wire

Wire Security Whitepaper

Wire Swiss GmbH*

October 22, 2020

Contents

1 **Introduction** 1

2 **Registration** 1
- 2.1 User Registration 1
  - 2.1.1 Registration by e-mail 2
  - 2.1.2 Registration by phone 2
  - 2.1.3 Passwords 2
  - 2.1.4 Further data 3
- 2.2 Client registration 3
  - 2.2.1 Further data 4
  - 2.2.2 Metadata 4
  - 2.2.3 Notifications 5
- 2.3 End-of-life of a client 5
- 2.4 Push token registration 5

3 **Authentication** 6
- 3.1 Tokens 6
- 3.2 Login 7
  - 3.2.1 Password login 7
  - 3.2.2 SMS login 7
- 3.3 Password Reset 7

4 **Messaging** 8
- 4.1 End-to-end encryption 8
  - 4.1.1 Prekeys 8
- 4.2 Message exchange and client discovery 9
- 4.3 Assets 10
- 4.4 Link previews 10
- 4.5 Notifications 11

5 **Calling** 11
- 5.1 Call signaling 11

*privacy@wire.com

5.2 Media transport . . . 11
5.3 Encryption . . . 12
5.3.1 1:1 calls . . . 12
5.3.2 Conference calls . . . 12
5.4 Encoding . . . 13
5.5 WebRTC . . . 14

6 Verification 14

7 Further encryption and protection 15
7.1 Transport encryption . . . 15
7.2 Local data protection . . . 15
7.3 App lock . . . 16
7.4 Backups . . . 16

A Cookie and access token format 16

1 Introduction

Wire runs on Android and iOS devices, on Windows, macOS and Linux as well as on web in browsers. Registered users engage in conversations whose contents are synchronized across all devices of a user.

This document provides an overview on the cryptographic protocols and security aspects implemented to protect the privacy of users.

2 Registration

Registration on Wire involves up to three steps, whereby only the first is strictly required in order to start using the service:

1. User registration.
2. Client registration.
3. Push token registration.

2.1 User Registration

Wire supports two basic registration flows, which can optionally be composed. All flows have in common that a profile name must be provided, which does not have to be unique. For more details on the data collected please see the Wire Privacy Whitepaper.

2.1.1 Registration by e-mail

Registration by e-mail (figure 2.1) requires a profile name and a valid e-mail address. To verify the e-mail address, the server generates a random verification

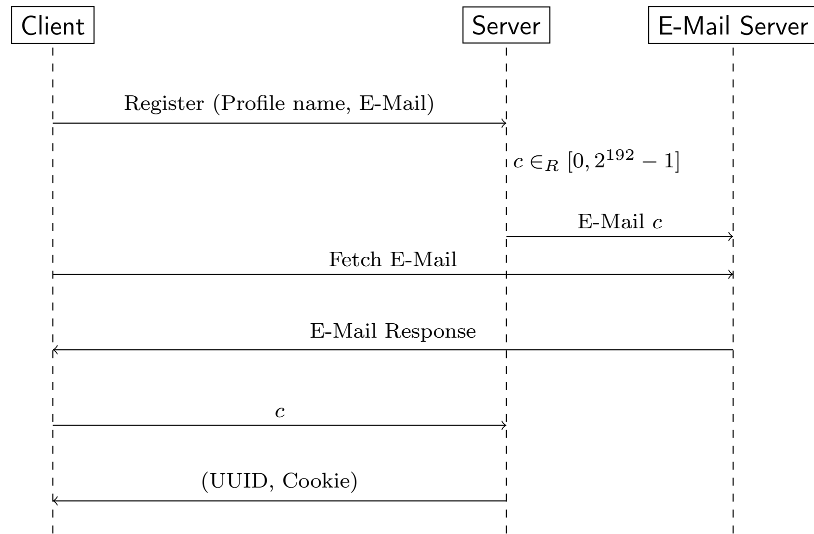

Figure 2.1: User Registration (E-Mail)

code $c \in_R [0, 2^{192} - 1]$ and sends it to the given e-mail address to complete the registration. The server allows only 3 attempts to send the correct verification code before the code is automatically invalidated and a new code needs to be requested. Verification codes expire after two weeks.

Upon successful registration the client receives a randomly generated user ID (UUID v4) and an authentication cookie.

2.1.2 Registration by phone

Registration by phone number (figure 2.2) requires a profile name and a valid phone number. Before the client application sends the actual registration request, it asks the server to send a verification code $c \in_R [0, 10^6 - 1]$ via SMS to the phone number the user provided. The actual registration request includes c. A client only has 3 attempts to send the correct verification code before it is invalidated, in which case a new code needs to be requested. Verification codes expire after two weeks.

Upon successful registration the client receives a Wire internal ID (UUID v4) and an authentication cookie.

2.1.3 Passwords

Passwords are not stored in plain text on the servers, instead they are passed into the *scrypt* key derivation function [4] with the parameters $N = 2^{14}, r = 8, p = 1$ and a random salt $s \in_R [0, 2^{256} - 1]$. The resulting hashes are stored along with the salt and the parameters in the form $\log_2 N \| r \| p \| base64(s) \| base64(hash)$. Clients only keep passwords in volatile memory.

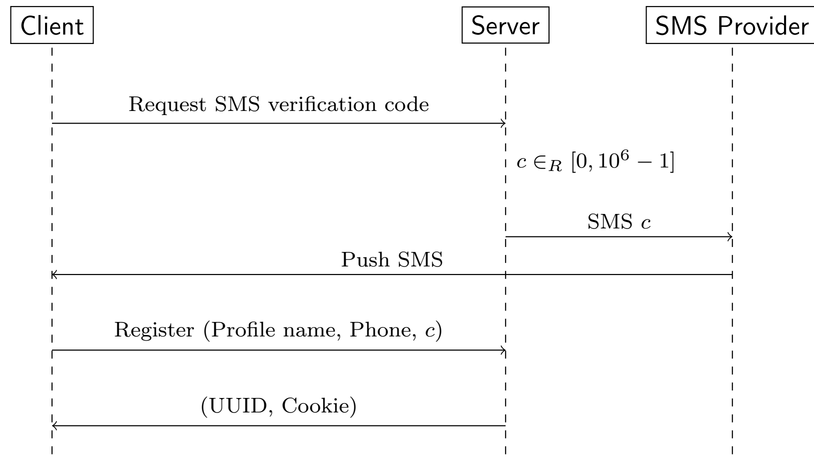

Figure 2.2: User Registration (Phone)

2.1.4 Further data

The following additional data is stored by the servers:

- Locale: An IETF language tag representing the user's preferred language.
- Accent Color: A numeric constant.
- Picture: Metadata about a previously uploaded public profile picture, including a unique ID, dimensions and a tag.
- Cookie Label: A label to associate with the user token that is returned as an HTTP cookie upon successful registration.
- App settings: Preferences such as emoji setting, link preview setting, sound alert setting are stored.

2.2 Client registration

Client registration (figure 2.3) is required in order to participate in the exchange of end-to-end encrypted content. The concept of user accounts is less relevant, as encrypted content is exchanged between two clients.

A user can register up to 8 client applications (usually different devices) in total: 7 are *permanent* 1 is *temporary*. Attempts to register more than 7 permanent clients will result in an error and require a permanent client to be removed. Registering a new temporary client will replace the old one.

These restrictions limit the amount of computation clients need to perform when sending encrypted messages, as messages are encrypted individually between clients.

The prekeys are used by other clients to initiate cryptographic sessions with the newly registered client and are defined in section 4.1.1 on page 8.

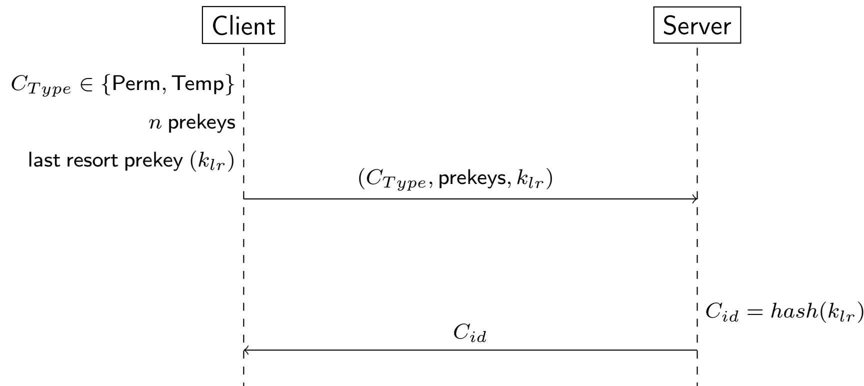

Figure 2.3: Client Registration

Upon successful client registration the server returns a client ID (C_{id}) which is unique per user ID.

2.2.1 Further data

The following data will also be collected during client registration:

- Class: The device class: *Mobile*, *Tablet* or *Desktop*.
- Model: The device model, e.g. iPhone 7.
- Label: A human-readable label for the user to distinguish devices of the same class and model.
- Cookie label: A cookie label links the client to authentication cookies (cf. section 3 on page 6). When such a client is later removed from the account, i.e. when a device is lost, the server will revoke any authentication cookies with a matching cookie label. Once set, cookie labels can never be changed.
- Password: If the user has a password, client registration requires re-authentication with this password, with the exception of the first registered client of an account. Similarly, removing a registered client also requires the password to be entered.

2.2.2 Metadata

The server collects the following metadata for every newly registered client and makes it available it to the user:

- Timestamp: The UTC timestamp when the client was registered.
- Location: The geographic coordinates of the IP address used to register the client.

This information is only collected to make notifications about new registrations more meaningful.

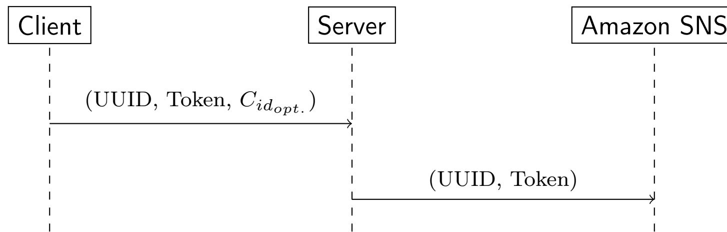

Figure 2.4: Push token registration

2.2.3 Notifications

When a new client is registered with an account, all existing clients of the same account are notified of that event. Additionaly, the user will be notified via e-mail. These notifications help the user to identify suspicious clients registered with their account, e.g. when login credentials are stolen.

2.3 End-of-life of a client

- The user logs out permanently of a client: All local state (on a device) of a client is deleted, including all secrets. The backend retains an entry for that client, including its public identity key.
- A user deletes a client(̈device)̈ from their account on another client: This causes the backend to drop the corresponding entry in the user's devices list. The backend also drops all cookies associated with that client. Additionally, the to-be-deleted client is notified by the backend and erases all local state on the device. The user is prompted for their password for this operation.
- A user account is terminated: All server-side data associated with that account is deleted, including the client list. Like in 2., clients are notified and proceed to erasing all their local data.
- A user uninstalls the Wire app on Android or iOS: In this instance the operating system deletes all data associated with the app, but the event is not communicated to the backend (due to technical infeasibility on mobile platforms). The backend retains an orphaned entry of that client in the user's devices list.

2.4 Push token registration

As a final registration step a client can register push tokens in order to receive push notifications over FCM or APNs when the device is offline (when there is no open websocket connection). Details about push notifications can be found in section 4.5 on page 11.

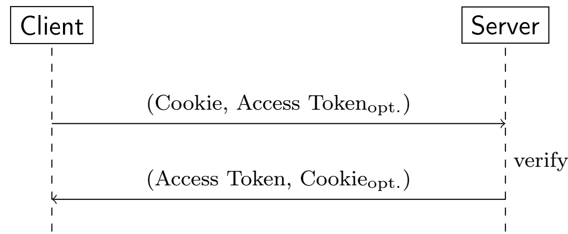

Figure 3.1: Token Refresh

3 Authentication

3.1 Tokens

API authentication is based on a combination of short-lived bearer tokens, referred to as *access tokens*, as well as long-lived *user tokens*. Access tokens are used to authenticate requests to protected API resources and user tokens are used to continuously obtain new access tokens.

User tokens are sent as HTTP cookies. All tokens are strings signed[1] by the server and include the user ID (UUID v4) and the expiration time as a Unix timestamp. The full format of user tokens and access tokens is specified in appendix A on page 16.

The scope of user tokens, and thus cookies, can be *persistent* or *session-based*, with the same semantics as those specified by the HTTP protocol. A client chooses the scope of the cookie during login. The HTTP cookie attributes restrict their use to the domain of the backend server, to the path of the token refresh endpoint as well as to the HTTPS protocol. Persistent cookies are stored in permanent, secure storage on the client, whereas session cookies are kept in ephemeral storage only (e.g. a browser session). Persistent cookies expire after 1 year and session cookies expire after 1 week.

Access tokens are comparatively short-lived (15 minutes). To refresh an expired access token a client uses a cookie to refresh the access token. If the cookie is valid, a new access token is generated and returned. When an access token is refreshed the server may additionally issue a new cookie, thus continuously prolonging the expiration date (figure 3.1). Such a cookie renewal typically occurs approx. every 3 months.

A user may have a maximum of 32 persistent cookies and 32 session cookies, both of which are replaced transparently from least recent to most recent.

After the initial registration, only a user login can generate a new long-lived user token and an access token.

Wire supports two different types of logins described below.

[1] A cryptographic Ed25519 signature attached to the string.

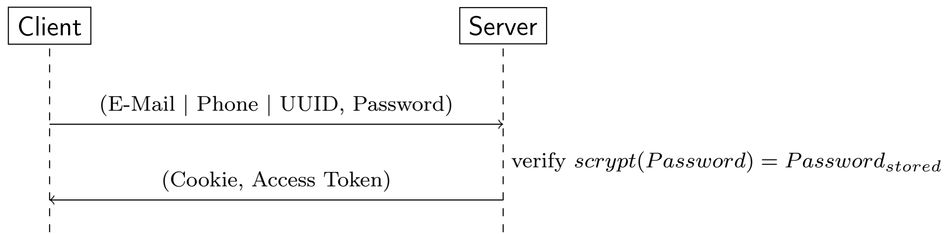

Figure 3.2: Login with password

3.2 Login

Users who have added a password to their account or have verified a phone number can login. Logins are classified as *session* or *persistent* logins, which corresponds to the desired scope of the resulting cookie. Clients can choose the type of login.

3.2.1 Password login

To login with a password a client provides a user ID, e-mail address or phone number and the password, which are transmitted over TLS. The server verifies the password using *scrypt* [4] and issues a new user token as an HTTP cookie and a new access token as shown in figure 3.2.

3.2.2 SMS login

Users who registered with a verified phone number can login via SMS. The procedure is the same as during registration and is subject to the same restrictions, however an SMS login code already expires after 10 minutes.

3.3 Password Reset

Wire provides a self-service password reset [5] for any registered user with a password and a verified e-mail address or phone number.

The procedure for a password reset via e-mail address is similar to the initial verification (cf. figure 2.1 and 2.2), with the following differences:

- There can be only 1 pending password reset for an account at any time. A new password reset cannot be initiated before the timeout window expires.
- The password reset codes are valid for 1 hour.

4 Messaging

Messaging refers exchanging text messages and assets (section 4.3). All messaging in Wire is subject to end-to-end encryption to provide users with a strong degree of privacy and security.

4.1 End-to-end encryption

End-to-end encryption (E2EE) takes place between two clients (cf. 2.2). Proteus [7] is the main cryptographic protocol. It is an independent implementation of the Axolotl/Double Ratchet [8] protocol, which is in turn derived from the Off-the-Record protocol, using a different ratchet[9].

Furthermore Wire uses the concept of prekeys [6] to use the protocol in an asynchronous environment. It is not necessary for two parties to be online at the same time to initiate an encrypted conversation.

Proteus uses the following cryptographic primitives (provided by libsodium [14]):

- ChaCha20 stream cipher [15]
- HMAC-SHA256 as MAC [16]
- Elliptic curve Diffie-Hellman key exchange (Curve25519 [17])

Key derivation is done using HKDF [18].

4.1.1 Prekeys

Every client initially generates some key material which is stored locally:

- Identity keypair: $(a, g^a) \in_R \mathbb{Z}_p \times \mathsf{Curve25519}$ where $g \in \mathsf{Curve25519}$
- A set of prekeys [6]: $(k_{(a,i)}, g^{k_{(a,i)}}) \in_R \mathbb{Z}_p \times \mathsf{Curve25519}$ where $0 \leq i \leq 65535$.

During client registration (section 2.2) a client uploads prekeys $(g^{k_{(a,0)}}, ..., g^{k_{(a,j)}})$ bundled with its public identity key g^a. These are eventually used by other clients to asynchronously initiate an end-to-end encrypted conversation, i.e. given a recipient's prekey $g^{k_{(a,i)}}$ and identity key g^a the sender can derive n initial encryption key even if the recipient is offline.

The prekey with ID 65535 is the so-called "last resort" prekey. Every prekey is intended to be used only once, which means that the server removes every requested prekey immediately. In order to not run out of prekeys the last resort prekey is never removed and clients should regularly upload fresh prekeys.

For further details on the remaining protocol flow and its security properties please refer to references [8], [9], [10] and [13].

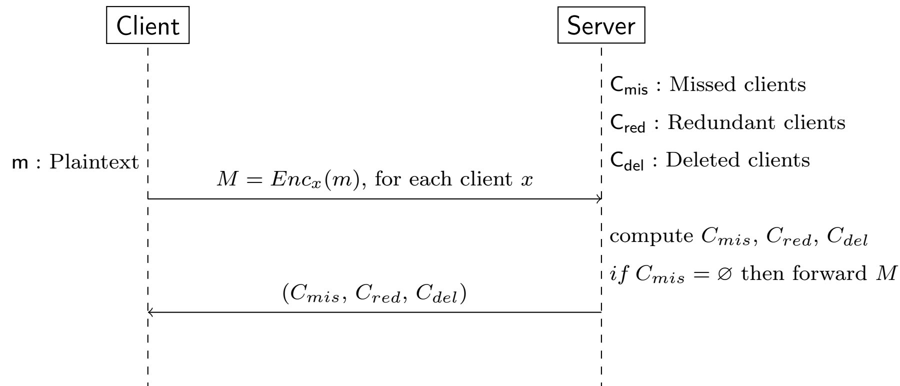

Figure 4.1: Client Discovery (Sender)

4.2 Message exchange and client discovery

To send an encrypted message the sending client needs to have a cryptographic session with every client it wants to send the message to (usually all clients of all participants of a particular conversation). It will encrypt the plain text message for every recipient and send the batch to the server. The server checks if every client of every user who is a participant of the conversation is part of the batch. If a client is missing, the server will reject the request and inform the sender of missing clients.[2] The sender can then fetch prekeys for the missing clients and prepare the remaining messages before attempting to resend the entire batch.

By the same mechanism clients are also informed about redundant clients, i.e. clients they have prepared an encrypted message for, but which are no longer part of the conversation. This includes deleted clients, i.e. clients which are redundant and known to have been deleted. The sender can use this information to update its own list of clients participating in a conversation and the corresponding cryptographic sessions.

Client discovery for the sender of a message is depicted in figure 4.1.

Conversely, when a client receives an encrypted message from another client with whom no prior cryptographic session exists, it initializes a new cryptographic session from the encrypted message.

To rule out man-in-the-middle attacks users need to compare identity key fingerprints out-of-band.

4.3 Assets

Assets are larger binary entities sent between users, such as pictures.

Profile pictures are uploaded as plaintext assets with technical metadata (e.g. width, height, file type) and are shared through a user's profile.

[2]Clients do have the ability to override this behaviour, but are always informed about missing clients.

Any other assets shared in conversations are end-to-end encrypted. Compared to regular text messages, the encryption of assets applies an optimization proposed in [11] to reduce the required computational overhead and network bandwidth for the sender. On Wire, the sending client does the following:

1. It generates a random symmetric key k for use with AES-256.
2. It encrypts the asset data with k using CBC mode with PKCS#5/7 padding and computes the SHA-256 hash of the resulting ciphertext.
3. It encrypts the key k together with the hash and other asset metadata for each recipient via the Proteus protocol.
4. It sends the encrypted asset data as well as the encrypted metadata payload for each recipient to the server.

The receiving client of an asset metadata message then does the following:

1. It decrypts the asset metadata using the Proteus protocol, thus obtaining the symmetric key k as well as the SHA-256 hash of the asset ciphertext.
2. It downloads the asset ciphertext, computes the SHA-256 hash and compares it to the received hash to verify the integrity of the asset data.
3. It decrypts the asset data using the key k.

As with regular text messages, only clients in the same conversation can receive asset metadata messages from one another and are authorized to download the corresponding asset ciphertext.

Assets are persistently stored on the server without a predefined timeout. This means that a client can repeatedly download and decrypt the same asset to conserve disk space on the device, since the client persistently stores the decrypted symmetric key k together with the SHA-256 hash. These credentials have the same sensitivity as the plaintext asset itself. Forward secrecy is not affected since the decryption key k is sent using the Proteus protocol.

4.4 Link previews

When users share links, the apps can generate link previews. This feature is optional and can be turned off. Link previews are generated on the sender's side only, by fetching open graph data (a picture and some text) from the website behind the link. The data is sent to the recipient and displayed there, but the recipient does not make any network requests to the website, unless the link is clicked or tapped.

4.5 Notifications

Messages are delivered by the server to recipients via notifications. Notifications are delivered by Wire over 3 different channels.

Websocket connections: Every authenticated client can establish a websocket connection over HTTPS. A client with an established websocket connection is considered *online*.

External push providers: Wire currently supports FCM and APNs as external push providers. This channel is used if a client is offline but has registered a valid FCM or APNs push token with the server. The content is encrypted and not visible to the external push providers.

Notification queues: Every message sent by a user, as well as most metadata messages are enqueued in a per-client notification queue that can be queried (and filtered) by every registered, authenticated client of a user. The notification queue allows clients to retrieve messages they may have missed. The retention period of notifications is 4 weeks.

5 Calling

Wire users can call each other in 1:1 or group conversations. Calls are initiated to all participants of a conversation and users who are not a participant of that conversation do not have access to the call. All calls are end-to-end encrypted.

Setting up a call involves three aspects: signaling, media transport and encryption. These are described in detail next.

5.1 Call signaling

Call signaling establishes a connection between clients. For 1: calls, their common capabilities are negotiated by exchanging SDP messages. For conference calls, additional signalling messages inform call participants in real time of who has joined and left the call. These messages are sent between clients as Proteus messages, using the same encryption as text messages.

5.2 Media transport

Once connected, endpoints determine a transport path for the media between them. Whenever possible the endpoints allow direct media flow between them, however some networks may have firewalls or NATs preventing direct streaming and instead require the media to be relayed through a TURN server. ICE identifies the most suitable transport path.

While TURN servers are part of the Wire infrastructure, they do not know the user ID of the users that use them. Clients use generic credentials to authenticate against the TURN servers, so that calls are indistinguishable for TURN servers. Therefore TURN servers cannot log identifiable call records.

5.3 Encryption

5.3.1 1:1 calls

Call media is exchanged between endpoints in an SRTP-encrypted media session. To initiate the session the SRTP encryption algorithm, keys, and parameters are negotiated through a DTLS handshake. The authenticity of the clients is also verified during the handshake by sending the expected fingerprints over the existing authenticated Proteus session.

5.3.2 Conference calls

Overview Wire conference calls use existing WebRTC mechanisms to establish peer connections between clients and Selective Forwarding TURN server (SFT). On those legs, all data is encrypted in the same way as on 1:1 calls. In addition, Wire clients use Insertable Streams [20] to end-to-end encrypt the content of media packets. The details of this encryption is described in the following paragraphs.

Cryptographic Primitives Conference calling uses AES-GCM-256 to encrypt the payload. For key derivation HKDF-SHA512 is used.

Eras and Epochs When Wire clients initiate a call, they allocate a session on the SFT for that particular call. Every 30 seconds the era counter is incremented by 1. Time is measured from the start of the call. When a user leaves the call a new epoch is triggered, the epoch counter is incremented by 1, and the era is reset to 0. The index i of the key for epoch e and era a is calculated as:

$$i = (e\&0xFFFF) << 16|(a\&0xFFFF)$$

Key Schedule The initial key (known as Call Secret, CS) is generated by a dedicated participant in the call — called the Key Generator — and sent to other participants as a Proteus message. The derived media key K — used to encrypt the media stream — is rotated every 30 seconds, independent of users joining or leaving to achieve Forward Secrecy. Additionally, the Key Generator generates a fresh key whenever users leave the call and distributes call information to the group participants to achieve Post-Compromise Security.

The media key K_i for era i is generated as follows using the 32-byte call secret CS generated by the Key Generator:

- S_i = HKDF(ikm = S_{i-1}, salt = call_id, info = “session_key”, len = 32)
- K_i = HKDF(ikm = S_i, salt = call_id, info = “media_key”, len = 32)

S_0 is set to CS, as is S_i for the first era in an epoch. CS is derived as follows:

- CS = HKDF(ikm = r, info = “cs”, salt = confpart_data, len = 32)

where r are 32 random bytes and confpart_data is a subset of the most recent CONFPART message from the SFT containing the timestamp, sequence number, and list of participants.

When a user joins, the Key Generator sends S_i for the current era as a Proteus message to all call participants and optionally S_j, which corresponds to the new CS, for the future epoch if one has been generated.

When a user leaves, the epoch counter is increased and a new key is generated as S_i (where $i = (e\&0xFFFF) << 16$). This key is then sent to all users currently in the call as a Proteus message. The next era transition moves to the new epoch key.

When the Key Generator leaves the SFT indicates the next user in the list to be the new Key Generator. This new Key Generator waits 5 seconds before triggering a new era, generating a new epoch key and starts the era timer, so that 30 seconds later the new epoch key is used.

Clients keep key material from up to 4 previous eras to ensure that they can decrypt all frames. The code actively ignores keys from anyone not recognized as the Key Generator to prevent other clients trying to set the Call Secret.

IV Derivation Because the 12 bytes needed for the IV are relatively large compared to the size of real-time media packages, the IV is not randomly generated but derived from values known to all participants.

First, a base for the IV is derived as

$$B = \text{HKDF}(\text{key} = \text{UID}, \text{salt} = s, 12)$$

where UID is the client's anonymized user id and s a string identifying the stream, i.e. either "video_iv" or "audio_iv".

The IV is then computed as

$$FIV = B \oplus (FRM||KID)$$

where $(FRM||KID)$ denotes the concatenation of the frame counter and the key ID. In particular B is XOR'ed with the frame counter (32 bit unsigned) at bytes $0-3$ and the key index (32 bit unsigned) at bytes $4-7$.

5.4 Encoding

The codec used for streaming is Opus for audio and VP8 for video. Opus can use variable bit rate encoding (VBR) or constant bitrate encoding (CBR). Users can choose to enable CBR in the settings. CBR has the advantage of eliminating potentially undesired information about packet length, but might have an impact on call quality on slow networks[21]. It is sufficient if one of the two parties of a call enables the CBR option, CBR will then always be used for calls of that user. When CBR is used, the calling screen will display 'CONSTANT BIT RATE'. In group calls the CBR option is only enforced on the legs connected to the participant that enabled the CBR option. It is up

to the participants to set the option to ensure that all legs use CBR encoding. The calling screen of group calls will not display 'CONSTANT BIT RATE', even when the option is set. In video calls the CBR option affects the audio streams like in audio calls, but the calling screen will not display 'CONSTANT BIT RATE'.

5.5 WebRTC

Wire is fully compliant with WebRTC and existing IETF standards. As a result, Wire native endpoints can also securely exchange media with any WebRTC compliant web-browser such as Google Chrome or Mozilla Firefox.

These are the main IETF standards used by Wire:

- UDP (RFC 768[22])
- RTP (RFC 3550[23])
- ICE (RFC 5245[24])
- STUN (RFC 7350[25])
- TURN (RFC 5766[26])
- SDP (RFC 4566[27])
- SRTP (RFC 3711[28])
- DTLS (RFC 4347[29])
- DTLS-SRTP (RFC 5764[30])
- Opus (RFC 6716[31]).

6 Verification

As mentioned in 4.1, each client has its own cryptographic identity. In order to ensure that no man-in-the-middle attack can take place, the cryptographic identities should be verified out-of-band. For that purpose the fingerprint of each client in a conversation can be viewed and compared. After comparing, the client can be marked as *verified*. Once all clients in a conversation are marked as verified, the conversation itself will be treated as a verified conversation and a blue shield next to the conversation name will appear.

If a new client is added to the verified conversation, users are warned by a system message. The blue shield will disappear until the new clients have been verified.

7 Further encryption and protection

7.1 Transport encryption

Wire clients interact with backend servers over HTTPS connections supporting only TLSv1.2. Only cipher suites that support forward secrecy (PFS) are used. The server indicates the order preference of cipher suites and communicates HTTP Strict Transport Security (HSTS) [1] to all clients.

To mitigate man-in-the-middle attacks caused by rogue or compromised CAs or caused by undesired root certificates installed on the client- side, Wire clients pin the public key of the leaf certificates to a set of hard-coded values. This means that clients expect the public key of the leaf certificate to be a certain value. If this is not the case the TLS handshake is aborted, and the connection is never initiated. The pinned keys are hard-coded into the apps and are used for all outgoing connections to the Wire backend.

On mobile devices, clients use the operating system?s API for TLS connections. The desktop app uses Electron, which is based on Chromium that uses BoringSSL as a TLS library.

In addition to requests to HTTP resources, clients can maintain a websocket connection to receive real-time push notifications, as well as register for push notifications through external transports such as FCM [2] on Android and APNS [3] on iOS. See section 4.5 on page 11 for details on push notifications.

7.2 Local data protection

Wire apps store the content of conversations such as text messages, images and other assets locally on the device. Depending on the platform, different protection mechanisms exist:

1. iOS: Local data is stored using Core Data and in files (both protected in with NSFileProtectionCompleteUntilFirstUserAuthentication). Conversation content, cryptographic key material and other sensitive data is not synced with iCloud or iTunes backups. Local data can only be accessed from the Wire app, it is inaccessible to other apps thanks to the iOS sandboxing.
2. Android: Local data is stored using SQLite and in files. Conversation content, cryptographic key material or other sensitive data is not synced with Android Backup Service. The local data can only be accessed from the Wire app, it is inaccessible to other apps thanks to the Android permissions. The app sometimes keeps cached data (i.e. downloaded images) on the external storage (SD card). Those files are encrypted using AES128, each file uses a different random key which is stored in the private database.
3. Desktop clients: Local data is stored using IndexedDB. The data is stored in the user's folder. It is strongly recommended to use full disk encryption like FileVault on macOS or Bitlocker on Windows.

7.3 App lock

On iOS and Android opening the application can be protected by an app lock mechanism (Settings, Options, Lock with Passcode). The app can be unlocked with the biometric sensors of the device (if available and enabled) or passcode in the same way the device can be unlocked. If this type of authentication from the operating system is not available, a password can be used to unlock the app.

7.4 Backups

On iOS and Android, backups are encrypted with XChaCha20Poly1305 and Argon2 is used for key derivation.

Appendices

A Cookie and access token format

⟨*token*⟩	::=	⟨*signature*⟩ ‘`.v=`’ ⟨*version*⟩ ‘`.k=`’ ⟨*key-index*⟩ ‘`.d=`’ ⟨*timestamp*⟩ ‘`.t=`’ ⟨*type*⟩ ‘`.l=`’ ⟨*tag*⟩ ‘`.`’ ⟨*type-specific-data*⟩
⟨*version*⟩	::=	⟨*Integer*⟩
⟨*key-index*⟩	::=	⟨*Integer*⟩
⟨*timestamp*⟩	::=	⟨*Integer*⟩
⟨*type*⟩	::=	‘`a`’ | ‘`u`’
⟨*tag*⟩	::=	‘`s`’ | ‘’
⟨*type-specific-data*⟩	::=	‘`a=`’ ⟨*access-data*⟩ | ‘`u=`’ ⟨*user-data*⟩
⟨*access-data*⟩	::=	⟨*UUID*⟩ ‘`.c=`’ ⟨*Word64*⟩
⟨*user-data*⟩	::=	⟨*UUID*⟩ ‘`.r=`’ ⟨*Word32*⟩

References

[1] https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

[2] https://firebase.google.com/docs/cloud-messaging/

[3] https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html

[4] http://www.tarsnap.com/scrypt.html

[5] https://en.wikipedia.org/wiki/Self-service_password_reset

[6] https://whispersystems.org/blog/asynchronous-security/

[7] https://github.com/wireapp/proteus

[8] https://github.com/trevp/axolotl/wiki

[9] https://whispersystems.org/blog/advanced-ratcheting/

[10] https://whispersystems.org/blog/simplifying-otr-deniability/

[11] https://whispersystems.org/blog/private-groups/

[12] https://github.com/WhisperSystems/Signal-Android

[13] https://eprint.iacr.org/2014/904.pdf

[14] https://github.com/jedisct1/libsodium

[15] https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant

[16] https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

[17] https://en.wikipedia.org/wiki/Curve25519

[18] https://tools.ietf.org/html/rfc5869

[19] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[20] https://w3c.github.io/webrtc-insertable-streams

[21] https://medium.com/@wireapp/call-security-constant-bit-rate-encoding-and-improving-webrtc-a85be6caa43a

[22] http://tools.ietf.org/html/rfc768

[23] http://tools.ietf.org/html/rfc3550

[24] https://tools.ietf.org/html/rfc5245

[25] https://tools.ietf.org/html/rfc5389

[26] https://tools.ietf.org/html/rfc5766

[27] https://tools.ietf.org/html/rfc4566

[28] https://tools.ietf.org/html/rfc3711

[29] https://tools.ietf.org/html/rfc4347

[30] http://tools.ietf.org/html/rfc5764

[31] https://tools.ietf.org/html/rfc6716