Security Whitepaper

Wire Germany GmbH
May 2025

Introduction

Wire is a complete end-to-end encrypted messaging platform. Users can collaborate and organize their chats in groups, send files to each other, and have audio/video calls. User accounts are organized in the form of Teams, while each user can only be part of one Team.

Wire's server component, the backend, can be used as-a-service or self-hosted - in a private cloud or even in environments disconnected from the Internet. Federation with other backends is also possible. Wire client applications are available for Android, iOS, Windows, macOS, Linux, as well as for web browsers.

About this document

This document provides an overview about the features and cryptographic protocols of the product for a technical audience. Wire has an extensive online documentation available under docs.wire.com which describes the product, architecture and even installation of the backend components in detail.

Open Source

All security-critical parts of Wire are available on Github under an GPLv3 license. All clients and the backend can be downloaded, inspected and built by anyone. All parts for running a fully functional deployment for hobbyist use and security researchers are available on Github. The Wire backend supports reproducible builds. Parts of the codebase that would allow or help our competitors to copy Wires business, are not open source.

Terminology

This section intends to clarify the meaning of some ordinary terms which could be misinterpreted by the reader, but does not intend to provide full definitions of all concepts and features used. Such definitions are introduced where needed.

Wire-specific terminology

This section introduces Wire-specific terminology in alphabetical order.

Backend The Backend of a Wire instance describes all components providing the Wire service. Wire clients contact the backend to exchange data with other clients. The backend also provides data exchange between its own clients and clients of federated *domains*.

Client Each device of a user that is logged into the user's account acts as a client. A client is the user interface to securely communicate with other users. One user can have multiple clients, which all have a synchronized view on the conversations of a user. Thus, when sending chat messages, other users are usually addressed as a set of clients.

Client-ID Each client has an identity, its Client-ID, randomly assigned by the backend. A fully qualified client-ID consists of user-ID, client-ID, and backend domain.

Conversation An encrypted data exchange channel between two or more users is called a conversation. A conversation exclusively between two users is called one-to-one conversation.

Domain An instance of a *backend* is called a domain and is addressable via its domain name.

Federation Different Wire *backends* can be connected through federation to allow users from one domain to communicate with users from other directly federated domains.

Group A set of multiple users in a conversation are called a group. Members of a group can be added and removed.

Team Teams act as an organizational unit on a *backend*. Multiple users can be managed in form of teams. Each user on a *backend* can only be part of one team.

User Through the login process users authenticate themselves against the *backend*. A user can have multiple clients to run Wire on different devices.

Identity management-specific terminology

ACME Automatic Certificate Management Environment. A protocol defined in RFC 8555 for automated certificate enrollment, renewal, and revocation. It defines ACME server and ACME client roles.

Certificate Authority (CA) A certificate issuer authoritative for a specific group or type of subjects, often scoped within a single Internet domain.

Certificate Issuance The general process of obtaining a valid certificate (includes enrollment and renewal).

Certificate Revocation A document signed by a relevant CA declaring that a previously issued, and otherwise valid, certificate is no longer trustworthy.

Certificate Signing Request (CSR) A self-signed document used by a subject to request a certificate. A piece of information asserted (claimed) about an entity.

Credentials An assertion of identity (and optionally other identity-related properties) that is cryptographically verifiable. Certificates, W3C verifiable credentials, and W3C verifiable presentations are all forms of a credential.

CRL Certificate Revocation List. A list of revoked certificates typically provided by a certificate authority or other interested party (for example browser vendors and public interest foundations often provide CRLs for Internet domain name certificates).

DPoP OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer, RFC 944. A specification for constructing proofs and authorization tokens which prove possession of a private key.

Identity Authority* A service which is authoritative for identities in a specific area of control. In the context of this document, specifically the combination of certificate authority and ACME server.

Identity Provider (IdP) A service which provides a source for validating user identity in a federated identity system, for example using OAuth/OIDC and/or SAML.

JOSE JSON Object Signing and Encryption. A set of standards including JSON Web Signing (JWS) and JSON Web Encryption (JWE).

JWT JSON Web Token. Pronounced "jot". A format for conveying authorization tokens using JWS. Consists of a JOSE header section, a "claims section", and a signature.

OAuth The Open Authorization family of standards, very popular for user authentication. Version 2.0 of the framework is defined in RFC 6749.

OIDC OpenID Connect. A set of profiles of OAuth 2.0 defined at the OpenID Specifications. OIDC has largely replaced SAML for new SSO applications.

Relying Party (RP) OAuth 2.0 client application requiring end-user authentication and claims from an OpenID provider.

Self-signed Certificate A certificate where the subject and the issuer are the same entity.

SSO Single Sign-On An authentication scheme that allows a user to log in with a single ID to any of several related, yet independent, software systems.

SubjectAltName (SAN) SAN is the subject's alternative name fields in X.509. This field has effectively replaced the subject field as the SAN can be used to clearly express different types of subjects, including URIs, email addresses, and DNS names. Multiple identifiers are permitted in the SAN.

X.509 certificate A specific format of a certificate. A certificate is a document that asserts that a specific public key is associated with the subject. The certificate is signed by the issuer with a specific signature algorithm and valid for a particular time period. Certificates outside the validity period are either not yet valid or expired. In the context of this document "certificate" will always mean an X.509 certificate.

MLS-specific terminology

Authentication Service (AS) MLS-specific term from RFC 9420. Authentication in MLS is only defined as an abstract functionality. An authentication service has to provide two functions:

1. Issue credentials, which provide bindings between identities and signature key pairs, to clients.
2. Provide functionality to allow a member to verify a credential of group members.

In Wire's implementation, the *Delivery Service* and *Authentication Service* are one component: the Wire backend.

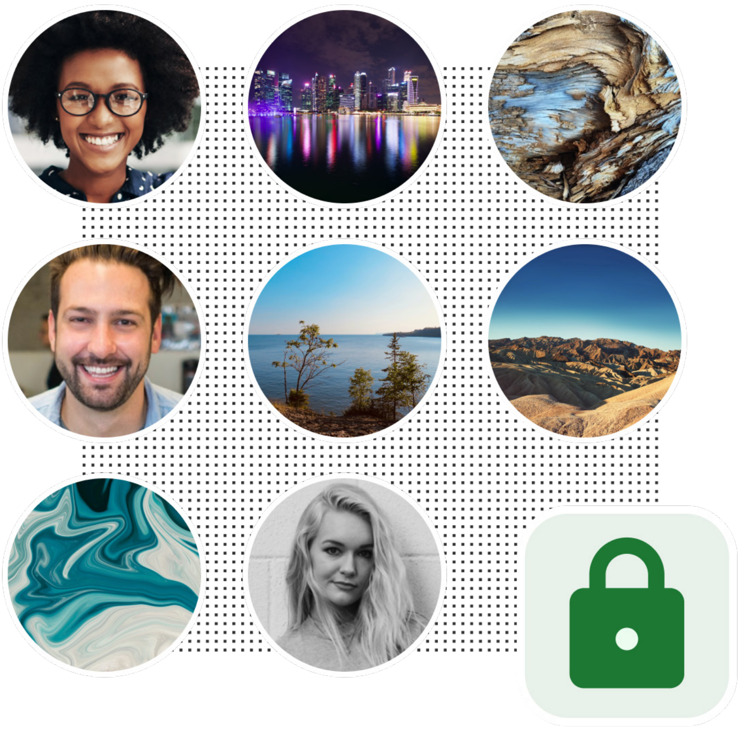
Delivery Service (DS) MLS-specific term from RFC 9420. The *Delivery Service* routes MLS messages among the participants in the protocol.

Security-specific terminology

Post-Compromise Security (PCS)
The notion of Post-Compromise Security was introduced by Cohn-Gordon et al. Intuitively, a protocol provides PCS if, after a full state compromise, the involved clients can recover in such a way that the adversary cannot decrypt future messages.In their definition, "full state compromise" refers to an adversary taking a snapshot of the client's state.

Without further assumptions, achieving PCS is only possible if the adversary only observes traffic passively during a period of time after a full state compromise.PCS is typically achieved if the compromised client uses randomly sampled key material after the compromise, to either negotiate a new unrelated key or to inject entropy into some previously agreed-upon key and communicating this action to its partner.

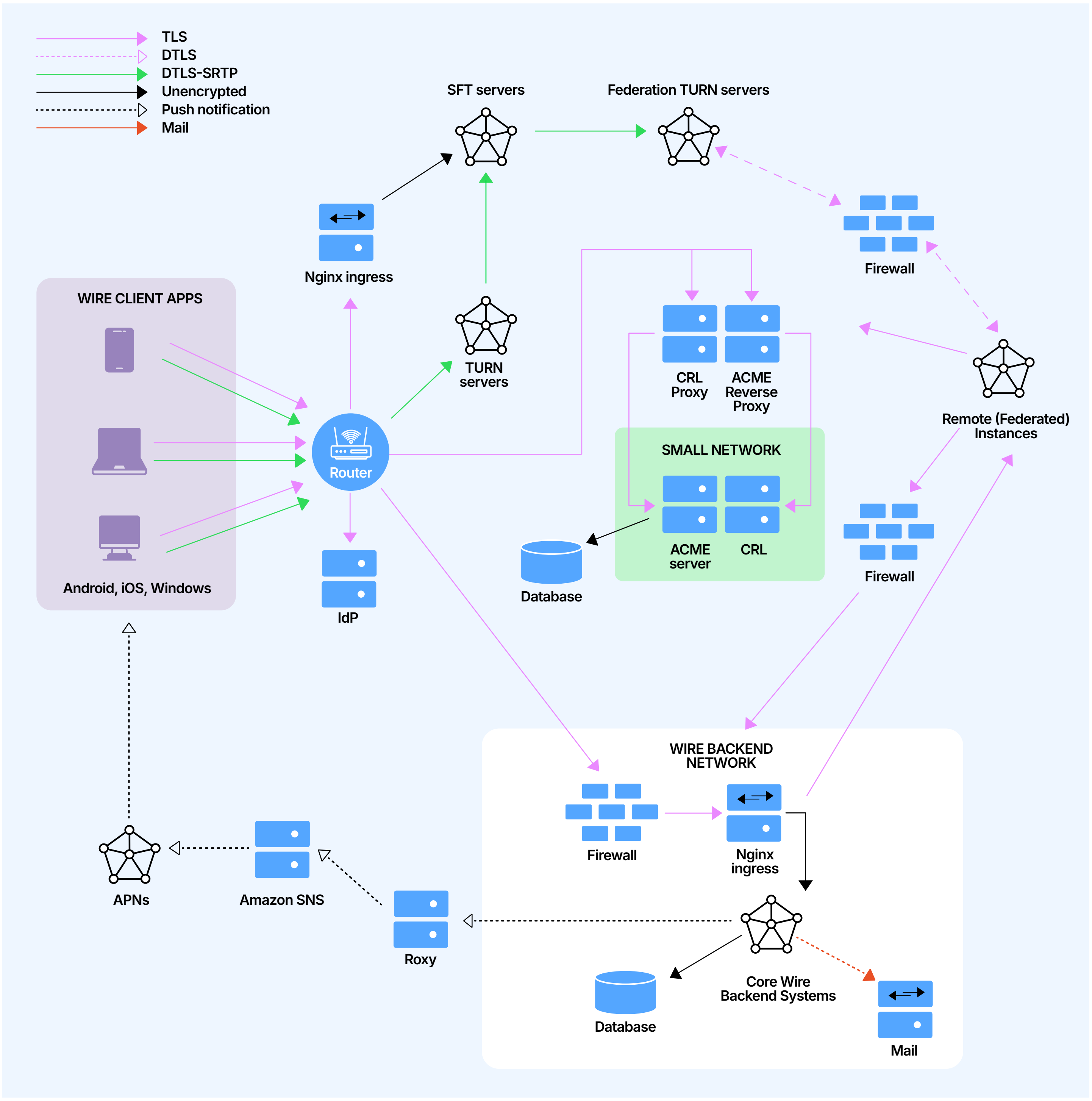
Forward Secrecy (FS)
Forward Secrecy is the counterpart to PCS, i.e. intuitively a protocol provides Forward Secrecy if, with a full state compromise, the adversary cannot compute keys exchanged in *past* protocol sessions. This means that messages sent at a certain point in time are secure in the face of later compromise of a group member.



Architecture overview

Below architecture diagram shows an example architecture of a Wire on-prem installation that leverages the ID Shield feature for automated end-to-end identity verification.

Overview of the Wire platform's components with a focus on the backend (server-side parts). The blue box denotes the core Wire server, the green box the optional CA used for identity verification, and the TURN/SFT servers at the top are used for calling support.



Registration

Registration on Wire involves three steps:

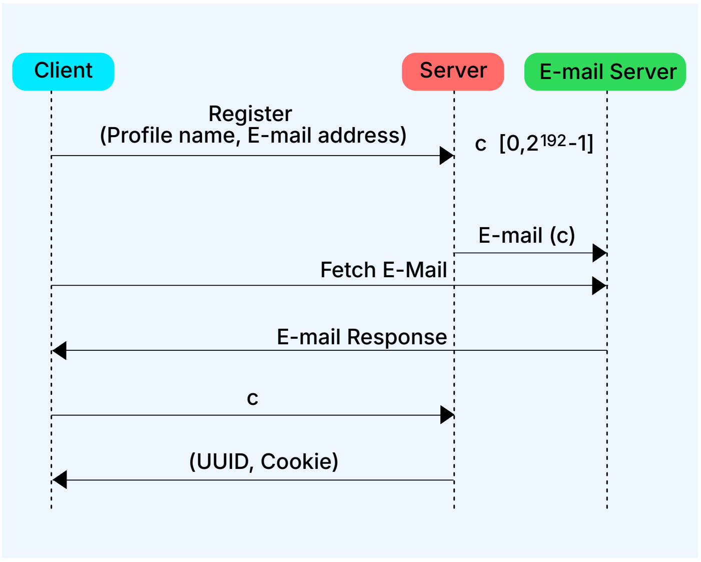
1. User registration - create the user account on the backend
2. Client registration - make a client known to the backend
3. Push token registration - allow the client to receive notifications

User Registration

A user must only register once, in order to create the account.

Registration by E-Mail

In order to create the user account, the backend server verifies that the client has control over the given e-mail address.



Registration by e-mail requires a profile name and a valid e-mail address. To verify the e-mail address, the server generates a random verification code c and sends it to the given e-mail address. The server only allows 3 attempts of the client to send the correct verification code. Afterwards, the code is automatically invalidated and a new code needs to be requested. Verification codes expire after 14 days. Upon successful registration, the client receives a randomly generated user ID (UUID v4) and an authentication cookie.

Registration by SCIM

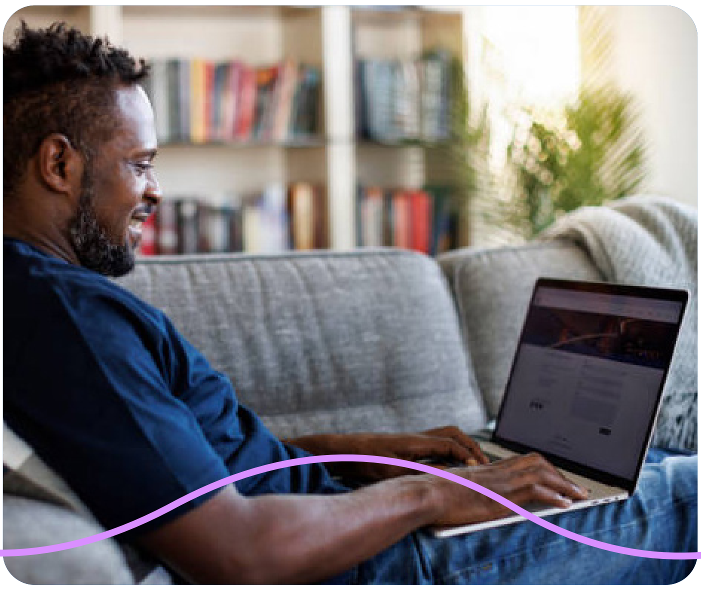
User accounts can also be added automatically using SCIM (System for Cross-domain Identity Management, RFC 7643. The SCIM client software must be authorized by a Team Owner to use the SCIM-API in order to create and update user account information.

User Login

Wire supports user login by account password or SSO (Single Sign On) via SAML (Security Assertion Markup Language). Details on this can be found in the Login section.

Passwords

Passwords are not stored in plain text on the server.Instead, upon login they are passed into the `scrypt` key derivation function 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 $log_2 (N) || r || p || base64(s) || base64(hash)$. Password hashing can also be configured to use argon2id instead of scrypt. Clients only keep passwords in volatile memory.



Password Policy

The default password complexity (as enforced by clients) is as follows:

- at least 8 characters long
- at least 1 lowercase letter
- at least 1 uppercase letter
- at least 1 number
- at least 1 special character

Further User Data

The following additional data is stored by the backend:

- **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.

If your are interested in more details about Wire's data privacy, have a look at the Wire Privacy Whitepaper that summarizes data processed by Wire clients and backends.

Client Registration

After creating an account, a user can register Wire client applications. Client registration 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.

It is possible to register up to 8 client applications (usually different devices) in total: 7 are permanent, 1 is temporary. Temporary devices are Wire web browser clients. Registering a new temporary client will replace the old one. This restriction on the number of clients limits the amount of computation required for sending encrypted messages. Upon successful client registration, the server returns a client ID (C_{id}) which is unique per user ID.

URI format of Wire handles and Wire Client-IDs

Both Wire handles and Wire client-IDs need to be represented in end-to-end identity certificates, in ACME messages and related DPoP challenges. The most convenient way to represent these identifiers in X.509 certificates and to distinguish one type from the other is using an URI.

Client-ID format

The fully-qualified Wire client-ID URI format is shown below.

Example: wireapp:2XlgqeneTyOPiefTp6R-SA!ce6af3facf225073@example.com`

- **wireapp:** the specific protocol (the Wire protocol)
- **2XlgqeneTyOPiefTp6R-SA** an unpadded base64url representation of the user-ID part of a qualified Wire client-ID. This corresponds to the traditional UUID representation of: **d97960a9-e9de-4f23-8f89e7d3a7a47e48**
- **!** the separator between the user-ID and client-ID portions
- **ce6af3facf225073** the unqualified, 64 bit client-ID in hexadecimal
- **@** the separator between the client-ID portion and the domain
- **example.com** the backend domain

Wire Handle format

The fully-qualified Wire handle URI format is shown below.

Example: wireapp:%40alice.smith@example.com

- **wireapp:** the specific protocol (the Wire protocol)
- **%40** the URI encoding of the @ character which signals a Wire handle
- **alice.smith** the unqualified portion of the Wire handle without the leading "@" character
- **@** the separator between the unqualified portion of the handle and the domain
- **example.com** the backend domain

Further Client Data

The following data will also be collected during client registration:

- **Class:** The device type; such as Mobile, Tablet or Desktop.
- **Model:** The device model, e.g. iPhone 12.
- **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. 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. Similarly, removing a registered client also requires the password to be entered.

Metadata

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

- **Timestamp:** The UTC timestamp when the client was registered.

Notifications on client registration

When a new client is registered with an account, all existing clients of the same account are notified of that event. Additionally, 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.

End-of-life of a client

→ **The user permanently logs out of a client:** All local state on the device of the client application is deleted, including all secrets. The backend retains an entry for that client, including its public key material.

→ **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.

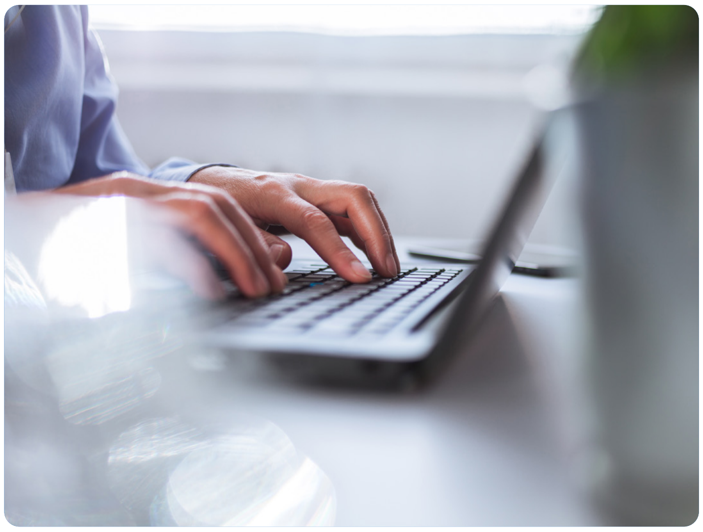
→ **A user account is terminated:** All server-side data associated with that account is deleted, including the client list. The user's clients are notified, like above, and proceed to erase all their local data.

→ **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.

Push token registration

As a final registration step, a client can register push tokens in order to receive push notifications over Google's Firebase Cloud Messaging (FCM) or Apple Push Notification Service (APNs). Those services are used by the backend to notify the client app about available messages while the client does not have an active WebSocket connection open to the backend. In addition to that, Wire also offers a F-Droid version of the Wire app that is preconfigured to avoid FCM and receive notifications via websockets.

Details about push notifications can be found in the respective section.



Authentication & Authorization

Access to the functions of the backend service APIs require authentication.

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.

Both token types have the following structure:

```
token ::= <signature> "." <version> "." <key-index> "." <timestamp> "." <type> "." <tag> "." <type-specific-data>
signature ::= Ed25519 signature
version ::= "v=" Integer
key-index ::= "k=" Integer (> 0)
timestamp ::= "d=" Integer (POSIX timestamp, expiration time)
type ::= "t=" ("a" | "u" | "b" | "p") ; access, user, bot, provider
tag ::= "l=" ("s" | "") ; session or nothing
type-specific-data ::= <access-data> | <user-data> | <bot-data> | <provider-data>
access-data ::= "u=" <UUID> "." "c=" <Word64> ("i=" <Word32> | "")
user-data ::= "u=" <UUID> "." "r=" <Word32> ("i=" <Word32> | "")
bot-data ::= "p=" <UUID> "." "b=" <UUID> "." "c=" <UUID>
provider-data ::= "p=" <UUID>
```

They are strings signed by the backend and include the user ID (UUID v4) and the expiration time as a Unix timestamp.

The long-lived user tokens have a binding to the corresponding user (u) and a 32 bit connection identifier (r), generated by the backend. Additionally, a user token may contain a binding to a client ID.

User tokens are sent as HTTP cookies and are therefore also called cookies. The scope of user tokens 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. 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.

The short-lived access tokens have a binding to the corresponding user (u) and contain an additional 64 bit random number (c) that stays constant through renewals. Additionally, an access token may contain a binding to a client ID (i).

Access tokens are comparatively short-lived (15 minutes). To refresh an expired access token, a client uses a cookie to obtain a new 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. Such a cookie renewal typically occurs approx. every 3 months.

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

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.

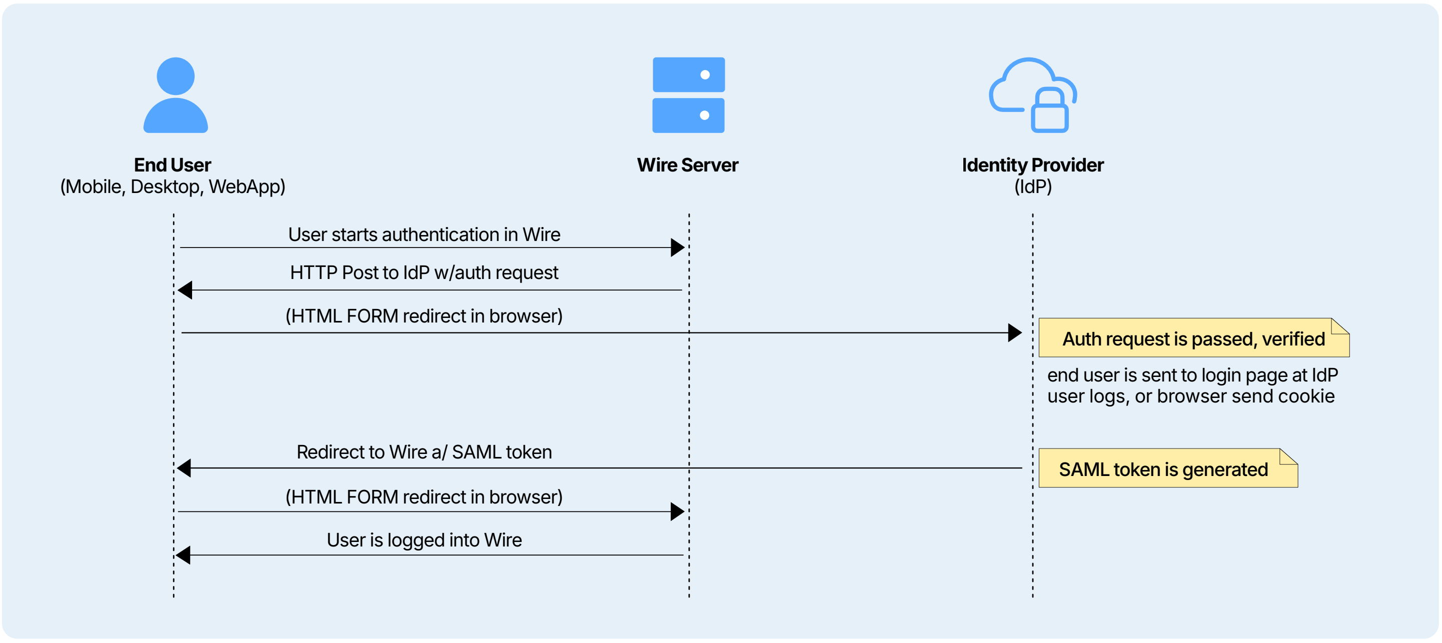
Login using a password

To login with a password, a client provides a user ID, e-mail address and the password, which are transmitted over TLS. The server verifies the password using scrypt (see section Passwords) and issues a new user token as an HTTP cookie as well as a new access token.

Login using SAML SSO

SSO (Single Sign-On) is technology allowing users to sign into multiple services with a single identity provider/ credential. For this, a Team administrator must configure the SAML connection with the IdP to be used.

Using SSO, the login flow then works as follows:


End User
(Mobile, Desktop, WebApp)
Wire Server
Identity Provider
(IdP)
User starts authentication in Wire
HTTP Post to IdP w/auth request
(HTML FORM redirect in browser)
Auth request is passed, verified
end user is sent to login page at IdP
user logs, or browser send cookie
Redirect to Wire a/ SAML token
SAML token is generated
(HTML FORM redirect in browser)
User is logged into Wire

Password Reset

Wire provides a self-service password reset for any registered user with a password and a verified e-mail address. The procedure for a password reset via e-mail is similar to the initial verification (see section Registration by E-Mail), 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.

A password reset code has two parts, key and code. The key is a SHA256 hash of the user ID. The code is a random base64 encoded string of 24 bytes. If someone tries to guess the code for a given key incorrectly 3 times in a row, the key-code is deleted from the backend and the password cannot be reset with this key anymore. Resetting the password invalidates all cookies, so that the user has to log in again on existing devices.

Messaging

Messaging refers to exchanging text messages and various types of files, called "assets" (see section Asset Encryption). All messaging in Wire is subject to end-to-end encryption, in order to provide users with a strong degree of privacy and security. End-to-end encryption takes place between two (or more) client applications. A device where such a client application is installed is typically called client device, be it a smartphone, laptop, or desktop computer.

For end-to-end encryption (E2EE), Wire supports the older **[Proteus]** protocol as well as the newer **[Messaging Layer Security (MLS)]**. Wire implements both protocols cryptographically independent of each other.

Client identity verification

Each client device has a cryptographic public identity key, which is used to establish end-to-end encryption to that device. In order to rule out man-in-the-middle attacks, a client device needs to establish trust in the used identity keys of their communication peers. This ensures that each client really communicates with the client it believes it talks to.

Establishing trust in the used identity keys works differently depending on the used protocol:

Proteus

For Proteus, users must manually compare the public identity key fingerprints they see on their device with the fingerprints displayed at their communication peer's devices. This comparison must be done for every device used for communication. Ideally, this should be done over another secure channel or in person to prevent attackers from intercepting or tampering with the verification process.

A chat is displayed with a blue-shaded shield icon as soon as all devices taking part in the chat are manually marked as verified.

MLS

In MLS, this verification can be automated if clients additionally trust the backend servers to authenticate users properly. For this optional on-prem feature, the backend operates a CA and issues X.509 certificates. See section X.509 Credentials for details.

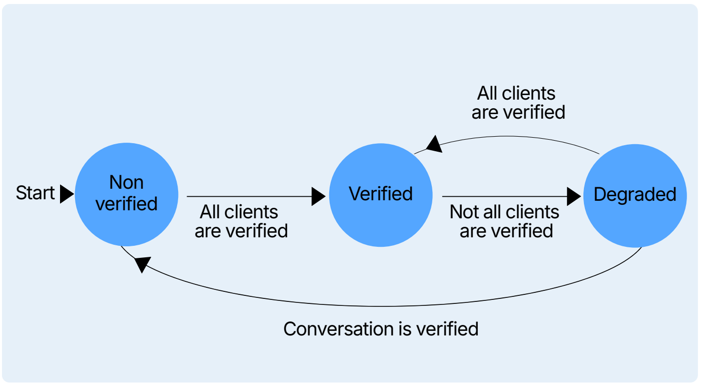
A chat is displayed with a green shield icon (ID Shield) as soon as all devices taking part in the chat are verified to have valid X.509 certificates.

Conversation and device verification states

A conversation in Wire has a verified state that is one of:

- → non-verified
- → verified
- → degraded

Conversations start in the non-verified state. A conversation is verified when all the participants and participants' devices (including the current user) are verified. Once a conversation is verified, it can degrade (go back to not being verified) if a non-verified user or device is added, or if one of the users or devices becomes unverified.



"Non Verified" State

A conversation is "non verified" if it is not "verified" nor "degraded" (see later). Non verified conversations can later become verified.

"Verified" State

A conversation is verified if all the participants (users that are member of the conversation, including the current user) are verified. A user is verified if all known devices of the user are verified. When every device of a conversation is locally marked as verified (this also includes the clients of the sending user) a conversation will be marked as verified by displaying a shield next to the conversation name. In case of a successful automated device verification through MLS, a green shield is displayed. Wire does not provide any kind of synchronization of the verified state of conversations or clients.

Joining the two definitions from before: A conversation is verified if all known devices of all participants are verified.

If the condition is broken, the conversation becomes degraded.

"Degraded" State

When a conversation becomes degraded, the user is informed with system messages. In the degraded state the user can still receive messages, but must either mark the conversation as non verified or verify all new clients to be able to send messages into that conversation.

Common situations that cause a conversation to degrade are:

- → a new user is added to the conversation;
- → a participant in the conversation adds a new client which is not verified;
- → a device is no longer verified because a certificate expired or is revoked for a device of a user that is in that conversation.

End-to-end Encryption

Proteus

Proteus is an independent implementation of the Axolotl/ Double Ratchet protocol, which is in turn derived from the Off-the-Record protocol, using a different ratchet. Furthermore, Wire uses the concept of prekeys 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 Sessions

Proteus provides end-to-end encryption guarantees between two endpoints (clients). For this, the clients establish a "Proteus session" between each other. Each session is uniquely identified by the pair of identity keys of the clients involved, as the client mandates that there can only be one session at a time between two clients.

Sessions can be manually "reset" by the user, meaning that an existing session is discarded and replaced by a new one. This action then applies only to the device for which it was initiated in the device details page in the GUI. The initiating client then bootstraps a new session, as if no previous session existed. The receiving client parses those new incoming messages, detects that a new session has been initiated and discards the previous session.

The following sections detail the establishment of a Proteus session and describe how it can be used to encrypt messages.

Primitives

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

- ChaCha20 stream cipher for message encryption
- HMAC-SHA256 as MAC for message authentication
- X25519 (Elliptic curve Diffie-Hellman based on Curve25519) for key agreement
- HKDF (HMAC-SHA256) for key derivation (RFC 5869).
- Ed25519 for digital signatures (RFC 8032)

Proteus key material

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

- Identity keypair: $\$(a, g^{a})\ \{\epsilon\}_R\ Z_p \times \text{Curve25519}\$\textit{where}\$g \in \text{Curve25519}\$$
- A set of prekey pairs: $\$\ (\ k(_{a,i}), g^{k(a,i)})\ \{\epsilon\}_R\ Z_p \times \text{Curve25519}\$\textit{where}\$0 \leq i \leq 65535\$$.

Prekeys

Prekeys are required to asynchronously initiate Proteus sessions between two clients. A public prekey consists of the static, public identity key of a client together with an ephemeral public DH key. A client regularly uploads a set of such prekeys to the backend.

When a client becomes aware that prekeys have been used to initiate sessions, it will upload new prekeys to replenish the pool on the backend. The client tries to ensure that the backend has 100 unused prekeys available.

These prekeys are eventually used by other clients to asynchronously initiate an end-to-end encrypted conversation, since a prekey allows to establish the initial encryption keys even if the recipient is offline. As soon as the first round-trip has occurred within a session, newly sampled key material is automatically introduced through the DH ratcheting.

Last Resort Prekey

Every prekey is intended to be used only once, which means that the server removes a requested prekey immediately. Thus, it is possible that the backend runs out of unused prekeys for a client. This can happen e.g. when a client has been offline for a while and hasn't been able to upload new prekeys, while other clients have initiated Proteus sessions with it.

For this, one prekey is the so-called "last resort" prekey. It is always available on the backend and never removed by design. The last resort prekey has the lifetime of a client.

X3DH

The X3DH (triple Diffie-Hellman) key agreement protocol is used to establish a shared secret, called master_secret, between two clients.

Overview

Let Alice be the initiating client and Bob be the responding client.

- Alice fetches one public prekey for Bob from the backend
- Alice generates an ephemeral prekey and uses it for the key agreement with the prekey of Bob's client (see section "Details of the key agreement" below)
- Alice sends a prekey message to Bob that contains the public values of her ephemeral prekey and optional encrypted application payload (e.g. an actual text message)
- Bob receives the prekey message, completes the key agreement in turn, and can decrypt the application payload

Details of the key agreement

Bob has the ephemeral DH key pair $\$(\text{esk}_{\text{bob,}}\text{ek}_{\text{bob}})\$$ and $\textit{hisidentitykeypair}\$(\text{isk}_{\text{bob}}, ipk_{\text{bob}})\$.\textit{Bob'sprekeyisthebund} \leq \$(epk_{\text{bob}}, ipk_{\text{bob}})\$$.

Alice generates her ephemeral DH key pair $\$(esk_{\text{alice}}, epk_{\text{alice}})\$$ and $\textit{hasheridentitykeypair}\$(isk_{\text{alice}}, ipk_{\text{alice}})\$$.

> [!NOTE] Identity keys are Ed25519 keys but are converted to X25519 keys during the X3DH phase. This authenticates the key agreement without using a signature.

Alice computes the triple DH agreement as follows:

$$m * er_{sec\ r}et = DH(esk_{alice}, epk_{bob}) || DH(esk_{alice}, ipk_{bob}) || DH(isk_{alice}, epk_{bob})$$

Bob computes:

$$m * er_{sec\ r}et = DH(esk_{bob}, epk_{alice}) || DH(isk_{bob}, epk_{alice}) || DH(esk_{bob}, ipk_{alice})$$

Here, concatenation of octet strings is denoted by ||.

Both clients now have the same master_secret and can use it to initiate a Proteus session. Note that the key agreement also authenticates the peer by only establishing the same master_secret on both sides if the other side has knowledge of its private identity key isk.

Diffie-Hellmann ratcheting

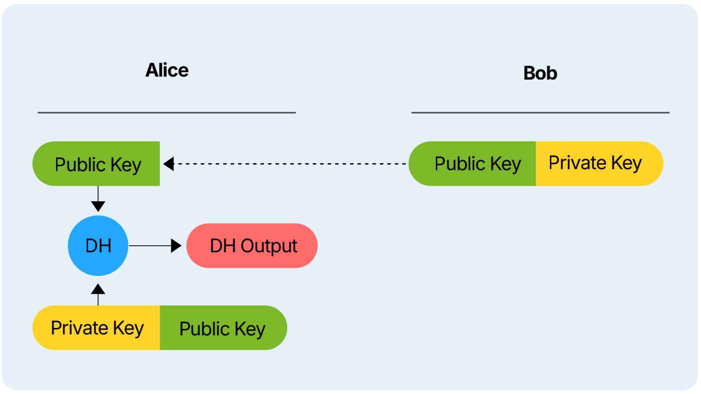
With a shared master secret, both parties could now send encrypted messages to one another. However, Proteus aims to achieve both Post-Compromise Security (PCS), and a more fine-grained Forward Secrecy (FS) as the one already achieved by X3DH.

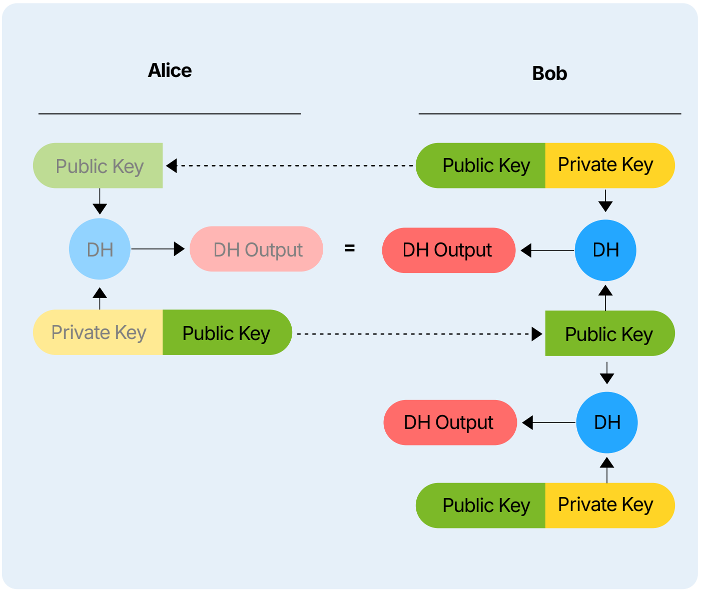
The DH ratchet helps Proteus to achieve Forward Secrecy not only for keys exchanged in past sessions, but keys exchanged in past DH ratchet steps. The KDF ratcheting explained in section KDF Ratcheting increase the FS guarantees even further.

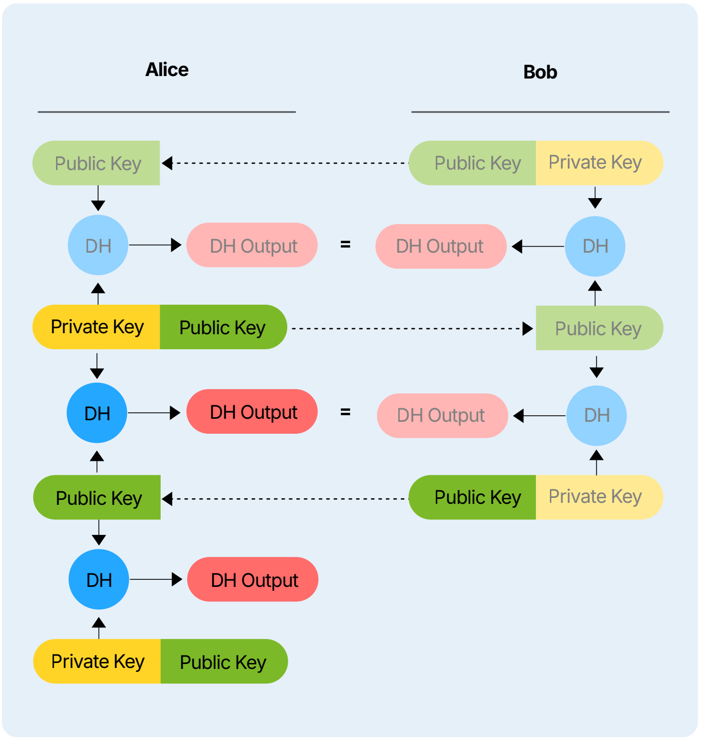
Proteus achieves Post-Compromise Security by using a DH ratchet. The DH ratchet protocol is based on the previously agreed upon master_key, as well as Bob's (i.e. the receiver's) ephemeral key pair.

To initiate the ratchet, Alice generates a X25519 DH key pair (a Diffie-Hellman public and private key) which becomes her current ratchet key pair. Every message from either party begins with a header which contains the sender's current ratchet public key. When Bob receives a new ratchet public key, a DH ratchet step is performed, which replaces Bob's current ratchet key pair with a new key pair:

The following illustrates a sequence of DH ratchet steps, where the DH output is the shared ratchet_secret (Source: Double Ratchet Algorithm)





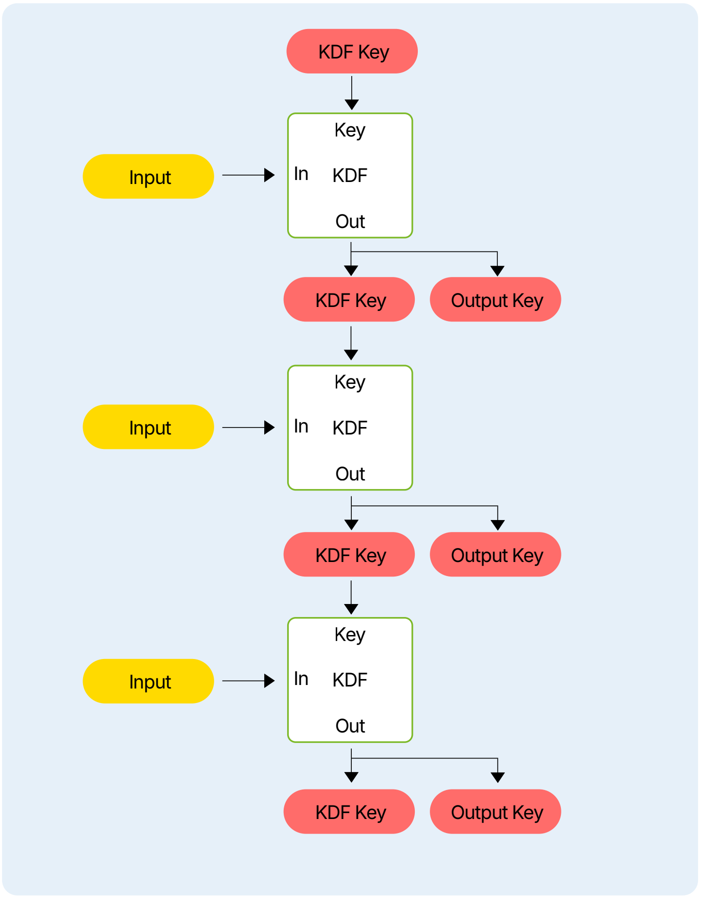


This results in a "ping-pong" behavior as Alice and Bob take turns replacing ratchet key pairs in each stage. An eavesdropper who briefly compromises one of the parties might learn the value of a current ratchet private key, but that private key will eventually be replaced with an uncompromised one. At that point, the Diffie-Hellman calculation between ratchet key pairs will define a DH output unknown to the attacker.

KDF Ratcheting

Since Proteus is an asynchronous protocol, there is no guarantee that round trips will occur, e.g. when one of the clients is offline while the other one sends messages. In this scenario, more fine-grained forward secrecy can be achieved by KDF (Key Derivation Function) ratcheting forward existing key material.

Starting from an initial KDF key, each KDF ratchet step takes an additional bit string as input and yields a new KDF key in addition to an output key.



Double Ratchet - Proteus key schedule

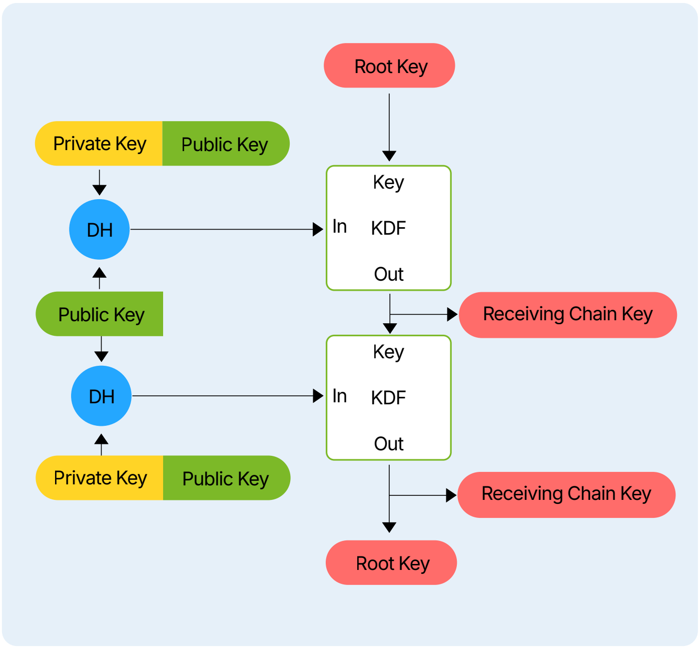
The combination of DH ratchet and KDF ratchet yields a double ratchet.

The double ratchet uses an initial root_key_0, and the ratchet_ secret values from a continuous DH ratchet to create KDF sending and receiving chains (depending on who sent the DH key in a particular step) for each DH ratchet step. The purpose of these two chains, each again a KDF ratchet, is to ensure that each derived key is only used once to protect a message (different keys are used for encryption and authentication).

The first root_key_0 and chain_key_0_0 are derived via

```
root_key_0 || chain_key_0_0 = HKDF(
input: master_secret,
info: "handshake",
length: 64)
```

The following figure from Double Ratchet Algorithm illustrates the interaction between DH and KDF ratchet for a receiving chain.



For each DH ratchet step, a root_key_n root key is used along with a ratchet_secret_n to derive the 32 byte chain_key_n_0, as well as a 32 byte root_key_n+1:

```
root_key_n+1 || chain_key_n_0 = HKDF(
    input: ratchet_secret_n,
    salt: root_key_n,
    info: "dh_ratchet",
    length: 64)
```

The chain_key_n_0 is then used as initial chain key for a sending or receiving chain, depending on which party sent the DH public key in this DH ratchet step.

From the initial chain_key_n_0, as well as any chain_key_n_m, a new message_key_n_m and chain_key_n_m+1 can be derived when the client wants to encrypt a message:

```
message_key_n_m = HMAC-SHA256(message: "0",
key: chainkey_n_m)
chain_key_n_m +1 = HMAC-SHA256(message: "1",
key: chainkey_n_m)
```

The keys resulting from the individual sending and receiving chains are used to derive keys for encryption and authentication of individual messages.

Key buffering

When Proteus protocol messages arrive out of order or are dropped completely, ratcheting the KDF chain forward can lead to problems when decrypting chat messages. Thus, clients ratchet the chain forward a number of messages, while buffering intermediate keys for which no messages were yet received.

In order to limit this buffering, the KDF ratcheting has a threshold of 1000 dropped protocol messages. When clients detect that the number of missing messages is above the threshold (by comparing a message counter), further messages are dropped by the client and an error message is shown to the user. This upper bound mitigates DoS attacks between clients and is never reached in normal operation.

Key Deletion

Throughout the execution of Proteus, keys are deleted after they were used to derive their follow-up key(s). The exception for this are message_key_n_m, which are deleted after they were either used to encrypt or decrypt a message.

Message encryption

Messages are encrypted using symmetric encryption through ChaCha20 with key symmetric_key_n_m.The corresponding nonce is derived from a message counter of the hash ratchet chain.

The integrity of messages is ensured by a HMAC-SHA256 tag created by the key hmac_key_n_m.

```
symmetric_key_n_m || hmac_key_n_m = HKDF(
                    input: message_key_n_m,
                    info: "hash_ratchet",
                    length: 64)
```

Messaging Layer Security (MLS)

The core functionality of MLS (RFC 9420) is an authenticated key exchange (AKE) for groups. The established keys are then used to protect messages sent in that group. Oversimplified, it is possible to describe MLS as TLS for groups, with evolving keys as group memberships change. Therefore, MLS defines mechanisms to synchronize group modifications (add and remove members) within a group. This serialization is done by the message *Delivery Service* in MLS, which is implemented by the centralized backend of Wire. Thus, MLS as implemented by Wire does not use group consensus protocols.

MLS provides Forward Secrecy and Post-Compromise Security at every state of the group. The following section describes how MLS achieves those guarantees.

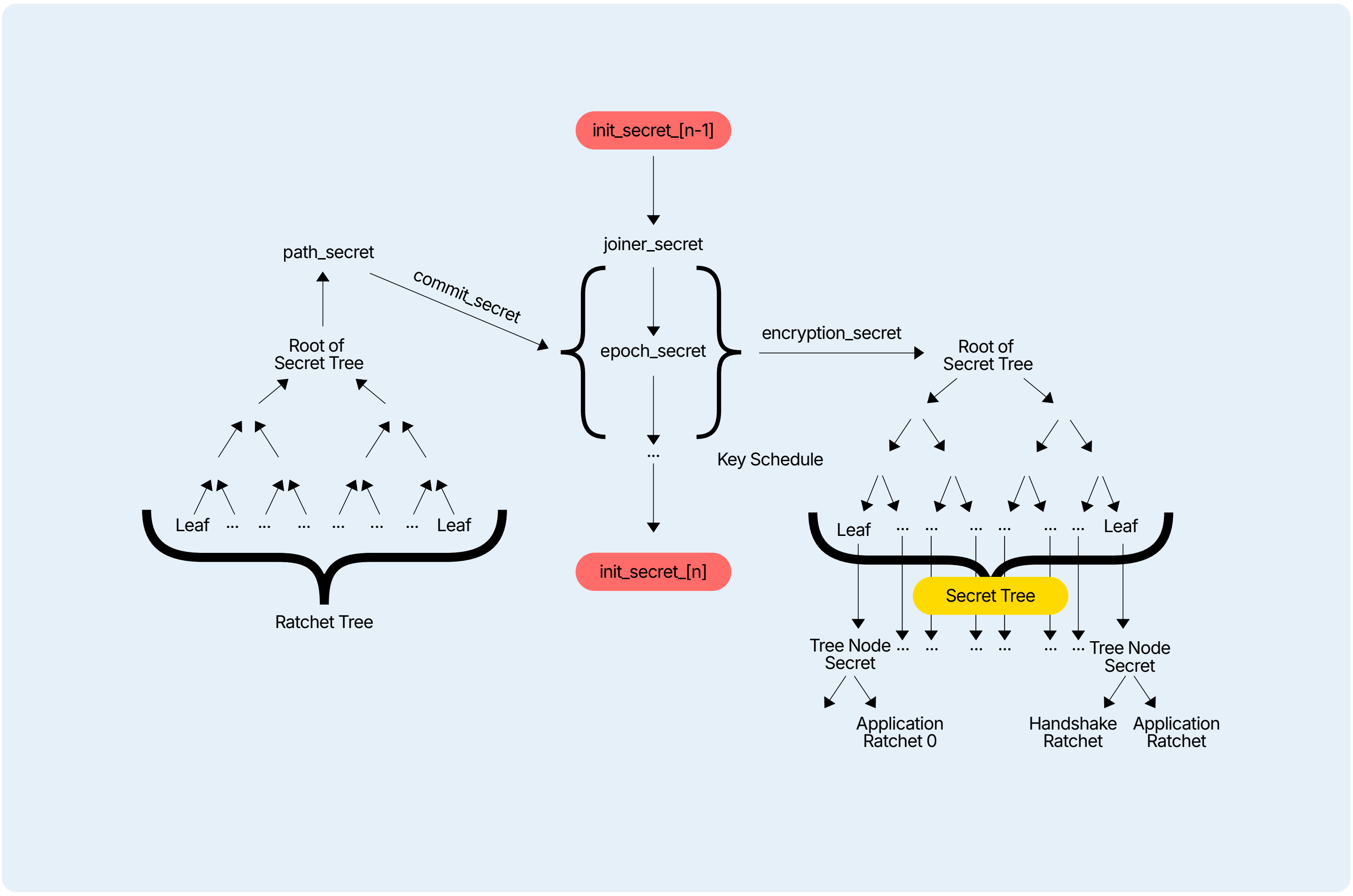
Cryptographic State and Group Evolution

The cryptographic state of an MLS group is defined by three components. Each client maintains a local instance of them:

→ A *Ratchet Tree*, which describes who is currently a member of an MLS group and how to encrypt changes to the Ratchet Tree for all relevant participants. Each unique state of the Ratchet Tree is called an epoch.
→ A *Key Schedule*, which defines a fresh base from which all secrets of an epoch are derived.
→ A *Secret Tree*, which allows every member to derive sender secrets for every member in the group. Those secrets are then used for message protection. The Secret Tree is based on secrets exchanged through the Ratchet Tree and derived by the Key Schedule.

This cryptographic state evolves from one epoch to the next epoch by applying modifications to the Ratchet Tree. This process is also called committing changes to the tree. All group members locally commit changes in a synchronized manner. A modification of the Ratchet Tree provides a new commit_secret, which updates the Key Schedule and, indirectly, the Secret Tree. This ensures that, for example, removing a member from a group cryptographically guarantees that they can not decrypt subsequent messages sent in that group. Thus, post-compromise security is provided between epochs by Ratchet Tree updates. Forward secrecy is provided between epochs by deleting past versions of the Ratchet Tree.

In order to accomodate real world conditions and ensure usability, Wire client's stores the key material up to the last three epochs. The out-of-order tolerance is set to two while the maximum-forward-distance is set to 1000. For further explanation why this is needed see the OpenMLS documention.



The RFC 9420 for MLS describes these cryptographic operations in detail.

MLS Cipher Suites

An MLS group uses a single cipher suite that was selected by the creator of the group. The cipher suite defines the following primitives, which are used for all cryptographic operations in a group:

- Hybrid Public Key Encryption (HPKE, RFC 9180) parameters:
- A Key Encapsulation Mechanism (KEM)
- A Key Derivation Function (KDF), defined by the KEM
- An AEAD encryption algorithm
- A hash algorithm
- A signature algorithm

The names of MLS cipher suites follow the pattern MLS_SecurityLevelInBitsKEMAEADHashAlgSignatureAlg.

The cipher suites currently used by Wire (public cloud offering) is:

MLS_128_DHKEMP256_AES128GCM_SHA256_P256

Additionally, new on-premises installations can be configured to use one of the following ciphersuites:

- MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519
- MLS_128_DHKEMP256_AES128GCM_SHA256_P256
- MLS_256_DHKEMP384_AES256GCM_SHA384_P384
- MLS_256_DHKEMP521_AES256GCM_SHA512_P521

Cryptographic Objects in MLS

Identity keys

When a new client is instantiated, it locally generates a long-term identity key pair for each supported signature scheme (depending on the supported cipher suites). The corresponding private key is only stored locally on the device. The public key is sent to the backend and can be retrieved by other clients. Note that the key type corresponds to the selected cipher suite(s): if only the MLS_128_DHKEMP256_AES128GCM_SHA256_P256 cipher suite is supported, a client only generates an elliptic curve P-256 keypair as identity key.

Credentials

Credentials are used to authenticate the identity of an MLS group member (i.e., a Wire client). The information stored in the credential can be verified by the MLS Authentication Service in use for a group. Credentials are intended to be reused among different groups using the same cipher suite. Thus, a client may use multiple Credential types simultaneously.

Basic Credentials

A basic credential is a bare claim of an identity without included cryptographic proof. MLS does not specify the format of the basic credential. In case of Wire, the basic credential consists of the fully qualified client-ID.

X.509 Credentials

Automatic cryptographic end-to-end identity verification is only possible when using the x509 credential type. This type provides a cryptographic binding between a public identity key and a fully qualified client-ID.

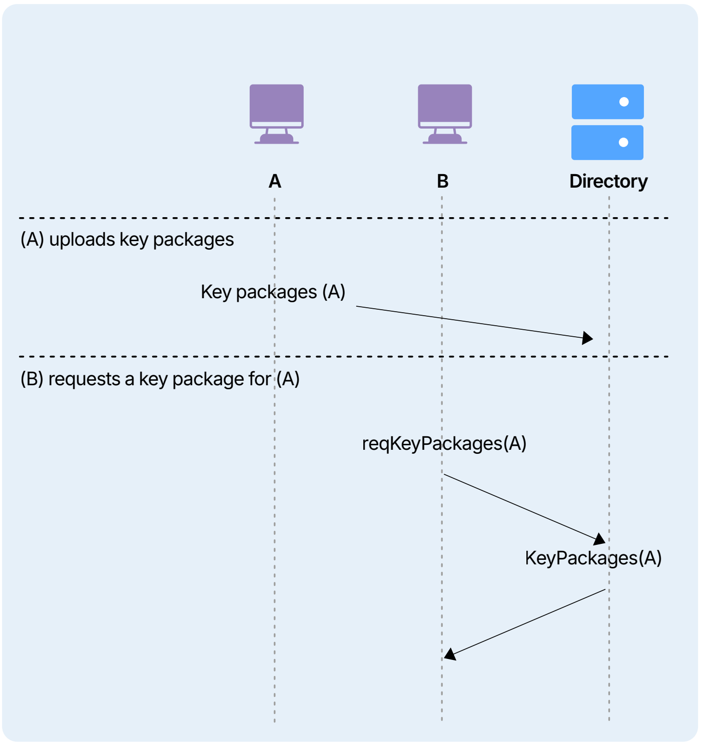
Whenever a new credential is introduced to a group, the credential must be validated. This includes a full certificate chain validation.

The section ID Shield provides more details on the issuance and verification of X.509 Credentials.

Key Packages

MLS uses so-called Key Packages for asynchronous addition of a client to a group. Clients publish key packages on the directory of the Delivery Service, i.e., the Wire backend.

This key distribution mechanism is displayed here:
In order to support multiple MLS cipher suites, there is one set of key packages for one cipher suite.



Each key package is intended to be used only once to join exactly one group. To achieve this, the directory returns every key package only once and removes them from the directory in the same step. At the moment, Wire clients will generate and upload 100 key packages per cipher suite and refill the key packages for a cipher suite if the number of available key packages drops below 50.

> [!NOTE] Even though the MLS protocol specification allows it to reuse a key package in case of last resort, when no more unused key packages are available, Wire does not implement the support for last resort key packages.

A key package consists of:

- → The supported protocol version
- → The cipher suite (section MLS Cipher Suites) for this key package, defining the compatibility and the used cryptographic primitives
- → A public key (KeyPackage.init_key), only to be used for encryption of the Welcome message
- → The Leaf node that will be added to the Ratchet Tree. This contains the Credential used to authenticate the client's identity key
- → Optional additional MLS extensions used by the client
- → The cryptographic signature of the previous data, verifiable with the identity key from the credential

When the Wire client uploads KeyPackages that contain an X.509 Credential, the Wire server verifies that the identity public key covered by the "leaf certificate inside that Credential in the LeafNode of the KeyPackage" matches the client's uploaded identity key for the cipher suite/signature algorithm of the KeyPackage.

Ratchet Tree

MLS uses a so-called *Ratchet Tree* to describe the current state of a group and distribute shared secrets within the group. The use of a tree to manage the membership state of all clients is the core component, which allows to perform efficient changes to the group while only having to perform updates for relevant subtrees of a group.

Tree terminology

Tree nomenclature

A tree consists of *nodes*. Nodes are connected in a hierarchical order where a node can be the *parent* of its *child*. A node is a leaf if it has no children. The node without a parent is the *root* node of a tree. Nodes with children and parents are intermediate nodes. All children with the same parent node are *siblings*.

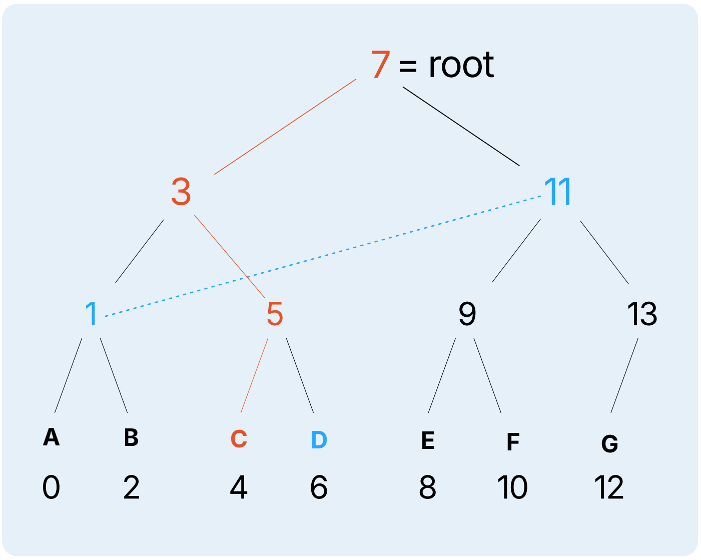
A *subtree* of a node is a subset of a tree where the node becomes the root node and only the children below this node (children, children of their children, ...) are part of the new subtree.

MLS only uses binary trees, which means that all parents have exactly two children, a left child and a right child.

Paths

The *direct path* of a leaf node describes all nodes on its path from that node to the root node.

The *copath* of a node describes all siblings of the nodes from the direct path.



Ratchet Tree structure

The Ratchet Tree is used to efficiently distribute encrypted updates (i.e., Commits) to the future members of the group in the next epoch. Its data structure is a binary tree representing group membership and asymmetric key knowledge within an MLS group. It consists of two different types of nodes:

Leaf nodes Tree nodes with no children. A leaf node is either empty (blank) or represents a client. Such a non-empty Leaf node contains a public HPKE key encryption_key, a public signature key signature_key, and the client's Credential, among other details. The signature_key is used to sign the Leaf node and is authenticated by the Credential.

Parent nodes Every other tree node is a parent node. A Parent node contains a public HPKE key encryption_key, among other details.

The encryption_key is used for encrypting updated key material for that client or set of clients beneath that node.

The Ratchet Tree has two views, a public and a private one.

The public view provides the public keys for each node, known by all members. All set (non-empty) leaf nodes hold a public key. Parent nodes usually hold a public key, too. However, parent nodes, especially when being newly created, have no key data assigned.

The private view represents the knowledge of private keys, which is different for each client. In the private tree all clients know their own private key for their leaf node. Additionally, clients might know the private parts of the encryption_keys of their parent nodes.

Ratchet Tree updates

MLS Proposals and Commits are used to distribute changes to the Ratchet Tree.

Key Schedule

In each epoch, the group keys are derived locally by every client using the Extract and Expand functions from the KDF, defined in the group's cipher suite.The secrets can be derived either by knowing the previous `init_secret` and `commit_secret` from the Ratchet Tree or by knowing the `joiner_secret`, as provided in `Welcome` messages for newly added clients.

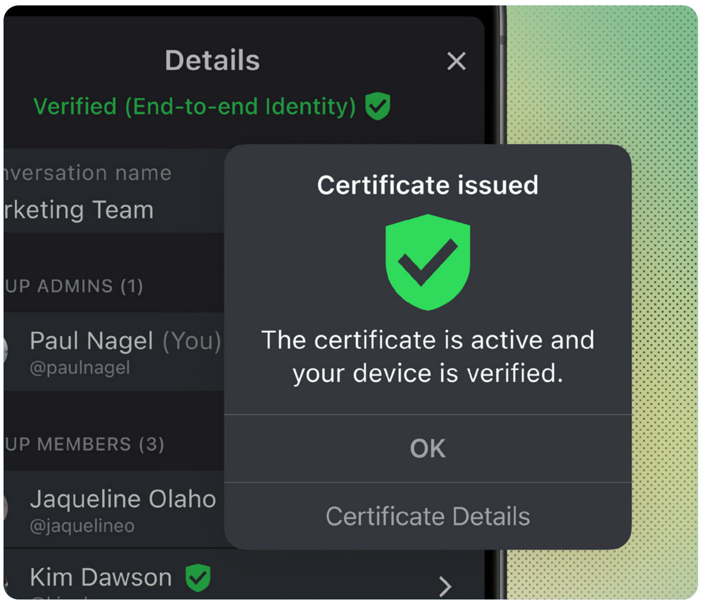
MLS provides a method to include pre-shared keys (psk) into the Key Schedule, by providing a PSK-Proposal, which references a psk that should be used. Wire does not use or implement psks.

Secret Tree

For all encrypted messages exchanged within one epoch MLS uses hash-based ratchets to derive one-time-use encryption keys. All hash ratchets origin in `encryption_secret` from the key schedule.

The Secret Tree uses the structure of the Ratchet Tree to derive all tree node secrets and thereby assign the ratchet secrets to each leaf. Based on this chained key derivation to each leaf, MLS provides a *sender ratchet*.

Each sender has their own sender ratchet, and each step along the ratchet is called a generation. For handshake and application messages, a sequence of keys is derived via this sender ratchet. The ratchet is iterated for every message that needs to be encrypted, so the key and nonce derived from a ratchet are intended for one-time use only. This process is comparable to the KDF Ratcheting in Proteus.



End-to-end identity verification (ID Shield)

For on-prem installations using MLS, Wire offers automatic end-to-end verification of identities ("ID Shield") using X.509 certificates.

This section details issuance and verification of X.509 Credentials, which is specific to MLS.

X.509 Credentials facilitate automatic verification of:

→ the identity of MLS-capable clients,
→ the identity of users of those clients, and
→ the entire MLS conversation.

Limitations

Only for Teams

Currently end-to-end identity verification (E2EI) is designed as a Team-based feature. This means that the feature can be enabled for specific Wire Teams (or all Teams on a backend - with the same configuration values). This also means that individual non-Team users will not have access to E2EI.

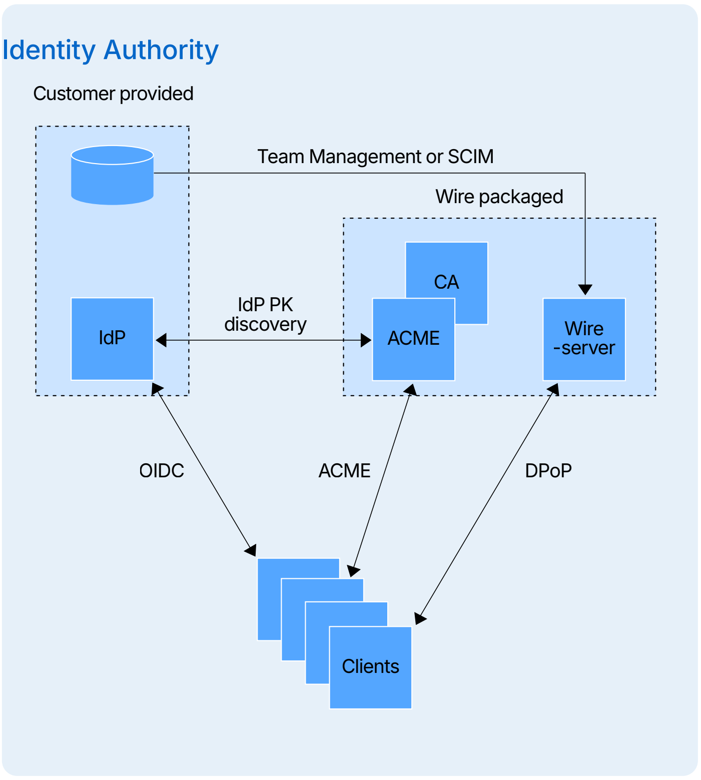
No guest users

Guests (temporary users without Wire account, active for 24 hours) in MLS conversations can not participate in E2EI. This is because it is not feasible to verify the user's identity as part of the E2EI proof flow.

Introduction to components and services

In general, the PKI is structured as follows: The backend's MLS Authentication Service issues X.509 leaf certificates to clients, which uses those certificates to generate MLS X509Credentials. For this, the Authentication Service has an X.509 CA certificate and a corresponding private key. This CA certificate is subsequently called intermediate CA certificate. The root CA certifies the intermediate CA certificate. Its private key is stored offline and is the single trust anchor used by clients of that backend.

To provide automatic certificate management, the following components are required:
→ OpenID Connect (OIDC)-compliant Identity Provider (IdP) Wire can use an existing customer deployment for this.
→ "Identity Authority", consisting of a) X.509 Certificate Authority (CA) b) ACME server. In Wire, the Identity Authority is implemented by Smallstep's step-ca.
→ The existing backend interface



In the case of Wire, the X.509 certificate authority (CA) and the ACME server is the same component (step-ca).

Due to the trust relationship of CA, IdP, and Backend, the end-to-end identity feature is only available for on-prem installations which is managed by the customer itself.

X.509 Certificate Authority (CA)

Administrators need to configure the CA with a certificate suitable for issuing other certificates.

The public key of the CA, which is part of the Identity Authority, can be trusted directly by the clients (e.g., fetched from the ACME server during initial enrollment and stored on application level). This mechamism is based on trust on first use.

In general, the following components of a PKI need to be generated and run as part of E2EI:
1. Root CA
2. Intermediate CA
3. Cross-signed intermediate CA for every federated domain

Wire recommends storing this CA key in dedicated hardware, like a hardware authentication card (for example, Yubikey or other PKCS#11 card) or a Hardware Security Module. From an operational standpoint, Wire flexibly adapts to whichever CA operation policies the customer may have.

The following subsections detail the used keys:

E2EI PKI root CA

The root CA of a backend has a private key in the PKI of the backend's X.509 Credentials.

Type Typically NIST ECDSA P-256 with SHA2-256. Other signature algorithms like RSA or Ed25519 are also possible.

Creation By the backend administrators, during the setup of the backend or root CA key rollover

Storage Offline system or dedicated hardware

Deletion Upon decommissioning of the E2EI feature or key rollover

For the root certificate, Key Usage and Basic Constraints X.509v3 extensions must be configured according to this template:

```
{
  "subject": {
    "organization": "alpha.example.com",
    "commonName": "alpha.example.com Root CA"
  },
  "issuer": {
    "organization": "alpha.example.com",
    "commonName": "alpha.example.com Root CA"
  },
  "keyUsage": [
    "certSign",
    "crlSign"
  ],
  "basicConstraints": {
    "isCA": true,
    "maxPathLen": 1
  }
}
```

A suitable expiration date for the certificate could be multiple years.

E2EI PKI intermediate CA

The intermediate CA of a backend has a private key for issuing the backend's X509Credentials. The intermediate CA's private key is also used to sign the CRL. The intermediate CA is signed by the local backend's root CA. Moreover, federated domains cross-sign the public key of this intermediate CA.

Type Typically NIST ECDSA P-256 with SHA2-256. Other signature algorithms like RSA or Ed25519 are also possible.

Creation By the backend administrators, during the setup of the backend or intermediate CA key rollover.

Storage The private key should be stored in a secure location such as a Hardware Security Module, or on the cluster (encrypted with a secret stored in a k8s secret resource), or in an equivalent secure location. The private key must be accessible to the CA in the Identity Authority (Smallstep).

Every approach has pros and cons and should be evaluated by the customer according to risk assessment.

Deletion Upon decommissioning of the E2EI feature or key rollover

For the Intermediate Certificate, Key Usage and Basic Constraints X.509v3 extensions must be configured according to this template:

```
{
  "subject": "alpha.example.com Intermediate CA",
  "keyUsage": [
    "certSign",
    "crlSign"
  ],
  "basicConstraints": {
    "isCA": true,
    "maxPathLen": 0
  },
  "nameConstraints": {
    "critical": true,
    "permittedDNSDomains": permittedDNSDomains,
    "permittedURIDomains": permittedURIDomains
  }
}
```

where `permittedDNSDomains` must contain the host name at which the Smallstep server is served, for example, `acme.alpha.example.com`. The field `permittedURIDomains` is the federation domain exposed by the Wire instance, for example, if users on that instance have the Wire identifier `@max@alpha.example.com`, this should be set to `alpha.example.com`.

It is possible to rotate this certificate; when doing so, a new dedicated key pair should be used. Wire recommends a conservative expiration of 6 months.

Cross-signed intermediate CA

The cross-signed Intermediates must be created for every federating domain. As a first step, a CSR signed with the private key of the federating parties' intermediate certificate, must be created.

This CSR must be securely transmitted and then signed by the other parties Root, using the same intermediate template as specified earlier. The resulting certificate must be served as `federatedRoot` to ones own clients.

This needs to be repeated, every time a new federating domain is added or certificates in the chain expire.

Certificate revocation list (CRL)

The certificate issuer server maintains a Certificate Revocation List (CRL), which is a signed list of certificates revoked by a particular issuer within a specified validity period. The CRL can be accessed via HTTPS through a URL included in the certificates.

Further details on the CRL can be found in Section Certificate Revocation List.

ACME server

The ACME server uses the ACME protocol to automate certificate management. Clients communicate with the IdP and the Wire backend to obtain proofs of control over their identifiers and present those proofs to the ACME server. If those proofs are valid, the ACME server issues a certificate using the CA.

The ACME server is configured with several categories of information:

→ where to contact the IdP, the clients use to prove control over their Wire handle and display name,
→ where to contact the Wire server, the clients use to prove control over their client-ID,
→ how to trust the Wire server by providing the public signature keys the Wire Server uses to sign the DPoP proof,
→ where to fetch the IdP's OIDC discovery document that contains the endpoint for fetching the IdP's public key,
→ certificate parameters such as allowed certificate lifetimes and signature algorithms,
→ how to determine the Wire handle and display name format from the information provided by the IdP.

To handle the last category, the configuration includes transformations needed between the user information provided by the IdP and the exact format of the display name and handle identifier included in client certificates. For instance, assuming that the unqualified Wire handle and the local part of an email address are consistent for a specific team, the ACME server could then validate the Wire handle based on an email address returned in the IdP's OIDC response. Likewise, the ACME server could check for the presence of custom OIDC claims or membership in a certain group.

Together with the CA, the ACME server forms the Identity Authority. The Identity Authority must be reachable by all Wire clients over a single TCP port number. The Identity Authority has no connection to any Wire server component and should not be in the same firewall zone as the Wire server.

Client-ID proof interface of the backend

Wire clients use this interface to obtain a proof of control over their client-ID.

Customer-provided component: OIDC IdP

An OpenID Connect (OIDC)-compliant Identity Provider (IdP) is a required external service needed for the E2EI feature. The IdP has to support Individual Claims Requests. Ideally, the IdP supports SCIM for automated user provisioning and uses that to manage user accounts at the Wire server.

The OIDC IdP must be reachable by the Wire clients. It should be in a different firewall zone than the Wire backend or the Identity Authority. The OIDC IdP must be reachable for the ACME server to discover its public key pair for signature verification.

Possible instances of an IdP include an on-prem solution such as Keycloak, Gluu, or OpenIAM, a cloud service such as Microsoft Azure Identity, Google Identity, Okta, or Auth0, or an OAuth proxy such as dex (which could authorize based on another source such as an LDAP directory).

E2EI high-level process summary

Enrollment

The Wire client enrolls using the ACME protocol, and proves that it has control over its Wire handle, display name, and its Wire client-ID. All Wire identifiers used in this document are "qualified" identifiers.

→ The Wire handle and display name are verified with an OIDC-compliant Identity Provider for the domain. The naming convention is enforced because the IdP uses SCIM to provision users to the Wire server. This ensures that the display name and the Wire handle is the same on the IdP and the Wire server.
→ The Wire client-ID and the display name is verified with the responsible Wire server.

Once these identifiers are verified, the ACME server issues a certificate associating these three elements with the client's public identity key.

Step by step:

1. At minimum every 24 hours and on login, the Wire client checks if end-to-end identity is enabled in the feature configuration for its Wire Team.
2. If enabled, the Wire client connects to the ACME server starting with the discovery URL for the Team; then it follows the ACME flow to request a new certificate.

3. During the flow, the Wire client needs to prove to the ACME server that it has control over their user's Wire handle and display name by presenting an OIDC identity token from the IdP.

   Typically, the IdP will be configured in such a way that the user will need to re-enter their login credentials.

The ACME server checks that the OIDC identity token is indeed signed by the IdP.

The client must also obtain a DPoP proof from the Wire server which contains its client-ID and user display name.

The ACME server verifies that this DPoP proof is signed by the Wire server's DPoP signing key.

The full list of checks done by the ACME server is omitted here for brevity.

4. If all checks were successful, then the ACME server provides a certificate for the client. Once the client obtains its certificate, it updates its KeyPackages and the LeafNodes in all its MLS groups to include the certificate.

Presentation and Verification

The Wire client presents the certificate in its MLS groups by including the certificate in the Credential of its MLS LeafNodes. The Wire client also includes its certificate in its MLS KeyPackages, which are used by other clients to add this client into new MLS groups

Clients verify the certificates of the other clients:

→ when receiving an MLS Welcome message, validate the certificates of all the peers in the newly joined group;
→ when receiving an MLS Commit with a new peer or new certificate for an existing peer;
→ when adding a new peer to an existing conversation (validate the certificates in the KeyPackages of the new clients to be added).

The Wire client will display a verification badge if all the clients in the MLS group/conversa- tion are verified. If a formerly verified conversation has a non-verified client (for example, if a new client is added or a certificate expires), the conversation will become degraded. See section Conversation and device verification states for more details.

Renewal

The Wire client periodically renews its own certificates well before they expire.

Each Client requests a new certificate, such that even if the client goes offline for the "server message hold interval" (default value - 28 days) - the client will renew early enough to insure it always has a valid certificate.

Whenever the client receives a new certificate, the client replaces its old KeyPackages with fresh ones, and updates its LeafNodes in all its MLS groups.

Revocation

The certificate issuer server maintains a Certificate Revocation List (CRL) - a signed list of revoked certificates from a specific issuer, with a specific validity time. The CRL is available via HTTPS from an URL embedded in the certificates. The Wire client checks on login for the CRLs of its issuer, and that of any federated issuers of E2E identity credentials in its conversations, and shortly before those CRLs expire (the default validity is 24 hours). If there is a conversation with another client that has a revoked certificate, the user will be warned and a conversation no longer will be verified.

The Certificate Revocation List is stored for future validations. If the revocation list contains any new entries, all conversations containing the relevant user are verified again to identify potentially revoked clients.

Revocation is a rare, strong action indicating that a device or piece of software is no longer trustworthy. It is akin to canceling a credit card, in that there is no certificate issuance possible after a revocation. Consequently, the user or admin of a client with a revoked certificate should also delete the corresponding client.

Asset Encryption

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

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

Assets shared in conversations are end-to-end encrypted. Compared to regular text messages, the encryption of assets applies an optimization proposed here 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 uploads the encrypted asset data to the server
4. It encrypts the key k together with the hash and other asset metadata via Proteus/MLS and sends it to the recipients.

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

1. It decrypts the asset metadata using Proteus/MLS, 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/MLS protocol.

Link Previews

When users send 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 (that is, a picture and some text) from the website behind the link. This data is added to the chat message, sent to the recipient, and displayed there. The recipient does not make any network requests to the website, unless the recipient clicks or taps the link.

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 notification providers: Wire currently supports FCM and APNs as external push notification providers. This channel is used if a client is offline but has registered a valid FCM or APNs push token with the server.

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.

A note on ephemeral messages (“Self Deleting” Messages)

Timed messages carry a time-to-live indication, specified by the sender. When the receiving client displays the message for the first time, it calculates an expiry date based on the current time and the provided time-to-live. Based on this value the client is able to verify when the time-to-live has passed. When this is the case, the receiving client will remove the message from the local database.

Ephemeral messaging is not designed as a security feature, rather than a convenience to have conversations cleaning themselves up on their own. There is no guarantee that an ephemeral message will be treated as such on the receiving side. It may prevent unauthorized users with device access from reading previous messages within a conversation when using only the application. However, this should not be considered a security feature, as traces of those messages may still exist in the client's local database.

Client-Server Protocol Description (Chat Server Protocol)

The API descriptions of previous and the current version can be found here.

WebSocket connections

In addition to requests to HTTP resources, every client that successfully authenticated with the backend can establish a WebSocket connection over HTTPS.

The WebSocket connection is used when the app is in an active state and the client has an ongoing connection with the backend. A client with an established WebSocket connection is considered online. It is used to fetch new messages from the notification queue.

Additionally, clients can maintain the WebSocket connection even when the app is in an inactive state to receive real-time push-notifications from the backend, when push notification services from an app environment are not available. The WebSocket connection over HTTPS uses the same TLS connection as described in section Transport Encryption (TLS).

Calling

Calling in Wire comes in two flavors: one-to-one calls (Section 1:1 Calls) and conference calls (Section Conference Calls). One-to-one calls are calls between two clients, whereas conference calls can host more than two clients. Both flavors have the same technological foundation and heavily rely on WebRTC for media encoding/decoding, encryption and media routing. Conference calls use an additional server-side component (Selective Forwarding TURN Server) as well as an additional encryption method.

Call signaling

All calls are initiated through the end-to-end encrypted session. Call signalling parameters to establish a connection between Wire endpoints and negotiating their common capabilities is done by exchanging SDP messages. In the case of conference calls, SDP messages are sent as HTTPS messages between a client and a Selective Forwarding TURN (SFT) server (see section Selective Forwarding TURN Server (SFT)).

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 a topology (e.g. with 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. TURN servers are part of the Wire backend infrastructure but are standalone components that are not connected to the rest of the backend components and therefore do not share data with them. They do not know the user-ID of the users that use them and act purely as relay servers for media streams.

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. TURN servers and the backend only share a long-term secret key that is used to symmetrically sign the generic credentials used by the clients to authenticate to the TURN server.

The credentials are emitted by the backend. They expire after 24 hours and need to be refreshed by the clients. The TURN server can verify the signature with the long-term secret key. The purpose of these credentials is to prevent DoS attacks against the TURN server. In the case of a conference call, the client starting the conference transmits the TURN servers and credentials to the SFT server as SFT servers do not have their own connection to the backend.

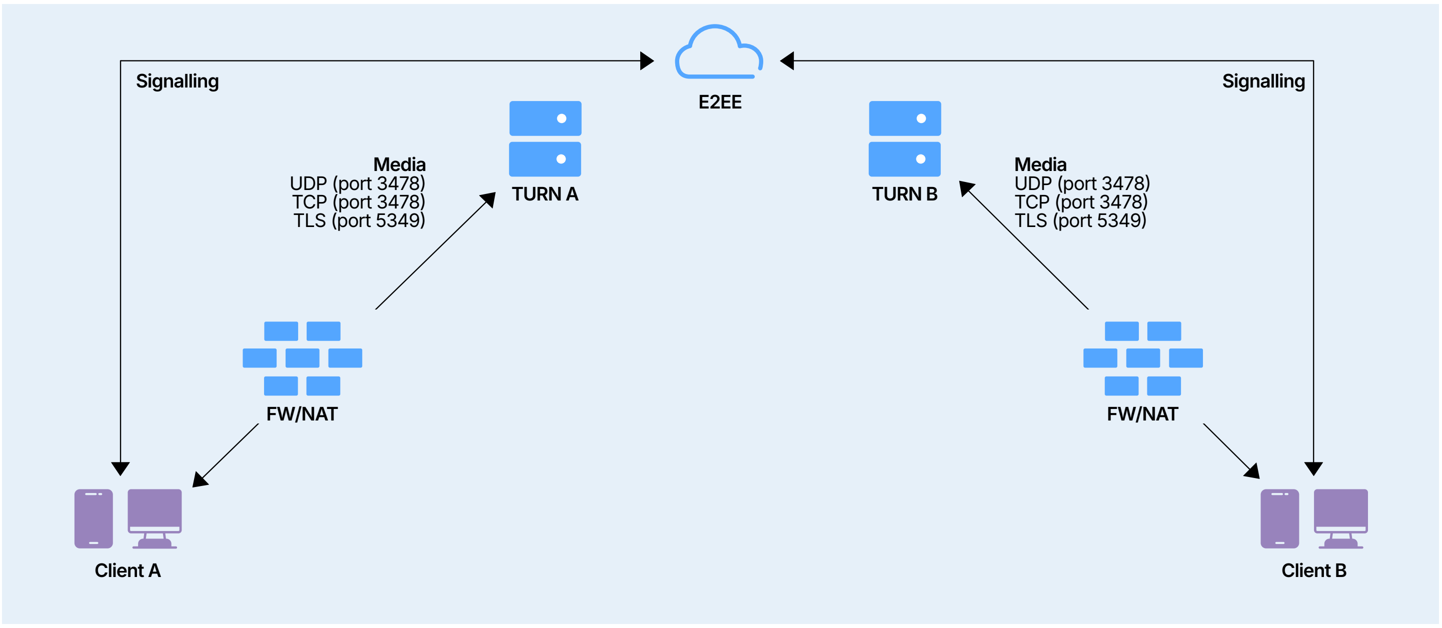
Call setup

1:1 Calls

The following is an example for setting up a one-to-one call with client A calling client B. Client A connects to TURN server A and client B to TURN server B. In practice these two TURN servers could be the same server. The separation was chosen to reflect the fact that the external side of the TURN servers connects via UDP. Clients may also directly connect via UDP to either other clients that are directly reachable or to a TURN server that a client is connected to.

Before a call can be set up, clients need to receive a call configuration from their associated backend. This

Overview



configuration is received when clients come online after they were offline for a longer time, and it is frequently refreshed while being online. The refresh interval (TTL) can be set on the backend and is transmitted to clients in the configuration. The configuration contains all available TURN servers, credentials to connect to the TURN server, and all available transport protocols. TURN servers can be configured to allow any combination out of UDP, TCP, and TLS.

They are listening on the following ports:

→ 3478 for UDP
→ 3478 for TCP
→ 5349 for TLS

For conference calls the call configuration also contains URLs to SFT servers. To allow for load balancing over multiple SFT servers on the backend side, the call configuration is refreshed immediately before starting a conference call. This way the backend can always distribute SFT servers with available capacity for a conference. A typical call configuration for one TURN server and all transports, and one SFT server received by clients may look like this:

```
{
  “ttl”: 3600,
  “ice_servers”: [
    {
    “urls”: [“turn:turn01.de.somedomain.
    com:3478?transport=udp”],
    “credential”:”qvt5kHU7vQ5HK6JxihBIFY60fVm8FT-
    FiRIv2LKdOJi6LX8yauMoXGSzRY/6MEokaCFerN-
    WkbNyYh02ngOXFtgA==”,
    “username”:”d=1618436350.v=1.k=0.t=s.r=olgeadtu-
    aoxmtkhz”
    },
    {
    “urls”: [“turns:turn01.de.somedomain.
    com:5349?transport=tcp”],
    “credential”: “QanQMQZvRZwQmojx3D/78IsZZLGw-
    bGabqTOREUigf2vihwuSppWMz9PIytkvbBTy-
    jDYR21/79coGJ8ZJ/3I9Og==”,
    “username”: “d=1618436350.v=1.k=0.t=s.r=ogm-
    drqxmirpaiyss”
    },
    {
    “urls”: [“turn:turn01.de.somedomain.
    com:3478?transport=tcp”],
    “credential”: “e2snEvOH1mWaUgWaYvXG5i53Xy-
    mAhJQWxENNLK5GDBoeTnAo8rb9Ne+pfSgG-
    16WeyQqHSBVAXbaeZ3kzVWN0NQ==”,
    “username”: “d=1618436350.v=1.k=0.t=s.
    r=pekwyrmcocpgicqq”
    }],
  “sft_servers”: [
    {
    “urls”: [“https://sft01.sft.somedomain.com:443”]
    }]
}
```

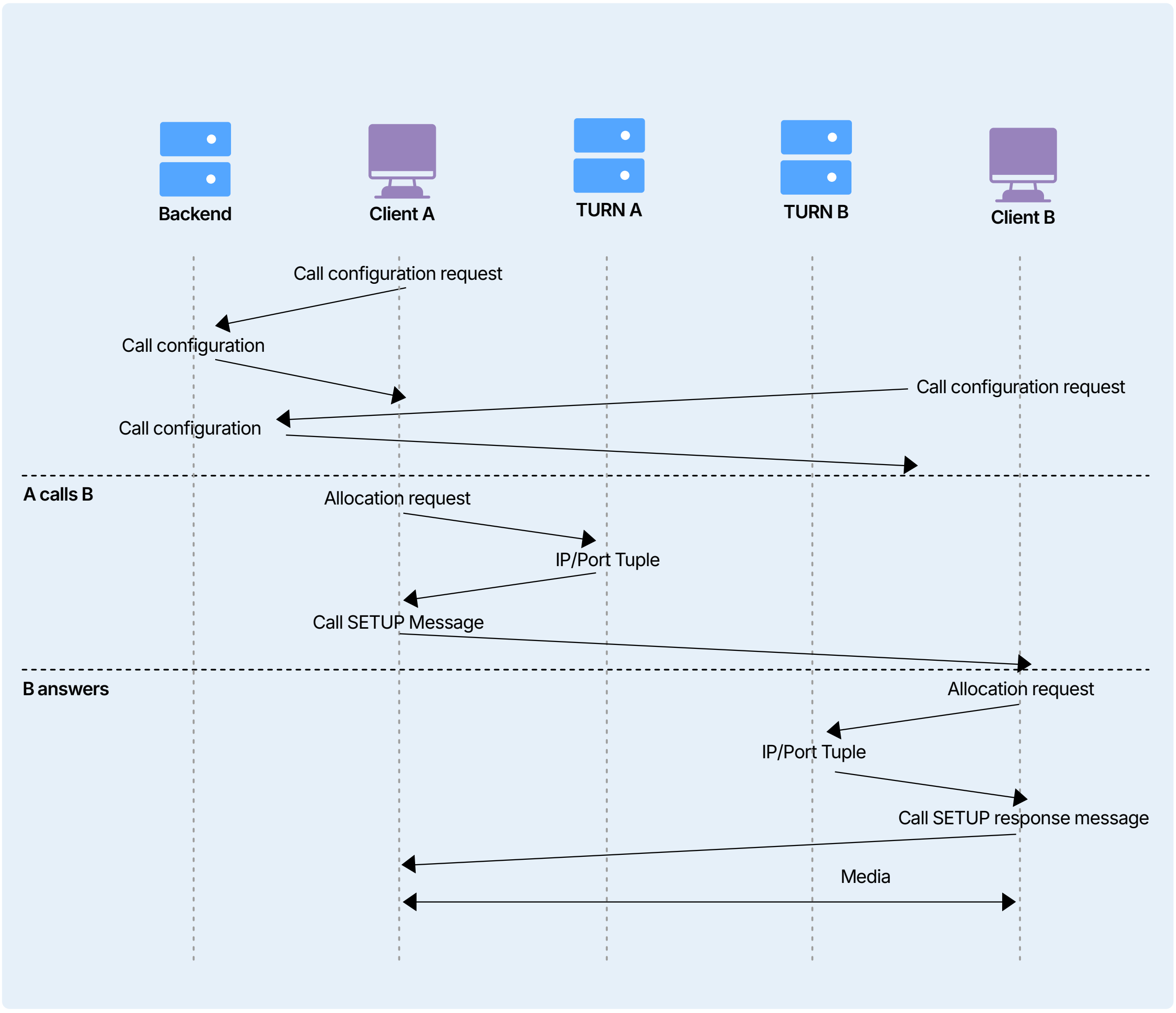
In the example client A would receive a call configuration from the backend that includes TURN server A in combination with UDP, TCP, and TLS transport. On the other side, client B would receive a similar call configuration from the backend as well that includes TURN server B.

Note that neither client A or B has or requires any knowledge about the call configuration on the other side (B or A) at the time a call is initiated.

Also note that even though the example above only shows one TURN server, for redundancy reasons, there might be multiple TURN, and multiple SFT servers provided in the configuration.

Signaling flow

When client A sets up a call to client B it contacts all TURN servers that were listed in the call configuration, in the above example TURN server A, with an allocation request. TURN server A then allocates and returns a UDP port on the "external" network for client A.



Client A now is reachable from the outside via the tuple of external IP address of TURN server A and the allocated UDP port. All data that is sent to this tuple will be forwarded to client A.

The next step in the call setup process is to send this allocated tuple to client B in a call setup message via an E2EE message. When client B receives the setup message it will run through the same procedure as client A. Client B contacts TURN server B with an allocation request. TURN server B then allocates and returns a UDP port on the "external" network for client B. Client B at this point is reachable from the outside via the tuple of external IP address of TURN server B and the allocated UDP port. All data that is sent to this tuple will be forwarded to client B. Client B sends this tuple to client A in an answer to the call setup message from client A via an E2EE message.

Now both clients, client A and client B, run through a connectivity check where they try to reach the other client on all possible routes. Ways to reach the other client includes the TURN allocation, but also local address or server reflexive address may be included. In the above example it is assumed that both clients reside in networks that are not directly reachable from the other side (or want to mask their IP addresses). Therefore, a connection from client A will be established through TURN server A connecting to TURN server B, forwarded to client B. Client B will connect through TURN server B to TURN server A, forwarded to client A.

A path between client A and client B has been established, and both clients can start streaming media.

Calling in federated environments

A call between two federated participants is not different from a call between two participants on the same domain. Both participants exchange connection capabilities as E2EE messages and setup their connection based on the available connection endpoints.

Federated backends may additionally provide TURN servers to provide external connectivity.

Conference Calls

This section specifies the end-to-end encryption used for Wire's conference calling. All messages between clients are sent with the selected E2EE protocol (section End-to-end Encryption) and inherit the security properties accordingly, i.e. authenticity and end-to-end encryption. This version implements a base-line security that is comparable with other end-to-end encrypted conferencing solutions today. The goal however is to move to a sframe-based solution on top of MLS.

Selective Forwarding TURN Server (SFT)

The SFT is the main component in the conference calling architecture. Its job is to gather encrypted streams from each client and fan them out to the others over a single connection. In order to establish a call, clients initially connect to the SFT via HTTPS and exchange connection information via SDPs in SETUP messages. Once established, the SFT and clients exchange media and data-channel messages over UDP. For clients that can not connect directly via UDP refer to previous chapters on how clients may use TURN servers to connect to the SFT server.

The HTTPS connection between clients and the SFT uses the same TLS mechanism and parameters described in section [Transport Encryption (TLS)]. In that respect, the SFT acts as just another RESTful backend API.

Calling Messages

Wire uses JSON for encoding calling messages. Messages are sent via HTTPS post/response, via E2EE session or via the data channel between clients and the SFT. Messages only relevant for current call participants are sent via targeted E2EE messages to clients in the ongoing call (with MLS Wire uses an MLS subconversation with ID "conference" to send the message to all actively participating clients). The handling of the encryption/decryption keys is negotiated on the MLS layer.

Encoding

The codec used for streaming is Opus for audio and VP8 for video. Opus can use variable bit rate encoding (VBR) or constant bit rate encoding (CBR).

Conference calls always use CBR encoding.

CBR has the advantage of eliminating potentially undesired information about packet length but might have an impact on call quality on slow networks. 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 video calls the CBR option affects the audio streams like in audio calls, but the calling screen will not display "CONSTANT BIT RATE".

Encryption

Call media is always exchanged between endpoints in an SRTP-encrypted media session. To initiate the session the SRTP encryption algorithm, keys, and parameters are negotiated through a DTLSv1.2 handshake. The authenticity of the clients is also verified during the handshake in the following steps:

- → Each client generates an ephemeral key pair ahead of time.
- → The fingerprint of the public key is sent to the other client as part of the SDP message during the initial signalling. As mentioned above, signalling messages are exchanges over the E2EE session and therefore inherit from its authentication properties.
- → During the DTLS handshake the public keys are exchanged between clients (through ServerHello/ClientHello).
- → The clients compare the public keys to the fingerprints from the SDP.
- → If they detect a mismatch, the DTLS handshake is aborted and no connection is established. The handshake procedure utilizes the authenticated key exchange of DTLS (and the authenticated E2EE session) to guarantee confidentiality and authenticity of call data.

For the DTLS phase, both devices negotiate cipher suites similar to a TLS handshake from the following list:

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_
SHA256
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_
SHA256
```

In addition, Wire clients use Encoded Transforms (formerly known as Insertable Streams) to end-to-end encrypt the content of media packets.

Symmetric cipher
AES256-GCMTRC (used for payload encryption)

Key derivation
HKDF with HMAC-SHA512

Forward secrecy is addressed by the fact that keys are rotated every time participants join or leave a call, using the E2EE session between the call participants as key transport.

Federation

In a federated environment, backends interact with each other. These connections are secured by mutual TLS (mTLS). The current MLS based federation implementation adds two components to the normal backend setup:

- → A Federation Server acts as the single point of incoming traffic for federated services. This is implemented as a part of the Kubernetes nginx ingress service with server certificates and mandatory client certificate authentication.
- → A Federator server processes all federation requests (incoming and outgoing).

The federation server component only accepts connections with client certificates, while the Federator component is in possession of a client certificate to present to other instances. Conversely, the Federation Server component needs a server certificate to authenticate itself to other parties. Both components may use the same certificate, if the certificate holds both purpose flags (for client authentication and server authentication).

The federation server is a part of the Kubernetes nginx ingress service with the same cipher configuration as for regular TLS connections (see section Transport Encryption (TLS)).

The list of acceptable certificate authorities for both incoming client certificate verification and for verification of server certificates of outgoing connections can be configured at backend deployment time in the Kubernetes/helm chart configuration.

Both default to the system CA store and a verification depth of 1, but this can be changed by the backend administrator.

Through Federation, Wire enables users from one domain to communicate with users of another domain it federates with. Federation uses two different domain names: The backend domain and the infrastructure (or infra) domain. All references to "domain" without any qualifier refer to the backend domain. The backend domain is the user visible part of the federation infrastructure. It is used to qualify usernames when presented to the user. A qualified username in the alpha.org backend domain would be @usera@alpha.org.

The infrastructure domain is used for actual communication between the backends. The client/server certificates for Federator and federation server need to be issued in the (subject alternative) name of the infrastructure domain. A DNS SRV lookup is used to map from a backend domain to its infra domain, for both outgoing connections (where the infra domain also needs to be used for address lookup) and for incoming connections (where the infra domain in the presented client certificate needs to be associated with the backend domain that the incoming connection claims to represent).

The mutual authentication between the federation components assures that all messages passing between instances are assigned their correct source and target domain names. Within an instance, for all messages relating to external domains, the username is qualified with the external domain name. The backend administrator can configure an allowlist of allowed backend domains. Domains can only federate with each other after actively setting up the Federation. Administrators are able to configure the search policy for a remote backend e.g., so that no users are returned by federated searches. Additional information on Federation can be found in Wire's Public Documentation.

Legal Hold

Legal hold introduces a new type of device, named legal hold device, for the sake of satisfying corporate compliance rules without sacrificing end to end encryption with full transparency.

Legal hold device

A legal hold device is a device bound to a team user account but managed outside of the user's control - it is managed by the Legal hold service. That is, only team admins can remove that device from a user's account again. The management/ operation of said Legal hold service is of the responsibility of the team.

This type of device is visually distinguishable, for full transparency, with a clear, constant indication on the client UI. There can be only one such device per user account and they can only be added to the user's device list with the user's consent and confirmed with a password prompt (if they have one - there can be exceptions such as SSO users).

Legal hold service

This Legal hold service can be provided by the backend, or hosted separately at customers' premises.

Interaction

A team admin may ask members of the team to be put under legal hold. Once a person is prompted, they should then verify that the presented signature matches the one generated by the legal hold service to avoid any potential MITM attack.

Once a user accepts the legal hold request, then a device is added to that user's account. This device, also known as legal hold device, then receives a copy of each message this user sends or receives.

Note that every user talking to someone under legal hold (including, of course, the self user) is made aware by means of displaying a red dot on the user's profile.

Further security & cryptography

Transport Encryption (TLS)

For TLS, the same server settings are used for the RESTful API, WebSocket connections, Browser sessions or users (e.g. Team admin web page), and federation communication to remote backends:

The backend only supports TLSv1.2 and TLSv1.3 as well as the following ciphersuites:

TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
TLS_AES_128_GCM_SHA256,
TLS_AES_256_GCM_SHA384

Cipher suites can be configured for each backend individually. The default cipher suites only use cipher suites that support Forward Secrecy (FS), to make sure that session keys will not be compromised, even if the long-term keys are compromised. The server indicates the order preference of cipher suites and communicates HTTP Strict Transport Security (HSTS) to all HTTPS-clients.

To mitigate man-in-the-middle attacks caused by rogue or compromised certificate authorities 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 binary client applications and are used for all outgoing connections to the Wire backend, including checks for updates of the Webapp in the Wire Desktop application.

On mobile devices, clients use the operating system's API for TLS connections.

The desktop client uses Electron, which is based on Chromium that uses BoringSSL as a TLS library.

App Lock

The Wire Client App allows to enable an application lock mechanism. This enforces user authentication using the methods provided by the underlying platform (biometrics, passcode, etc.) to gain access to the app in the same way the device itself is unlocked. When this feature is initially enabled and no platform-based locking functionality is available (e.g. Touch ID on iOS), the client app asks the user to set an unlock password.

If "Lock with Passcode" is turned on, the application lock is automatically engaged when the Wire Client App is inactive for at least 1 minute (configurable by the Team administrator). The user then has to enter the unlock password in order to further interact with the app again.

Local storage of backups

Wire client applications have a function to export the conversation history (the bulk of all messages, organized in different conversations). The choice of generating encrypted or unencrypted backups is left to the user. The process works as follows:

- The user is asked to provide a dedicated backup password
- The Argon2 memory-hard key derivation function (argon2i with 6 iterations and 134 MB memory per iteration, with a random salt of 16 bytes) turns the user-entered password into an encryption key
- The derived key is used to encrypt the backup with XChacha20Poly1305

On the application level, backups files contain the user ID and can only be imported by the same user who has created the backup.

Second factor authentication

Wire backends can be configured to mitigate the impact of compromised Wire logins by requiring a second factor authentication (2FA) token for important operations:

- User login
- Registration of a new client
- Creation of a new team SCIM token
- Deletion of a team

The requirement for 2FA secured authentication can be configured to be enforced on the backend level or per team. When a client triggers the requests, the backend generates a random token (a 6-digit number) for the specific action and sends it to the email linked to the user performing the action. 2FA tokens are valid for 15 minutes. After providing a wrong code the third time, the token will be invalidated and can no longer be used to access the account. Additionally, a token is only valid for a specific action (login/ client registration or SCIM token creation) for which it was requested.