Doctolib

Doctolib Connect Security Whitepaper

A Secure Foundation for Confidential Healthcare Communication

March 5, 2026

Disclaimer

This Security Whitepaper is intended to provide information about the security design and implementation of Doctolib Connect. While we have made every effort to ensure the accuracy and completeness of this document, we make no warranties or representations, express or implied, regarding its content.

Updates and Changes

We may update this Security Whitepaper from time to time to reflect changes in our understanding of security best practices or to address new threats and vulnerabilities. We encourage readers to review this document periodically for the latest information.

Copyright

Contact

Doctolib
54 Quai Charles Pasqua
Levallois-Perret
92300, Ile-de-France
France

Changelog

v2.1	05-03-2026	Small revision to reflect versioning changes.
v2.0	17-01-2025	Revised earlier version to add additional information on key storage, case keys, and various clarifications.

Table of Contents

1 Introduction

Doctolib Connect is dedicated to providing a secure and reliable asynchronous communication platform for healthcare professionals. Recognizing the sensitive nature of healthcare data, Doctolib prioritizes security in all aspects of its platform design and implementation. Connect's security foundation is built upon the following principles:

Proven Cryptography Doctolib Connect employs well-established cryptographic techniques based on open-source components, ensuring transparency and peer-reviewed security. These cryptographic choices are made in consultation with academic cryptographic experts, further validating their robustness.

Rigorous Security Testing Doctolib Connect maintains a proactive security posture and has conducted penetration and vulnerability testing to assess and address potential weaknesses in the platform. These efforts ensure a resilient and secure environment for sensitive healthcare communications.

Transparency and Trust This whitepaper provides a comprehensive overview of Doctolib Connect's cryptographic design decisions and threat model. By transparently outlining our security approach, we aim to foster trust and confidence among healthcare professionals using our platform.

Balancing Security and Usability Doctolib recognizes the importance of balancing robust security with ease of use. Our platform is designed to provide a seamless and intuitive user experience while maintaining the highest security standards to protect sensitive healthcare information.

2 Overview of Connect End-to-End encryption

2.1 Asymmetric encryption

This section describes how `crypto_box` is used for asymmetric encryption, for secure communication between contacts. Doctolib Connect leverages the NaCl library[1] as the basis for its End-to-End encryption (E2EE), the same library used as the basis of Doctolib's Tanker. Specifically, for message exchange, Connect uses NaCl's `crypto_box` function [6] to provide authenticated encryption.

[1]Peter Schwabe Daniel J. Bernstein Tanja Lange. *NaCl: Networking and Cryptography library*. `https://nacl.cr.yp.to/`

When reading the following section, please refer to Figure 1 for a visual overview.

2.1.1 Keys

On the client, the `crypto_box_keypair` function randomly generates a private, or secret key (**sk**) and a corresponding public key (**pk**):

```
sk, pk = crypto_box_keypair()
```

2.1.2 Key sharing

At time of user registration, **pk** is sent to the Doctolib backend, where it is stored as part of the user's Contact data. In contrast, **sk** is securely stored on the user's device (see the Key Storage section, page 13).

2.1.3 Contact list

The sender has access to a list of contacts. Each of these contacts has an associated public key, **pk**.

2.1.4 Message exchange

When sending a message to a contact, the `crypto_box` function is used: this function encrypts and authenticates a message **m** using the sender's secret key **sk**, the receiver's public key **pk**, and a (secure random) nonce **n**. The `crypto_box` function returns the resulting ciphertext **c**:

```
n = secure_random_bytes(32)
c = crypto_box(m, n, pk, sk)
```

Ciphertext **c**, along with nonce **n**, is sent through the Doctolib backend to the recipient. There, it is decrypted using `crypto_box_open`. This function verifies and decrypts ciphertext **c** using the receiver's secret key **sk**, the sender's public key **pk**, and the nonce **n**. The `crypto_box_open` function returns the resulting plaintext **m**:

```
m = crypto_box_open(c, n, pk, sk)
```

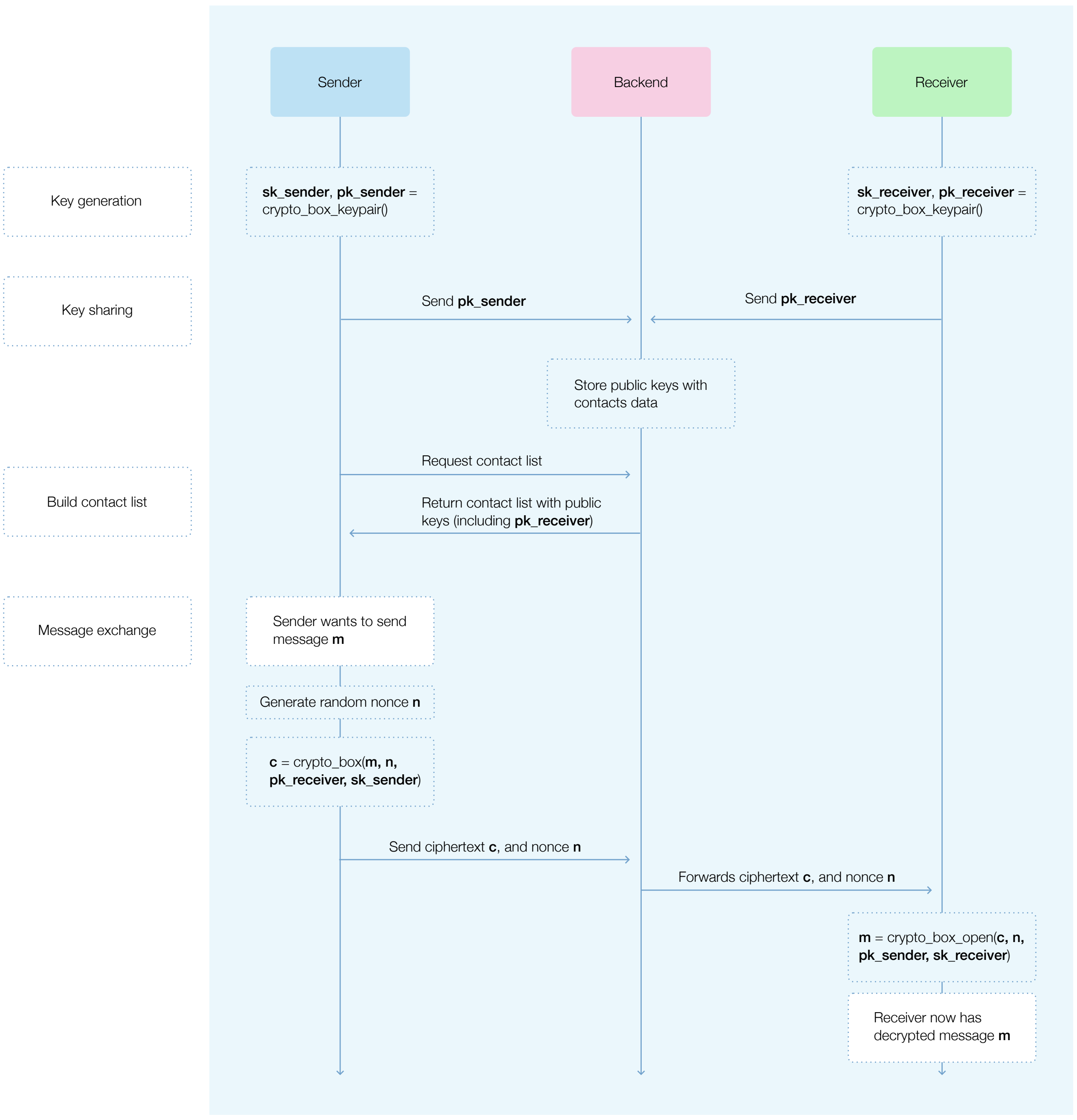
Sender
Backend
Receiver
Key generation
sk_sender, pk_sender = crypto_box_keypair()
sk_receiver, pk_receiver = crypto_box_keypair()
Key sharing
Send pk_sender
Send pk_receiver
Store public keys with contacts data
Build contact list
Request contact list
Return contact list with public keys (including pk_receiver)
Message exchange
Sender wants to send message m
Generate random nonce n
c = crypto_box(m, n, pk_receiver, sk_sender)
Send ciphertext c, and nonce n
Forwards ciphertext c, and nonce n
m = crypto_box_open(c, n, pk_sender, sk_receiver)
Receiver now has decrypted message m

Figure 1: Message exchange flow

2.1.5 Security Model

As specified in the official `crypo_box` documentation [6]:

> *"The `crypto_box` function is designed to meet the standard notions of privacy and third-party unforgeability for a public-key authenticated-encryption scheme using nonces. For formal definitions see [1] Distinct messages between the same sender, receiver set are required to have distinct nonces. Nonces are long enough that randomly generated nonces have negligible risk of collision."*

Connect uses random nonces for message exchange. The `crypto_box` function is not meant to provide non-repudiation. On the contrary: the `crypto_box` function guarantees repudiability. A receiver can freely modify a boxed message, and therefore cannot convince third parties that this particular message came from the sender. The sender and receiver are nevertheless protected against forgeries by other parties.

2.2 Symmetric encryption

This section describes how `crypto_secretbox` is used for symmetric encryption in various other parts of the application (see Keys used, and their purpose, page 10). This involves a single key for both encryption and decryption. Connect leverages NaCl's `crypto_secretbox` function for symmetric encryption, which encrypts and authenticates a message **m** using a 256-bit secret key **k** and a nonce **n**. The nonce is a unique value used only once per message m to ensure that encrypting the same message twice produces different ciphertexts. The `crypto_secretbox` function returns the resulting ciphertext **c**:

```
k = secure_random_bytes(32)
c = crypto_secretbox(m, n, k)
```

The `crypto_secretbox_open` function verifies and decrypts a ciphertext **c** using a secret key **k** and a nonce **n**, returning the resulting plaintext **m**:

```
m = crypto_secretbox_open(c, n, k)
```

2.3 Nonce Usage and Reuse

2.3.1 Random Nonces

In most cryptographic operations (e.g., message exchange, local file encryption and patient case encryption) Connect employs random nonces of 32 bytes generated using a secure random number generator. This ensures that nonce collisions, which can compromise the confidentiality and integrity of encrypted data, pose a negligible risk.

Each randomly generated nonce is unique for its specific cryptographic session and purpose, safeguarding the encryption process from potential vulnerabilities.

2.3.2 Incrementing Nonces for Attachments

For attachment encryption, Connect uses a deterministic approach by employing an incrementing nonce. This approach starts with 1 and increments sequentially for each attachment chunk. This method allows recipients to easily reconstruct the nonce without the need for additional metadata or nonce transmission, reducing communication overhead while maintaining security since a new symmetric key is used for each attachment.

2.4 Cryptographic primitives

The cryptographic primitives used by NaCl are not necessarily the most common and well-known choices such as AES and RSA. NaCl uses elliptic-curve cryptography, not RSA; it uses an elliptic curve, Curve25519, that has several advanced security features; it uses Salsa20, not AES; it uses Poly1305, not HMAC; and for an elliptic-curve signature system it uses EdDSA, not ECDSA. There are various good reasons behind these choices, including performance and more robustness against cryptographic attacks.

`crypto_box` is `curve25519xsalsa20poly1305`, a combination of Curve25519, Salsa20, and Poly1305.

`crypto_secretbox` is `crypto_secretbox_xsalsa20poly1305`, a combination of Salsa20 and Poly1305.

`crypto_sign` is `crypto_sign_edwards25519sha512batch`, a combination of Curve25519 in Edwards form and SHA-512.

For more information on NaCl, the underlying cryptographic primitives, and the rationale for their use, please refer to the website [5] and the paper [7].

2.5 Reason behind using NaCl

Doctolib has chosen to utilize NaCl as its core cryptographic foundation for several compelling reasons, primarily centered around its unique combination of simplicity and robust security. NaCl is designed with a focus on ease of use and minimizing the potential for developer error. By providing high-level, well-defined cryptographic functions, NaCl abstracts away many of the complexities of lower-level cryptographic primitives. This significantly reduces the risk of introducing vulnerabilities due to incorrect implementation or misuse of cryptographic algorithms. Despite its simplicity, NaCl does not compromise on cryptographic strength. It is built upon carefully selected, modern cryptographic primitives that have undergone extensive scrutiny and are widely considered secure. This ensures that Doctolib Connect's communication platform benefits from strong encryption and authentication mechanisms.

3 Keys used, and their purpose

3.1 Box Key (asymmetric)

The Box Key is an NaCl keypair generated during the registration process using `crypto_box_keypair`, and used for encrypting messages, as described in the previous section:

- The public Box Key (**pk**) is stored on the server and accessible to authenticated clients. When a user reinstalls the application and generates a new Box Key, a contact update event is sent through the Connect protocol to inform their contacts about the updated public key.
- The private Box Key (**sk**) is stored securely on the user's device (see Key Storage, page 13)

Lifetime: The Box Key remains constant throughout the installation's lifetime. However, it can be revoked by the server under specific circumstances:

- **New Installation**: When a user installs the application again, a new Box Key is generated and provided to the server, effectively revoking the previous key.
- **Account Deletion**: An authorized Doctolib operator can manually revoke the Box Key when a user's account is deleted.
- **Security Concerns**: The server can revoke a Box Key if there are security concerns or suspected compromise.

3.2 Sign Key (asymmetric)

The Sign Key is an NaCl signing keypair, also generated during the registration process. It is used for client authentication during API calls, ensuring that requests are coming from legitimate Doctolib users. The Sign Key is generated using the `crypto_sign_keypair` function:

```
sk_sign, pk_sign = crypto_sign_keypair()
```

The lifetime and revocation of the Sign Key are identical to the Box Key, described above.

3.3 Database Key (symmetric)

The Database Key is a 256-bit key used by SQLCipher, which leverages AES-256 to secure the underlying SQLite database files. This key encrypts the database on the user's device, which contains contacts, messages, and associated metadata. Generated during registration, this key persists for the lifetime of the installation and cannot be changed without reinstalling the application.

3.4 Local File Key (symmetric)

For encrypting locally stored files, a 256-bit symmetric key called the Local File Key is used. Encryption is performed using NaCl's `crypto_secretbox`, with a secure randomly generated nonce prepended to the encrypted byte array. This key is generated during installation and remains unchanged unless the application is reinstalled.

3.5 Attachment Key (symmetric)

The Attachment Key is a 256-bit symmetric key that is generated by the client each time an attachment (such as an image, video, or audio file) is uploaded to the server. These keys are single-use. Attachments are divided into 2 MB chunks before encryption. This facilitates efficient handling and transmission of large files. The nonce used for encrypting each attachment chunk is derived from the index of the chunk (starting with 1 for the first chunk, then incrementing.) This deterministic approach allows the recipient to easily derive the nonce without requiring it to be transmitted separately, reducing communication overhead. The NaCl `crypto_secretbox` implementation is used for symmetric encryption, and the key is shared with each intended recipient using asymmetric encryption (`crypto_box`), using their public Box Key, **pk**.

3.6 Case Key (symmetric)

A Case Key is a 256-bit symmetric key that is generated by the client and used to encrypt the various fields (blocks of information) of a so-called Patient Case, a dedicated group conversation used to discuss a single patient. For each Case, a new Case Key is generated.

- **Generation**: The Case Key (**`case_key`**) is randomly generated by the client using `secure_random_bytes(32)`, a cryptographically strong random number, ensuring its uniqueness and unpredictability.
- **Encryption**: The Case Key is used to encrypt the Patient Case information using `crypto_secretbox` (See Symmetric encryption, page 8.) This ensures the confidentiality and integrity of the case data.
- **Sharing with Participants**: To enable all participants to access the encrypted case information, the Case Key is securely shared with them through this process:
  - The Case Key is individually encrypted for each participant using their public Box Key (**`contact_pk`**), the sender's private Box Key (**sk**) and a nonce **n** using `crypto_box`. This ensures that only intended recipients can decrypt and access the Case Key.
  - The encrypted Case Key is then transmitted to each participant.

```
case_key = secure_random_bytes(32)
for_each(contact in participant_list)
    c = crypto_box(case_key, n, contact_pk, sk)
```

In addition to the case information, messages in this case are also encrypted using this same case key. Unlike for normal group chats, this allows for adding members to the conversation at a later date, while ensuring they can decrypt and access the historic content of the conversation, allowing frictionless collaboration.

3.7 Master key (symmetric)

In order to securely store the various keys described above on the device, a master key is used. More information about this can be found in the Master Encryption Key section, page 13.

3.8 TLS Certificate

Finally, the TLS Certificate is issued by Sectigo CA. It utilizes an RSA 2048-bit key and SHA-256 for secure communications.

4 Key Storage

4.1 Android

Doctolib Connect employs a multi-layered approach to securely store and protect cryptographic keys on Android devices. This approach leverages the Android Keystore System and encryption to ensure the confidentiality and integrity of sensitive key material.

4.1.1 Master Encryption Key

Connect utilizes the Android Keystore System to generate and store a master encryption key. This key is used to encrypt the various individual encryption keys used within the application. The Android Keystore provides a hardware-backed, isolated environment for storing this master key, making it significantly more difficult for unauthorized access or extraction.

- **Master Key Generation**: A master key is generated using AES (Advanced Encryption Standard) with the following parameters:
  - **Key Algorithm**: KeyProperties.KEY_ALGORITHM_AES
  - **Block Mode**: KeyProperties.BLOCK_MODE_CBC
  - **Encryption Padding**: KeyProperties.ENCRYPTION_PADDING_PKCS7
- **Security Features**: The Android Keystore provides several security features to protect the master key, including:
  - **Hardware-backed Storage**: On devices with dedicated secure hardware, the master key is stored in this hardware for enhanced protection.
  - **Non-Exportability**: The master key's raw material remains non-exportable, preventing extraction from the device even with root access.
  - **Isolated Environment**: The Keystore provides an isolated environment, protecting the master key from unauthorized access by other applications.

4.1.2 Individual Key Encryption

The individual encryption keys used within the Connect app (e.g., Box Key, Sign Key, Database Key), as well as the PIN code are encrypted using the master key stored in the Android Keystore.

- **Encryption Method**: The Android Keystore is used to encrypt the individual keys using AES in CBC mode with PKCS7 padding.
- **Initialization Vector (IV)**: A secure random IV is generated for each encryption operation.
- **Storage**: The encrypted keys, along with their corresponding IVs, are stored in SharedPreferences in Base64 encoded format.

4.1.3 User Authentication

The user’s PIN is also securely stored within the Android Keystore. Users can authenticate to the Connect app using one of the following methods:

- **Biometric Authentication**: Convenient and secure authentication using fingerprint or other biometric modalities supported by the device.
- **PIN Entry**: Users can enter their 5-digit PIN to authenticate.

4.1.4 Benefits

This multi-layered approach provides robust security for cryptographic keys on Android devices:

- **Master Key Protection** The Android Keystore safeguards the master key, minimizing the risk of unauthorized access.
- **Individual Key Encryption**: Encrypting individual keys with the master key adds an extra layer of protection, even if SharedPreferences data is compromised.
- **Hardware-backed Security**: Leveraging the Android Keystore’s hardware-backed features enhances key security on compatible devices.
- **Non-Exportability**: The non-exportable nature of the master key and the encryption of individual keys prevent key material from being extracted from the device.

Refer to Android Keystore System [2] for more information.

4.2 iOS

On iOS, all keys are securely stored in the iOS Keychain, initialized with the **kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly** parameter enabled.

The data in the keychain item cannot be accessed after a restart until the device has been unlocked once by the user. After the first unlock, the data remains accessible until the next restart. This is recommended for items that need to be accessed by background applications. Items with this attribute do not migrate to a new device. Thus, after restoring from an Apple backup on a different device, these items will not be present and messages can not decrypted. However using the Connect Backup/Restore process will guarantee the messages access on the new device. For more information, please refer to the Apple Documentation [3].

4.2.1 User Authentication

Similarly to Android, users can authenticate to the Connect app using one of the following methods:

- **Biometric Authentication**: Convenient and secure authentication using fingerprint or other biometric modalities supported by the device.
- **PIN Entry**: Users can enter their 5-digit PIN to authenticate.

5 Key Invalidation

Both the Box Key and Sign Key can be invalidated or revoked by the server under specific circumstances. This mechanism ensures that outdated keys are no longer used.

5.1 Invalidation Triggers

Key invalidation can be triggered by the following events:

- **New Installation**: When a user reinstalls the Connect application on their device, a new Box Key and Sign Key are generated. The server automatically invalidates the previous keys associated with the user's account.
- **Manual Revocation**: Authorized Doctolib operators have the ability to manually revoke a user's Box Key and Sign Key. This might be necessary in cases of suspected account compromise, or at the user's request.

5.2 Invalidation Mechanism

Key invalidation is implemented through a contact update message sent by the server. This message is transmitted through the same secure communication channel used for other Connect protocol messages. This contact update message informs other users about the invalidation of the keys, prompting them to update their records and use the new keys for future communication with the affected user. The process can be initiated by a dedicated invalidation API call, accessible only to authorized operations personnel.

5.3 Decryption of Messages with Revoked Keys

Connect employs a fallback mechanism to decrypt messages sent using older keys. The backend stores the old public keys for each contact. For each incoming message, the decryption process attempts to use the sender's previously active keys, one after another, until decryption succeeds. This approach ensures that messages that were sent prior to the key update but before the recipient had a chance to receive them become permanently inaccessible due to key updates.

6 Perfect Forward Secrecy (PFS)

Perfect Forward Secrecy [11] (PFS) is a cryptographic feature that ensures the confidentiality of past communications even if a long-term private key is compromised. PFS achieves this by generating ephemeral session keys for each session, which are discarded after the session ends.

Currently, Connect does not implement PFS for its end-to-end encryption protocols. This decision is influenced by the need to balance performance, user experience, and technical complexity. While the absence of PFS does not compromise the security of ongoing communications, we acknowledge that adding PFS would enhance the resilience of Doctolib Connect's platform against certain advanced threat models.

As we continue to evolve Connect's security infrastructure, incorporating PFS is a potential focus area for future development. This reflects our ongoing commitment to providing state-of-the-art security for healthcare professionals.

7 Open Source Cryptography Implementations

The cryptographic framework of the system is built on several open-source projects that ensure robust security across various platforms.

- For database encryption, the SQLCipher Project [12] is utilized, providing transparent AES-256 encryption of SQLite databases.
- Message encryption and other cryptographic functions rely on the NaCl project [5]. This library, known for its high-speed cryptographic operations, is foundational to the system's security architecture. Additional resources and documentation for NaCl are available on the NaCl project page and the corresponding whitepaper [7].

For platform-specific implementations:

- **Android**: The cryptographic functionalities are extended through projects like Libsodium JNI [9], an Android wrapper for NaCl, and the Android-specific version of SQLCipher [12].
- **iOS**: On iOS, cryptographic operations are implemented using Swift Sodium [10], a Swift wrapper for NaCl, and SQLCipher.swift [4] with SQLCipher for database encryption.
- **Backend**: on the server side TweetNaCl [8], a minimalistic version of the NaCl library in Java is used.

8 Protocol

The system's wire format and domain objects are generated using Google Protocol Buffers, as defined in `Envelope.proto`. At the highest level, messages are encapsulated within an "envelope." This envelope, protected by TLS 1.3, contains metadata that allows the backend to route the message for "store and forward" operations. The backend holds the message until it is acknowledged (or "Acked") by the recipient, at which point the message is deleted from the server[2]. Each message is uniquely identified by a combination of `message-id` and `timestamp`. However, it is important to note that currently, an authorized client has the ability to rewrite the message history.

[2] If you use Connext Web, the message is not deleted immediately, as it needs to be available for the web client.

8.1 Receiving a message

To receive a message, the client sends an HTTP POST request to

```
/api/{api version}/messages/receive
```

The request headers include **`x-siilo-uid`** and **`x-siilo-sig`** (see Signature, page 19) alongside `Protos.Ack` in the body. Upon receiving this `Ack`, the server deletes the acknowledged messages from its storage. The server's response includes `EnvelopeList`, which contains a list of envelopes in the same format as those used by the send API.

8.2 Sending a message

To send a message, the client sends an HTTP POST request to

```
/api/{api version}/messages/send
```

The request headers include **`x-siilo-uid`** and **`x-siilo-sig`** in the headers alongside `Protos.Envelope` in the body. A successful operation results in a `204 No Content` response. An envelope includes the sender's user ID and, if applicable, the associated group ID. This allows for sending text messages, typing events (i.e., indicating a user is typing), and read receipts (indicating a user viewed a message).

8.2.1 Group Messages

For group messages, the same envelope is used with different payloads, each encrypted with the public key of the intended recipient. This is similar to messages encryption to a single recipient (see Overview of Connect End-to-End encryption, page 6) but encrypting them for each recipient separately using `crypto_box`. This ensures that only members who are part of the group have access to the messages in the conversation, and nobody can be added to the group at a later date and have access to the message history of the conversation. Although encrypting the message individually for each group participant increases the overall data size, this overhead is negligible given the typical message length of approximately 200 bytes.

8.2.2 Attachments

As discussed in more detail in the Attachment Key (symmetric) section on page 11, attachments, unlike regular text messages, are not encrypted asymmetrically, but symmetrically instead, using `crypto_secretbox`. The symmetric key used to encrypt an attachment is then shared to the recipient asymmetrically with `crypto_box`.

8.2.3 Patient Case Messages

As discussed in more detail in the Case Key (symmetric) section on page 11, patient case messages are encrypted with a symmetric key using `crypto_secretbox`, which is shared with all patient case participants through asymmetric encryption using `crypto_box`.

8.3 Signature

All API calls are protected by a digital signature included in the HTTP headers as **`x-siilo-uid`** and **`x-siilo-sig`**. The **`x-siilo-uid`** must match the **`user_id`** encoded within the **`x-siilo-sig`**. A request is signed as follows:

1. A `SignatureHeader` Protobuf object is constructed and converted to bytes using `toByteArray()`, forming the message **bm**. The `SignatureHeader` contains these fields:
   - timestamp
   - user_id
   - build_number
   - uri_path
   - client_type
2. The Sign Key is used to sign **bm** using NaCl's `crypto_sign`:

   ```
   sm = crypto_sign(bm, sk_sign)
   ```

3. The result **sm** is concatenated with **bm** (the original `SignatureHeader` byte array), then Base64 encoded and included in the **`x-siilo-sig` header**:

   ```
   x-siilo-sig = Base64(sm, bm)
   ```

9 Security mechanisms

The system employs several robust security measures to ensure the protection and confidentiality of user data:

- **End-to-End Encryption**: All communications are encrypted end-to-end using NaCl cryptographic primitives, ensuring that only the intended recipient can decrypt the messages (see Overview of Connect End-to-End encryption, page 6)
- **Application Access Control**: A pin code or biometric authentication is required to access the application, providing an additional security measure to protect user data (see User Authentication, page 14 for iOS and User Authentication, page 14 for Android).
- **Local Storage Encryption**: Data stored locally on the device is encrypted using SQLCipher (which implements AES-256) or NaCl, securing the database and other sensitive information.
- **The OS-level application sandbox**: which isolates Connect's app data and code execution from other apps.
- **Client-Side Certificate Pinning**: The server's public key is pinned on the client side, adding an additional layer of security to prevent man-in-the-middle attacks.
- **TLS Encryption**: Communication between the client and server is secured using TLS, with ciphers that support forward secrecy:
  - **TLS 1.3** (server has no preference):
    - **TLS_AES_128_GCM_SHA256 (0x1301)**: ECDH x25519 (eq. 3072 bits RSA) FS
    - **TLS_AES_256_GCM_SHA384 (0x1302)**: ECDH x25519 (eq. 3072 bits RSA) FS
    - **FS TLS_CHACHA20_POLY1305_SHA256 (0x1303)**: ECDH x25519 (eq. 3072 bits RSA) FS
  - **TLS 1.2** (suites in server-preferred order):
    - **TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)**: ECDH x25519 (eq. 3072 bits RSA) FS
    - **TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)**: ECDH x25519 (eq. 3072 bits RSA) FS

10 Threat Model

The platform is designed with stringent security goals to ensure the confidentiality, integrity, and authenticity of user communications and data. These goals include:

- **Confidentiality and integrity in transit**: Ensuring that messages remain confidential and unaltered while being transmitted between the sender and receiver.
- **Device-level confidentiality**: Protecting the confidentiality of messages stored on the device, particularly in scenarios where the device might be lost or stolen.
- **Server-Side confidentiality**: Safeguarding the confidentiality of messages temporarily stored on the server as they await delivery.
- **Minimized data retention**: Avoiding the storage of personal information such as phone numbers and emails for non-Doctolib users, thereby reducing unnecessary data exposure.
- **User identity verification**: Verifying the identities of users on the network to prevent unauthorized access and impersonation.

To thoroughly assess potential security threats, this document will model threats using a structured template:

1. **Description**: This section provides a brief outline of the attack or attacker in question, detailing the nature and intent of the potential threat.
2. **Prerequisites**: Here, the conditions required for the attack to be implemented are outlined. This may include specific technical conditions or access to certain information or resources.
3. **Prevention**: This section discusses the measures in place to prevent the attack or to reduce its likelihood. It covers both technical and procedural safeguards designed to counteract the threat.
4. **Difficulty**: This assesses how practical it is to carry out the attack, categorized as:
   - **Easy**: The attack requires no specialized knowledge, skills, or equipment.
   - **Moderate**: The attack demands a moderate level of knowledge or information, such as a strong computer science background or non-public information about the target.
   - **Difficult**: The attack necessitates specialized knowledge or detailed information about the target, such as expertise in Informatics Security or personal insights gained through prolonged social contact with the target.
   - **Very Difficult**: The attack requires highly specialized knowledge or information approaching the level of a nation-state actor or an individual within the top 3% of the security field.
5. **Severity**: This section evaluates the potential impact of a successful attack, independent of its difficulty. It considers the consequences and damage that could result from the breach.
6. **Conclusions**: Finally, an assessment of what a successful attack could achieve is provided, along with an explanation of the design decisions made to mitigate this specific threat vector, where applicable.

10.1 Intercepted network traffic

Description An attacker intercepts network traffic between a participant and the server. For instance, if a user connects their device to a compromised Wi-Fi network controlled by the attacker, the attacker can capture and analyze the traffic before it is forwarded to its intended destination.

Prerequisites For this attack, the attacker must first compromise a device in the network path, such as a Wi-Fi router. Additionally, they must bypass the transport layer encryption to access the unencrypted data.

Prevention The platform implements several safeguards to mitigate this threat:

- **Certificate Pinning**: Clients use certificate pinning to guard against many man-in-the-middle attacks by ensuring that only trusted certificates are accepted.
- **Cryptographic Signing**: Each request includes a cryptographic signature with a timestamp, valid for only 5 minutes. This limits the window for replay attacks.
- **End-to-End Encryption**: In addition to TLS, end-to-end encryption protects user content, ensuring that even if transport layer security fails, the data remains confidential.

Difficulty Difficult. Successfully executing this attack requires specialized knowledge and capabilities to both intercept network traffic and break transport layer encryption (TLS 1.3).

Severity Low. Although the attacker might gain access to certain types of information, the overall impact is limited.

Conclusions Even if the attacker compromises network traffic and bypasses TLS 1.3, their achievements are constrained:

- **Message Replay**: The attacker could potentially replay messages within the server's signature validity window, triggering duplicate system notifications like typing indicators. However, they cannot spoof content, read notifications, or access end-user messages.
- **Metadata Exposure**: If the transport layer is compromised, metadata such as the Connect user ID of participants and message length may be exposed.
- **Encrypted Content**: The attacker can intercept encrypted content transmitted during their network access. Despite the absence of forward secrecy in the message encryption, they would only gain access to messages that were intercepted. Breaking the user's private key would only reveal the specific messages captured during this period.

10.2 Non-technical attempt to access the application

Description An attacker gains brief physical access to your device through a social engineering attack. For example, someone may ask to use your device to make a call due to their battery being dead. Given the time constraint, the attacker cannot perform more invasive methods, such as connecting to the device with a debugger.

Prerequisites The attacker must have physical access to the device and attempt to access your data through non-technical means, such as guessing the pin code at the application level.

Prevention Several layers of security are in place to mitigate this risk:

- **OS-Level Pin Code**: The device's operating system requires a pin code for access.
- **Application-Level Protection**: The app is secured with a 5-digit numeric pin code that must be entered if the app has not been accessed within the past 10 minutes. Additionally, incorrect pin code attempts trigger progressively longer timeouts.

Difficulty Difficult. The attacker must have physical access and is limited to non-technical methods of bypassing security, such as guessing the pin code. Advanced technical attacks are not feasible due to the time constraint.

Severity High. The attack could potentially allow the attacker to access sensitive application data if they can guess the pin code within the limited time window. This threat is significant especially if the attacker is targeting the individual specifically.

Conclusions The design allows access to the application if the pin code has been entered within the last 10 minutes. This is a trade-off between usability and security, catering to scenarios where quick access is needed. In cases where the device is left unattended, the pin code is likely to expire before an attacker attempts access. For enhanced security, users can configure their OS-level pin code to activate when the device enters sleep mode, providing an additional layer of protection against unauthorized access.

10.3 Unrestricted physical access to the device

Description The device is left unattended in a scenario where the attacker has sufficient time to access it using specialized tools, such as a debugger. The attacker is not constrained by the need to quickly return the device undetected and can perform in-depth tasks, such as "rooting" the device or physically accessing its internal components.

Prerequisites The attacker must have physical access to the device and is free to use advanced methods to bypass security measures, including rooting the device or manipulating its hardware.

Prevention

- **OS-Level Pin Lock**: The primary defense is the OS-level pin lock, which is robust against unauthorized access. On iOS devices with Touch ID, the biometric security adds an additional layer of protection, making it highly resistant to attempts to bypass the OS-level passcode. On Android devices, the application's data directory is encrypted by the OS-level security code. However, some devices are vulnerable to bootloader attacks that may bypass this encryption.

Difficulty Moderate. While specialized tools and techniques are required to access a device in this scenario, it is possible with sufficient time and resources. The attacker may leverage vulnerabilities or physical access methods, making this attack feasible but not trivial.

Severity High. If the attacker successfully gains access to the device, they could potentially bypass the application-level pin protection and access sensitive data. This level of access poses a significant risk, especially if the device's OS-level security is not robust.

Conclusions The 5-digit pin code provides a limited addressing space, which is considered insufficient as a sole protective measure. It serves primarily as a form of obfuscation rather than robust security. The best defense against this threat is strong OS-level protection. On iOS, devices with Touch ID configured offer a high level of resistance to unauthorized access. For Android devices, while the OS-level encryption provides substantial protection, users should be aware of potential vulnerabilities related to bootloader attacks. Proper configuration and use of available security features are crucial in minimizing the risk of successful attacks in this scenario.

10.4 Physical access to Doctolib servers

Description An attacker gains physical access to the Doctolib servers, either due to the attacker being a trusted host provider or successfully breaching the hosting facility. This situation could involve the attacker compromising the server infrastructure directly.

Prerequisites The attacker needs physical access to the Doctolib servers, which could be achieved through either direct infiltration of the data center or by exploiting vulnerabilities in the host provider's security.

Prevention

- **Trusted hosting provider**: Doctolib's servers are hosted on Amazon Web Services (AWS), which has security measures in place to protect its customers. AWS holds ISO 27001 certification, ensuring adherence to high security and data protection standards.
- **Security measures**: Regular audits, monitoring, and compliance checks are part of the hosting provider's procedures to prevent unauthorized physical access and ensure overall security.

Difficulty Difficult. Gaining physical access to the servers requires significant effort and resources. The attacker would need to bypass both physical security measures and potentially sophisticated datacenter protections.

Severity : High.

- **Messaging server**: If the messaging server is compromised, user metadata, such as hashed telephone numbers and communication patterns (e.g., contact graphs), could be exposed. However, the actual message content would remain secure due to end-to-end encryption.
- **Elastic search server**: Compromise of the elastic search server would expose publicly available information from the application, but without the application's rate limiting features. This data would be limited to what users can access through the application itself.
- **Database server**: A breach of the database server would have a broader impact, potentially exposing the entire database, which includes all user data. The attacker could access a comprehensive dump of the database, not just intercepted data.

Conclusions If an attacker successfully compromises the server infrastructure, the impact varies based on the server type. While (historic) message content remains secure due to end-to-end encryption, user metadata and publicly available information could be at risk. In the case of database access, the scope of the breach extends to all data stored in the database. To mitigate these risks, it is essential that users verify their key fingerprints to ensure message confidentiality and protect against potential impersonation by an attacker who might insert their own key.

10.5 Social engineering attacks

Description This threat encompasses attacks involving social engineering and impersonation within the Doctolib Connect system. It includes:

- **Impersonating a Real Connect user**: An attacker pretends to be an existing Connext user.
- **Impersonating a Non-Connect user**: An attacker poses as someone who is not a registered Connect user.
- **Impersonating a Doctolib employee**: An attacker pretends to be a Doctolib employee to gain access or information.

Prerequisites

- **Application installation**: The attacker must be able to install the Connect application.
- **SMS falsification**: If targeting a specific user, the attacker may need to falsify SMS origin to deceive the target.

Prevention

- **Unverified state indicator**: New users are marked as "unverified" with clear UI indicators. Users must fill in profile details, which are verified by Doctolib employees. Verified users receive a verification mark.
- **Key fingerprint verification**: Users can verify each other's identity through a face-to-face meeting to compare the hash or fingerprint of their public keys. This helps ensure the cryptographic identity matches the real person.

Difficulty Easy. Social engineering relies on exploiting human psychology rather than technical vulnerabilities. Impersonating someone within the system is straightforward if the attacker can convince the target.

Severity Low to medium.

Conclusions

- **Social engineering risks**: Social engineering attacks exploit human factors and can be prevalent where there is incentive. The challenge of identity verification without a central authority persists. Doctolib is exploring solutions like IRMACard to improve this aspect.
- **Employee impersonation**: Phishing tactics used to impersonate Doctolib employees are akin to scams in other sectors. The primary risk involves revealing the user's pin code. Combined with physical theft of the device, this could lead to unauthorized access. However, the practical application of this attack is limited as it requires physical presence and substantial effort. Users should be aware that legitimate Doctolib employees will not request their pin code and should resist disclosing it.

In summary, while social engineering attacks are a common risk, Doctolib's verification mechanisms and user vigilance are key to mitigating these threats.

10.6 An attacker who is a trusted Doctolib employee

Description This threat involves a Doctolib employee with elevated privileges who might misuse their access to the system to gain unauthorized access to end-user data. Such personnel currently include three individuals with full operational access.

Prerequisites

- **Employee status**: The attacker must be a Doctolib employee with elevated privileges, granted full operational access.
- **Access level**: This access level pertains to a limited number of employees who are entrusted with critical system capabilities.

Prevention

- **Access limitation**: The number of employees with such high-level access is minimized to reduce potential abuse.
- **Peer review**: Code changes are subject to peer review by at least one other individual, making a code-based attack more challenging and requiring collusion.
- **Audit trails**: Access credentials to the production environment are assigned individually, ensuring that all actions are traceable.
- **Experience requirement**: Employees with elevated privileges must have been with the company for a minimum of four years, ensuring a level of trust and stability.

Difficulty Moderate.

Severity Medium.

Conclusions

- **Data access**: Employees with this level of access can view all user metadata, including contact lists and phone numbers. However, they do not have direct access to message content, such as text messages or images, without breaking the end-to-end encryption.
- **Security comparison**: From a security perspective, such employees are akin to attackers who have compromised network traffic and broken transport-level encryption. While they have access to full SSL certification, this does not grant them additional advantages over an attacker who has compromised transport security.
- **Confidentiality assurance**: Message confidentiality is maintained as long as users have mutually verified their key fingerprints. Without this verification, an attacker could potentially substitute their own key to impersonate users.

In summary, while the risk from internal abuse by privileged employees is mitigated by strict access controls and audit mechanisms, the integrity of message content remains protected by end-to-end encryption. Continuous vigilance and stringent access management practices are essential to safeguard against such internal threats.

10.7 Compromised root certificate authority

Description This threat involves an attacker gaining control over a globally trusted Root Certificate Authority (CA), enabling them to issue fraudulent TLS certificates that are accepted as valid due to the Public Key Infrastructure (PKI) trust model.

Prerequisites

- **Root CA compromise**: The attacker must have compromised a Root CA that is widely trusted by internet users. This allows them to issue TLS certificates that would be recognized as valid by third parties.
- **Certificate issuance**: With control over the Root CA, the attacker can forge certificates to impersonate legitimate organizations or service providers.

Prevention

- **Certificate pinning**: To mitigate this risk, we use TLS certificate pinning. This technique ensures that only certificates from our specified certificate provider are accepted, rejecting any certificates issued by unauthorized or compromised CAs.

Difficulty Very difficult.

Severity Medium to high.

Conclusions

- **Sophistication required**: The complexity of successfully compromising a globally trusted Root CA makes it less likely for such an attack to specifically target our service. The effort required for this attack is significant, thus reducing the probability of occurrence.
- **Potential targets**: The most critical certificates that an attacker might forge would be those of major entities like Apple/Google (which could alter application builds), Amazon (which could impersonate Amazon APIs used for server management), and the Doctolib certificate itself.
- **Mitigation impact**: While certificate pinning offers strong protection against forged certificates, the overall threat level remains medium to high due to the potential impact on trust and data security if such an attack were successful.

In summary, the risk from a compromised Root CA is mitigated by using certificate pinning, which safeguards against unauthorized certificates. However, the severity of the potential impact underscores the importance of continued vigilance and robust certificate management practices.

Reference List

[1] Jee Hea An. *Authenticated encryption in the public-key set- ting: security notions and analyses*. `https://eprint.iacr.org/2001/079`.

[2] Google Android. *Android Keystore system*. `https://developer.android.com/privacy-and-security/keystore`.

[3] Apple. *Apple Security Documentation*. `https://developer.apple.com/documentation/security/ksecattraccessibleafterfirstunlockthisdeviceonly`.

[4] Stephen Celis. *SQLite.swift*. `https://github.com/stephencelis/SQLite.swift`.

[5] Peter Schwabe Daniel J. Bernstein Tanja Lange. *NaCl: Networking and Cryptography library*. `https://nacl.cr.yp.to/`.

[6] Peter Schwabe Daniel J. Bernstein Tanja Lange. *Public-key authenticated encryption: crypto_box*. `https://nacl.cr.yp.to/box.html`.

[7] Peter Schwabe Daniel J. Bernstein Tanja Lange. *The security impact of a new cryptographic library*. `http://cr.yp.to/highspeed/coolnacl-20120725.pdf`.

[8] Peter Schwabe Daniel J. Bernstein Tanja Lange. *TweetNaCl*. `https://tweetnacl.cr.yp.to/`.

[9] Open source. *libsodium-jni*. `https://github.com/joshjdevl/libsodium-jni`.

[10] Open source. *Swift-Sodium*. `https://github.com/jedisct1/swift-sodium`.

[11] Wikipedia. *Forward Secrecy*. `https://en.wikipedia.org/wiki/Forward_secrecy`.

[12] Zetetic. *SQLCipher*. `https://www.zetetic.net/sqlcipher/`.