End-to-End Encryption in Jitsi Meet

Saúl Ibarra Corretgé, Emil Ivov

2021-08-24 (1.0)

This document describes the current state of *End-to-End* (E2EE) in Jitsi Meet. Jitsi Meet is a collection of projects designed for building highly scalable and secure video conferencing solutions.

Throughout this document we'll see the project architecture and how security and encryption are implemented all throughout.

Contents

Introduction

Jitsi Meet is a set of Open Source projects that provide a turn-key video conferencing solution.

While Jitsi's origin dates to 2003, it currently is WebRTC [1] compatible, with web browsers (or compatible endpoints) being the main endpoint used to participate in meetings.

Threat model

When designing E2EE features, the following actors could be considered the most likely adversaries:

- Insiders: parties involved in the maintenance of the Jitsi Meet installation.
- Outsiders: parties who gained illegitimate access to a component in a Jitsi Meet installation, for example a Jitsi Videobridge.
- Participants: any individual part of the meeting, who will have access to everyone else's audio / video once authorized to be in the meeting.

Note: In order to facilitate analysis and comparison, we intentionally employ terminology similar to that of other providers.

We presently focus on defending against *outsider* attacks.

Due to how modern software is distributed (frequent client and server updates under the same authority) it is our position that protecting against *insiders* is an extremely difficult aspiration.

Goals

E2EE is another layer of security in addition to the always present **transport encryption**. It is not meant to be used as a mechanism for authorizing users into a meeting.

Limitations

The current E2EE implementation in Jitsi Meet requires support for *Insertable Streams* [2]. This API is currently not supported in all browsers.

Mobile support varies depending on the type of endpoint:

- iOS browsers: not supported.
- Android browsers: supported as long as insertable streams are available.
- Native Jitsi Meet apps / SDK: not supported yet, work is in progress.

PSTN access to meetings is not possible when E2EE is used.

E2EE meetings are currently limited to 20 participants due to the signalling overhead.

Architecture

A basic Jitsi Meet setup is comprised of several components:

- Web server: serves the HTML / CSS and other assets, and proxies the signalling channel towards the XMPP server.
- XMPP server: acts as the signalling hub for communication across participants and other server components.
- Jicofo: conference focus; creates meetings rooms and is in charge of negotiating audio / video for participants.
- Videobridge (JVB): media router; it routes audio / video packets to all participants in a meeting.

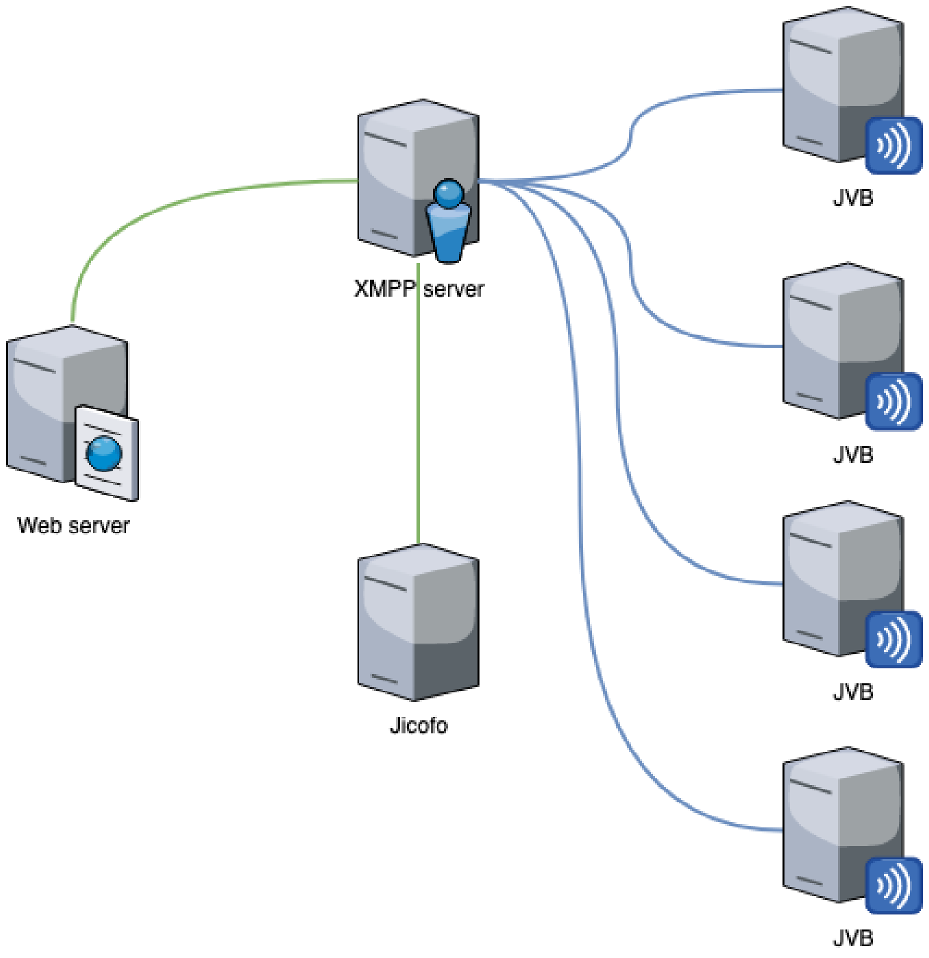

Figure 1: Architecture

The current architecture uses XMPP [3] for signalling. This signalling is transported via HTTPS or WebSocket (using secure web sockets). *WebRTC* [1] requires the origin site to use an encrypted transport, otherwise its capabilities are not available to websites.

Transport encryption

The audio and video content are received by the Jitsi Videobridge and forwarded to other participants in the conference. As specified in *WebRTC Security Architecture* [4] *DTLS-SRTP* [5] is used as the transport encryption.

Transport encryption is applied hop by hop. That is, the videobridge will have access to the (decrypted) payload, as some codec metadata is required in order to efficiently route video packets.

As mentioned earlier, this is one of the main reasons to implement E2EE, in order to prevent access to the media the videobridge receives.

End-to-End Encryption

While WebRTC does provide strong transport encryption capabilities with DTLS-SRTP being mandatory, in practice the use of *Selective Forwarding Units* [6] means encryption is terminated at the SFU boundary and it's thus hop by hop.

E2EE in Jitsi is implemented by adding an extra layer of encryption, that is, encrypting the audio / video media at the source, *before* it's encrypted with DTLS-SRTP. This way, when the SFU (videobridge) decrypts the DTLS-SRTP payload it won't be able to access the actual media contained within the payload.

As mentioned in the limitations, this extra layer of encryption can currently only be implemented in browsers supporting *insertable streams* [2].

Jitsi Meet implements a slight variant of the *SFrame* [7] specification for achieving E2EE, we call it JFrame.

The specification leaves key management out, so we implemented that in Jitsi in two ways: managed mode and unmanaged mde.

We have moved away from unmanaged mode at this time, but we may bring it back and make the mode configurable in the future, see appendix II.

Encryption

Encryption is performed with AES-GCM (with a 128 bit key) and the WebCrypto API.

AES-GCM needs a 96 bit initialization vector which we construct based on the SSRC, the RTP timestamp and a frame counter which is similar to how the IV is constructed in *SRTP with GCM* [8].

This IV gets sent along with the packet, adding 12 bytes of overhead. The GCM tag length is the default 128 bits or 16 bytes.

At a high level the encrypted frame format looks like this:

```
  +------------+--------------------------------------+^+
  |unencrypted payload header (variable length)       | |
+^+------------+--------------------------------------+ |
| |                                                   | |
| |                                                   | |
| |                                                   | |
| |                                                   | |
| |              Encrypted Frame                      | |
| |                                                   | |
| |                                                   | |
| |                                                   | |
| |                                                   | |
| | ---------+-------------------------+-+---------+----|
| | payload  |IV...(length = IV_LENGTH)|R|IV_LENGTH|KID |
| | ---------+-------------------------+-+---------+----|
|                                                       |
+--+Encrypted Portion        Authenticated Portion+-----+
```

We do not encrypt the first few bytes of the packet that form the VP8 payload (10 bytes for key frames, 3 bytes for interframes) nor the Opus TOC byte.

This serves two purposes:

- Allows the SFU to have access to the required metadata (such as the SVC layer or the keyframe indicator) to properly route the packet, while not having access to the payload nor knowing if it's E2EE or not
- Fools the decoder in the browser into processing the video frame (as the header is correct), resulting in a "pixelated rainbow" pattern of sorts

Key management

Encryption keys are generated automatically and shared with participants in a meeting.

This model dissociates E2EE and authorization, which are conflated in unmanaged mode. Once a participant is part of a meeting they will get every other participant keys.

Each participant will generate 256 bits of random material which will be used to derive a key used for encryption:

$$DerivedKey = HKDF(HMAC, material, salt, keyLen)$$

- HMAC: HMAC-SHA256

- material: randomly generated
- salt: JFrameEncryptionKey
- keyLen: 128

This results in a 128 key used to encrypt media using *AES-GCM*. These keys will be short lived, as we'll see below.

Key distribution

Each participant will share their key with every other participant in the meeting using a secure channel.

In the current implementation, the builtin signalling transport (XMPP) is used to negotiate an E2EE channel using *Olm* [9].

Olm is an implementation of the *Double Ratchet* [10] cryptographic ratchet, created and used by the Matrix project. This implementation has been peer reviewed by NCC Group [11].

When E2EE is enabled every participant will establish an Olm session with every other participant. These sessions provide an E2EE communication channel between every two participants, which cannot be eavesdropped by the signalling servers.

Once the channel is established between two participants they send their keys to each other.

Each participant will use the corresponding participant's key to decrypt their media.

Key rotation

As participants join and leave the meeting keys will be replaced so former participants can no longer decrypt any new media, and any new participants cannot decrypt any previous media.

When a participant leaves the meeting a full key rotation procedure is carried out. This is the same process as when creating the initial key: new random material will be created, key derived and distributed over the Olm channel.

When a new participant joins the meeting each participant will ratchet their key (derive a new key based on the previous one) and share the new key with the new participant.

$$K(i) = HKDF(K(i-1),' JFrameRatchetKey', 32)$$

This requires no signalling since all participants are able to ratchet the keys if they are unable to decrypt a message due to it being encrypted with the new key. Participants will make several (currently 8) ratchets before giving up trying to decrypt data.

Comparisons

Table 1: E2EE communication products.

	Zoom	Signal	Jitsi Meet
Key management	Custom (over TLS)	Double ratchet	Double ratchet
Media encryption	AES-GCM	AES-CTR	AES-GCM
SAS verification	Yes	Yes	No
Web browser support	No	No	Yes
Participant limit	200	8	20
Open Source	No	Yes	Yes
Vulnerability to outsiders	Low	Low	Low
Vulnerability to insiders	Very High	Medium-High	Medium-High
Vulnerability to participants	High	High	High

Table 1 shows a comparison with other communication products providing similar features.

Vulnerabilities

The “vulnerability” lines in the comparisons table try to outline the areas where providers offer strong protections, and those where they do not.

Among the commonalities that almost go without saying is the fact that the content of a meeting is always available to all participants who are in a position to compromise its privacy in a variety of ways (e.g., simply recording it locally and distributing it after the fact). This is why we rate all three providers as highly vulnerable to participants.

Also common is that all three providers, appear do a relatively good job at protecting the meeting content from outsiders that could have compromised the media relay where all meeting content is transiting.

Protecting against insiders is likely tre trickiest of all three vectors as the tools that exist to verify authenticity are provided by the very same insiders (a very common case in today’s world of cloud distribution for software). In order to authenticate a meeting, participants therefore need to be able to verify the tools themselves.

Such verification is entirely impossible for Zoom as the source code is unavailable, which is why we have it marked as very highly vulnerable to insiders.

In the case of Signal and Jitsi, all source code is available to auditors, however actual audits themselves are rarely public and when they are, they are rarely being re-run for every new version of the product.

Even in cases where audits are available or (in some very rare cases) users able to complete them themselves, it is necessary for them to also compare the build they are actually using against the audited source code, which requires what is often referred to as reproducible builds.

Neither Signal nor Jitsi offer reproducible builds for their Desktop, Web or iOS versions.

Both Signal and Jitsi, however, allow reproducible builds to be compared against Android APK binaries, which is why we rate their vulnerability to insiders as Medium-High (i.e., lower than Zoom's but still incomplete and hard to achieve in practice)

It is worth noting that Signal's team have gone an extra mile in this effort and provide a script for running the actual comparison.

Whitepapers

Zoom's E2EE whitepaper can be read *here* [12].

Signal's E2EE whitepaper can be read *here* [10].

Browser support

Presently Zoom and Signal don't support E2EE for video calls in their browser applications. They do provide E2EE in their desktop and mobile applications.

In contrast, Jitsi provides E2EE capabilities in supported browsers (and browser-like desktop applications such as Electron) while native mobile app support is still in progress.

Possible next steps

The current implementation provides Jitsi Meet users with End-to-End Encryption capabilities in an *unverified* manner. All keys are ephemeral and not saved anywhere, thus providing plausible deniability.

Implementing user verification using Olm's builtin *Short Authentication String (SAS)* [13] mechanism is a possible next step.

The IETF has been working in the *Messaging Layer Security* [14] specification, which would simplify the required signalling and avoid the need for a full-mesh of E2EE communication channels. We are monitoring these efforts closely, hoping it could allow us to lift the 20 participants limit.

Contact

We take security very seriously and develop all Jitsi projects to be secure and safe.

If you find (or simply suspect) a security issue in this procedure or any of the Jitsi projects, please report it to us via email to security@jitsi.org.

We encourage responsible disclosure for the sake of our users, so please reach out before posting in a public space.

Appendix I: JFrame

SFrame is not standardized yet and has gone through some changes throughout the currently available 3 draft versions. Our implementation of SFrame is based on the "00" draft, but it puts the metadata as the packet trailer instead of the header, because it makes the code simpler.

Appendix II: Unmanaged Mode

Note: Unmanaged mode is no longer supported and was part of the first E2EE implementation in Jitsi Meet.

In unmanaged mode participants are prompted for a passphrase when activating E2EE (it can alternatively be injected as a URL parameter or passed in via an API call). This passphrase is then used to derive an encryption key using PKDF2:

$$DerivedKey = PKDBF2(PRF, passphrase, salt, iterations, keyLen)$$

with the following values:

- PRF: HMAC-SHA256
- passphrase: user supplied key
- salt: MUC name
- iterations: 100000
- keyLen: 128

This results in a 128 key used to encrypt media using *AES-GCM*.

No mechanism is provided for exchanging keys between users, that has to happen out of band.

Since there is no key distribution mechanism, in this mode the key is shared amongst all participants and used to symmetrically encrypt / decrypt media. Due to the lack of a key distribution mechanism the key usually remains unchanged throughout the duration of the meeting.

References

[1] “WebRTC: Real-time communication between browsers,” 2011. [Online]. Available: https://www.w3.org/TR/2021/REC-webrtc-20210126/. [Accessed: 26-Jan-2021]

[2] “WebRTC encoded transform,” 2019. [Online]. Available: https://w3c.github.io/webrtc-encoded-transform/. [Accessed: 01-Jul-2021]

[3] P. Saint-Andre, “Extensible messaging and presence protocol,” 2004. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6120. [Accessed: 01-Mar-2011]

[4] E. Rescorla, “WebRTC security architecture,” 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8827. [Accessed: 01-Jan-2021]

[5] “Datagram transport layer security (DTLS) extension to establish keys for the secure real-time transport protocol (SRTP),” 2010. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5764. [Accessed: 01-May-2010]

[6] “RTP topologies (selective forwarding middlebox),” 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7667#section-3.7. [Accessed: 01-Nov-2015]

[7] “The messaging layer security (MLS) architecture,” 2020. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-omara-sframe-00. [Accessed: 19-May-2020]

[8] “AES-GCM authenticated encryption in the secure real-time transport protocol (SRTP),” 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7714#section-8.1. [Accessed: 01-Dec-2015]

[9] “Olm: An implementation of the double ratchet cryptographic ratchet in c++.” [Online]. Available: https://matrix.org/docs/projects/other/olm

[10] M. Marlinspike, “The double ratchet algorythm,” 2016. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/. [Accessed: 20-Nov-2016]

[11] “Olm cryptographic review,” 2016. [Online]. Available: https://www.nccgroup.com/globalassets/our-research/us/public-reports/2016/november/ncc_group_olm_cryptogrpahic_review_2016_11_01.pdf. [Accessed: 01-Nov-2016]

[12] “E2E encryption for zoom meetings,” 2020. [Online]. Available: https://raw.githubusercontent.com/zoom/zoom-e2e-whitepaper/master/zoom_e2e.pdf. [Accessed: 15-Dec-2020]

[13] "Short authentication strings (SAS) in ZRTP," 2021. [Online]. Available: https://en.wikipedia.org/wiki/ZRTP#Authentication. [Accessed: 24-Aug-2021]

[14] "The messaging layer security (MLS) architecture," 2018. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-mls-architecture/. [Accessed: 08-Mar-2021]