Practically-exploitable Vulnerabilities in the Jitsi Video Conferencing System

Robertas Maleckas
ETH Zürich
Switzerland
robertas.maleckas@alumni.ethz.ch

Kenneth G. Paterson
ETH Zürich
Switzerland
kenny.paterson@inf.ethz.ch

Martin R. Albrecht
King's College London
UK
martin.albrecht@kcl.ac.uk

ABSTRACT

Jitsi Meet is an open-source video conferencing system, and a popular alternative to proprietary services such as Zoom and Google Meet. The Jitsi project makes strong privacy and security claims in its advertising, but there is no published research into the merits of these claims. Moreover, Jitsi announced end-to-end encryption (E2EE) support in April 2020, and prominently features this in its marketing.

We present an in-depth analysis of the design of Jitsi and its use of cryptography. Based on our analysis, we demonstrate two practical attacks that compromised server components can mount against the E2EE layer: we show how the bridge can break integrity by injecting inauthentic media into E2EE conferences, whilst the signaling server can defeat the encryption entirely. On top of its susceptibility to these attacks, the E2EE feature does not apply to text-based communications. This is not made apparent to users and would be a reasonable expectation given how Jitsi is marketed. Further, we identify critical issues with Jitsi's poll feature, which allow any meeting participant to arbitrarily manipulate voting results. Our findings are backed by proof-of-concept implementations and were verified to be exploitable in practice.

We communicated our findings to Jitsi via a coordinated disclosure process. Jitsi has addressed the vulnerabilities via a mix of technical improvements and documentation changes.

1 INTRODUCTION

Jitsi Meet [6] is a free and open source video conferencing platform marketed under strong privacy and security claims. Contrary to its arguably more mainstream competitor offerings, Jitsi emphasises privacy, openness, and flexibility over advanced features or partner integrations. Meanwhile, the increasingly common reports [25] of state-sponsored digital surveillance of high-profile public figures in recent years have likely contributed to a growing demand for better eavesdropping resistance and untraceability in communication software. By explicitly positioning itself as such, Jitsi has gained the attention of various NGOs [16] and civil society groups working to educate high-risk communities on tools and practices of secure communication, particularly as an alternative to proprietary software and telecommunication services which are often under complete control of their respective operators [39]. Over time, it has become a popular choice of video conferencing appliccation among higher risk users such as political activists and whistleblowers, and has been used by Edward Snowden [47], endorsed by the Tor Project [42], Mozilla [15], and others.

As video conferencing software rose to unprecedented levels of popularity throughout 2020 and into 2021, so did the public scrutiny of its security. In the process, some organisations have since banned such third-party software from internal use, citing privacy and security concerns. Partially in response, several major videoconferencing vendors adopted some form of end-to-end encryption (E2EE) [18, 48], with Jitsi announcing [4] work on this feature in April 2021.

Jitsi's users can choose to host their meetings on public servers (official and community run), or run their own private instances. This makes adoption tricky to measure; however, Jitsi powers the video calls in Element [30] (the Matrix flagship client – a communication platform with 60M+ users [11]), and according to Jitsi's website [6], is used internally by a number of sizeable organisations, including governments [20]. Interestingly, despite its growing popularity and open nature, no publicly available independent security audit of the platform is available beyond one mention on its community forum about one having been done internally [2]. Given this state of affairs, we conducted our own analysis of Jitsi Meet, with a focus on its cryptographic components.

1.1 Contributions

After some preliminaries in Section 2, in Section 3 we present an analysis of Jitsi's custom E2EE design. This includes discussions on the threat model, key management procedures, implementation details, wire format, and how the feature fits into the rest of the system. We describe the cryptographic primitives used along with their potential risks and choices of implementation.

Then, in Section 4 we present our findings including several shortcomings of the E2EE implementation and overall system security. We demonstrate how a rogue service provider can mount practical attacks against the E2EE layer, with varying impact dependent on the threat model used. Furthermore, we discuss the security claims made on Jitsi's official website with regard to the implementation, identifying discrepancies between perceived and practical security guarantees. Lastly, we show how an insecure protocol design can be exploited to manipulate votes and re-route P2P client traffic over the service provider's infrastructure.

1.2 Disclosure

We disclosed the vulnerabilities to Jitsi's developers on 9th August 2022. They acknowledged four of the six reported attacks — Mallory-in-the-middle (4.1), compromised media integrity (4.2), vote manipulation (4.4), and E2EE announcement forging (4.5) — as-is, noting the voting protocol issue was a duplicate of another independent report. They considered the remaining two — a mismatch of marketing claims and guarantees of the E2EE feature (4.3) and the risk of silent denial of service attacks on the P2P mode (4.6) — as documentation rather than design or implementation issues.

The developers asked for an extension to the typical 90-day disclosure deadline citing engineering capacity constraints and overhead in the release process, and we agreed on a 120-day timeframe.

Jitsi began rolling out security patches to their public repositories in mid-September, following up with security advisories after the changes made it to the stable release.

The developers implemented and released mitigations to most of the reported issues and issued security advisories within the 120-day disclosure deadline. They committed to improve their documentation to more accurately reflect the guarantees and pitfalls of their E2EE implementation. To date, no fix for the lack of E2EE in chats is available, but the documentation has been updated to specifically note this.

We note all of our references to Jitsi's website predate the changes made in response to our work. Furthermore, we will use the present tense throughout, despite the described vulnerabilities having been fixed now.

1.3 Related Work

There has been surprisingly little technical security analysis of video conferencing systems. This may be due to their generally closed-source nature. Berson [14] gave an early analysis of Skype. White et al. [46] showed how to reconstruct approximate transcripts of encrypted VoIP conversations using traffic analysis. A recent study [22] provides an analysis of Zoom's E2EE feature, exposing impersonation attacks against the system.[1] Jitsi is declared in a Mozilla blogpost [15] as providing "strong privacy protections". Our findings are not consistent with this view. Kagan et al. [26] collected publicly available images of video conference meetings and extracted personal information about the participants, violating privacy expectations.

Hasan and Hasan [21] used STRIDE to develop a general threat model for video conferencing systems. Reisinger et al. [35] survey security and privacy in Unified Communication (UC), a catch-all term for communications systems providing video conferencing, messaging and other services as Jitsi does. Their analysis includes an overview of Jitsi but not an in-depth security analysis. Cohney et al. [17] also covered Jitsi as part of a broader study of the potential harms of virtual classrooms in U.S. universities. Again, their study, being broader than ours, does not provide the same depth of analysis.

2 PRELIMINARIES

2.1 Olm

Olm [29] is an open-source implementation of the Double Ratchet Protocol [34] to exchange encrypted messages between two parties sharing a secret key. It combines a symmetric Key Derivation Function (KDF) and a Diffie-Hellman based asymmetric ratchet to provide certain forward secrecy (FS) and post-compromise security (PCS) protections. The symmetric ratchet ensures that past keys cannot be derived from current key material (FS), while the DH ratchet limits the ability of a single compromised key to decrypt future sessions (PCS).

[1] See also https://citizenlab.ca/2020/04/move-fast-roll-your-own-crypto-a-quick-look-at-the-confidentiality-of-zoom-meetings/ for a less formal study of Zoom.

Similar to Signal's (Extended) Triple Diffie-Hellman (X3DH) [40], Olm uses the Triple Diffie-Hellman (3DH) key agreement to bootstrap the Double Ratchet Algorithm. In short, 3DH is an extension of the "classic" Diffie-Hellman exchange which allows two parties to establish a shared secret over a public channel in a way that's secure against passive eavesdroppers. Compared to a regular DH exchange, 3DH additionally provides a cryptographic binding of the shared secret to both parties' respective long-term identity keys.

We use I_A and E_B for identity and ephemeral Diffie-Hellman keys of Alice and Bob, respectively. The same symbol implicitly refers to the private key when used by its owner, or the public key when transmitted to or used by the other party. We use multiplicative notation for group operations, e.g. g^x. With $R_i, C_{i,j}, T_i$, and $M_{i,j}$ we refer to root, chain, ratchet, and message DH keys as defined in the Olm documentation [29], although here the roles A and B are reversed. The KDF implementation specifics are omitted for brevity.

2.2 System architecture

At its core, Jitsi is based on a client-server architecture in a *hub-and-spoke* configuration. The complexity involved in addressing practical issues (e.g. availability and scale) is typically hidden from the end user, who instead sees a single entry point URL.

Behind the facade of a single logical "server", the system functionality is in fact divided across multiple modular components. A bare-minimum setup involves four distinct pieces of server software, and a compatible client. In practice, more advanced deployments may add further components and replicas thereof; however, the rest of this section focuses on a minimal deployment.

2.3 Minimalist setup

The self-hosting guide [10] provides a minimal single-machine setup suitable for personal use or a small group of users. The four core server components include: (1) A standard web server (*nginx* by default, others supported); (2) An XMPP[2] server (*Prosody*, limited support for alternatives); (3) Jitsi Video Bridge (JVB); (4) Jitsi Conference Focus (JICOFO).

Jitsi's modular design allows administrators to customise and even replace certain system components. However, in practice the project is primarily developed using a "preferred" deployment model, and significant changes to it may cause issues with compatibility. We too therefore use the default choice of core components — *Prosody* and *nginx*.

2.4 Methodology

Our analysis targets the stable release 2.0.6865-2 of Jitsi Meet [6], released on Jan. 28th, 2022.

2.4.1 Static code analysis. Our analysis began with a review of a technical whitepaper [18] and openly available source code [1] published in Jitsi's original blog post [4] demonstrating the E2EE feature. While the core encryption primitives were mostly contained within a single module, they referenced many external Jitsi libraries whose security properties were not immediately obvious. Similarly, the whitepaper omitted substantial background details

[2] https://xmpp.org/

on the message sequences and communication channels underlying the proposed E2EE protocols. As a result, any comprehensive attempt at assessing its security necessitated a further deep dive into the broader surrounding system including Jitsi's application and library source code, installation scripts, configuration files, and technical documentation.

2.4.2 *Dynamic debugging.*

After the "offline" analysis, we followed the official self-hosting guide [10] to set up a private Jitsi instance. This was done on a virtual machine in the cloud, provisioned with 2 CPU cores and 4GiB of memory, running Debian 10 "Buster". We simulated common usage scenarios and configurations using multiple Jitsi client application instances, and readily available server configuration options. The system behavior was observed on the server and client components via diagnostic logs and the web browser's built-in Javascript debugger, respectively.

2.4.3 *Specification review.*

Many of Jitsi's features lean heavily on standard protocols and off-the-shelf software implementations thereof. As such, we studied the relevant protocol specifications, Internet standards, and software library documentation to verify the security properties of the system as a whole.

2.4.4 *Purpose-built exploits.*

Most of our findings were verified to be exploitable in practice by developing proof-of-concept exploits against the private Jitsi instance. Depending on the threat model being considered, this involved malicious code or configuration changes to both client- and server-side Jitsi components, as well as building a custom WebRTC / Jitsi client application from the ground up.

2.5 Software components

The following sections discuss each component's role in the overall system.

2.5.1 *Client software.*

While Jitsi offers client software for multiple platforms, we mainly focus on the JavaScript-based client as the first one to support E2EE. Its core functionality is available in `lib-jitsi-meet` [9] — a low-level library which handles server connectivity, WebRTC sessions, and all relevant conference events. The Jitsi Meet web client [6] serves as a "reference implementation" providing a user interface and additional features based on the core library.

2.5.2 *Web server.*

The web server is the primary public-facing component which acts as the entry point into Jitsi conferences. It has two key responsibilities: answering regular HTTP requests, and forwarding XMPP traffic to and from the XMPP server. The web server hosts the Jitsi Meet web application files, allowing the user's browser to obtain and run it directly without prior setup. XMPP messages are exchanged over HTTP or WebSockets, where the web server functions as a *reverse proxy* in front of the XMPP server. In both cases, all traffic between the client and the web server is tunneled over TLS.

2.5.3 *XMPP SIGNALING server.*

Jitsi uses the XMPP protocol and various extensions thereof for virtually all non-media communications and signalling. The XMPP server is responsible for routing such traffic between clients, and hosting server-side business logic underlying most conference features. The following sections describe some of its primary functions beyond the core XMPP primitives.

Multi User Chat (MUC). The MUC extension forms the core of every Jitsi conference, with its concepts mapping to and used directly by the participants. At their most basic, MUC rooms provide a convenient medium to manage user access, exchange text and signalling messages, and share status updates. Jitsi uses ephemeral MUC rooms, whose lifetime is managed by the conference focus, described in Section 2.5.4.

Addressing scheme. Jitsi uses two logical "address spaces" in its XMPP communications. Upon the first connection, such as opening the landing page in a browser, the XMPP assigns a "temporary, unique bare JID [Jabber ID] `<localpart@domain.tld>` to the client." [38, §3] These JID fall under the domain of the Jitsi instance such as `meet.jit.si`, and can be considered "global" (within a single XMPP server). This address space is used in the initial messages to the focus before an XMPP room is created.

Whilst joining a MUC room, participants obtain *occupant JID* of the form `<room@service/nick>`, which identify them within the context of the room [37, §4.1]. The "service" part typically takes the value of a subdomain, such as `conference.meet.jit.si`. The room-scoped address space is used for most of the conference functionality including Jingle negotiation, text chat, polls, and more.

Presence updates. Participants use XMPP presence to communicate status updates to the MUC room of the conference. This primitive is used in protocols such as joining a MUC room, and to communicate user attributes such as their name and room role [37, §5.1] to others. Jitsi further extends the presence updates for its "raise hand" functionality, feature support flags (e.g. screen sharing and E2EE), and various state synchronization such as the E2EE toggle (Section 3.3).

Clients send the presence messages to the XMPP server, which distributes them to the other conference participants. In the context of MUC rooms, users typically address their presence to their current (or desired) *occupant JID*. The server uses this value in the `from` attribute of messages it propagates to others. As a result, participants do not learn each others' *full JID* [37, §4.1], and cannot communicate directly outside of the MUC room.

2.5.4 *JICOFO.*

The conference focus acts as the MUC Room Owner [37, §4.1], creating rooms upon user request, and destroying them after all participants have left. Additionally, it manages client-server media sessions by participating in Jingle negotiations, and controlling JVB instances.

Authentication. The conference focus is considered to have administrative privileges over the media sessions, and therefore requires authentication to log into the XMPP server. This is done using the SCRAM-SHA1 protocol [32] based on a shared password set in the server configuration. At a high level the authentication process can be summarised as follows:

(1) The verifier (XMPP server) generates a challenge consisting of a random salt s and iteration count i and sends it to the prover (focus).

(2) The prover computes i iterations of an HKDF using the SHA-1 hash function, challenge salt and stored password to compute a proof, and sends it back to the verifier.
(3) The verifier mirrors the computation and compares the result to the' provers response.

The full protocol includes additional steps and details which are omitted from the above summary for brevity.

Addressing. The focus owns special JID in both the "global" and room-scoped address spaces, described in Section 2.5.3. The former is used for room allocation requests, whereas the latter appears in Jingle negotiations after participants have already joined. The different addresses can be thought of as alternative interfaces, ultimately serviced by the same component.

Media session management. As users join, the focus initiates Jingle media session negotiations with each one. During this exchange, it uses the Colibri (Conferences with Lightweight Bridging) protocol to allocate JVB channels, and incorporates their transport and authentication details into its own Jingle messages. Since media routing requires significantly more bandwidth, offloading it to the JVB allows the Jicofo to scale to more conferences before exhausting its computational and network capacity.

2.5.5 JVB. The JVB is Jitsi's software implementation of a Selective Forwarding Unit (SFU). Also known as a Selective Forwarding Middlebox (SFM), an SFU is an infrastructure component which dynamically multiplexes the participants' media streams [45]. This ensures the users only have to send their streams to the server just once, therefore eliminating a scalability bottleneck. The JVB is designed to be a modular, WebRTC-compatible SFU, and is therefore not coupled to many Jitsi-specific concepts. Instead, it provides basic control interfaces, and operates under full control of the JVB.

Control interfaces. The primary functions of the bridge such as conference creation and endpoint allocation are controlled through Colibri protocol messages [23]. For interoperability with non-XMPP systems, the JVB implements an HTTP REST control interface with a message structure serialised to JSON.

Within Jitsi Meet, JVB instances are controlled via the "native" XMPP-based Colibri interface. Upon startup, bridges connect to a special preconfigured, authenticated MUC room. The focus uses the room to track available bridges and load-balance conferences across them. The Colibri protocol runs over MUC private messages between the bridge and the JVB, similarly to the Jingle exchanges used to negotiate media with conference participants.

2.6 Signaling protocol stack

Jitsi's architecture uses two major protocol stacks: WebRTC for media sessions, and XMPP for everything else. The XMPP traffic includes text chat, E2EE Olm channels, media negotiations, and various other features (polls, meeting lobbies, etc.)

3 JITI'S SECURITY ARCHITECTURE

We present an in-depth look at Jitsi's custom E2EE. Contrary to transport security measures such as TLS, E2EE aims to extend the security guarantees to guard against malicious servers, the logical adversary considered in such settings.

3.1 Threat model

In Jitsi's E2EE threat model, potential actors behind a malicious server are further subdivided into insiders and outsiders. This distinction is made based on the legitimacy of the parties' access to the server. Insiders are defined as "parties involved in the maintenance [. . .]", while outsiders are those "[. . .] who gained illegitimate access to a component [. . .], for example a Jitsi Videobridge" [18, p. 3].

The authors assert that preventing insider attacks is an "extremely difficult aspiration" and that the feature focuses on defending against outsider attacks. The difficulty is attributed to the practical challenges associated with auditing frequently updated cloud software. For instance, an inside attacker could distribute malicious updates to obtain unauthorised access to meeting contents.

While this level of access may also be attainable by a sufficiently powerful outside attacker, the threat model appears to primarily focus on defending against a compromised JVB. Its access to the audio and video content is cited as "one of the main reasons to implement end-to-end encryption" [18, p. 5], stating that "outsiders [. . .] could have compromised the media relay where all meeting content is transiting." [18, p. 8] The latter claim is not entirely accurate, since textual contents such as chat messages, polls, and other metadata are routed over the XMPP server rather than the JVB.

Additionally, the Olm sessions are said to "provide an end-to-end encryption communication channel between every two participants, which cannot be eavesdropped by the signaling servers." [18, p. 7] Despite a lack of more explicit claims, this seems to signal intent for the E2EE design to defend against compromised XMPP servers. In small deployments such as the default self-hosted installation [10], the XMPP server may even run on the same machine as the JVB. It is therefore not inconceivable to consider it equally prone to illegitimate outsider access.

The threat model also includes malicious participants, acknowledging that unencrypted meeting content is available to all attendees by design. In fact, user authorisation to join meetings is explicitly listed as a non-goal of E2EE [18, p. 3].

Our analysis considers the effectiveness of Jitsi's E2EE implementation in securing meetings against malicious server infrastructure. More precisely, we consider the following major software components as potential threat vectors: web server, XMPP Signaling server, JICOFO and JVB.

We do not necessarily distinguish insider and outsider threats, but instead focus on the attacks each compromised component may enable on the E2EE security. Authenticity of client software is assumed to stay more consistent with the outsider attack model of illegitimate access to server components. As discussed in more detail in Section 4.1.2, the attacker's ability to compromise client software is not always a given, and would invalidate the system's security as a whole.

3.2 E2EE key management

Jitsi's E2EE implementation makes use of several cryptographic keys, both explicitly and within third-party components. In the following sections, we use the term *media key* to refer to the symmetric keys conference participants use to secure media traffic.

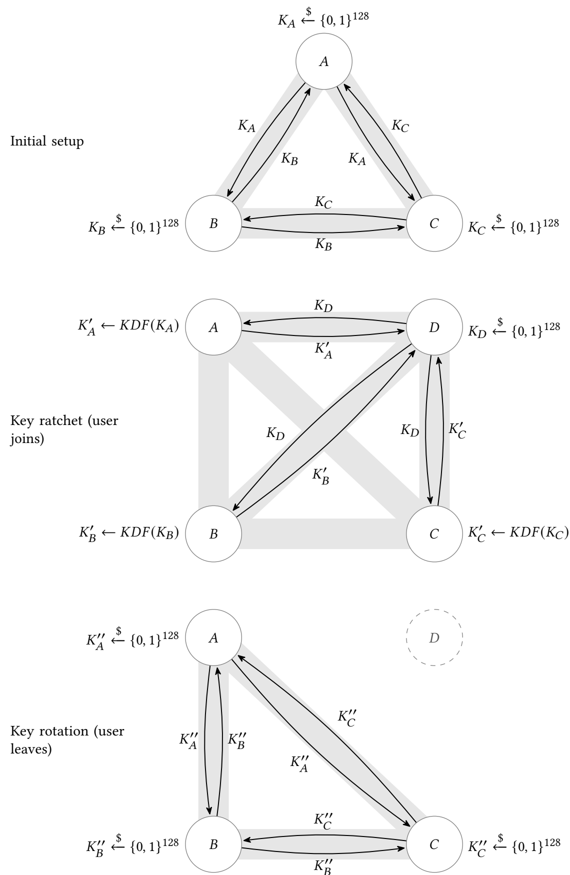


Figure 1: Distribution of media keys within a Jitsi conference. Grey backgrounds depict the pairwise E2EE Olm channels used to distribute media keys. K_A denotes Alice's media key.

3.2.1 Overview. Every E2EE conference participant has a symmetric media key to encrypt their outgoing media streams. These keys are randomly generated by the Jitsi client software and distributed to all other participants over E2EE-secured Olm messaging channels. Each participant maintains a set of their peers' media keys to decrypt incoming media streams. The keys are ratcheted forward when new participants join and re-generated when current participants leave to provide a certain level of FS and PCS. As a result of the re-keying procedures performed with every change to the participant list, the keys are relatively short-lived in practice, and even with a fixed set of participants are deleted once the conference ends. Figure 1 shows the key distribution and updates.

The 3DH key agreement protocol used in the Olm library involves long-term identity keys. In Jitsi, these are generated upon joining a conference and scoped to its lifetime. We note that even though the public parts of participants' identity keys are published to the MUC room through XMPP presence, this is not used in the Olm session establishment protocol.

Meeting participants run a JSON-based protocol over private MUC messages to establish pairwise Olm sessions and share media keys among all supported clients. This is done upon any of the following conference events: (1) E2EE is switched on; (2) A new participant joins while E2EE is on; (3) A current participant leaves while E2EE is on.

(1-2) trigger a full protocol run, establishing Olm sessions and exchanging media keys. On the other hand, (3) only causes a rotation of media keys between the remaining participants over already existing Olm sessions. This provides PCS as previous participants are unable to decrypt future traffic encrypted using fresh keys.

In all cases, participants decide their roles as the initiator and responder by comparing their MUC occupant JID. Before running the protocol, initiating participants check if the responder supports E2EE based on a special flag published through XMPP presence. Other clients are simply skipped.

3.2.2 Olm session establishment. The session establishment part of the protocol consists of a simple request-response exchange, illustrated in Fig. 4a. In a successful run, the initiator receives the responder's media key, and the parties agree on a shared secret to exchange further messages encrypted using the Double Ratchet protocol [34]. We note the inclusion of the responder's E2EE key is redundant, as it is shared again in the media key exchange part of the protocol that always follows session establishment.

Protocol steps. As a prerequisite, both parties start out with long-term identity keypairs I_A and I_B. The two parties then engage in the following protocol:

(1) Alice generates a UUID and an ephemeral DH key pair E_A.
(2) Alice sends her public identity and ephemeral keys together with the UUID and a "session-init" tag to Bob.
(3) Upon receiving a "session-init" message, Bob generates an ephemeral DH key pair E_B. He then combines Alice's public keys I_A and E_A with his private keys I_B and E_B in a 3DH key exchange to derive a shared secret S_{AB}. Bob generates a ratchet key T_0, uses S_{AB} to derive the initial root, chain, and message keys.
(4) Bob encrypts his media key using the message key $M_{0,0}$, and sends it to Alice in a "session-ack" message. It also includes Bob's public identity, ephemeral, and ratchet keys, as well as the UUID and Alice's public ephemeral key received in Item 2.
(5) Upon receiving a "session-ack" message, Alice verifies that the UUID matches the value generated in step 1. Like Bob, she uses 3DH to derive the shared secret S_{AB}, and compute the message key $M_{0,0}$. Alice then uses it to decrypt the ciphertext and get Bob's media key.

3.2.3 Media key exchange. Similar to the session establishment protocol, the media keys are shared in a pair of request-response messages. The keys are encrypted using a previously established Olm channel. Figure 4b depicts the protocol run immediately following a session establishment; further reruns differ only in the state of the underlying KDF chains, such as chain indices.

Here, the term *key exchange* refers to the back-and-forth sharing of session keys which pre-exist outside of this protocol, rather than the more traditional sense of establishing fresh keys during the run of the protocol.

Protocol steps. As a prerequisite, Alice and Bob share a root key R_0, Bob's sending chain key $C_{0,1}$, and a DH ratchet key T_0 owned by Bob, with its public part known to Alice.

(1) Alice generates a UUID and a new DH ratchet key pair T_1. She uses the previous root key R_0 and a DH exchange between ratchet keys T_0 and T_1 to derive new root and sending-chain keys R_1 and $C_{1,0}$. A message key $M_{1,0}$ is derived from $C_{1,0}$.
(2) Alice encrypts her media key using the message key $M_{1,0}$ and sends it to Bob along with the UUID, her public ratchet key T_1, and the tag “key-info”.
(3) Upon the receipt of a “key-info” message, Bob combines the public ratchet key T_1 with his private ratchet key T_0 and the previous root key R_0 to derive a new root key R_1 and receiving-chain key $C_{1,0}$. He uses $C_{1,0}$ to derive the message key $M_{1,0}$ and decrypt Alice's media key. Bob then generates a fresh DH ratchet key T_2, and repeats the procedure using R_1, T_1 and T_2 to derive a new root key R_2 and sending-chain key $C_{2,0}$. He then derives a message key M_2.
(4) Bob sends his media key encrypted under M_2 and his public DH ratchet key T_2 to Alice, together with the UUID received in step 2 and the tag “key-info-ack”.
(5) Having received the “key-info-ack” response, Alice checks the UUID against the value generated earlier. She then repeats the procedure to initialize a new receiving chain using her private DH key T_1, Bob's public ratchet key T_2, and the previous root key R_1. Alice can then compute the message key $M_{2,0}$ and decrypt the message to get Bob's media key.

3.2.4 Group key distribution. The Olm session establishment and the Media key exchange protocols form the basis for a two-party media key exchange. In larger conferences, the protocols are run between every pair of participants to distribute the media keys to the entire group. This is done asynchronously and in parallel, based on a simple algorithm outlined in Listing 1.

```
1  for each peer in conference do
2    if peer supports E2EE and has lexicographically greater
         ID then
3      send Olm session initiation to peer
4  for each pending session do
5    wait for response or timeout
6
7  for each peer in conference do
8    if Olm session with peer exists then
9      send E2EE key to peer using Olm
10 for each pending key exchange do
11   wait for response or timeout
```

Listing 1: Media key exchange algorithm outline (pseudocode).

Participant limit. In a group of N participants, Jitsi's media key distribution approach involves $\frac{N\cdot(N-1)}{2}$ pairwise Olm channels. Consequently, the protocol overhead incurred on the XMPP server grows quadratically with the number of participants. To avoid saturating the signaling bandwidth, Jitsi imposes an artificial hard limit of 20 participants in E2EE-enabled conferences. If the limit is exceeded, the feature is automatically switched off and can not be re-enabled, displaying a warning instead.

Error handling. We note that the protocol implementation includes fairly robust error handling to avoid exceptions. It, however, does not contain retry mechanisms in case of failure. If this happens, the conference will continue, although the affected parties will not have access to each others' media keys and therefore the E2EE-secured media streams. For instance, this may happen when either party uses an unsupported client, or the protocol messages are dropped in-transit. If protocol messages are received unexpectedly, such as a “key-info” without a preexisting session, an error is logged to the browser console and the current protocol run is aborted. Olm sessions are scoped to every pair of participants; failures do not affect the parties' protocol runs with others.

3.2.5 Key generation.

Olm identity and ephemeral keys. Identity and ephemeral keys involved in the Olm session establishment and media key exchanges are generated internally within the library. Under the hood, it uses a pseudorandom number generator (CSPRNG) outputs of either the browser API (Web Crypto [44]) or close equivalents within *Node.js*.

Media keys. Jitsi's media frame encryption involves a master secret initialised using a CSPRNG, and an hash-based KDF (HKDF)-derived key used for encryption. Participants keep both the derived key and the master secret. The key is used directly to encrypt and decrypt media, while the secret is what gets ratcheted before repeating step to obtain a new key (Section 3.2.6).

3.2.6 Re-keying. As described in Section 3.2.1, a key rotation involves multiple protocol runs over the network, and is therefore not instantaneous. It may also routinely happen throughout the duration of a conference as participants leave.

Jitsi maintains uninterrupted operation through the use of keyrings to keep multiple, numbered keys for every participant. This allows them to decrypt traffic under either the current or updated key amidst a rotation or ratchet procedure.

Rotation. When a participant leaves a meeting, the XMPP server broadcasts their presence update to all other conference participants. This event triggers every client to perform a key rotation, generating fresh keys and running the “key-info” protocol described in Section 3.2.3 to distribute them to others.

Ratcheting. Similar to participant departure, the entry of a new user is communicated using the XMPP presence mechanism. This causes the remaining participants to ratchet their media keys forward. Unlike key rotation, ratcheting is performed locally without any out-of-band (wrt. media streams) coordination. Instead, clients simply begin using the ratcheted keys, which the decryption code detects and self-adjusts to. This is discussed further in Section 3.7.1.

Similar to the media key derivation, the key ratcheting uses an HKDF to first ratchet the master secret forward, and then derive a new media key.

3.3 State management

The E2EE feature state is controlled on the client side, and synchronised to conference participants through XMPP presence updates. Clients use the same mechanism to advertise E2EE support and public identity keys for Olm session establishment (although they are sent again in protocol runs).

3.3.1 Controlling end-to-end encryption state. Conference participants can toggle the E2EE feature on and off using a switch in the

security options menu. It is off by default, and only visible to participants using supported clients. If any of the conference participants do not support the feature, a small warning is shown, explaining that some clients may not be able to see or hear the conference.

Despite the E2EE switch being hidden on unsupported clients, they can still control the feature through an API accessible via the browser console. In other words, the client checks its E2EE compatibility while rendering the options menu, but not in the underlying state handling.

3.3.2 Audiovisual cues. Upon joining an E2EE-enabled conference or toggling the feature on, an automated voice message ("end to end encryption is on") is played on all clients. Toggling the feature off triggers a similar announcement. Interestingly, the messages are played even if E2EE is not supported by the client, such as when joining an E2EE-enabled conference using Firefox. This likely relates to insufficient safeguards around the state handling logic discussed above.

In addition to the audible cues, the E2EE state is also displayed at the top of the conference view in the form of a green padlock icon. The icon is absent by default, or when some of the participants do not support E2EE. We observed some inconsistent behaviour of the visual indicator on unsupported clients, which would occasionally be shown despite the unavailability of E2EE in said clients.

3.3.3 Conference events. Certain parts of Jitsi's E2EE implementation like re-keying (Section 3.2.6) and state keeping (Section 3.3.1) rely on standardised XMPP behaviours. Most notably, presence broadcasts in the conference's MUC room are used as triggers for re-keying, and the E2EE feature itself. They are also used to track the list of participants, and switch between P2P and JVB modes accordingly.

3.4 Implementation

E2EE in Jitsi is applied to encoded media frames using the WebRTC Encoded Transform [12] draft API, formerly known as WebRTC Insertable Streams. We present a brief overview of the data flow involved in sending E2EE-secured video below:

(1) A raw media frame (an audio sample or a video buffer) is first encoded using a lossy codec such as Opus [43] or VP8 [13].
(2) The encoded media frames and source metadata are passed as plaintext inputs to the E2EE function.
(3) A stateful E2EE function encrypts the frame under the currently set media key.
(4) The encrypted outputs are split into RTP packets and sent over the network to another user (P2P mode) or the JVB.

Step 3 is performed by Jitsi's application code, while the other steps are handled by the implementation of WebRTC — commonly, the web browser. The decryption procedure is essentially an inverse of the above sequence, performed by the receiver. Figure 2 depicts the flow of a single media stream either sent directly (P2P in one-on-one conferences) or over the bridge (in all other cases).

3.5 Encryption

Jitsi implements a variant of *SFrame* — an Internet Draft specification which attempts to standardise media frame-level E2EE in conferences [33]. The modified version, referred to as *JFrame*, is

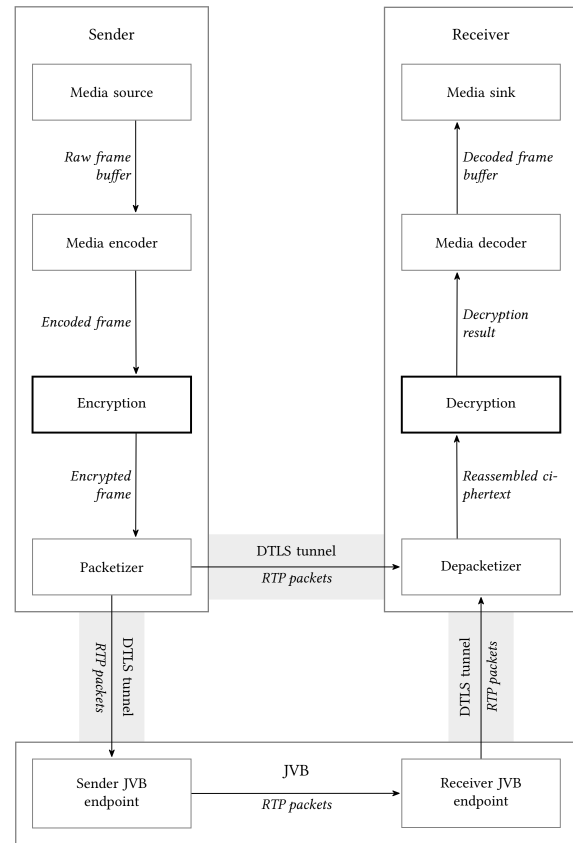


Figure 2: Logical data flow of a single media stream within Jitsi routed directly (P2P) or via the JVB. E2EE is implemented as a pair of Encoded Transforms marked by a darker outline in the diagram. All other components shown within the sender and receiver are not part of the application code.

based on the "00" draft, and primarily differs from it by placing encryption metadata in the trailer rather than the header. The frame format is described in Section 3.6.

The E2EE encryption layer uses AES-GCM with 128-bit media keys [18, p. 5]. The code relies on the AES-GCM cryptographic primitive defined in the *Web Crypto API* specification [44], implemented by the browser. The default authentication tag length of 128 bits is used.

Jitsi uses a 96-bit IV, stating that "AES-GCM *needs* a 96 bit initialization vector [. . .]" [18, p. 5]. We note the cipher supports IV 1 to 2^{64} bits long [19, §5.2.1.1], and Jitsi possibly refers to the recommended length which allows for simpler initialisation of the pre-counter block [19, §7.1].

The IVs are constructed by concatenating three fixed-length values from the frame metadata and the encryption function state:

(1) The SSRC of the media stream (32 bits);
(2) The RTP timestamp at which the frame was sampled (32 bits);
(3) An SSRC-scoped counter of outgoing encrypted frames (32 bits).

This construction avoids IV collisions with very high probability. If a collision were to occur, an eavesdropping attacker would

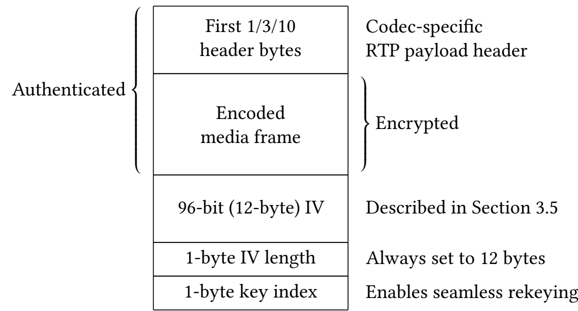


Figure 3: Jitsi E2EE media frame format (*"JFrame"*).

gain access to the XOR of the two plaintext frames. With some knowledge of VP8 frame structure, an attacker might (partially) recover the underlying plaintexts. An attacker could also recover the authentication key and forge ciphertexts [24]. However, given the relatively short-lived nature of video conferences the impact can be considered minimal.

3.6 Frame format

An E2EE frame contains a variable-length header (additional authenticated data), the ciphertext of the encoded frame (Section 3.5), the IV (Section 3.5) with its length, and the key index (Section 3.2.6). The frame layout is depicted in Fig. 3, and forms the RTP payload before being passed on to the packetiser (Section 3.4).

3.6.1 Unencrypted header. The first several bytes (codec-dependent) of the header are authenticated (but not encrypted) as associated data under the AEAD scheme. The primary purpose of this appears to be functional: allowing the video bridge to "[...] continue detecting keyframes [...]". However, this could be signaled in a single bit, and the "[inclusion of additional bytes] is a bit for show [...]" as "it generates funny garbage pictures instead of being unable to decode". In other words, the unencrypted header is sufficient to trick the media decoder into accepting an encrypted frame as valid and attempting to decode it. Section 3.7.2 discusses this behavior in more detail.

The plaintext header length is determined independently for each frame and depends on the codec as well as the frame type (key frame or *delta* frame). Opus (audio) headers contain a single table-of-contents (TOC) byte, while video (VP8) headers carry 3- or 10-byte frame tags with metadata. The following paragraphs list the unencrypted contents in each case.

3.6.2 Encrypted frame contents. With the first header bytes designated as additional authenticated data, the remainder of the frame is encrypted using AES-GCM. The output of the Web Crypto API is a concatenation of the ciphertext and the authentication tag [44, §27.4] and is simply appended to the authenticated header.

3.6.3 IV length. Despite the fixed method of IV construction presented above, the IV length is encoded in the frames. Its purpose is not quite clear, as both Jitsi's encryption code and the *SFrame* draft use fixed-length IVs. This may be a left-over from a prior iteration of the design.

3.6.4 Key index. The index of the key used to encrypt the frame is encoded as an 8-bit number in the frame trailer. It is analogous to the key ID field used in *SFrame*, which serves the same purpose and only differs in its encoding [33, §4.2].

3.7 Decryption

Upon receiving an encrypted media frame, the decryption function uses the key index contained in the last byte of the frame trailer to retrieve a decryption key from the keyring. In case of failure, the encrypted frame is returned as output. Otherwise, the function parses the supposedly variable-length IV, authenticated header and ciphertext, and passes them along with the key to the Web Crypto API for authenticated decryption [44, §27.4].

3.7.1 Self-ratcheting. If the frame parsing, authentication or decryption fail for any reason, the code keeps ratcheting the key forward and retrying for up to eight attempts. In case of a successful decryption, the respective keyring entry is updated with the ratcheted value. If failed attempts exceed the maximum allowed ratcheting window, the original key is restored, and the encrypted frame is returned as output.

A nearly identical self-ratcheting mechanism is present in the *SFrame* draft, and is described as a more efficient alternative to always re-generating keys [33, §4.3.5.1]. In contrast to performing a multiparty key exchange over the network, the ratcheting is a local operation. The *JFrame* implementation contains a subtle difference to *SFrame*, ratcheting even in case of authentication or parsing failures. The documentation acknowledges this, albeit does not provide a reason. It may simply help code reuse by sharing a common error handling path; such a philosophy would be consistent with the design reusing E2EE contexts for both encryption and decryption logic in the code.

3.7.2 Decryption output. With the exception of ratcheting, decryption failures generally result in the encrypted frame being passed on to the decoder as-is. As presented in Section 3.6.1, the unencrypted header bytes are enough to pass the decoder's validation. The code comments acknowledge this behaviour: "this just passes through to the decoder. Is that ok?" While it is mentioned "[the developers] might want to reduce [the header length] to 1 unconditionally in the final version", the "feature" has remained as-is for over two years. This follows a similar pattern to the encryption implementation, opting to carry on instead of triggering errors when unexpected scenarios occur.

4 VULNERABILITIES

We have identified several design, implementation, and presentation issues which undermine the effective security of certain features. Despite strong security claims made on their official website, we have found Jitsi's E2EE implementation to be greatly limited in scope and only target an extremely narrow threat model. Even against the explicitly stated adversary, the E2EE feature fails to provide media integrity, while extending the attacker capabilities to a more typical threat model renders it useless. Furthermore, improper E2EE state handling may be abused to instill a false sense of security in unsupported clients. Lastly, the poll design is fully

exploitable by anyone in the conference. The following sections present these findings in more detail.

4.1 Unauthenticated key exchange

We report a vulnerability in Jitsi's media key management which allows a malicious XMPP server to defeat E2EE without alerting the conference participants. It stems from the fact the public Olm identity keys are not authenticated neither before nor after the Olm session establishment.

The X3DH specification states: "If authentication is not performed, the parties receive no cryptographic guarantee as to who they are communicating with." [28, §4.1] The same applies to the closely-related 3DH handshake used in Olm. As a result, the XMPP server can perform a classic man-in-the-middle (MITM) attack on the 3DH key exchange, and intercept the participants' media keys.

4.1.1 Overview. The attack starts out with a regular run of the Olm session establishment protocol as described in Section 3.2.2. The messages between Alice and Bob are relayed by the XMPP server, and thus are under the control of the adversary (Eve). Acting similarly to an active network adversary, Eve intercepts Alice's initial message and responds with her own public identity and ephemeral keys, posing as Bob. She modifies Alice's message, replacing the public keys with her own before forwarding it on to Bob, posing as Alice. Eve completes the 3DH key exchanges with Alice and Bob, establishing shared secrets S_{AE} and S_{BE}, respectively. This allows her to derive the root, chain, and message keys used in the media key exchange.

Without any authentication of the public keys, Alice and Bob are unable to verify their true owner's identity. As a result, they remain convinced of having established a shared secret S_{AB} directly with each other, and are entirely oblivious to the attack.

After the 3DH key exchanges are complete and Olm sessions are established, Alice initiates the media key exchange. Eve hijacks the message, decrypts Alice's media key, re-encrypts it under the session with Bob and relays it to him. Bob responds with his media key, which Eve intercepts and relays to Alice in the same manner. Eve repeats the steps for future runs of the protocol as needed to obtain newly generated keys.

A successful attack allows Eve access to both current and future E2EE used by Alice and Bob, which can be used to decrypt and even forge E2EE traffic from either party.

4.1.2 Threat model. It could be argued that with control over the server, distributing a modified client application is an easier attack vector towards compromising conference confidentiality or integrity. However, such an attack is, in principle, detectable. Moreover, client application distribution *can* be separated from the server, i.e. this property is not inherent in Jitsi's protocol design.

Indeed, Jitsi client functionality is available in multiple forms. The web application is served to the user's browser on-the-fly and could indeed be modified by a compromised web server. On the other hand, desktop and mobile versions[3] as well as integrations in third-party software generally use other distribution channels. Thus, compromised server infrastructure does not necessarily imply a capability to alter the client software, while authentic clients are still vulnerable to the MITM attack.

[3]We note, however that as of May 2022, E2EE for Android and iOS is still under development [3].

4.1.3 Practical impact. In practice, obtaining the media key at the XMPP server is only one step towards breaking into E2EE-enabled conferences. This is because the media traffic traverses over the media bridge or directly between peers in the case of one-to-one meetings. However, re-routing the media traffic to an attacker-controlled endpoint is a straightforward extension of the attack, as the XMPP server handles the negotiation of media channels.

4.1.4 Proof of concept. To verify this finding, we built a functional proof-of-concept demonstrating this attack by injecting malicious code into the XMPP server. This is accomplished by a small Lua module which hooks into the server's plugin system and listens to certain events. The cryptographic operations underlying the session establishment are delegated to the Olm library, which is compiled alongside with and linked to the XMPP server executable.

The module intercepts messages which are part of the Olm session establishment protocol, and modifies them to establish two channels with either participant instead. The module then waits for the media key exchange, and logs the captured keys before relaying them to their intended recipients.

As described in Section 4.1.3, this forms just one part of a full attack to gain access to the conference's media contents. To complete the demonstration, we have built a second module to hide a malicious conference participant from the others. The module drops XMPP messages announcing the malicious user's presence to everyone but the conference focus. As a result, the focus still allocates JVB channels routing media to the hidden user, however other clients never see their presence.

We note this is just one of many ways a malicious XMPP server could gain access to the media traffic to make use of an intercepted media key. Some such possibilities are further discussed in Section 4.6.

4.1.5 Remediation. Key authentication is a fundamental consideration in the design of public key cryptography based applications, and typically requires making some trade-offs to improve usability. One example of this is the TLS public key infrastructure used to authenticate connecting parties on the Internet. This model, however, does not trivially apply to the default, pseudonymous nature of Jitsi. Instead, voice and video calling applications with E2EE often display a Short Authentication String (SAS), which can be verbally compared by the communicating parties to detect MITM attacks. Following our disclosure of this vulnerability, Jitsi implemented a SAS-based authentication option to their client.

4.2 E2EE conference integrity

As detailed in Section 3.7.2, decryption failures in an E2EE context lead to the input being passed through to the media decoder as-is. While uncommon, this could, for instance, be caused by a key negotiation timeout in adverse network conditions. In this case, the meeting would carry on with some participants being unable to decrypt each others' media streams due to the lack of appropriate keys. Normally, attempting to decode an effectively pseudo-random ciphertext as a video frame creates a colorful noise pattern. This can make for an easy-to-understand demo of media E2EE, as seen in

Jitsi's blog post announcing the feature [4]. However, we found this seemingly harmless behavior is prone to abuse. More specifically, a compromised video bridge — the primary adversary in Jitsi's E2EE threat model [18] — can mount an attack breaking the integrity of E2EE conferences whose traffic traverses said bridge.

4.2.1 Overview. Consider the scenario in which the decryption function receives a regular, unencrypted VP8 video frame wrt. the steps described in Section 3.7. Attempting to decrypt this plaintext would produce an output where its probability of matching the expected frame trailer structure and authentication tag is negligible. Regardless of the exact point of failure (key retrieval, frame parsing, authentication, or decryption), the code treats it as an error and passes the unmodified input to the decoder. Being a regular VP8 frame, the input is successfully decoded and displayed in the client's view of the associated video stream.

4.2.2 Practical implications. This vulnerability allows an attacker-controlled video bridge to inject forged media into an E2EE-enabled conference, effectively breaking its integrity. The bridge still cannot decrypt other streams, hence their confidentiality is not impacted. In practice, this can make attacks more difficult, as the bridge can only "blindly" replace authentic media without knowing its contents. To circumvent this, a malicious bridge operator may collude with a conference participant to gain access to the meeting contents and use them to guide their attack.

4.2.3 Matching frames to streams. Meeting participants maintain exactly one media session with the video bridge regardless of the conference size. The JVB avoids mixing media streams by design [8], instead acting as a multiplexer. The client application distinguishes between the separate streams using their SSRC identifiers. As a result, a bridge can impersonate any conference participant as the origin of forged frames by simply manipulating the SSRC field. In other words, any conference participant's webcam view or audible speech can be replaced with attacker-chosen contents.

4.2.4 Proof of concept. Using an unsupported client makes it trivial to verify that unencrypted video is displayed as-is to clients that do support the feature. As described in Sections 3.2.4 and 3.3.1, such clients can still join E2EE conferences and their presence does not switch the feature off. They are unable to see or hear others' encrypted media, however can be seen and heard by everyone including E2EE-aware clients due to their disregard of decryption errors.

4.2.5 Remediation. The attack could be prevented through more careful error handling in the implementation to avoid proceeding further upon encountering parsing, authentication, or decryption errors. Additionally, the encrypted frame format could be updated to be explicitly incompatible with regular frames, and thus get rejected by the WebRTC stack, should the custom E2EE code let it through.

Following the disclosure, Jitsi updated their implementation to drop invalid frames instead of passing them to the decoder.

4.3 E2EE scope

Despite an "experimental" label still visible in the options menu, E2EE in Jitsi is marketed as a general security feature with no apparent caveats. As of June 2022, the official security page states: "Does Jitsi support end-to-end encryption? The short answer is: Yes, we do!" [7]

While not technically false, the broad claim can arguably be misconstrued as a level of security comparable to Signal or WhatsApp. This is demonstrably not the case, as none of the text-based communication is ever end-to-end encrypted. All exchanges through group chat, private messages, poll questions and votes, as well as user display names and avatars are always accessible by the XMPP server irrespective of the E2EE toggle. Various other exchanges such as P2P Jingle negotiations (Section 4.6) are similarly outside of the E2EE scope.

Whether a deliberate choice or an oversight, E2EE for media streams but not text messages seems like a design oddity. Simple message exchanges are significantly less bandwidth-intensive than video, and do not rely on cutting-edge draft API to integrate. Moreover, there exist attempts to standardize E2EE in XMPP such as *OMEMO* [41]. While the definitive reasons are unknown, the following sections discuss several plausible scenarios.

4.3.1 Practical attack surface. Sizable Jitsi deployments are likely to run multiple bridges per XMPP server, simply due to the disproportionate bandwidth cost of video routing. For practical reasons such as geographical placement and significant fluctuations in demand, they are also more likely to run on third-party infrastructure. It could therefore be argued the attack surface across many JVB instances in the cloud is greater than that of one or several self-hosted signaling servers.

4.3.2 Product features. At its core, Jitsi is a video conferencing application primarily used for its media streaming capabilities. Jitsi first announced E2EE at a similar time as some of its competitors. If time-to-market were a concern, prioritizing its implementation for video and audio streams may have been a conscious product development decision. Being the first to use the new WebRTC Encoded Transform API for E2EE could also be seen as a marketable feature.

4.3.3 Official communications. Interestingly, the fact that E2EE does not apply to text messages is never explicitly pointed out in the feature announcement, security page, E2EE whitepaper, or in the application itself. Official communications either describe the feature in broad terms, or focus on its implementation details with references to WebRTC. Upon a closer look at the architecture and explanations of the feature it may already be possible to deduce (or question) its scope. Unfortunately, this nuance is not captured in the simplistic statements about E2EE support and is easy to overlook.

4.3.4 Proof of concept. This flaw can be verified in practice in a number of ways. First, the Chromium developer tools allow participants of a Jitsi conference to observe the "raw" XMPP traffic, secured only via TLS. Sending and receiving text messages and interacting with polls can be observed as plaintext stanzas traversing the XMPP channel. As expected, enabling E2EE has no effect on this traffic.

Server-side module. In addition to the client-side observations, we have developed a malicious server module similar to the one described in Section 4.1.4 to corroborate the finding. The module

listens to MUC events, and saves all group and private messages to the server's diagnostic logs. Again, the contents remain visible even after E2EE is enabled.

In-room demonstration. For showcasing purposes, the module waits for a pre-determined trigger message from a conspiring user in the conference room. Once this happens, the module will begin forwarding others' private message exchanges to the attacking user in the conference

Remediation. A number of XMPP extensions provide options for message encryption, and could be used to protect the chat messages or other XMPP-based features from the influence a malicious signaling server. However, Jitsi opted to not extend E2EE to its text based communications, instead simply updating their documentation following disclosure.

4.3.5 Conclusion. Having conducted an in-depth analysis of the code and the practical demonstration described above, we have proven Jitsi's E2EE does not apply to any text-based communication. While said traffic takes a different route (XMPP instead of the JVB), this is a detail largely irrelevant to the average user, who may expect the claims about E2EE support to cover both media and text.

4.4 Vote manipulation

Among its out-of-the-box features, Jitsi allows conference participants to create polls and vote on multiple-choice questions. This is provided by an XMPP server module, and a custom messaging protocol over MUC stanzas. We have identified several critical flaws in the feature's design, allowing any conference participant to manipulate others' responses and forge votes arbitrarily.

4.4.1 Overview. Fundamentally, the poll feature design places too much trust in the clients' inputs, neglecting the potential for malice. The protocol messages needlessly include participant JID and names which could be derived from the XMPP sessions instead. The fields are not authenticated in any way, and can be trivially manipulated by any participant to impersonate others. What is more, the utter lack of validation or cross-checking allows forging a virtually unlimited number of votes under arbitrary voter names beyond the participant list.

4.4.2 Proof of concept. The potential attacks can be trivially demonstrated using the browser's developer console, which exposes convenient JavaScript functions to send XMPP messages to the server. By crafting modified poll creation and voting payloads, we could create polls under others authors' names, and forge votes as described above. The attack can be carried out by a malicious client in the meeting, with no modifications to or involvement of any server components. Of course, since the underlying protocol is not secured by E2EE (Section 4.3), the votes can be observed and manipulated by the XMPP server as well.

4.4.3 Conclusion. The poll feature was originally developed at a week-long hackathon open to the public, which could explain its unpolished security posture. Even though "[it had been] requested many times over", "[prior attempts] never achieved the required amount of completeness [. . .]", but "[the hackathon entry] ticked all the boxes [. . .]" and was awarded the 1st place prize, later getting officially adopted by Jitsi Meet.

Regardless of its humble origins, the feature has received some interest from the public for use in "legally binding votings like in an association general meeting" [5]. Unfortunately, it is clearly unfit for such purposes, and could lead to adverse consequences if exploited "in the wild".

4.4.4 Remediation. Client-side voter impersonation can be addressed with a straightforward switch from the identities contained within protocol messages to those associated with the messaging channels instead. Jitsi implemented such updates after the vulnerability disclosure. We note that this remains vulnerable to server-side attacks, as it is based on the non-E2EE text signaling channels.

4.5 Faux E2EE in unsupported clients

Jitsi's E2EE implementation relies on the WebRTC Encoded Transform API described in Section 3.4. The draft spec is implemented in Chromium versions 83 and above, and is thus available in several mainstream browsers including Edge, Chrome, Brave, and Opera [7]. Notably, it is unsupported in Firefox as of June, 2022 [31]. End-to-end encryption is also unavailable in Jitsi's mobile applications using native WebRTC protocol implementations.

4.5.1 Overview. As described in Section 3.3.1, unsupported clients hide the E2EE toggle, but not other audio & visual status indicators. These can still be triggered by "enabling" the feature using the console or via XMPP presence updates. Moreover, the ciphertext pass-through vulnerability presented in Section 4.2 means the video sent by unsupported clients is seen by everyone, including those with E2EE support. Under certain circumstances, this behavior can be exploited to trick participants into a false sense of security.

4.5.2 No supported clients. If none of the conference participants support E2EE, enabling it as described above has no effect on the media streams being exchanged. Yet, it triggers an audible announcement stating that "end-to-end encryption is on", and displays a green padlock icon to the participants. A conference participant, as well as the XMPP server can therefore falsely convince a group of unsupported clients their meeting is secured using E2EE.

4.5.3 Mixed client support. Enabling E2EE when only a strict subset of the participants support it causes their video and audio to appear as illegible noise to those that do not. The audio streams in particular tend to generate rather loud, unpleasant sound patterns. Thus, users are unlikely to carry on in this configuration for any significant amount of time.

In a "mixed" scenario of supported and unsupported clients, a compromised XMPP server can still carry out a weaker variant of the attack described in Section 4.5.2. Since the feature state is synchronized via XMPP presence, the server can selectively modify messages sent to clients based on whether they support E2EE. More specifically, the server can signal the feature is off to clients who support it, and signal it is on to those that do not. This way, the E2EE-ready clients keep the feature off, continuing to be seen and heard by everyone and therefore not revealing the attack. Those who do not support the feature receive a trigger to enable it, sounding the false announcement without any actual encryption being added.

4.5.4 Proof of concept. As mentioned in Section 3.3.1, unsupported clients can still access the E2EE switch through the browser developer console. The attack can thus be easily demonstrated by setting up a conference with several instances of Firefox, and toggling said switch. As expected, the participants can see and hear the false status indicators without actually encrypting their media or any indication of the attack.

4.5.5 Remediation. The issue has been addressed by Jitsi through some changes in the state-handling within the client application following vulnerability disclosure.

4.6 P2P mode

In one-on-one conferences, Jitsi attempts to use a direct P2P connection instead of routing through the JVB. This serves a practical purpose to reduce unnecessary load on the bridge, and can effectively leverage the DTLS-SRTP tunnel as E2EE without the need for an additional encryption layer. Yet, both the P2P mode itself and the design of P2P-JVB switching allows a compromised XMPP server to intercept P2P media traffic without detection.

4.6.1 Security claims. Jitsi's official security page claims "very strong protection even if you don't explicitly turn on e2ee", explaining that "in [P2P mode], audio and video are encrypted using DTLS-SRTP all the way from the sender to the receiver [...]" [7]. The statement is technically true w.r.t. the WebRTC specification [36, §6.5], but fails to capture the fact Jitsi's design provides no mechanisms to independently verify either party's identity. As an aside, nothing is said about text-based communication still traversing the server (Section 4.3).

Later in the same section, the security page states: "Since Jitsi is built on top of WebRTC, a deeper look into its security architecture is very important when evaluating Jitsi's security aspects." [7]. The referenced document in fact explicitly addresses the above issue: "[...] the signaling server can potentially mount a man-in-the-middle attack unless implementations have some mechanism for independently verifying keys." [36, §9.1] Jitsi has no such mechanism, and its P2P mode is therefore susceptible to trivial MITM attacks by a compromised XMPP server.

MITM attack details. The WebRTC negotiations include metadata such as network information and self-signed TLS certificate fingerprints. To mount the above attack, the XMPP server can trivially modify the signaling messages, tricking both parties into connecting to an attacker-controlled endpoint. This can be used in conjunction with the MITM attack on Olm sessions (Section 4.1), which allows a compromised XMPP server to intercept both the media traffic, and E2EE keys, if used. Notably, the P2P mode does nothing to prevent this attack due to the trust model of WebRTC and a lack of additional measures on Jitsi's part.

4.6.2 Transparent fallback to the JVB. The mode of operation (JVB or P2P) is "[...] transparent to the user." [7] It could be argued this is an implementation detail which should not concern users. However, considering Jitsi's claims on the security of P2P connections (Section 4.6.1), knowing the current mode gains some relevance. After all, encryption "[...] from the sender to the receiver [...]" [7] could make the E2EE switch seem redundant in one-on-one meetings.

Independently of the MITM attacks discussed in Sections 4.1 and 4.6.1, a compromised XMPP server can abuse the P2P feature design to reliably disable it, routing traffic over a bridge instead. It can then gain access to the media streams using the XMPP control interface of the JVB.

Blocking negotiation. A P2P connection is established using a Jingle protocol run. In case of failure, the client silently falls back to JVB mode without any apparent indication. The current mode can only be found in connection diagnostic menus, and is not displayed by default. The XMPP server can therefore effectively disable the P2P entirely by simply blocking Jingle negotiation messages between two clients (excluding the bridge).

Proof of concept. To test the reliability of this attack in practice, we built an XMPP server module similar to those described in Sections 4.1.4 and 4.3.4. It intercepts Jingle's `session-initiate` messages [27, §7.2.10] and only keeps those originating from or addressed to the JVB. This effectively prevents the P2P mode from functioning, while keeping JVB traffic unaffected.

Remediation. The potential impact of this issue has been reduced by patching the E2EE integrity-breaking vulnerability, and the introduction of SAS. Jitsi has additionally updated the relevant documentation following our disclosure.

5 CONCLUSIONS

In general, our results demonstrate the importance of a comprehensive approach to security engineering, particularly in complex, multi-component systems. Jitsi's E2EE design appears to be fixated on the narrow goal of ensuring that WebRTC streams cannot be eavesdropped on by a rogue JVB. As a result, the design overlooks critical issues such as E2EE of chat messages, and flawed key management which allows the signaling server to defeat the E2EE layer entirely. Curiously, an effort is made to share media keys using Olm, which is unnecessary if we assume the signaling server is honest, and ineffective without proper authentication otherwise.

In addition to the fundamental E2EE design issues, Jitsi's code exhibits a general lack of secure coding practices, leading to more weaknesses within the system. Our analysis shows how a fail-open decryption implementation, likely left over from a demo, leads to a compromise of E2EE conference integrity in Jitsi's own explicit threat model. Furthermore, the poll feature, initially developed in a hackathon, was merged into the mainline project despite its broken trust model. Lastly, insufficient state consistency checks in the client allow for some more subtle social engineering attacks.

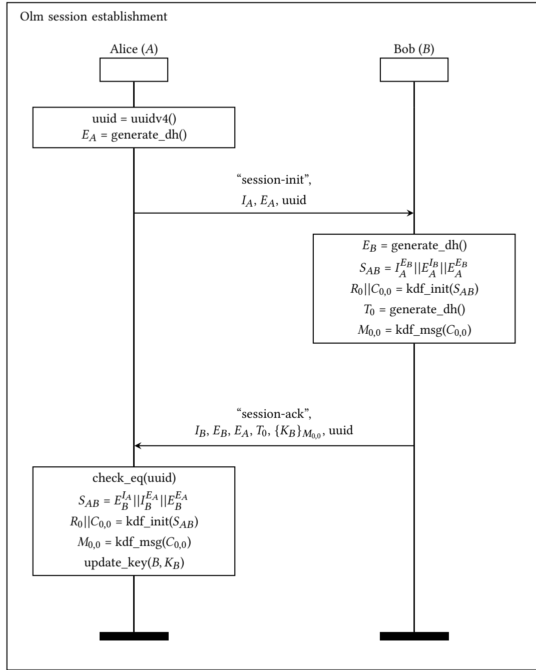
To end on a positive note, after a hesitant start. Jitsi engaged positively with us during disclosure. They made a significant engineering investment to improve the E2EE design, adding an authentication mechanism based on short authentication strings and fixing the majority of the vulnerabilities we identified. However, we regret that Jitsi addressed some of the security issues through changes to documentation rather than technical improvements. This likely puts too much burden on the average user of a mass-market product.

REFERENCES

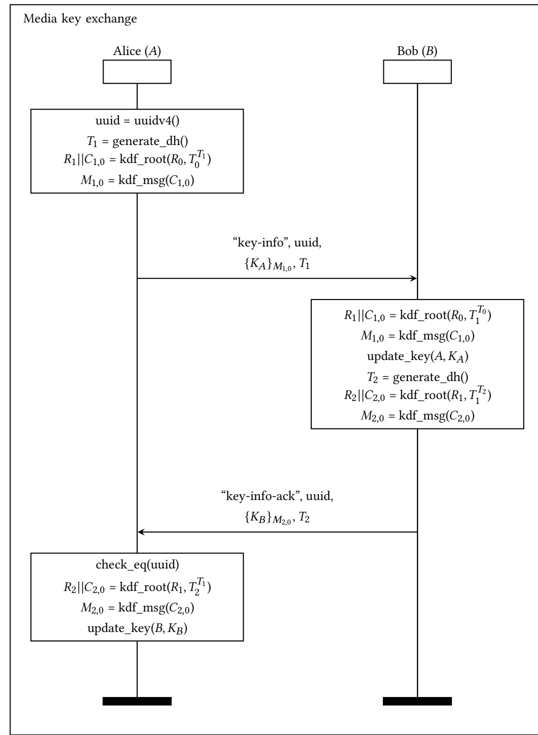
[1] 8x8, Inc. [n. d.]. Jitsi GitHub page. Retrieved 2022-04-09 from https://github.

com/jitsi
[2] 8x8, Inc. 2020. Security audit? Jitsi community forum. Post #2. Retrieved 2022-12-03 from https://community.jitsi.org/t/security-audit/25401/2
[3] 8x8, Inc. 2020. Support E2EE for Android and iOS. Jitsi Meet GitHub repository. Issue 8148. Retrieved 2022-05-24 from https://github.com/jitsi/jitsi-meet/issues/8148
[4] 8x8, Inc. 2020. This is what end-to-end encryption should look like! Retrieved 2022-06-22 from https://jitsi.org/blog/e2ee/
[5] 8x8, Inc. 2021. Ability to create polls inside Jitsi. Jitsi Meet GitHub repository. Pull Request 9166. Retrieved 2022-06-25 from https://github.com/jitsi/jitsi-meet/pull/9166
[6] 8x8, Inc. 2022. Jitsi Meet. https://meet.jit.si
[7] 8x8, Inc. 2022. Jitsi Meet Security & Privacy. Retrieved 2022-06-19 from https://jitsi.org/security/
[8] 8x8, Inc. 2022. Jitsi Videobridge / Open Source Video Conferencing for Developers. Retrieved 2022-06-17 from https://jitsi.org/jitsi-videobridge/
[9] 8x8, Inc. 2022. lib-jitsi-meet GitHub repository. https://github.com/jitsi/lib-jitsi-meet/tree/4baeb98964c6
[10] 8x8, Inc. 2022. *Self-Hosting Guide - Debian/Ubuntu server*. 8x8, Inc. Retrieved 2022-04-07 from https://jitsi.github.io/handbook/docs/devops-guide/devops-guide-quickstart/
[11] Martin R. Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. 2022. Practically-exploitable Cryptographic Vulnerabilities in Matrix. To appear at IEEE S&P'23, https://nebuchadnezzar-megolm.github.io/.
[12] Harald Alvestrand, Guido Urdaneta, and Youenn Fablet. 2022. *WebRTC Encoded Transform*. W3C Working Draft. W3C. https://www.w3.org/TR/2022/WD-webrtc-encoded-transform-20220519/
[13] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. 2012. *VP8 Data Format and Decoding Guide*. RFC 6386. RFC Editor. https://www.rfc-editor.org/rfc/rfc6386.txt
[14] Tom Berson. 2005. SKYPE SECURITY EVALUATION. https://download.skype.com/share/security/2005-031%20security%20evaluation.pdf
[15] Ashley Boyd. 2020. Which Video Call Apps Can You Trust? https://blog.mozilla.org/en/privacy-security/which-video-call-apps-can-you-trust/
[16] Chiara Castro. 2022. Meet the people helping activists to fight against digital surveillance. https://www.techradar.com/features/meet-the-people-helping-activists-to-fight-against-digital-surveillance
[17] Shaanan Cohney, Ross Teixeira, Anne Kohlbrenner, Arvind Narayanan, Mihir Kshirsagar, Yan Shvartzshnaider, and Madelyn Sanfilippo. 2021. Virtual Classrooms and Real Harms: Remote Learning at U.S. Universities. In *Seventeenth Symposium on Usable Privacy and Security, SOUPS 2021, August 8-10, 2021*, Sonia Chiasson (Ed.). USENIX Association, 653–674. https://www.usenix.org/conference/soups2021/presentation/cohney
[18] Saúl Ibarra Corretgé and Emil Ivov. 2021. End-to-End Encryption in Jitsi Meet. Retrieved 2022-04-21 from https://jitsi.org/wp-content/uploads/2021/08/jitsi-e2ee-1.0.pdf
[19] Morris Dworkin. 2007. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D. National Institute of Standards & Technology. https://doi.org/10.6028/NIST.SP.800-38D
[20] Ghita Ennadif. 2020. Spanish and French governments turn to open source video-conferencing platform. https://joinup.ec.europa.eu/collection/open-source-observatory-osor/news/open-source-videoconferences
[21] Raiful Hasan and Ragib Hasan. 2021. Towards a Threat Model and Security Analysis of Video Conferencing Systems. In *18th IEEE Annual Consumer Communications & Networking Conference, CCNC 2021, Las Vegas, NV, USA, January 9-12, 2021*. IEEE, 1–4. https://doi.org/10.1109/CCNC49032.2021.9369505
[22] Takanori Isobe and Ryoma Ito. 2021. Security Analysis of End-to-End Encryption for Zoom Meetings. *IEEE Access* 9 (2021), 90677–90689. https://doi.org/10.1109/ACCESS.2021.3091722
[23] Emil Ivov, Lyubomir Marinov, and Philipp Hancke. 2013. COnferences with LIghtweight BRIdging (COLIBRI). https://xmpp.org/extensions/xep-0340.html
[24] Antoine Joux. 2006. Authentication failures in NIST version of GCM. *NIST Comment* (2006), 3.
[25] Balakumar K. 2021. Pegasus Spyware: Is your mobile ever really safe from being hacked? https://www.techradar.com/news/pegasus-spyware-is-your-mobile-ever-really-safe-from-being-hacked
[26] Dima Kagan, Galit Fuhrmann Alpert, and Michael Fire. 2020. Zooming Into Video Conferencing Privacy and Security Threats. *CoRR* abs/2007.01059 (2020). arXiv:2007.01059 https://arxiv.org/abs/2007.01059
[27] Scott Ludwig, Joe Beda, Peter Saint-Andre, Robert McQueen, Sean Egan, and Joe Hildebrand. 2005. Jingle. https://xmpp.org/extensions/xep-0166.html
[28] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH Key Agreement Protocol. Retrieved 2022-04-15 from https://signal.org/docs/specifications/x3dh/x3dh.pdf
[29] matrix.org. [n. d.]. olm / An implementation of the Double Ratchet cryptographic ratchet in C++. Retrieved 2022-04-15 from https://matrix.org/docs/projects/other/olm
[30] matrix.org. 2022. Jitsi in Element. Developer documentation. Retrieved 2022-12-03 from https://github.com/vector-im/element-web/blob/7defcf3957d8e2674d83944b46cd1fe583834f4d/docs/jitsi.md
[31] Mozilla Foundation. [n. d.]. Support RTCRtpScriptTransform (formerly webrtc insertable streams). Retrieved 2022-06-25 from https://bugzilla.mozilla.org/show_bug.cgi?id=1631263
[32] C. Newman, A. Menon-Sen, A. Melnikov, and N. Williams. 2010. *Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms*. RFC 5802. RFC Editor. https://www.rfc-editor.org/rfc/rfc5802.txt
[33] E. Omara, J. Uberti, A. Gouaillard, and S. Murillo. 2020. *Secure Frame (SFrame)*. Internet-Draft. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-omara-sframe-00 Work in Progress.
[34] Trevor Perrin and Moxie Marlinspike. 2016. The Double Ratchet Algorithm. Retrieved 2022-04-07 from https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf
[35] Thomas Reisinger, Isabel Wagner, and Eerke A. Boiten. 2023. Security and Privacy in Unified Communication. *ACM Comput. Surv.* 55, 3 (2023), 55:1–55:36. https://doi.org/10.1145/3498335
[36] E. Rescorla. 2021. *WebRTC Security Architecture*. RFC 8827. RFC Editor. https://www.rfc-editor.org/rfc/rfc8827.txt
[37] Peter Saint-Andre. 2002. *Multi-User Chat*. XEP 0045. XMPP Standards Foundation. https://xmpp.org/extensions/xep-0045.html
[38] Peter Saint-Andre. 2006. *Best Practices for Use of SASL ANONYMOUS*. XEP 0175. XMPP Standards Foundation. https://xmpp.org/extensions/xep-0175.html
[39] Martin Shelton. 2016. Research Methods With Media Activists Under Surveillance. https://mshelton.medium.com/research-methods-with-media-activists-under-surveillance-979cef44fa55
[40] Signal Messenger LLC. [n. d.]. Signal Messenger. https://signal.org
[41] Andreas Straub, Daniel Gultsch, Tim Henkes, Klaus Herberth, Paul Schaub, and Marvin Wißfeld. 2015. OMEMO Encryption. https://xmpp.org/extensions/xep-0384.html
[42] The Tor Project. 2020. If you want an alternative to Zoom: try Jitsi Meet. It's encrypted, open source, and you don't need an account. https://meet.jit.si. Retrieved 2022-12-03 from https://twitter.com/torproject/status/1244986986278072322
[43] JM. Valin, K. Vos, and T. Terriberry. 2012. *Definition of the Opus Audio Codec*. RFC 6716. RFC Editor. https://www.rfc-editor.org/rfc/rfc6716.txt
[44] Mark Watson. 2017. *Web Cryptography API*. W3C Recommendation. W3C. https://www.w3.org/TR/2022/WD-webrtc-encoded-transform-20220519/
[45] M. Westerlund and S. Wenger. 2015. *RTP Topologies*. RFC 7667. RFC Editor. https://www.rfc-editor.org/rfc/rfc7667.txt
[46] Andrew M. White, Austin R. Matthews, Kevin Z. Snow, and Fabian Monrose. 2011. Phonotactic Reconstruction of Encrypted VoIP Conversations: Hookt on Foniks. In *32nd IEEE Symposium on Security and Privacy, S&P 2011*. IEEE Computer Society, 3–18. https://doi.org/10.1109/SP.2011.34
[47] Anna Wiener. 2020. Taking Back Our Privacy. https://www.newyorker.com/magazine/2020/10/26/taking-back-our-privacy
[48] Zoom Video Communications, Inc. 2021. E2E Encryption for Zoom Meetings. Retrieved 2022-07-02 from https://github.com/zoom/zoom-e2e-whitepaper/blob/master/archive/zoom_e2e_v3_2.pdf

A ADDITIONAL FIGURES



(a) Olm session establishment protocol flow.



(b) Media key exhange protocol flow.

Figure 4: Protocol flows