Swish:
Usable, Privacy-first Collaboration

January 2021

1 Overview

This document outlines a threat model, a set of desired security properties, and a high-level system design for Swish - a private, end-to-end encrypted collaboration platform.

In its most basic sense, we propose a system that ensures that no sensitive information (including documents, document titles, and messages) is ever stored, seen, or processed in plaintext by our servers. This is achieved using end-to-end encryption as well as additional safeguards, including robust authentication methods, out-of-band key verification, and two-step verification.

2 Threat Model

Swish's threat model assumes that adversaries may access and potentially modify any data sent over a network to or from a client (even data sent over encrypted network connections).

We assume the server is "honest but curious:" We assume that Swish's servers will not deliberately deny access to data, and will not serve compromised client applications over the web. For the most security-conscious users, this second component of the threat model is supported by using subresource integrity; out-of-band public key verification (see further sections of this document); and other technical properties, with additional verification methods possible and in development.

2.1 Desired Properties

1. **End-to-end encryption of all sensitive information**: The contents and certain metadata (including title, time created, and last modified date) associated with every document created or uploaded by a user is only visible to the user or shared collaborators. A generic "document" may be a rich-text file, a spreadsheet, a group conversation, or other types.

2. **Resistant to man-in-the-middle attacks**: Even without a fully trusted communication channel between the server and clients, our model will reveal no sensitive data.

3. **Resistant to user abuse attacks**: As we expand and improve our services, we are particularly sensitive to user abuse as a threat vector. One user abuse attack includes sharing unwanted information or documents with a particular target user.

4. **Resistant to impersonation attacks**: Our model must be resistant to impersonation of any of the parties (server or clients).

5. **Makes phishing difficult**: Given an initial online release, our model should prevent adversaries from compromising user accounts even if their password is compromised, including using 2FA, and, in the future, hardware tokens.

6. **Uncompromisingly usable and beautifully designed:** Swish is significantly more private than the most widely-used collaboration platforms. Given this, building a usable product with these security properties is critical to ensuring that users do not switch to less-secure alternatives due to poor usability.

3 System design

3.1 Overview and encryption protocols

Public-key authenticated encryption serves as the fundamental basis of our security model. Under this schema, each user is issued a long-term public signing key and a medium to long term public key for encryption. Each public key is associated with a corresponding private key generated using Curve25519. We are currently using tweetnacl-js as our encryption library for both asymmetric public-key authenticated encryption (tweetnacl.box) and secret-key authenticated encryption (tweetnacl.secretbox).

While a user's encryption and signing public keys can be freely shared and used to securely send or verify information, all private keys - which can decrypt information or generate signatures - must be kept private. We discuss how this is done in the following section.

3.2 Login, creating accounts, and private key storage

For a new user Alice, our account creation and login system is designed to:

1. Keep Alice's private keys safe
2. Ensure sensitive information - such as Alice's password and private keys - are never sent beyond her browser window, and
3. Resist brute-force and dictionary attacks.

Account creation is as follows:

1. Bob enters his email and generates a secure password (more than n digits, upper/lower case letters, numbers, and special characters). Random encryption and signing keypairs are generated (tweetnacl.js) for Bob's account in-browser.
2. On the browser, we run Argon2id to derive a symmetric key from Bob's password. Using HKDF, this symmetric key is used to generate one symmetric key for login (using the Secure Remote Password protocol), and another symmetric key for encrypting Bob's secrets (i.e. private keys). This second key is called Bob's `password_derived_secret`.
3. We use Bob's `password_derived_secret` to encrypt sensitive data associated with Bob's account; this encrypted data is called Bob's `encrypted_user_data`. This includes Bob's private keys but not his password. This encrypted information is stored by our server but can only be decrypted with Bob's `password_derived_secret`. The next time Bob logs in, Bob downloads his `encrypted_user_data` from the server, then decrypts the `encrypted_user_data` in-browser by using the `password_derived_secret`. The `password_derived_secret` never leaves the browser.

In this system, Bob's public keys are publicly visible, while his private keys are encrypted end-to-end. His password and `password_derived_secret` are never stored, not even as encrypted data. Bob's `password_derived_secret` and password are also never sent over any network, even as encrypted data.

We use the secure remote password (SRP) protocol to authenticate user login. After Bob is authenticated using SRP, the server sends Bob's `encrypted_user_data` as well as a signed JSON Web Token (JWT) to indicate that Bob has properly logged in within a certain amount of time. In our security model, the time-limited JWT is used for "read-only" operations, including downloading encrypted documents. This is discussed in further detail below.

3.3 Two-factor authentication (2FA)

On login, a user can setup two-step authentication to add an additional level of security to their account. Currently, we support two-step authentication using a one-time password from the following authenticator apps: Authy, Google Authenticator, and Duo Mobile.

If Alice chooses to set up two-step authentication, she generates a secret using an authenticator library (`otplib`) in-browser; this is used to display a QR code to her. Alice is then prompted to register her device using the QR code, read an OTP from the device, and enter that OTP once. If the code is successfully entered, Alice's 2FA secret is sent to the server over HTTPS.

When the server receives the 2FA secret, it encrypts that secret using secret-key authenticated encryption (tweetnacl.secretbox), and stores it in Swish's database.

This 2FA key is later decrypted by the server and used to check Alice's one-time password when she tries to login again. Note: The DB provider (currently AWS RDS, either directly or managed by Heroku, adds an additional layer of symmetric encryption with key rotation as described here). WebAuthN can be integrated as an additional verification method here.

In end-to-end encrypted systems, two-factor authentication increases reliance on the server to store and authenticate access to accounts. However, as Swish's initial release is accessed via a web browser, 2FA offers greater account access protection in the context of an honest-but-curious server.

3.4 Account Recovery and Password Changes

When users forget their password, it's convenient for them to have a way to recover their account and reset their password. This is achieved using a recovery key (a symmetric key similar to the `password_derived_secret`). A user can enable account recovery in their settings, which generates a recovery key. The recovery key is used to encrypt the user's private data (i.e. private keys); this encrypted user data is stored on the server, along with a hash of the recovery key.

When a user requests account recovery, their identity is first verified through email. The server sends an email to the user containing a random six-digit passcode which they can use to prove access to their email account. After this step, the user enters their email, recovery key, and a new password. The client hashes the recovery key, which is sent to the server. The server checks if the hash matches the stored recovery key hash, and that the client retains the correct email code. If both match, the server sends the encrypted user data to the client, which then decrypts it with their recovery key. Note that storing a hash of the recovery key is not like storing a hash of a user's password; while a password may be predictable and reused, the recovery key is a randomly generated symmetric key.

From this point, the process is similar to choosing a new password when an account is created - a `password_derived_secret` is derived from the new password, which is then used to encrypt the user's private data, and the encrypted user data is uploaded to the server, replacing the previous data encrypted with the old `password_derived_secret`. In future implementations, account recovery can be accomplished using n-of-m Shamir Secret Sharing to maintain end-to-end encryption while increasing usability. For example, using 2-of-3 secret sharing, one secret share could be stored in the user's device, another stored by Swish's server, and another provided for "paper" storage. In this model, usability may be significantly higher for many use cases.

4 Documents

A Document is the fundamental unit of collaboration on Swish. When interacting with our platform, users create, share, and save documents of different kinds, including rich text documents, spreadsheets, PDFs, and groups or direct conversations.

Swish's document model keeps both document metadata and document contents private to shared collaborators (and hidden from all others, including Swish). Document metadata includes information including the document's title as well as the document's created at/last modified at times. Document contents include the data stored inside a document (such as the content of a text document or spreadsheet). Both metadata and contents are end-to-end encrypted. In the next sections, we describe how document end-to-end encryption is maintained, including across sharing, unsharing, and link sharing.

4.1 Document encryption model

Every document is associated with a short-term symmetric `session_key`. The `session_key` is used to encrypt all document contents and metadata that are stored on the server. In order to support real-time collaboration and simple sharing mechanisms, all collaborators - say Alice and Bob - have access to the same `session_key`. However, because Alice and Bob have different asymmetric key pairs, they each have unique encrypted copies of the same session key (encrypted with `public_key_Alice` and `public_key_Bob`, respectively).

Using this symmetric `session_key`, we can outline processes for creating, editing, and sharing documents.

4.2 Opening a document d_1

To open a document, Bob:

1. Downloads the stored copy of `session_key_d_1` that has been encrypted with `public_key_Bob`. Call this key `session_key_d1_Bob`.

2. Decrypts `session_key_d1_Bob` using `private_key_Bob`.

3. Using `session_key_d1_Bob`, Bob is now able to decrypt, read, and edit the contents or metadata of d_1.

4.3 Sharing a document d_2

Reading and editing document d_2 requires access to the session key `session_key_d2`. Say Alice creates a document d_2 (and is the only shared user on d_2). Now, she wants to share a document with Charlie. Alice:

1. Sends a HTTPS request to the server to retrieve Charlie's public key. In future sections, we describe a mechanism for out-of-band public key verification to reduce trust in the network and in Swish's public key registry.

2. Encrypts `session_key_d2` with Charlie's public key, and sends this encrypted key back to the server. Alice also includes a signature on the encrypted key in the request.

3. When Charlie signs in, the server sends Charlie all document IDs and session keys - now including `session_key_d2_Charlie` - that he can access.

4. Charlie decrypts the session key, queries the server for the metadata and contents of d_2, and can now read the document.

5 Real-time collaboration

Real-time collaboration among shared users on a document is end-to-end encrypted using the document's session key. On Swish, conflict resolution is performed using a CRDT, which allows each collaborator to maintain an in-browser copy of the document and perform change resolution as live document updates are received from other users.

To initiate a collaboration session using the CRDT, two shared users open the document and create Websocket connections to the Swish server. They include the unique identifier of the document for collaboration. At this point, both users begin broadcasting encrypted updates to the CRDT through this Websocket connection to other users listening for collaboration updates. Each user decrypts the CRDT updates (using a document's symmetric key) and applies these updates to their local copies of the document data structure.

6 Building a filesystem

In addition to a document's metadata and contents, we add the fields parent and children to support nested documents and filesystems. This requires changing behavior for sharing and unsharing, particularly around scalability for large or many documents. Swish's collaboration models are designed to efficiently scale to large numbers of documents and users.

While the document's title, modified time, and content remains private, the document identifiers stored in parent and children fields are known to the server. While the entire filesystem is not authenticated, each parent/child relationship is authenticated with a signature. Furthermore, even though these fields express some hierarchical relationship among documents, these relationships are only expressed in document UUIDs (and do not reveal document titles, for example).

6.1 Sharing a document d_2 (filesystem version)

Now, consider the case where Alice wants to share a user Bob on a document d_2, where d_2 has child documents. In this case, instead of sending a single encrypted session key, Alice launches a set of sharing requests to share d_2 and (recursively) all of its child documents.

6.2 Unsharing a document d_3

Given the threat model defined above, we enforce that unsharing a user from a document also requires changing the document's session key. If the session key were not changed, it is conceivable that an adversary (either an unshared user or a malicious party with access to a compromised account) with access to an old document session key could intercept and decrypt network traffic related to a document. To unshare Bob from document d_3, Alice:

1. Generates a new `session_key` for d_3,
2. Re-encrypts metadata and contents with the new key,
3. Sends a request to the server, which replaces the encrypted metadata and contents with the new versions and deletes Bob's stored copy of the session key to access document d_3.

We can batch multiple unshare operations together on the client that can be individually processed by the server.

6.3 Unsharing a document d_3 (filesystem version)

In order to support scalable unsharing in the context of our filesystem, we introduce the concept of expiring access to a document. As described below, unsharing via expiration allows us to significantly increase scalability for unsharing from large numbers of documents. It also allows us to maintain secrecy of documents among shared collaborators.

6.4 Expiring Access

Say Alice wants to share user Dorothy on a document with an expiration date. User Dorothy's access to d_4 can be expired as follows:

1. Alice - who is going to unshare Dorothy - sends a request to the server to expire Dorothy's access to d_4 with a given authenticated `expiry_date`
2. After checking permissions, and ensuring `expiry_date` $>$ `today_date` $+ \delta$, the server adds an (authenticated) `expiry_date` field to Dorothy's permission entry.
3. When d_4 is modified or re-downloaded by any shared user U after Dorothy's permission expires, U's client recognizes that the permission entry has expired and re-encrypts document d_4 (as in typical unsharing). Under this model, Dorothy is blocked by the server from accessing d_4, and all future versions of d_4 will be encrypted with a new key. This maintains our desired characteristics of end-to-end encryption where only shared collaborators can decrypt document contents. Although there may be a period of time between "expiring" Dorothy's access and another user re-encrypting the document, the document will remain unchanged during this period, and Dorothy will be unable to access future modified versions.

Now, to unshare d_3 in the case where d_3 has many child documents, Alice sends two requests:

Unshare all child documents, and their document children recursively, via expiration. Unshare the root document with a typical unshare (and re-encryption).

Expiring access introduces enormous scalability and efficiency improvements to unsharing documents, particularly in consideration of two alternatives wherein a client would either:

(a) Re-encrypt every unshared document (unfeasible), or

(b) Not rotate document `session_key`s (undesirable given security model wherein only shared collaborators should have access to a document's `session_key`)

In the future, additional security properties may be added to further develop expiring access.

6.5 Link sharing a document d_2

Link sharing is a critical usability feature for modern collaboration. Today, link sharing is one of the most poorly-regarded sharing mechanisms given unknown security properties and the risks of unintentional disclosure. Swish's link sharing mechanism maintains end-to-end encryption such that not even Swish can gain access to a document's URL. In this section, we discuss **"security-first links."**

Security-first links store information in the URL that remains private to the client (using a URL fragment) and employ an authentication technique extremely similar to the user login method documented above. In order to generate a sharable link for d_2, Alice:

1. Generates a random key `link_key_d2`, and encrypts `session_key_d2` with `link_key_d2`, and encrypts the `link_key` with `session_key_d2`. It is critical that Alice encrypts the link key with the session key so she can recover the document link when redownloading the document (for example, after logging out and logging back in).

2. Using `link_key_d2` as a "password," Alice generates a salt and verifier used for Secure Remote Password. As in user account creation, the salt, verifier, and encrypted keys are stored by the server. However, it is impossible for the server to decrypt the `link_key` or the `session_key`.

Alice now has access to a URL link `https://<client_name>/<docID>#link_key` that can be securely transmitted to another individual, either through an end-to-end encrypted chat on Swish or another communication channel. Note: As noted above, Alice encrypts `link_key_d2` with `session_key_d2` in addition to encrypting `session_key_d2` with `link_key_d2` in order to recover the full link URL when accessing the document.

In order for Bob to access d_2 given the link URL, Bob:

1. Parses the URL to extract `link_key` and `docID`

2. Performs the SRP login process to request the salt and send a proof back to the server, which responds with the encrypted `session_key_d2`

3. Bob then decrypts `session_key_d2` with `link_key_d2`. At this point, Bob can download and read document d_2.

4. If Alice has enabled editing for the link, Bob sends a copy of `session_key_d2` encrypted with Bob's public key. The server stores this key and a new permission entry on d_2 for Bob to access in the future.

In the future, in order for Alice to access the link, Alice uses the version of `link_key` encrypted with the `session_key` to reconstruct the link URL.

7 Document chunks and expiration

In our data model, document contents are represented as an array of chunks. Each chunk individually contains a piece of encrypted data (encrypted with the document `session_key` and authenticated by the user encrypting the individual chunk). Splitting a document into chunks can yield significant efficiency gains depending on the size and type of document stored.

The sequence of chunks is authenticated to prevent unintentional reordering or omission of pieces of the document by Swish servers or any malicious actor. This includes authenticating the chunk's zero-indexed sequence number and a boolean flag indicating whether chunk C_n is the final chunk in the document. When adding a new chunk, the client updates the signature on the previous chunk (C_{n-1}) to flip this boolean field (to indicate that the old previous chunk is no longer the final chunk).

To support chunks with expiring content (which could be used to support expiring documents or messages), we add an `expiry_date` property as well as an `expiring_content` field to store an expiring encrypted message. When included, the `expiry_date` of a chunk is also authenticated with the entire chunk. After `expiry_date` has passed for chunk c_a in document d_m, the server:

No longer includes the `expiring_content` of chunk c_a when clients download a copy of d_m, and Regularly deletes all expired `expiring_content` fields from the database; for example, the server will run a job every day to delete `expiring_content`.

In addition to requesting a new copy of the document from the server, the client removes a chunk from its stored copy of the document whenever it expires, just in case a user's connection to the server is severed.

8 Public key verification

Private communication requires trust in mechanisms to receive and verify other users' public keys. Swish allows other users to view and verify other users' public signing keys through a user interface for "verification numbers" - a bip39 (or alternate) encoding of another user's signing public key.

Each individual's encrypted user data (see above) includes a data structure to store public signing keys for other verified users. Given this model, when user Alice marks user Charles as "verified," Alice stores a copy of Charles' public signing key in her encrypted user data (or another end-to-end encrypted data store maintained by the server). In all future interactions with Charles (adding to a group, sharing on a document) this stored copy of Charles' public signing key is verified against the one distributed by Swish's server.