, and the value is the hash of the signature of the last link in the sigchain (the ith). t is an enumerated value that specifies which type of chain for p this is. As we will see in Section 3.4.6, each party has multiple chains. r_i is a random value that was generated when the $i-1$st link was published. The $i-1$st link contains the value $H(r_i)$ in the field B.3 from Section 3.4.1. This simple mechanism achieves the same end as the PRF in CONIKs and SEAMLess, in that it's unpredictable in the tree where a team's sigchains is stored. This unpredictability prevents data leaks that would otherwise allow the owners of neighboring nodes in the tree to deduce when p's chain advances.

When a client requests a sighchain for party p, the server returns $i+1$ paths down from the root of the tree, if the sigchain is i links long. The first i point to leafs as described above, and the last is a proof that the $i+1$st link does not exist. For each path, the server returns neighbors necessary to trace the path back to the tree root. The server also returns the sequence of r_is, so

the client can verify they match the commitments in the sigchain. Note the first link location is not randomized with r_1, since there was no 0th link to commit to r_1. However, this link is already essentially random since the party's ID p will appear random at the first link.

3.4.4   Hostchains

Servers maintain hostchains so they can manage and rotate their signig keys, DNS names, and TLS keys. Like team and user chains, hostchains form a cryptographic chain, ensuring they can only be replayed in the intended order, even if modified in transit. Chain links have sequence numbers and contain the cryptographic hashes of previous links. When an administrator creates a new server, they first create a *hostkey*, a signing key-pair. This public key becomes the host's ID. The first chainlin contains this hostkey and several subkeys, one that serves as a TLS CA for the server, and one used to sign *zonefiles* for the server. The *zonefile* contains the DNS names for the server's various services (see Section 5.4). Subsequent chainlinks can change any of these keys or subkeys, as long as they are signed with keys valid up until that point. clients play these links back to map host IDs to DNS names as they establish connections to new servers.

this basic architecture to the problem of key distribution, rather than the data those keys might secure. Many other projects have riffed on this, from CONIKS [4], to the SEAMless [2] work out of Microsoft Research, to the widespread adoption of Key Transparency Signal, WhatsApp and iMessage. The question of federation has largely been ignored, as these sytems all shared the basic architecture of a single upstream server.

8 Conclusion

References

[1] Keybase. Available at `https://keybase.io`.

[2] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1639–1656, 2019.

[3] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[4] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.