The Federated Open Key Service (FOKS)

Maxwell Krohn (max@ne43.com)

March 14, 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:

1. The hash of the previous link in the chain (nil for the eldest)
2. The current sequence number of the sigchain (which is 1 for the eldest link)
3. A random commitment to the next tree location (see Section 3.4.4)
4. The user's ID and the server's host ID (see Section 3.4.2)
5. The user's new PUK public key
6. The user's new device key
7. The current Merkle root hash (see Section 3.4.3)
8. A "subchain tree location seed commitment" (see Section 3.4.5)
9. A cryptographic commitment to the user's username (see Section 3.9.1)
10. A cryptographic commitment to the user's device name (picked by the user)
11. The role of the new device (currently always owner).

12. For Yubikeys, a public “subkey’ (see Section 5.3)

The public portion of the PUK computes this signature first; then the device key computes a signature over the payload and the previous signature (signed with the PUK). 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 (1) is the collision-resistent hash of the package uploaded in the previous step. New PUK public keys (5) do not appear on device addition; and likewise, PUK signatures are only performed when new PUKs are introduced, so would not be present on device addition.

Teams

3.4.2 Hostchains

3.4.3 Merkle Tree

3.4.4 Location Hiding

3.4.5 Subchains

3.5 Provisioning

3.5.1 Device-to-Device

3.5.2 Yubikey-to-Device

3.5.3 Device-to-Yubikey

3.6 Teams

3.6.1 Invitation Sequence

3.7 Key Rotations

3.7.1 Removal Keys

3.8 Cryptographic Primitives

3.8.1 Key Derivation

3.8.2 Yubikeys

3.8.3 PQ-KEM and PIV Support

3.9 Privacy

3.9.1 Blinding and Commitments

3.9.2 Sigchain Visibility

3.10 Secret Key Management

3.10.1 Secure Enclaves

3.10.2 Passphrase-based Management

3.11 Beacon Server

4 Applications

4.1 Key-Value Store

4.2 Git

5 Implementation

5.1 Snowpack Protocol Language

5.2 Entities and IDs

5.3 Authentication

6 Evaluation

7 Related Work

The initial inspiration for FOKS is the SUNDR project [3], which first originated the idea of a fork-consistent blockchain of edits facilitated by a untrusted server. Like Keybase [1], FOKS applies 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.