Tilig Security Architecture

Tilig Security Team

2023-01-25 (v1.0.0)

Security Architecture

The goal of this document

This document provides an overview of the architecture for our authentication and backend services. This architecture aims to be simple, modular, and most importantly, secure.

The goal of Tilig

Tilig is a source-available password manager that aims to provide security through simplicity. Our goal is to enable users to enhance their online security without having to go through complicated steps or remember and continually re-enter long passphrases. We want users to be able to securely and conveniently access their stored usernames and passwords, and share them with their friends, family, and colleagues.

Architecture summary

This is a brief summary of Tilig's architecture and design philosophy.

1. **No master password** Tilig operates without requiring users to choose a master password or passphrase. This has consequences for Tilig's architecture and security model. Even without a master password, we want to make it impossible for Tilig to get access to our user's secrets. Other password managers refer to this as a "zero-knowledge" architecture.
2. We keep Tilig's backend, clients, and infrastructure as simple as possible, both in terms of code and architecture.
3. We use Firebase for User Authentication through federated identity providers.
4. A set of services will be deployed to Firebase, together these are called the Key Service (KS). The KS is responsible for:

    1. Authenticating users through Firebase Auth;
    2. Encrypting a user's key pair, so it is safe for storage in the backend;
    3. Decrypting a user's private key, so the user can decrypt their own items.

    The Key Server is stateless; services in the KS do not maintain any persistent state, except for authentication purposes.

    The Key Service is the most sensitive part of our architecture as it deals with both authentication and unencrypted user credentials.

We trust Firebase to secure their services and prevent abuse. At Tilig, we restrict access to these services as much as possible, allowing only a select subset of personnel to deploy changes. We also minimize the amount and complexity of deployed code and rely on standard libraries and vetted dependencies whenever possible.

5. The Tilig API (TA) handles all other business logic. TA is hosted on a managed Kubernetes cluster provided by Digital Ocean. Separating the API from the KS allows for faster iteration, easier feature addition, a more conventional programming paradigm (compared to a serverless architecture), and greater hosting flexibility.

   There are a few important things to note about TA:

   1. The backend does not have any session management. Authentication is handled by the KS and Firebase Auth. The backend validates Authorization (Bearer) Tokens, but does not generate them.
   2. Secrets (tokens, credentials, encrypted secrets) obtained from the TA (for instance in the case of compromise or leakage) must not enable an attacker to compromise the KS or allow an attacker to decrypt a user's items. *Note: This means a single Authorization Token shared between the KS and the TA is not possible.*
   3. The TA only stores data that is either encrypted or of lower sensitivity.

Assumptions about client security

1. We hold users responsible for securing their own devices. We can offer additional security features, such as a biometric lock, to protect decrypted secrets. However, we cannot guard against a compromised browser or a malware-infected mobile device.
2. We use the default implementation of Firebase client libraries for authentication flows, including the defaults for short lived session tokens and long lived refresh tokens.
3. The amount of data that can be securely stored on a client depends on the security level of that client. Native clients, such as those for Android and iOS, offer stronger security guarantees for protecting stored data than browser-based clients.

Isolated Architecture

Our architecture is separated into distinct components with clearly defined responsibilities.

Firebase, Google Cloud, and Federated Identity Providers

Firebase, Google Cloud, and related services are managed by Google. The Key Service is hosted on Firebase. The federated authentication flows are also managed by Firebase.

We trust Google to correctly and securely manage these components. Tilig and Google both bear responsibility for keeping the Key Service secure through the shared responsibility model.

The architecture components hosted on or managed by Firebase are extremely sensitive, since it's the only place in our infrastructure, apart from clients, where unencrypted secrets exist (during key pair encryption and decryption) and it is where user authentication takes place. To reduce risks, we keep attack surfaces small by minimizing the amount of code running in Google's cloud and restricting (developer) access to these services.

Tilig API

Most of the business logic for Tilig resides in the API. The API is deployed on a Digital Ocean-managed Kubernetes cluster, which we provision and monitor. The API is designed as a Rails monolith that communicates with a Postgres database cluster and a Redis database cluster. The Kubernetes cluster also contains applications for ingress, network monitoring, logging, and security monitoring.

The Tilig API has a larger attack surface than the Key Service due to the amount of code, features, and dependencies. Additionally, more people have access to either the application code or the production environment, which is necessary for day-to-day operations. Therefore, we take extra care to ensure the security of this component in our infrastructure.

We reduce risks by only storing encrypted and low-sensitivity information in the API.

Clients

Client components include the Tilig web application, browser extensions, and native apps. These components run on the end-user's device; however, they are never "trusted" by the API. Authentication and authorization checks should always happen on the Key Service and the API, relying only on external attestations such as Bearer Tokens for verification.

Users are responsible for securing physical access to their own devices and ensuring the integrity of the client applications' execution environment. Tilig is responsible for adequately securing the client applications in a "clean" execution environment. This includes storing plaintext secrets in a way that cannot be stolen by another application, and only autofilling passwords on sites for which they were created or with explicit user instructions.

Architecture description

Figure 1 gives a schematic overview of Tilig's server-client architecture. The Key Service and Tilig API components are detailed below.

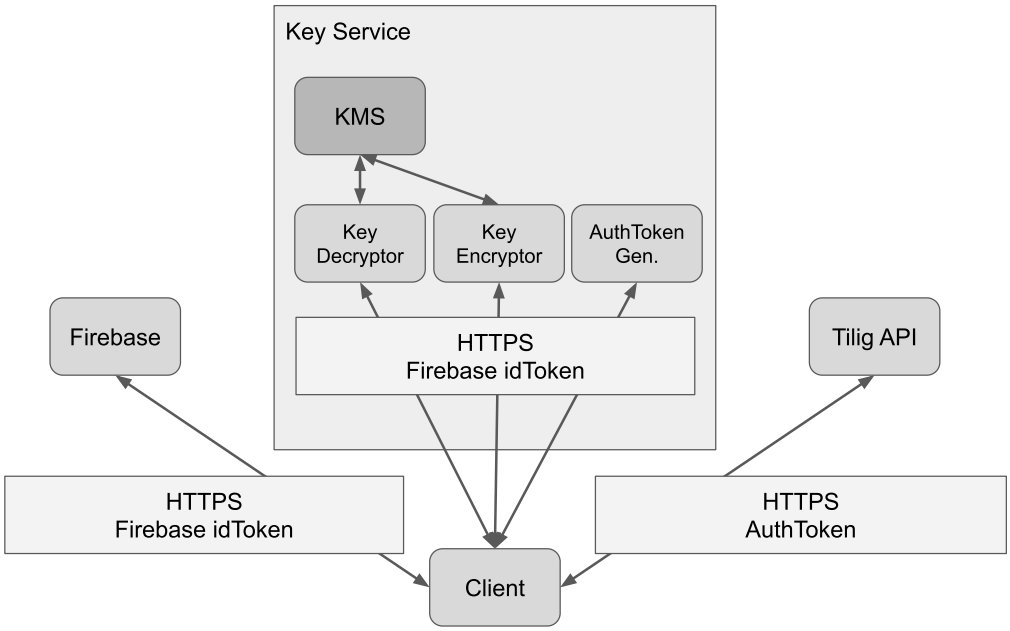

Figure 1: Server-client architecture

Key Service

The Key Service consists of three Firebase functions hosted on Google Cloud, and the default Firebase Authentication stack, with Google and Apple enabled as trusted federated identity providers at this time.

All three cloud functions will be as simple as possible, with minimal dependencies, to ensure a small attack surface. These functions do not need to be updated often.

Firebase Authentication

We delegate user authentication to Google's Firebase capabilities. Firebase Auth is responsible for user sessions, authorization tokens, refresh tokens, and OAuth token verification.

Key Encryptor / Key Decryptor

The Key Encryptor and Key Decryptor both have a single responsibility: encrypting private keys to make them safe to store in the backend, and decrypting the private key for a user when they add a new

device.

The Key Encryptor and Key Decryptor use RSAES-OAEP with a 3072-bit key and a SHA-256 digest for asymmetric encryption, and ECDSA on the P-256 Curve with a SHA-256 digest for asymmetric signing. Decryption and signing are performed by Google's Key Management Service (KMS). This means that actual decryption and signing take place outside of the control of Tilig's cloud functions. The private keys used will never leave the KMS servers, and Tilig will never have access to them, nor will any party outside of Google.

After encrypting a user's private key, meta data will be *bound* to the encrypted private key by signing the combined payload. At this time, we bind the user's Firebase ID and email to the encrypted private key. When decrypting, the Firebase ID and email will be obtained from the Firebase ID token used to authenticate to the cloud function. Before requesting the KMS to decrypt an encrypted private key, the associated meta data will be verified to be the same as that of the current user. Decryption will only be successful if the Firebase ID and email match.

Authentication Token Generator

The Authentication Token Generator cloud function can generate short-lived Authentication Token JWTs for clients with a valid Firebase Auth session. These JWTs can then be used to authenticate to the API; the API never generates Authentication Tokens itself.

If the Authentication Token expires, the client can request a new JWT from the cloud function using their existing Firebase Auth ID token. If this token is expired, which happens after one hour, the user can user their Firebase Auth refresh token to obtain a new idToken. The refresh tokens only expire when a user's Google Account email or password changes.

Tilig API

Tilig API is a Rails monolith that handles storing and securing user data. It is hosted on a Kubernetes cluster managed by Digital Ocean. The source code of the API is available to view for anyone on our public Gitlab project: https://gitlab.com/tilig/api

Users The main responsibility of the Tilig API is storing the encrypted items of users, authenticating users and providing the encrypted items to our client applications.

The Tilig API authenticates users based on an Authentication Token issued by the Authentication Token Generator of the Key Service. This token contains a user's Firebase ID and email address. The combination of the Firebase ID and email is used to identify a user.

Items Items stored in Tilig, such as login credentials, secure notes, WiFi passwords, credit cards, and other secrets, are referred to as “items” in the API. All items are encrypted on the client application before they are stored in the API. The keys used to encrypt and decrypt items are never stored in the API, and an item can only be decrypted by its creator on their own device.

Figure 2 shows the relationship between users and items. Attributes like timestamps, ids, meta data, and foreign keys have been omitted for brevity.

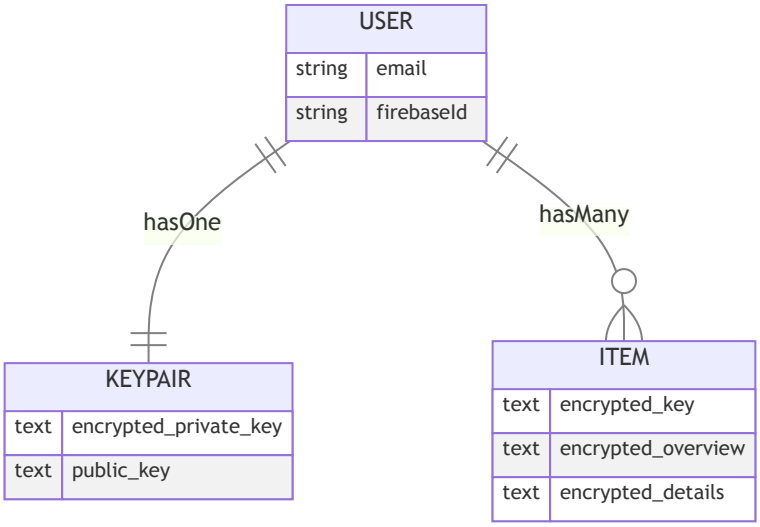

Figure 2: Users and items

Groups and sharing This section is under construction.

Client flows

This section describes the various interactions that a client can have with the Key Service and Tilig API.

Authentication

Figure 3 illustrates the authentication process for a client on a new device. It involves federated authentication from Google and Apple, Firebase Authentication, and the Authentication Token Generator cloud function.

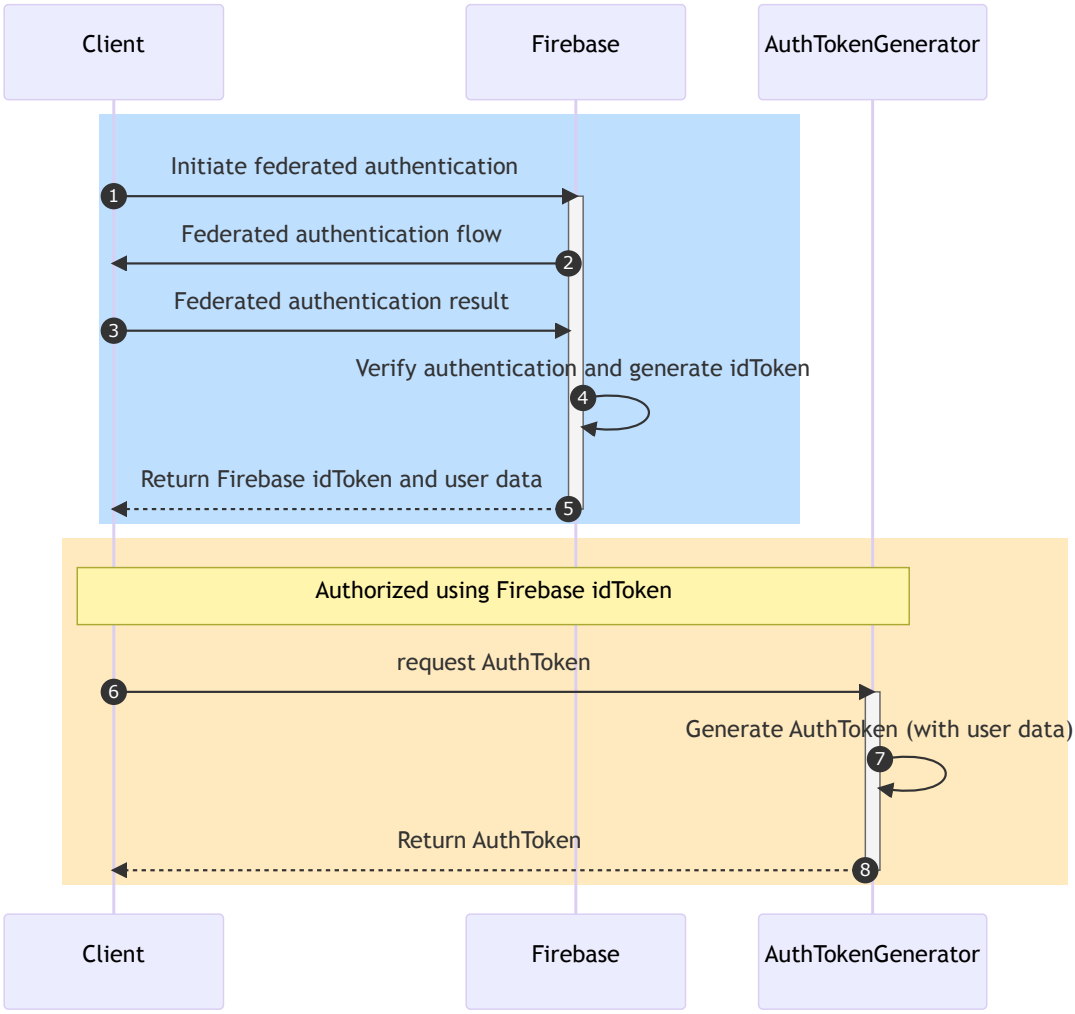

Figure 3: Authentication flow

Key pair storage

In Figure 4, we see how a user uses the Key Encryptor cloud function to encrypt their client-side generated key pair (Figure 4, Step 1) and securely store it in encrypted form in the Tilig API.

Tilig employs authenticated encryption based on X25519 key exchange, XSalsa20 stream cipher, and Poly1305 digest for all client-side asymmetric encryption operations. New key pairs are generated randomly using a cryptographically secure random number generator (CSRNG) available on each client platform. For all client-side cryptography, we use libsodium, which is based on Daniel J. Bernstein's NaCl library (pronounced "salt"). This library is available on all platforms that Tilig offers client applications for. It offers very fool-proof primitives which are hard to misuse.

Private keys are encrypted and bound to a user's meta data. The resulting payload is signed by Google's KMS. The signed payload can then be stored in the Tilig API.

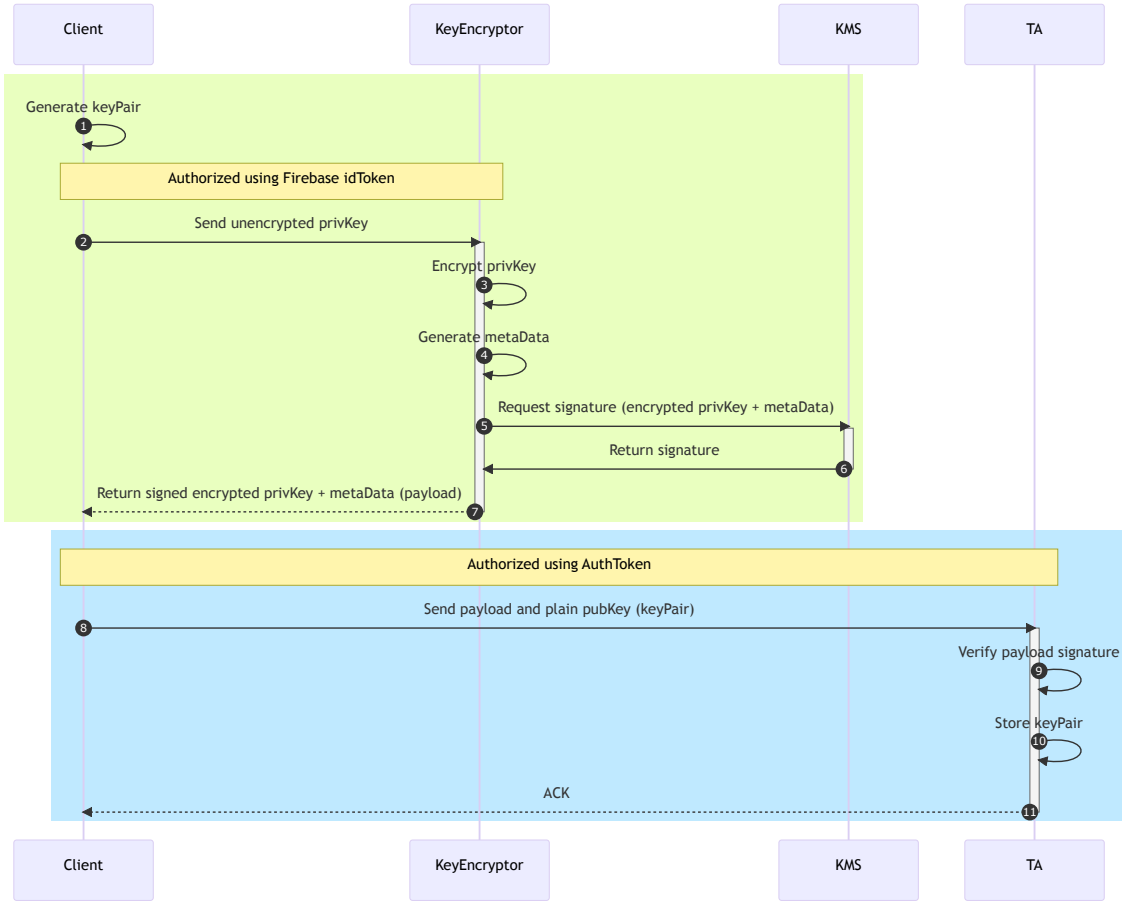

Figure 4: Key pair storage

Retrieve private key on new device

Figure 5 shows the steps a user must take to get their private key when signing in with a new device. It is assumed the user has already authenticated with both Firebase and the Tilig API.

The Key Decryptor will only decrypt a private key if it belongs to the user requesting the decryption. This is verified by comparing the meta data associated with the private key to the meta data of the authenticated user.

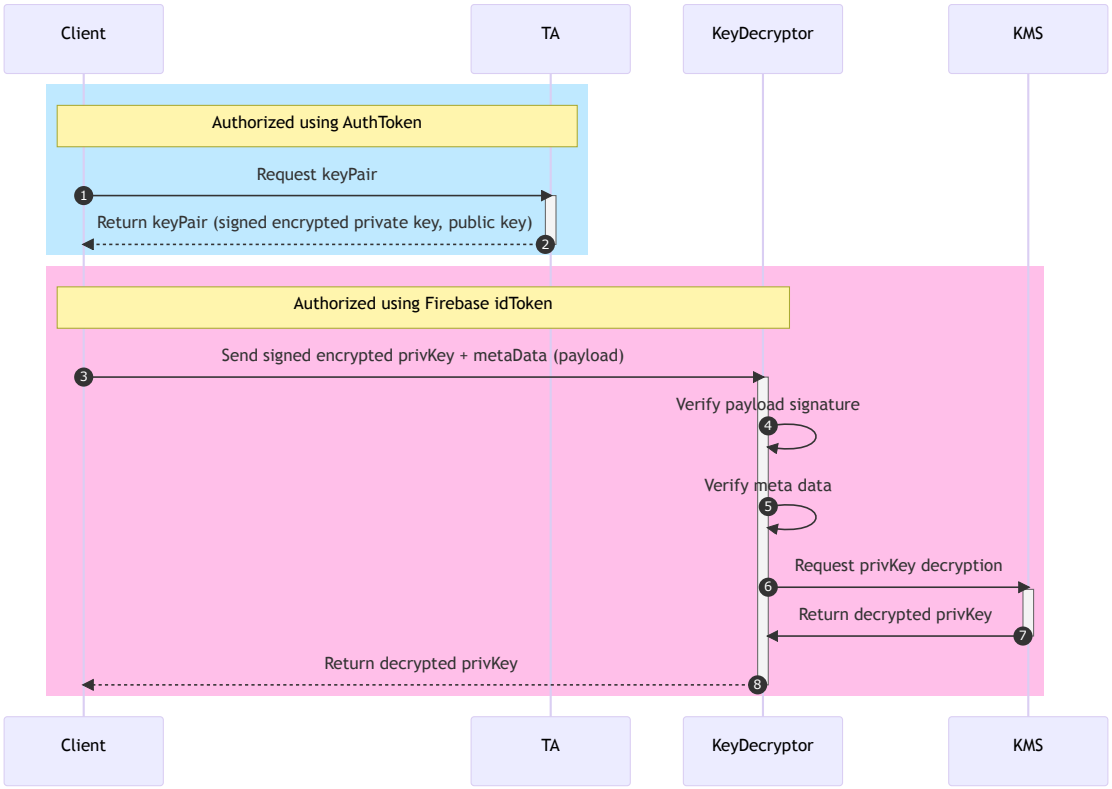

Figure 5: Key pair retrieval

Store an encrypted Item

Client-side encryption is distinct and independent from the private key encryption in the Key Service.

Tilig uses the libsodium library for client-side encryption. Each item is encrypted with it's own randomly generated key called the Data Encryption Key (DEK). Tilig uses symmetric authenticated encryption based on XSalsa20 stream cipher and Poly1305 digest. Every encryption operation uses random nonces from a cryptographically secure random number generator.

Data within an item is separated into two categories: overview data and sensitive details. Overview data is used for list display, sorting, and searching, and includes the item name, subtitles, and display information. Sensitive data, such as passwords and notes, is stored in the item details.

In Figure 6, we see how a user can encrypt an Item client-side, and store it in the Tilig API for later retrieval or synchronization with other devices. It is assumed the user is authenticated with the Tilig API.

Obtain and decrypt a stored Item

A user can use Tilig on multiple clients. When adding a new device, logging in on the web application, or synchronizing a client, the user retrieves their encrypted items from the Tilig API. After obtaining the item, the client decrypts the encrypted overview and, if necessary, the encrypted details.

In Figure 7, we can observe how a client retrieves an encrypted item from the Tilig API and decrypts it locally using their key pair and the encrypted data encryption key (DEK).

Sharing an item by link

This section is under construction.

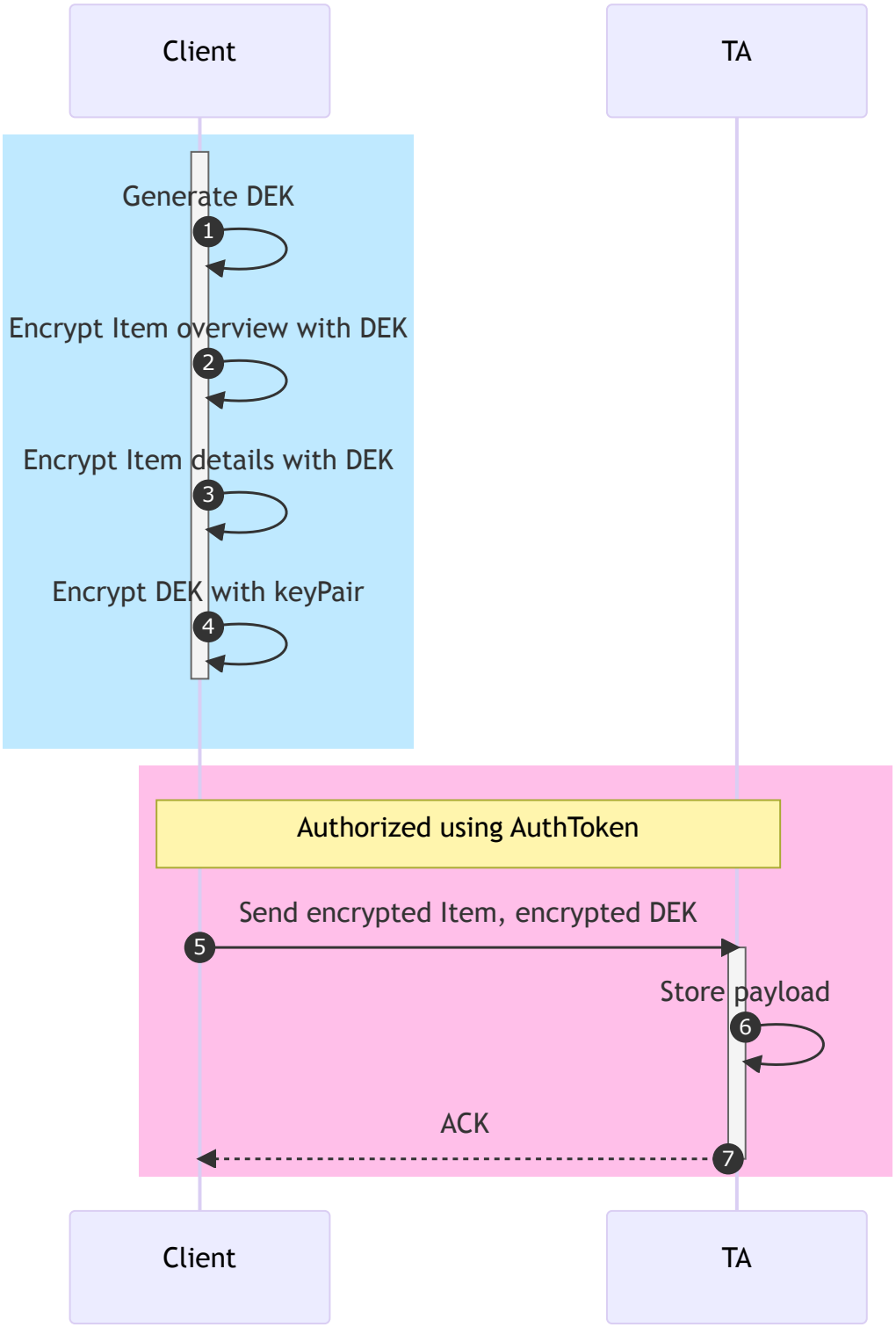

Figure 6: Item storage

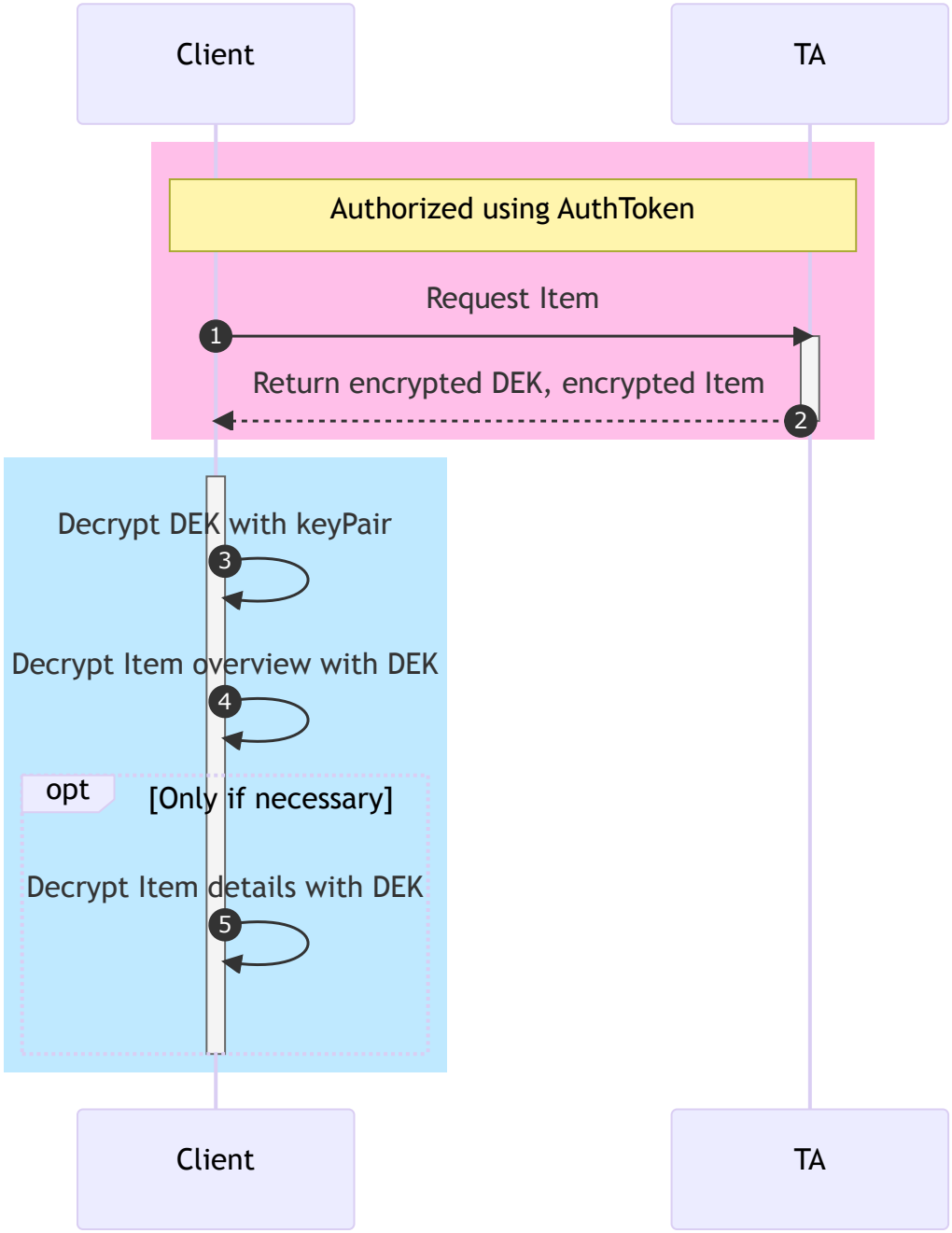

Figure 7: Item retrieval