The road to End-to-End Encryption in Jitsi Meet
How we did it, and how you can do it too!
WebRTC Security model
A quick look
WebRTC security refresher

- Restricted to “safe origins” in browsers
- DTLS-SRTP is mandatory (RFC8829, sec 5.1.1)
- Encrypted by design
WebRTC with SFU architecture

- Peer connections established with a server
- More scalable architecture
- The server has access to the media
SFU media processing
Why access to the media is was necessary

- Optimal video layer routing
- Keyframe detection
- Only the **packet header** is necessary
Wasn’t WebRTC end to end encrypted?
Sort of
End-to-End Encrypted…
Sort of

• When media is flowing Peer-to-Peer
  • But SFUs are needed for scaling
• Bad UX for certificate validation
• No indication if the tracks are swapped out
Building real E2EE for WebRTC
Why do we need E2EE?

- Go check Emil Ivov’s “e2ee beyond buzzwords” talk
- Eliminate the need to trust SFUs
Insertable Streams
The API that unlocked it

• JavaScript API for manipulating **full frames**
• Data is mangled before **transport encryption**
• WebCrypto APIs can be used for encryption
• Worker friendly
• Chromium only as of today
Insertable Streams
Insertable Streams

Encrypting transform

```javascript
// Create a PeerConnection
this.pc = new RTCPeerConnection({ encodedInsertableStreams: true });

// Add the stream and encrypt it
stream.getTracks().forEach(track => {
    const sender = this.pc.addTrack(track, stream);
    const insertableStreams = sender.createEncodedStreams();
    const transformer = new TransformStream({ transform: encrypt });

    insertableStreams.readableStream
        .pipeThrough(transformer)
        .pipeTo(insertableStreams.writableStream);
});

function encrypt(chunk, controller) {
    // AES encrypt with WebCrypto APIs ...
    controller.enqueue(chunk);
}
```
Insertable Streams
Decrypting transform

// Handle remote tracks and decrypt them
peerConnection.ontrack = e => {
    const transformer = new TransformStream({ transform: decrypt });
    const insertableStreams = e.receiver.createEncodedStreams();

    insertableStreams.readableStream
        .pipeThrough(transformer)
        .pipeTo(insertableStreams.writableStream);
};

function decrypt(chunk, controller) {
    // AES decrypt with WebCrypto APIs...
    controller.enqueue(chunk);
}
Don’t roll your own crypto

Hello SFrame!
SFrame
End-to-end encryption and authentication for media frames

• draft-omara-sframe (early stages)
• IETF WG formed
• Apple experimenting with a native implementation
• Bring your own key management (MLS, Signal, olm, other)
Insertable Streams in Jitsi Meet

- Encryption keys
  - AES-CTR 256bit + HMAC SHA-256 (truncated)
- Signing keys
  - ECDSA P-521
- “JFrame”, a slight variation of SFrame
- All encryption happens in a Worker
Insertable Streams

The Result
Key management
The missing piece
Unmanaged

Shared passphrase

- Users type a shared passphrase obtained out-of-band
- Encryption key is derived from the passphrase using PBKDF2
- The key never leaves the user machine
Managed
Hello olm!

- E2EE channel using Matrix’s libolm
- Randomly generated per-participant keys
- Automatic key rotation and ratcheting
- Keys are exchanged using the olm channel
- User verification using SAS
Implementation
Show me the code!

• Self-contained in lib-jitsi-meet
• ~1000 lines of code
• https://github.com/jitsi/lib-jitsi-meet/tree/master/modules/e2ee
Future

What’s next?
• Finish SAS validation

• Bring back unmanaged mode and make it configurable

• Collaborate with the IETF SFrame WG

• UI/UX polish
Thanks
We didn’t do this alone
• Philipp Hancke, for working with us to make E2EE possible in Jitsi Meet

• Matrix, for libolm and the help understanding how to use it properly

• Google, for championing the insertable streams effort

• Our community, for all the love and support