Doctolib

A new standard for online medical confidentiality

White paper

Table of contents

Doctolib

Introduction

End-to-end encryption

> *"End-to-end encryption is a system where only the users can read the messages. In principle, it prevents potential eavesdroppers – including telecom providers, Internet providers, and even the provider of the communication service – from being able to access the cryptographic keys needed to decrypt the conversation"*
>
> https://en.wikipedia.org/wiki/End-to-end_encryption

End-to-end encryption is ideal for securing sensitive data. With end-to-end protocols, only users can access their data. It implies that nobody can access encryption keys, not even the service provider.

Examples of end-to-end encryption include Signal, in which messages and videos are encrypted end-to-end, Apple's iMessage and Health data, and password managers such as Dashlane.

End-to-end encryption allows these companies to provide products that manipulate highly sensitive information. Users can use these services knowing that the service provider cannot access their (encrypted) data. It is particularly applicable to sensitive data such as passwords, health data, or otherwise sensitive communication.

There are, however, many limitations that prevent end-to-end encryption's wider adoption:

- high risk of losing access to encrypted data if users forget their passwords or lose their devices;
- technical limits for features development;
- high cost of implementation, requiring specialized developers and long term investment;

Finding a solution to these problems was the prerequisite for end-to-end encryption's adoption in a mass-market application such as Doctolib and an application to medical confidentiality.

Design philosophy and threat model

When designing Doctolib's encryption solution, our goal was to aim for the highest level of confidentiality for both patients and medical professionals. Our solution is based on the known principles of end-to-end encryption, adapted to our mass-market use case. As we have seen, security is always relative and is more a matter of limits.

Here are the constraints we imposed ourselves when designing the solution:

1. **It must be impossible for any Doctolib employee to access personal health information**. Even if someone gained access to all user data stored by Doctolib, they should not be able to read any protected health information.
2. **It must be impossible for any Doctolib employee to access encryption keys**. Encryption keys should be accessible only by end-users, and no one should be able to modify them.
3. End-user devices are assumed safe when in use. **When not in use, no health data or encryption keys should be accessible.**

This set of rules has allowed us to develop innovative features, pushing new standards for online medical confidentiality without compromising on user experience or functionality.

End-to-end encryption protocol

Most existing end-to-end encryption protocols are designed for securing communications, i.e. take place in a context where both parties are present at the same time, with short-term data retention.

Our use case is the complete opposite. We need to store data for a long period of time and we need to asynchronously share it between users. That's why we use the Tanker protocol, which was designed for this purpose.

Symmetric and asymmetric encryption

The basic blocks of any encryption protocol are the cryptographic primitives – or algorithms – that are being used. These basic blocks are industry-standard and are

available in open-source libraries. Within those basic blocks, two are particularly important to understand.

Symmetric encryption makes data unreadable by encrypting it with a key. This same symmetric key is then used to make the data readable again. With symmetric encryption, it is thus important to make sure only intended people have access to the encryption key.

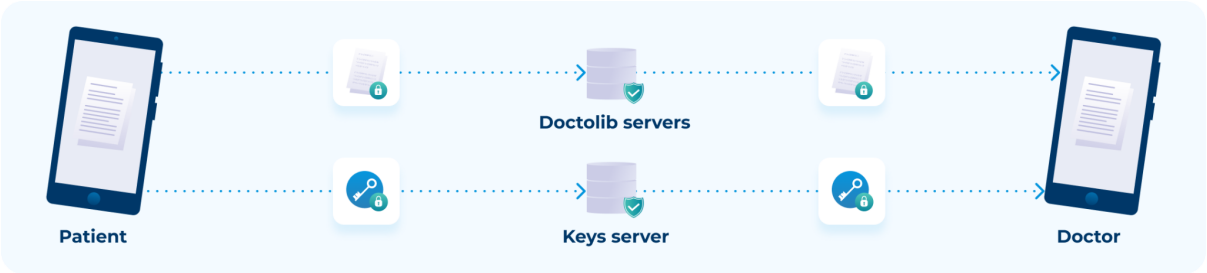
Asymmetric encryption works with a key *pair* instead of a single key. The first element of this key pair is a public key, which can encrypt data, making it unreadable, but cannot decrypt it. The second key, called the private key, can only decrypt encrypted data. Public keys can thus be sent to other users, who can use them to encrypt data that only the private key owner will be able to read.

Envelope encryption

The basic principle behind Doctolib's encryption and key sharing is envelope encryption. It combines symmetric and asymmetric encryption to provide a fast, efficient means to encrypt and share data between different users.

First, the protocol encrypts the resource (file, data, etc.) with a randomly-generated symmetric resource encryption key. This resource key is then asymmetrically encrypted using each recipient's public encryption key.

The encrypted resource keys can now safely be sent to the cloud and stored. In Doctolib's case, keys are stored in a separate keys server, even though Doctolib cannot decrypt them. These encrypted keys are then distributed to their respective recipients as needed.



The recipient can then decrypt the resource key using their asymmetric private encryption key before decrypting the encrypted resource using the resource key.

Envelope encryption allows Doctolib to store and distribute encryption keys without having access to them. This, in turn, allows users to securely upload and share encrypted data with other users.

Secret sharing to guarantee data access

As we have seen earlier, the Tanker protocol is based on envelope encryption. The fundamental element in envelope encryption is the end user's device, on which private keys are stored. But what happens when the end-user wants to access their data from a different device or loses their current device?

Our challenge was to allow data to be accessible from multiple devices and to guarantee that users won't lose access to their data without compromising the security model.

Devices and users

To allow users to access their data from multiple devices, the protocol allows existing devices to authorize new ones. This is done by signing the new device's public keys with the existing device's private keys and sharing data with the new device. This mechanism guarantees that only the user, who has access to an existing device, can add new devices.

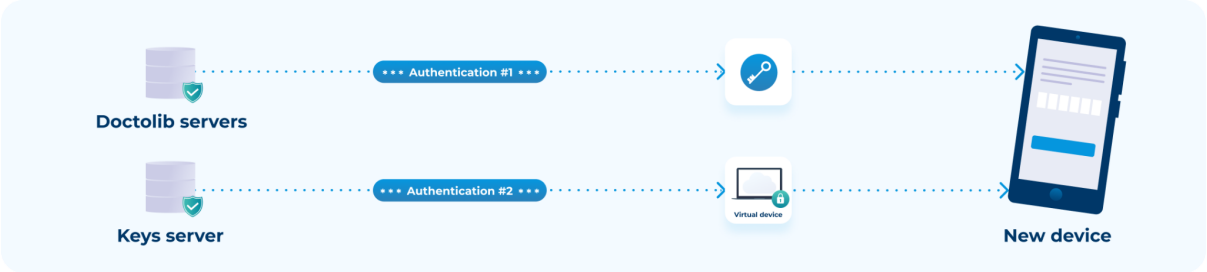
The only exception to this is the first device of each user, which is signed by a root signature key since there are no previous devices available.

When adding a new device, the existing one could take this opportunity to re-share all data to the new one. However, this would be highly inefficient and resource-intensive. To solve this issue, we use an additional level of asymmetric keys: the user keys. Instead of sharing data with devices directly, we share it with the user key. The user key itself is shared between devices, allowing any device to access any previously encrypted data at no additional cost.

Virtual devices and secret sharing

The implications of our device adding scheme are that users should always keep access to at least one registered device. This is unrealistic in real life, which is why we introduced the concept of virtual devices.

When a user uses Doctolib for the first time, two devices are created in the background. The first device is a virtual device, while the second one (authorized by the virtual device) corresponds to the physical device. The virtual device's private keys are not stored locally. Instead, they are encrypted with a symmetric key, the user secret, and stored in a dedicated keys infrastructure completely separate from our main servers. The user secret is stored separately in our main authentication servers. This scheme effectively splits the virtual device into two chunks, a process known as secret sharing.



To regain access to their data, users need to get both parts of the secret back. They must first authenticate with the main authentication server in order to get their user secret. Then the user independently authenticates with the key servers to get their encrypted virtual device. Only with both these elements can the user access their virtual device and use it to access their data on a new device.

Processes to keep data secure

For the secret sharing scheme to be secure, the two parts holding the user secret and the encrypted virtual device must be independent. To ensure independence, we have two completely separate infrastructures. These infrastructures are maintained by different teams with no overlapping access rights. That's a model called segregation of duties.

Admins of the main Doctolib infrastructure cannot access the keys infrastructure and vice versa. Developers working on the Doctolib servers cannot modify the keys

servers and vice versa. The two infrastructures are also hosted on different cloud providers to ensure maximum separation.

Our security processes and access rights are also regularly audited and are ISO-27001 certified.

Transparent double authentication

The two authentications required to get the virtual device back must also be independent. This means that at no time should the user provide information to the main Doctolib server that could be used to authenticate with the key server, and vice versa. To achieve this, different authentication means are used.

Transparent password authentication

The first way we perform independent authentications is completely transparent for the end-user. We achieve this by reusing the user's password to authenticate twice.

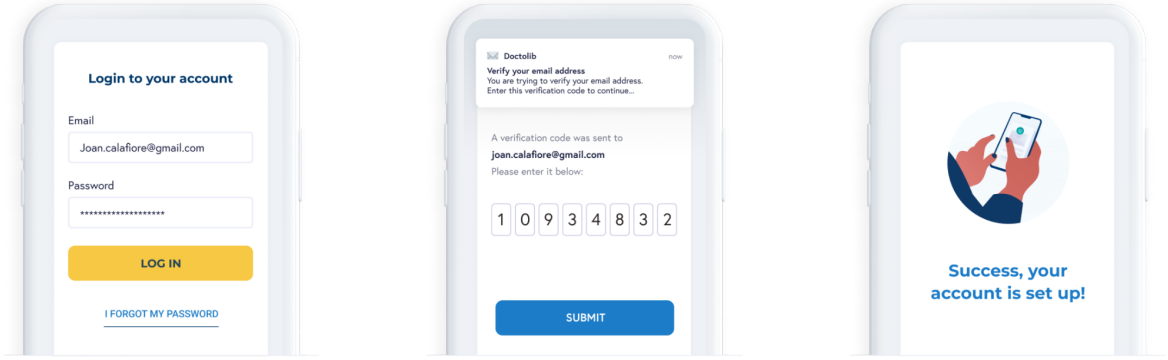
To prevent any part from being able to impersonate the user, neither the application server nor the keys server must ever see the password in clear. To avoid this, the password is hashed client-side, using two different hashing algorithms: one for the key server and one for the application server.

This way, authentication remains independent, and the users' accounts are secure.

Email verification backup

However, users still need to register an alternate identity verification method in case they forget their password. To this end, we use email verification.

When the user goes through the 'I forgot my password' flow, they receive a verification link. This link authenticates with the application server. Then, a second authentication happens with the key server this time. The user verifies their email address or phone number by entering an 8-digit code sent by the keys server.



This way, the user authenticates with the Doctolib servers with the link and with the keys server thanks to the email verification code.

Conclusion

This paper is only an introduction to the basic principles and choices we had to make when designing Doctolib's encryption protocol and is not exhaustive in any way. The entirety of the protocol is open-source, as well as the SDKs we use to implement it. They can be checked on GitHub.