WhatsApp

WhatsApp Encryption Overview

Technical white paper

Version 4 Preview (Applicable to Multi-Device Beta Only) Updated July 14, 2021
Version 3 Updated October 22, 2020
Version 2 Updated December 19, 2017
Version 1 Originally published April 5, 2016

Contents

Messaging Security

Introduction

This white paper provides a technical explanation of WhatsApp's end-to-end encryption system. Please visit WhatsApp's website at www.whatsapp.com/security for more information.

WhatsApp Messenger allows people to exchange messages (including chats, group chats, images, videos, voice messages and files), share status posts, and make WhatsApp calls around the world. WhatsApp messages, voice, and video calls between a sender and receiver that use WhatsApp client software use the Signal protocol outlined below. See "Defining End-to-End Encryption" for information about which communications are end-to-end encrypted.

The Signal Protocol, designed by Open Whisper Systems, is the basis for WhatsApp's end-to-end encryption. This end-to-end encryption protocol is designed to prevent third parties and WhatsApp from having plaintext access to messages or calls. Due to the ephemeral nature of the cryptographic keys, even in a situation where the current encryption keys from a user's device are physically compromised, they cannot be used to decrypt previously transmitted messages.

Beginning on July 14, 2021, a user can have multiple devices, each with its own set of encryption keys. If the encryption keys of one device are compromised, an attacker cannot use them to decrypt the messages sent to other devices, even devices registered to the same user. WhatsApp also uses end-to-end encryption to encrypt the message history transferred between devices when a user registers a new device.

This document gives an overview of the Signal Protocol and its use in WhatsApp.

Terms

Device Types

- Primary device - A device that is used to register a WhatsApp account with a phone number. Each WhatsApp account is associated with a single primary device. This primary device can be used to link additional companion devices to the account. Supported primary device platforms include Android and iOS.
- Companion device - A device that is added to an existing WhatsApp account by the account's primary device. Companion device platforms include Web, Windows, macOS, and Facebook Portal.

Public Key Types

- `Identity Key Pair` – A long-term Curve25519 key pair, generated at install time.
- `Signed Pre Key` – A medium-term Curve25519 key pair, generated at install time, signed by the `Identity Key`, and rotated on a periodic timed basis.
- `One-Time Pre Keys` – A queue of Curve25519 key pairs for one time use, generated at install time, and replenished as needed.

Session Key Types

- `Root Key` – A 32-byte value that is used to create `Chain Keys`.
- `Chain Key` – A 32-byte value that is used to create `Message Keys`.
- `Message Key` – An 80-byte value that is used to encrypt message contents. 32 bytes are used for an AES-256 key, 32 bytes for a HMAC-SHA256 key, and 16 bytes for an IV.

Other Key Types

- `Linking Secret Key` - A 32-byte value that is generated on a companion device and must be passed by a secure channel to the primary device, used to verify an HMAC of the linking payload received from a primary device. The transmission of this key from companion devices to the primary device is done by scanning a QR code.

Companion Linking

- Linking Metadata - An encoded blob of metadata assigned to a companion device during linking, used in conjunction with the companion device's `Identity Key` to identify a linked companion on WhatsApp clients.
- Signed Device List Data - An encoded list identifying the currently linked companion devices at the time of signing. Signed by the primary device's `Identity Key` using the 0x0602 prefix.
- Account Signature - A Curve25519 signature computed over the 0x0600 prefix, Linking Metadata, and companion device's public `Identity Key` using a primary device's `Identity Key`.
- Device Signature - A Curve25519 signature computed over the 0x0601 prefix, Linking Metadata, companion device's public `Identity Key`, and primary's devices public `Identity Key` using a companion device's `Identity Key`.

Client Registration

Primary Device Registration

At registration time, a WhatsApp client transmits its public `Identity Key`, public `Signed Pre Key` (with its signature), and a batch of public `One-Time Pre Keys` to the server. The WhatsApp server stores these public keys associated with the user's identifier.

Companion Device Registration

To link a companion device to a WhatsApp account, the user's primary device must first create an Account Signature by signing the new device's public `Identity Key` and the companion device must create a Device Signature by signing the primary's public `Identity Key`. Once both signatures are produced, end-to-end encrypted sessions can be established with the companion device.

To link a companion device:

1. The companion client displays its public Identity Key $(I_{companion})$ and a generated ephemeral Linking Secret Key $(L_{companion})$ in a linking QR code. $L_{companion}$ is never sent to WhatsApp server.
2. The primary client scans the linking QR code and saves $I_{companion}$ to disk.
3. The primary loads its own Identity Key as $I_{primary}$.
4. The primary generates Linking Metadata as $L_{metadata}$ and updated Device List Data containing the new companion as $L_{istData}$.
5. The primary generates an Account Signature for the companion, $A_{signature}$ = CURVE25519_SIGN($I_{primary}$, 0x0600 || $L_{metadata}$ || $I_{companion}$).
6. The primary generates a Device List Signature for the updated Device List Data, `ListSignature = CURVE25519_SIGN(`$I_{primary}$`, 0x0602 || ListData)`.
7. The primary serializes the Linking Data (L_{data}) containing $L_{metadata}$, $I_{primary}$ and $A_{signature}$.
8. The primary generates a Linking HMAC, `PHMAC = HMAC-SHA256(`$L_{companion}$, L_{data}`)`.
9. The primary sends `ListData, ListSignature,` L_{data} and `PHMAC` to WhatsApp server. See "Transport Security" for information about the secure connection between WhatsApp clients and servers.
10. The server stores `ListData` and `ListSignature`, and forwards L_{data} and `PHMAC` to the companion.

11. The companion verifies PHMAC, decodes Ldata into Lmetadata, Iprimary and Asignature, and verifies Asignature.

12. The companion saves $L_{metadata}$ and $I_{primary}$ to disk.

13. The companion generates a Device Signature for itself, $D_{signature}$ = CURVE25519_SIGN($I_{companion}$, 0x0601 || $L_{metadata}$ || $I_{companion}$ || $I_{primary}$).

14. The companion uploads $L_{metadata}$, $A_{signature}$, $D_{signature}$, $I_{companion}$, the companion's public Signed Pre Key (with its signature), and a batch of public One-Time Pre Keys to WhatsApp server.

15. The server stores the uploaded data associated with the user's identifier combined with a device specific identifier.

Initiating Session Setup

In order for WhatsApp users to communicate with each other securely and privately, the sender client establishes a pairwise encrypted session with each of the recipient's devices. Additionally, the sender client establishes a pairwise encrypted session with all other devices associated with the sender account. Once these pairwise encrypted sessions have been established, clients do not need to rebuild new sessions with these devices unless the session state is lost, which can be caused by an event such as an app reinstall or device change.

WhatsApp uses this "client-fanout" approach for transmitting messages to multiple devices, where the WhatsApp client transmits a single message N number of times to N number of different devices. Each message is individually encrypted using the established pairwise encryption session with each device.

To establish a session:

1. The initiating client ("initiator") requests the public Identity Key, public Signed Pre Key, and a single public One-Time Pre Key for each device of the recipient and each additional device of the initiating user (excluding the initiator).

2. The server returns the requested public key values. A One-Time Pre Key is only used once, so it is removed from server storage after being requested. If the recipient's latest batch of One-Time Pre Keys has been consumed and the recipient has not replenished them, no One-Time Pre Key will be returned. Additionally, for each companion device (for both the initiator's account and the recipient's), the server also returns the Linking Metadata ($L_{metadata}$), Account Signature ($A_{signature}$) and Device signature ($D_{signature}$) that was uploaded by the companion device when linked.

3. For every returned key set for a companion device, the initiator needs to verify Asignature by CURVE25519_VERIFY_SIGNATURE($I_{primary}$, 0x0600 || $L_{metadata}$ || $I_{companion}$), and $D_{signature}$ by CURVE25519_VERIFY_SIGNATURE($I_{companion}$, 0x0601 || $L_{metadata}$ || $I_{companion}$

|| $I_{primary}$). If any of the verification fails for a companion device, the initiator terminates the encryption session building process immediately and will not send any messages to that device.

After getting the keys from server and verifying each device identity, the initiator starts to establish the encryption session with each individual device:

1. The initiator saves the recipient's `Identity Key` as $I_{recipient}$, the `Signed Pre Key` as $S_{recipient}$, and the `One-Time Pre Key` as $O_{recipient}$.

2. The initiator generates an ephemeral Curve25519 key pair, $E_{initiator}$.

3. The initiator loads its own `Identity Key` as $I_{initiator}$.

4. The initiator calculates a master secret as `master_secret` = ECDH($I_{initiator}$, $S_{recipient}$) || ECDH($E_{initiator}$, $I_{recipient}$) || ECDH($E_{initiator}$, $S_{recipient}$) || ECDH($E_{initiator}$, $O_{recipient}$). If there is no `One Time Pre Key`, the final ECDH is omitted.

5. The initiator uses HKDF to create a `Root Key` and `Chain Key`s from the `master_secret`.

Receiving Session Setup

After building a long-running encryption session, the initiator can immediately start sending messages to the recipient, even if the recipient is offline.

Until the recipient responds, the initiator includes the information (in the header of all messages sent) that the recipient requires to build a corresponding session. This includes the initiator's ($E_{initiator}$ and $I_{initiator}$). Additionally, if the initiator is a companion device, it also includes its $I_{primary}$, $L_{metadata}$, $A_{signature}$ and $D_{signature}$.

When the recipient receives a message that includes session setup information:

1. If the sender is a companion device, the recipient needs to verify $A_{signature}$ by CURVE25519_VERIFY_SIGNATURE(Iprimary, 0x0600 || $L_{metadata}$ || $I_{companion}$), and $D_{signature}$ by CURVE25519_VERIFY_SIGNATURE($I_{companion}$, 0x0601 || $L_{metadata}$ || $I_{companion}$ || $I_{primary}$). If any of the verifications fail, the receiver stops building the encryption session and rejects the message from that sender device.

2. The recipient calculates the corresponding master_secret using its own private keys and the public keys advertised in the header of the incoming message.

3. The recipient deletes the One-Time Pre Key used by the initiator.

4. The initiator uses HKDF to derive a corresponding Root Key and Chain Keys from the master_secret.

Exchanging Messages

Once a session has been established, clients exchange messages that are protected with a `Message Key` using `AES256` in CBC mode for encryption and `HMAC-SHA256` for authentication. The client uses client-fanout for all the exchanged messages, which means each message is encrypted for each device with the corresponding pairwise session.

The `Message Key` changes for each message transmitted, and is ephemeral, such that the `Message Key` used to encrypt a message cannot be reconstructed from the session state after a message has been transmitted or received.

The `Message Key` is derived from a sender's `Chain Key` that "ratchets" forward with every message sent. Additionally, a new ECDH agreement is performed with each message roundtrip to create a new `Chain Key`. This provides forward secrecy through the combination of both an immediate "hash ratchet" and a round trip "DH ratchet."

Calculating a Message Key from a Chain Key

Each time a new `Message Key` is needed by a message sender, it is calculated as:

1. `Message Key = HMAC-SHA256(Chain Key, 0x01)`.
2. The `Chain Key` is then updated as `Chain Key = HMAC-SHA256(Chain Key, 0x02)`.

This causes the `Chain Key` to "ratchet" forward, and also means that a stored `Message Key` can't be used to derive current or past values of the `Chain Key`.

Calculating a Chain Key from a Root Key

Each time a message is transmitted, an ephemeral `Curve25519` public key is advertised along with it. Once a response is received, a new `Chain Key` and `Root Key` are calculated as:

1. $\texttt{ephemeral\_secret} = \texttt{ECDH}(\texttt{Ephemeral}_{\texttt{sender}}, \texttt{Ephemeral}_{\texttt{recipient}})$.
2. `Chain Key, Root Key = HKDF(Root Key, ephemeral_secret)`.

A chain is only ever used to send messages from one user, so message keys are not reused. Because of the way `Message Keys` and `Chain Keys` are calculated, messages can arrive delayed, out of order, or can be lost entirely without any problems.

Transmitting Media and Other Attachments

Large attachments of any type (video, audio, images, or files) are also end-to-end encrypted:

1. The WhatsApp user's device sending a message ("sender") generates an ephemeral 32 byte `AES256` key, and an ephemeral 32 byte `HMAC-SHA256` key.
2. The sender encrypts the attachment with the `AES256` key in CBC mode with a random IV, then appends a MAC of the ciphertext using HMAC-SHA256.
3. The sender uploads the encrypted attachment to a blob store.
4. The sender transmits a normal encrypted message to the recipient that contains the encryption key, the HMAC key, a SHA256 hash of the encrypted blob, and a pointer to the blob in the blob store.
5. All receiving devices decrypt the message, retrieve the encrypted blob from the blob store, verify the SHA256 hash of it, verify the MAC, and decrypt the plaintext.

Group Messages

End-to-end encryption of messages sent to WhatsApp groups utilize the established pairwise encrypted sessions, as previously described in the "Initiation Session Setup" section, to distribute the "Sender Key" component of the Signal Messaging Protocol.

When sending a message to a group for the first time, a "Sender Key" is generated and distributed to each member device of the group, using the pairwise encrypted sessions. The message content is encrypted using the "Sender Key", achieving an efficient and secure fan-out for the messages that are sent to groups.

The first time a WhatsApp group member sends a message to a group:

1. The sender generates a random 32-byte `Chain Key`.
2. The sender generates a random Curve25519 `Signature Key` key pair.
3. The sender combines the 32-byte `Chain Key` and the public key from the `Signature Key` into a `Sender Key` message.
4. The sender individually encrypts the `Sender Key` to each member of the group, using the pairwise messaging protocol explained previously.

For all subsequent messages to the group:

1. The sender derives a `Message Key` from the `Chain Key`, and updates the `Chain Key`.

2. The sender encrypts the message using `AES256` in CBC mode.

3. The sender signs the ciphertext using the `Signature Key`.

4. The sender transmits the single ciphertext message to the server, which does server-side fan-out to all group participants.

The “hash ratchet” of the message sender’s `Chain Key` provides forward secrecy. Whenever a group member leaves, all group participants clear their `Sender Key` and start over.

Message History Syncing

Immediately after linking a companion device, the primary device end-to-end encrypts a copy of messages from recent chats and transfers them to the newly linked companion device. This process, called Messaging History Syncing, generates bundles of the end-to-end encrypted messages using the same mechanism of encryption as described in the “Transmitting Media and Other Attachments” section. Steps 1 through 5 explain the specifics regarding key, IV, mac generation, as well as the encryption, transmission, and decryption of these end-to-end encrypted bundles. Once a companion device has successfully decrypted, unpacked, and stored all the messages of a given bundle, all the associated data (including the downloaded encrypted bundle blob, the pointer to the encrypted blob storage, and all the keys) are deleted from the companion device.

Call Setup

WhatsApp voice and video calls are end-to-end encrypted. When a WhatsApp user initiates a voice or video call:

1. The initiating client (“initiator”) establishes encrypted sessions with each of the devices of the recipient (as outlined in the Initiating Session Setup Section), if these haven’t been set up yet.

2. The initiator generates a set of random 32-byte `SRTP` master secrets for each of the recipient’s devices.

3. The initiator sends an incoming call message to each of the devices of the recipient. Each recipient’s device receives this message, which contains the end-to-end encrypted `SRTP` master secret.

4. If the responder answers the call from one of the devices, a `SRTP` encrypted call is started, protected by the SRTP master secret generated for that device.

The `SRTP` master secret is persisted in memory on the client device and used only during the call. WhatsApp servers do not have access to the `SRTP` master secrets.

Group Calling

WhatsApp group calls are end-to-end encrypted. Unlike one-to-one calls that setup keys only once, in group calls, keys are reset whenever a participant joins the call or leaves the call.

Key reset in group calls is achieved by the following steps:

1. When a participant joins or leaves the call, the WhatsApp server arbitrarily selects one of the active participants as the key distributor.

2. The key distributor generates a random 32-byte `SRTP` master secret.

3. The key distributor establishes an encrypted session with the active device of each connected participant in the current group call (as outlined in the Initiating Session Setup Section), if these haven't been setup yet.

4. The key distributor initiates one message with end-to-end encrypted `SRTP` master secret for each participant. When these messages are delivered, a `SRTP` encrypted group call ensues.

Note that in a group call, a participant becomes active when they initiate a group call or accepts the group call invitation from any of their devices. Therefore each active participant has exactly one active device.

The `SRTP` master secret is persisted in-memory, and is overwritten when a new `SRTP` master secret is generated and delivered. Whatsapp servers do not have access to them, and cannot access the actual audio and video media.

To ensure call quality and to avoid race conditions from conflicting user actions, the WhatsApp server stores the state of the current group call (for example: participant list, call initiator) and media metadata (e.g. resolution, media type). With this information, the WhatsApp server is able to broadcast participant membership changes and select one as key distributor to initiate key reset.

Statuses

WhatsApp statuses are encrypted in much the same way as group messages. The first status sent to a given set of devices follows the same sequence of steps as the first time a WhatsApp group member sends a message to a group. Similarly, subsequent statuses sent to the same set of devices follow the same sequence of steps as all subsequent messages to a group. When a status sender removes a receiver either through changing status privacy settings or removing a number from their address book, the status sender clears their Sender Key and starts over.

Live Location

Live location messages and updates are encrypted in much the same way as group messages. Currently, sending and receiving live locations is only supported on primary devices. The first live location message or update sent follows the same sequence of steps as the first time a WhatsApp group member sends a message to a group. But, live location demands a high volume of location broadcasts and updates with lossy delivery where receivers can expect to see large jumps in the number of ratchets, or iteration counts. The Signal Protocol uses a linear-time algorithm for ratcheting that is too slow for this application. This document offers a fast ratcheting algorithm to solve this problem.

Chain keys are currently one-dimensional. To ratchet `N` steps takes `N` computations. Chain keys are denoted as `CK(iteration count)` and message keys as `MK(iteration count)`.

$$\begin{tikzcd}
\mathrm{CK}(0) \arrow[d] & \\
\mathrm{CK}(1) \arrow[d] & \\
\dots \arrow[d] & \\
\mathrm{CK}(N-1) \arrow[r] & \mathrm{MK}(N-1)
\end{tikzcd}$$

Consider an extension where we keep two chains of chain keys:

$$\begin{tikzcd}
\mathrm{CK}_1(0) \arrow[r] \arrow[d] & \mathrm{CK}_2(0) \arrow[d] & \\
\mathrm{CK}_1(1) & \mathrm{CK}_2(1) \arrow[d] & \\
 & \dots \arrow[d] & \\
 & \mathrm{CK}_2(M-1) \arrow[r] & \mathrm{MK}(M-1)
\end{tikzcd}$$

In this example, message keys are always derived from CK_2. A receiver who needs to ratchet by a large amount can skip M iterations at a time (where M is an agreed-upon constant positive integer) by ratcheting CK_1 and generating a new CK_2:

$$\begin{tikzcd}
CK_1(0) \arrow[d] & & \\
CK_1(1) \arrow[r] \arrow[d] & CK_2(0) \arrow[r] \arrow[d] & MK(M) \\
CK_1(2) & CK_2(1) &
\end{tikzcd}$$

A value of CK_2 may be ratcheted up to M times. To ratchet N steps takes up to $[N \div M] + M$ computations.

After a sender creates a message key and encrypts a message with it, all chain keys on the path that led to its creation must be destroyed to preserve forward secrecy.

$$\begin{tikzcd}
\cancel{CK_1(0)} \arrow[d] & & \\
\cancel{CK_1(1)} \arrow[r] \arrow[d] & \cancel{CK_2(0)} \arrow[r] \arrow[d] & MK(M) \\
CK_1(2) & CK_2(1) &
\end{tikzcd}$$

Generalizing to D dimensions, a sender can produce D initial chain keys. Each chain key but the first is derived from the preceding chain key using a distinct one-way function: these are the right-pointing arrows in the diagram above. Senders distribute all D chain keys to receivers who need them, except as noted below.

$$\begin{tikzcd}
RNG \arrow[r] & CK_1(0) \arrow[r] & CK_2(0) \arrow[r] & \ldots \arrow[r] & CK_D(0)
\end{tikzcd}$$

Legal values for D are positive powers of two less than or equal to the number of bits in the iteration counter: 1, 2, 4, 8, 16, and 32. Implementors select a value of D as an explicit CPU-memory (or CPU-network bandwidth) tradeoff.

If a chain key CK_j (for j in $[1, D]$) has an iteration count of M, it cannot be used. This algorithm restores the chain keys to a usable state:

1. If j = 1, fail because the iteration count has reached its limit.
2. Derive CK_j from CK_{j-1}.
3. Ratchet CK_{j-1} once, recursing if necessary.

Moving from one iteration count to another never ratchets a single chain key more than M times. Therefore, no ratcheting operation takes more than $D \times M$ steps.

Signal uses different functions for ratcheting versus message key computation, since both come from the same chain key. In this notation `{x}` refers to an array of bytes containing a single byte `x`.

```
MK        = HmacSHA256(CK_j(i), {1})
CK_j(i+1) = HmacSHA256(CK_j(i), {2})
```

Each dimension must use a different function. Keys are initialized as:

```
j = 1 : CK_1(0) ≔ RNG(32)
j > 1 : CK_j(0) ≔ HmacSHA256(CK_{j-1}(0), {j+1})
```

And ratcheted as:

```
CK_j(i) ≔ HmacSHA256(CK_j(i-1), {j+1})
```

App State Syncing Security

Introduction

App State Syncing enables a consistent experience across devices, and is end-to-end encrypted. Prior to supporting companion devices, the WhatsApp client was the sole owner and the source of truth for all client settings and other data, referred to as App State. With the introduction of companion devices, App State is synchronized between all of the user's devices securely, using end-to-end encryption.

Example App State Syncing client settings and other data include the following:

- Chat properties, such as Muted, Pinned, Deleted
- Message properties, such as Deleted for Me, Starred
- Contact-related properties, such as contact names, broadcast list names
- Most recently used GIFs, stickers, emojis

App State does not include user messages' content, nor keys that could be used to decrypt messages, nor settings that might impact the secrecy of messages.

The synchronization of App State between a user's devices requires storage of end-to-end encrypted data on the WhatsApp server to facilitate the transmission between the different devices for the user's account. WhatsApp servers do not have access to the keys that could be used to decrypt the App State data that is stored.

A **Collection** is a representation of several use-cases that are grouped together. For example, various Chat Properties (see above) can be modelled as a single Collection. Collections are implemented as a dictionary (a set of Index-Value pairs) and are fixed to specific client versions.

Initially, Collections are empty. To modify a Collection, a client device submits a **Mutation** which either sets a new Value for a given Index (SET Mutation), or removes the pair from the Collection (REMOVE Mutation).

A group of Mutations submitted together constitute a **Patch**, which atomically changes the Collection from version N to version N + 1. The server maintains an ordered queue of Patches (**Patch Queue**), which consists of all Patches submitted in the last X days.

A server-side process, called a **Base Roller**, periodically builds a **Snapshot** which represents the state of the Collection after applying all Patches up until and including the Patch with version N. The first Snapshot is built from the entire Patch Queue, while subsequent Snapshots are built by applying new Patches to the previous Snapshot. A Snapshot can be used to initialize a newly registered device or to optimize the data traffic by sending a Snapshot instead of the list of Patches.

App State Syncing is designed to guarantee the secrecy and integrity of the data being synchronized. The pairwise encrypted sessions (as outlined in the Initiating Session Setup Section) are used for transferring secret keys between different devices of the same account.

Terms

- `Base Key` - Input key material used to generate the keys used to encrypt the data or provide its integrity.
- `Index MAC Key` - Key derived from the `Base Key` via HKDF and used to compute the HMAC of the index.
- `Value Encryption Key` - Key derived from the `Base Key` via HKDF and used to encrypt the combined Mutation index and value. The encryption is done via AES-256 in CBC mode.
- `Value MAC Key` - Key derived from the `Base Key` via HKDF and used to compute the `HMAC` of the combined Mutation index and value. Used on the MAC stage of `Encrypt-then-MAC` approach to provide authenticated encryption.
- `Snapshot MAC Key` - Key derived from the `Base Key` via HKDF and used to provide anti-tampering for Snapshots generated by Base Roller.
- `Patch MAC Key` - Key derived from the `Base Key` via HKDF and used to provide anti-tampering for Patches.

- `KeyID` - Unique identifier for the `Base Key`. `Base Keys` are rotated periodically and when a device is removed from the account to provide eventual future secrecy. An attacker in possession of a removed device and access to the server can no longer decrypt the content of `SET Mutations` submitted after the removal.

- `Operation` - Byte value which identifies a Mutation as `SET` or `REMOVE`.

Encryption of Mutations

In order for Base Roller to coalesce sequences of actions to the same index, it needs the index submitted to the server to be deterministic. HMAC of the index is used as an identifier of the index-value record the Mutations refers to. This also makes sure that the indexes that the server sees have the same length and prevent the server from guessing the record for which the Mutation is applied.

Values (together with indexes, as mentioned above) are encrypted using standard authenticated encryption (described below) with random IVs.

Combined index and value plaintext are supplemented with arbitrary length padding in order to enable some model of differential privacy on the type of the records.

1. Generate the I`ndex MAC Key, Value Encryption Key, Value MAC Key, Snapshot MAC Key`, and `Patch MAC Ke`y from `Base Key` by means of HKDF.

2. Compute `HMAC-SHA2-256` of the index.

3. Construct the plaintext by combining Index and Value with random padding (used to obfuscate the size of the Mutation from the server).

4. Construct the associated data by concatenating `Operation` with `KeyID`.

5. Apply `Encrypt-then-MAC` approach with `AES-256-CBC` keyed by `Value Encryption Key` and `HMAC-SHA2-512` keyed by `Value MAC Key`.

6. MAC computed on Step 2, ciphertext computed on Step 5, together with `Operation` and `KeyID` form an encrypted Mutation.

Anti-Tampering

The anti-tampering mechanisms described below are designed to prevent:

- Drop, reorder, or replay Mutation within a Patch
- Drop, reorder, or replay (including a construction of new Patches) Patches within a Collection or even between the Collection
- Drop or replay Mutations within a Snapshot constructed by a Base Roller

Snapshot Integrity

The server periodically runs a Base Roller which compacts the Patch Queue into a single Snapshot. Clients cannot predict when the Snapshot is going to be built (at the extreme the Snapshots could be built on every Patch). Thus, clients include an additional unforgeable checksum for each Patch in order to be able to verify all possible Snapshots built by the server.

Our approach relies on a homomorphic hashing algorithm called LtHash. It has the following two important properties:

- Set homomorphism: for any two disjoint sets `S` and `T`, `LtHash(S) + LtHash(T) = LtHash(S U T)`.
- Collision resistance: it is difficult (computationally infeasible) to find two distinct sets `S` and `T` for which `LtHash(S) = LtHash(T)`.

A 1024-bit variant called `LtHash16` and HKDF as an extensible output function (XOF) is used. For each Collection clients must maintain a 1024-bit value of `LtHash16` computed over the current Snapshot of the Collection. The MAC computed over the content of plaintext index and value together with authenticated data is used as input to the `LtHash`.

Upon receiving or constructing a new Patch, the set homomorphism property of `LtHash` is used to compute the new 1024-bit value corresponding to the new state (after applying the Patch in question):

- Assume that the current value of digest is `CurrentLtHash`, and Patch `P` is being processed.
- Build a set `R` of `MACs` of previous states of all records that are affected (deleted with REMOVE or overwritten with a SET operation).
- Build a set `A` of `MACs` of all SET records (in encrypted form) in the Patch `P`.
- Construct `NewLtHash = LtHash16Add(LtHash16Subtract(CurrentLtHash, R), A)`. See below on how these operations are defined.

`LtHash16Add` operation mentioned above is defined as follows:

```
LtHash16Add(H, A):
    R = H
    for Item in A:
        R = LtHash16AddSingle(R, Item)
    return R

LtHash16AddSingle(H, I):
    X = HKDF(1024, “WhatsApp Patch Integrity”, I)
    return PointwiseAdd16(H, X)
```

where `PointwiseAdd16` performs pointwise overflowing addition of two 1024-bit byte arrays interpreting them as arrays of 16-bit unsigned integers. Operation `LtHash16Subtract` is defined similarly to `LtHash16Add` replacing pointwise addition with pointwise subtraction.

Further, a MAC over the computed value of `LtHash16` concatenated with 8 byte Patch version and the name of the Collection is computed:

```
SnapshotMAC = HMAC_SHA_256(
    SnapshotMACKey,
    LtHash ||
    TO_64_BIT_NETWORK_ORDER(PatchVersion) ||
    TO_UTF8(CollectionName)
)
```

Patch Queue Integrity

To prevent tampering with content of a Patch, clients must compute the HMAC over the 32-byte MACs of each individual Mutation that is part of the Patch together with the Patch version number and Collection Name:

```
PatchMAC = HMAC_SHA_256(
    PatchMACKey,
    SnapshotMAC ||
    MutationMAC_0 ||
    MutationMAC_1 ||
    .. ||
    MutationMAC_N ||
    TO_64_BIT_NETWORK_ORDER(PatchVersion) ||
    TO_UTF8(CollectionName)
)
```

where `MutationMAC_i` is the last 32 bytes of value ciphertext of Mutation #i in the Patch, and `PatchVersion` is the version of the Patch about to be submitted (i.e. latest known version of the Collection plus one). Note that upon receiving a Patch, the client must verify it, including the expected version of the Patch (which must match the server-assigned version).

Both values `PatchMAC` and `SnapshotMAC` are included in the Patch and submitted to the server.

Verification

After downloading a Patch, clients must first verify its correctness by recomputing the `PatchMAC` and comparing it with the value included with the Patch. After that, clients verify that `SnapshotMAC` is correct as well by repeating the steps outlined above.

The Base Roller process on the server must preserve `SnapshotMAC` (and the `KeyID` used to generate it) of the latest Patch that was used to construct the Base Rolled Snapshot. This value is used by a client that received a Snapshot to independently verify its integrity by applying LtHash16 over all of its records and further compute the MAC as described above.

Key Rotation

A Key Rotation involves a client randomly generating a new key tuple and broadcasting it to all other devices. In the event of Key Rotation, all future Mutations must not use any previous key version. To preserve the ability of the server to coalesce the Mutations applied to a record when updating across a key boundary, a client must submit a `REMOVE` Mutation with the old key and a `SET` Mutation (if needed) with the new key. It is notable that a simultaneous `REMOVE` and `SET` occurring as the key version increases will be relatively easy for the server to correlate as equating to an update of the record. In collusion with anyone with access to the old key, WhatsApp would therefore be able to determine with high confidence the value of the new index; and might be able to assume that this means it has been updated.

The following two mechanisms are used to combat this:

1. Post-Rotation Update Obfuscation - When submitting a Mutation to the server, clients will use this opportunity to rotate some other number of records, ensuring that WhatsApp cannot determine which of the old indexes was being updated, and cannot directly map any of the old records to which new record represents them.

2. Asynchronous Key Catch-Up - Ensures that after a Key Rotation, there will be at some point in the future when no current records are encrypted with the preceding key(s). This means that on some cadence, clients will issue a series of `SET` and `REMOVE` to re-encrypt old records under a new key version, without updating the actual plaintext values. Catch-Up updates are indistinguishable from a logical `UPDATE` operation, so that the server in collusion with a removed device can never determine when an old record is being updated.

The key must be rotated whenever a device is being unregistered. Additionally, clients rotate the key periodically (for example once a month).

Each device maintains a list of the encryption keys together with additional data:

1. `KeyData` - Actual base key bytes

2. `KeyID` - ID of the key

3. `Fingerprint` - Data structure which identifies a list of devices existing at the moment when the key was generated (and thus was shared with)

4. `Timestamp` - Time when the key was created

The `KeyID` is composite and consists of 4 byte `Epoch` and 2 byte `DeviceID`. `Epoch` is selected randomly between 1 and 65536 by the primary devices during the registration of the first companion device, and after that increases by 1 every time a device rotates a key. The `DeviceID` component of the `KeyID` is used to resolve races between several devices rotating the key at the same time, so that all keys will receive unique `IDs`. To settle on a single key after such an event, clients prefer the key with the smallest `DeviceID` component when `Epoch` components are equal. Otherwise, always prefer the `KeyID` with the largest `Epoch`. Additionally, one (or in rare cases several) encryption keys can be active at any given time.

Key Rotation must happen under the following conditions:

- If a client detects that a previously known device was removed, it must locally mark all active encryption keys as expired.
- Upon receiving an `AppStateSyncKeyShare` message mark all keys with smaller Epoch as expired.
- Upon receiving a Mutation in any Collection mark all keys with a smaller Epoch as expired.
- When a client wants to submit a new Patch to the server it first must check the list of known keys. If there is one that is still active it uses it. Otherwise, it performs the Key Rotation. To rotate the key:
    1. Generate a new `KeyID` by concatenating `DeviceID` with an incremented `Epoch` (maximum value among all known encryption keys).
    2. Generate new key material from `CSPRNG`.
    3. Generate a new `Fingerprint` from the current registration data.
    4. Persist the key information and send it to all other devices using the corresponding pairwise encrypted sessions.
    5. Use the key to encrypt the Mutations and submit them to the server.
- In some cases, clients cannot determine whether a key is still valid based on event ordering alone. To compute whether the last known active key is valid or not, clients compare the key's `Fingerprint` with the current device registration data.

If a device receives a Mutation from the server and the `KeyID` is not known, a device can request to resend the encryption keys from other devices.

To guarantee that encryption keys are not shared with untrusted devices, all client applications only send them via authenticated pairwise encrypted sessions:

1. While performing Key Rotation, a device must send the new key to all other devices which are known to be authorized by the primary.
2. When a device receives a new key from a device which is not authorized by the primary this key is ignored.

To make sure that other devices will not inadvertently use an encryption key that should be expired on device removal, the device that performs the removal (the companion device itself or primary device) submits a Patch into all Collections marking all the current keys as expired. This Patch informs other devices that encryption keys with epoch less or equal to the provided epoch should not be used going forward.

Verifying Keys

WhatsApp users additionally have the option to verify the keys of their devices and the devices of the users with which they are communicating in end-to-end encrypted contexts, so that they are able to confirm that an unauthorized third party (or WhatsApp) has not initiated a man-in-the-middle attack on any of the pairwise encryption sessions between devices. Each device's sessions are independent from one another, and as a result devices of the same user must be verified independently. Verification can be done by scanning a QR code or by comparing a 60-digit number.

The QR code contains:

1. A version.
2. The user identifier for both parties.
3. The full 32-byte public `Identity Key` for all devices of both parties.

When either device scans the other's QR code, the keys are compared to ensure that what is in the QR code matches the `Identity Key` as retrieved from the server.

The 60-digit number is computed by concatenating the two 30-digit numeric fingerprints for each user's device `Identity Keys`. To calculate a 30-digit numeric fingerprint:

1. Lexicographically sort public Identity Keys for all of the user's devices and concatenate them.
2. Iteratively SHA-512 hash the sorted `Identity Keys` and user identifier 5200 times.
3. Take the first 30 bytes of the final hash output.
4. Split the 30-byte result into six 5-byte chunks.
5. Convert each 5-byte chunk into 5 digits by interpreting each 5-byte chunk as a big-endian unsigned integer and reducing it modulo 100000.
6. Concatenate the six groups of five digits into thirty digits.

Transport Security

Communication between WhatsApp clients and WhatsApp chat servers is layered within a separate encrypted channel using Noise Pipes with Curve25519, AES-GCM, and SHA256 from the Noise Protocol Framework for long running interactive connections.

This provides clients with a few nice properties:

1. Extremely fast lightweight connection setup and resume.
2. Encrypts metadata to hide it from unauthorized network observers. No information about the connecting user's identity is revealed.
3. No client authentication secrets are stored on the server. Clients authenticate themselves using a Curve25519 key pair, so the server only stores a client's public authentication key. If the server's user database is ever compromised, no private authentication credentials will be revealed.

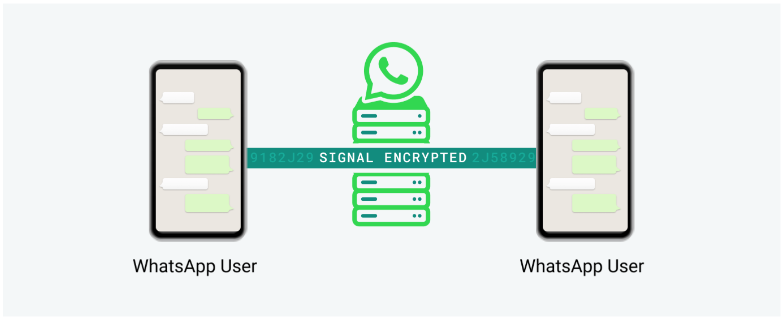
Note: In cases where a business user delegates operation of their Business API client to a vendor, that vendor will have access to their private keys - including if that vendor is Facebook. However, these private keys will still not be stored on the WhatsApp chat server. See below for details.

Defining End-to-End Encryption

WhatsApp defines end-to-end encryption as communications that remain encrypted from a device controlled by the sender to one controlled by the recipient, where no third parties, not even WhatsApp or our parent company Facebook, can access the content in between. A third party in this context means any organization that is not the sender or recipient user directly participating in the conversation.

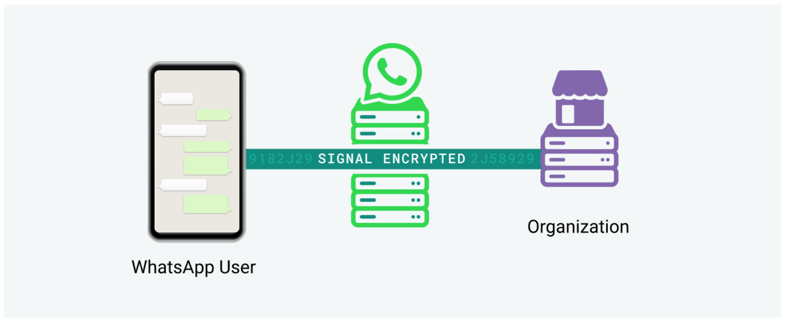
Implementation on WhatsApp Services

This is straightforward when it comes to two people communicating on their phones or computers using WhatsApp Messenger or the WhatsApp Business App: each person's WhatsApp endpoint is running on a device they control.

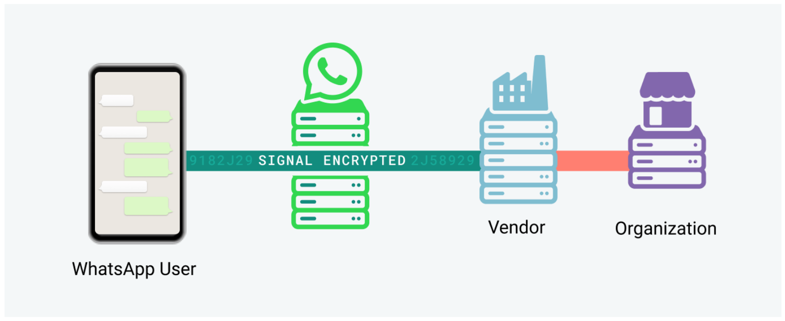


Some organizations may use the WhatsApp Business API, an application that can be deployed as a WhatsApp endpoint on a server. The Business API allows those organizations to programmatically send and receive messages.

WhatsApp considers communications with Business API users who manage the API endpoint on servers they control to be end-to-end encrypted since there is no third-party access to content between endpoints.



Some organizations may choose to delegate management of their WhatsApp Business API endpoint to a vendor. In these instances, communication still uses the same Signal protocol encryption and clients on or after version v2.31 are configured to generate private keys within the vendor-controlled API endpoint. However, because the WhatsApp Business API user has chosen a third party to manage their endpoint, WhatsApp does not consider these messages end-to-end encrypted.



In 2021, organizations who use the Business API will be able to designate WhatsApp's parent company, Facebook, as the vendor that operates the Business API endpoint on their behalf. Since such messages are not delivered directly to an endpoint controlled by the organization, WhatsApp does not consider chats with organizations who choose to use Facebook to operate their API endpoint to be end-to-end encrypted.

Encryption Has No Off Switch

All chats use the same Signal protocol outlined in this whitepaper, regardless of their end-to-end encryption status. The WhatsApp server has no access to the client's private keys, though if a business user delegates operation of their Business API client to a vendor, that vendor will have access to their private keys - including if that vendor is Facebook.

When chatting with an organization that uses the Business API, WhatsApp determines the end-to-end encryption status based only on the organization's choice of who operates its endpoint.

The encryption status of an end-to-end encrypted chat cannot change without the change being visible to the user.

Displaying End-to-End Encryption Status

Across all our services, WhatsApp makes the end-to-end encryption status of a chat clear. If the user's phone sees that it's communicating with an API endpoint that delegates operation of its API to a vendor, the phone will display this to the user. The user can also double check the encryption status within the chat or in the business info section of their app.

These changes will take effect in all WhatsApp versions after January 2021.

Conclusion

All WhatsApp messages are sent with the same Signal protocol outlined above. WhatsApp considers all messages, voice calls, and video calls sent between all devices controlled by a sender user and all devices controlled by a recipient user to be end-to-end encrypted. WhatsApp message history syncing and app state syncing are also protected by end-to-end encryption. Communications with a recipient who elects to use a vendor to manage their API endpoint are not considered end-to-end encrypted. If this occurs, WhatsApp makes it clear to users within the chat.

The Signal Protocol library used by WhatsApp is based on the Open Source library, available here:

http://github.com/whispersystems/libsignal-protocol-java/