WhatsApp

WhatsApp Encryption Overview

Technical white paper

December 19, 2017
Originally published April 5, 2016

Contents

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) and make WhatsApp calls around the world. WhatsApp messages, voice and video calls between a sender and receiver that use WhatsApp client software released after March 31, 2016 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. What's more, even if encryption keys from a user's device are ever physically compromised, they cannot be used to go back in time to decrypt previously transmitted messages.

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

Terms

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.

Client 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. At no time does the WhatsApp server have access to any of the client's private keys.

Initiating Session Setup

To communicate with another WhatsApp user, a WhatsApp client first needs to establish an encrypted session. Once the session is established, clients do not need to rebuild a new session with each other until the existing session state is lost through an external event such as an app reinstall or device change.

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 the recipient.
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.
3. 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 $O_{recipient}$.
4. The initiator generates an ephemeral Curve25519 key pair, $E_{initiator}$.
5. The initiator loads its own `Identity Key` as $I_{initiator}$.
6. 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.
7. 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 $I_{initiator}$.

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

1. The recipient calculates the corresponding `master_secret` using its own private keys and the public keys advertised in the header of the incoming message.
2. The recipient deletes the `One-Time Pre Key` used by the initiator.
3. 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 `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 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. The recipient decrypts the message, retrieves the encrypted blob from the blob store, verifies the SHA256 hash of it, verifies the MAC, and decrypts the plaintext.

Group Messages

Traditional unencrypted messenger apps typically employ "server-side fan-out" for group messages. A client wishing to send a message to a group of users transmits a single message, which is then distributed N times to the N different group members by the server.

This is in contrast to "client-side fan-out," where a client would transmit a single message N times to the N different group members itself.

Messages to WhatsApp groups build on the pairwise encrypted sessions outlined above to achieve efficient server-side fan-out for most messages sent to groups. This is accomplished using the "Sender Keys" component of the Signal Messaging Protocol.

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.

Call Setup

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

1. The initiator builds an encrypted session with the recipient (as outlined in Section *Initiating Session Setup*), if one does not already exist.
2. The initiator generates a random 32-byte SRTP master secret.
3. The initiator transmits an encrypted message to the recipient that signals an incoming call, and contains the SRTP master secret.
4. If the responder answers the call, a SRTP encrypted call ensues.

Statuses

WhatsApp statuses are encrypted in much the same way as group messages. The first status sent to a given set of recipients 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 recipients 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. 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 locations demand 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)`.

```
CK(0)
 ↓
CK(1)
 ↓
 …
 ↓
CK(N-1) → MK(N-1)
```

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

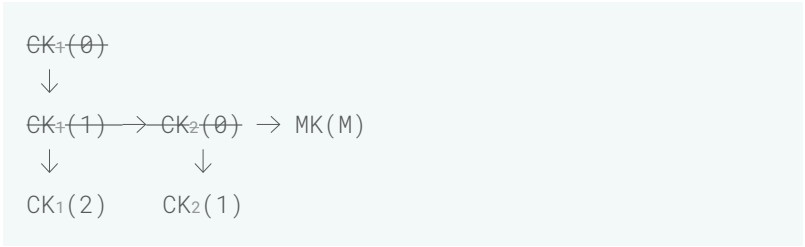
```
CK₁(0) → CK₂(0)
 ↓          ↓
CK₁(1)    CK₂(1)
             ↓
             …
             ↓
          CK₂(M-1) → MK(M-1)
```

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:

```
CK₁(0)
 ↓
CK₁(1) → CK₂(0) → MK(M)
 ↓          ↓
CK₁(2)    CK₂(1)
```

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.



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.

$$RNG \rightarrow CK_1(0) \rightarrow CK_2(0) \rightarrow \ldots \rightarrow CK_D(0)$$

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:

$$j = 1 : CK_1(0) \coloneqq RNG(32)$$
$$j > 1 : CK_j(0) \coloneqq HmacSHA256(CK_{j-1}(0), \{j+1\})$$

And ratcheted as:

$$CK_j(i) = HmacSHA256(CK_j(i-1), \{j+1\})$$

Verifying Keys

WhatsApp users additionally have the option to verify the keys of the other users with whom they are communicating so that they are able to confirm that an unauthorized third party (or WhatsApp) has not initiated a man-in-the-middle attack. This 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 both parties.

When either user 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 `Identity Key`. To calculate a 30-digit numeric fingerprint:

1. Iteratively SHA-512 hash the public `Identity Key` and user identifier 5200 times.
2. Take the first 30 bytes of the final hash output.
3. Split the 30-byte result into six 5-byte chunks.
4. 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.
5. Concatenate the six groups of five digits into thirty digits.

Transport Security

All communication between WhatsApp clients and WhatsApp servers is layered within a separate encrypted channel. On Windows Phone, iPhone, and Android, those end-to-end encryption capable clients use 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.

Conclusion

Messages between WhatsApp users are protected with an end-to-end encryption protocol so that third parties and WhatsApp cannot read them and so that the messages can only be decrypted by the recipient. All types of WhatsApp messages (including chats, group chats, images, videos, voice messages and files) and WhatsApp calls are protected by end-to-end encryption.

WhatsApp servers do not have access to the private keys of WhatsApp users, and WhatsApp users have the option to verify keys in order to ensure the integrity of their communication.

The Signal Protocol library used by WhatsApp is Open Source, available here: https://github.com/whispersystems/libsignal-protocol-java/