# Mattermost End-to-End Encryption Plugin

Adrien Guinet & Angèle Bossuat

Quarkslab

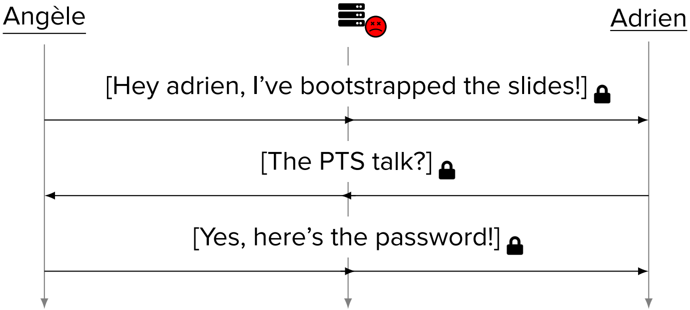
# Table of Contents

# Setup

## End-to-end encryption?

- **Encrypt** (and **sign**) messages on end devices just before sending them
- (**Verification**) and **decryption** also happen on the receiving end device(s)

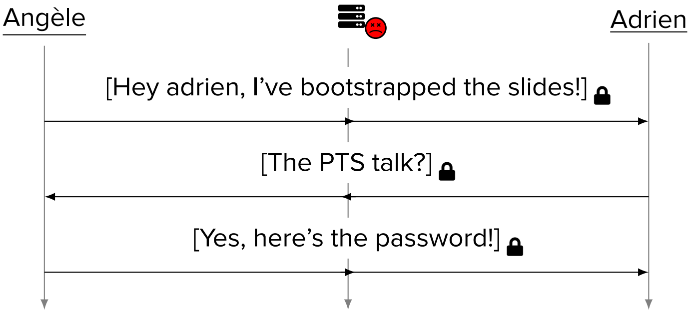
$\Rightarrow$ **End** (device)**-to-End** (device) **Encryption** (E2EE)!



# Setup

## Why?

- **Privacy**: I don't want anyone but Angèle to be able to read my messages
- **Authenticity**: am I really talking to Angèle?
- **Integrity**: is this really the message Angèle has sent?



# What/who are we protecting this from?

## Passive attacker model

- An attacker passively listens to what goes in & out of the server
- Passively = does not modify any communication

## Active attacker model

- An attacker actively listens and tampers with what goes in & out of the server
- Examples of what they can do:
  - deliver fake public keys for some users (a form of MitM)
  - deliver compromised Javascript to clients

# Was this not already done before?

## Anonymous Mattermost plugin

`https://github.com/bakurits/mattermost-plugin-anonymous`

- Uses RSA for key exchange, through `node-rsa` (full JS implementation) $\Rightarrow$ no usage of *unextractible keys* from WebCrypto
- No authentication of messages
- The choice of encrypted messages is done per message $\neq$ per channel (our need)

## Other E2EE chat software

- Mobile-based authentication: WhatsApp, Signal, Olvid
- *Classical* authentication: Matrix, rocket.chat

# Table of Contents

# Table of Contents

# Limitations due to Mattermost's design

## Things hard to do with the current interface

- Notifications in encrypted messages
  - No easy way to reuse the existing mechanisms from a plugin
- Encryption of attachments (e.g. images)
  - Hard to make it transparent for the end user (e.g. on-the-fly decryption of attached images)
- Encrypted messages modification...
  - ...that is unstable between Mattermost versions [1]
- Peer-to-peer key exchange protocols
- Searchable encryption
- Many small UI/UX details

[1] github.com/quarkslab/mattermost-plugin-e2ee#unable-to-update-messages-mattermost--61---64

# Limitations due to Mattermost's design

## Sticking to the plugin interface?

Why not modify Mattermost directly?

- Far more complex maintenance and deployment scenarios (recurrent rebase, docker image builds, etc...)
- The plugin interface provides an easy-to-deploy experience for the end-users (just upload a `.tar.gz` to your instance)

# Table of Contents

# Group messaging

There are several ways to do group messaging (cf our blogpost 😉):

- one shared symmetric channel key
- a channel is just many subchannels of two
- the soon-to-be standardized MLS protocol

# Group messaging

There are several ways to do group messaging (cf our blogpost 😉):

- one shared symmetric channel key
- a channel is just many subchannels of two
- the soon-to-be standardized MLS protocol

⇒ we use the P2P mode (like Signal), with an *ephemeral Diffie-Hellman key exchange* for each message (not like Signal)

# Overview

Steps performed when sending a message:

1. generate a random key and encrypt the message;

# Overview

Steps performed when sending a message:

1. generate a random key and encrypt the message;
2. for each recipient, compute a shared key to encapsulate the message key;

# Overview

Steps performed when sending a message:

1. generate a random key and encrypt the message;
2. for each recipient, compute a shared key to encapsulate the message key;
3. sign the context/public values and the encrypted message;

# Overview

Steps performed when sending a message:

1. generate a random key and encrypt the message;
2. for each recipient, compute a shared key to encapsulate the message key;
3. sign the context/public values and the encrypted message;
4. send everything necessary;

# Overview

Steps performed when sending a message:

1. generate a random key and encrypt the message;
2. for each recipient, compute a shared key to encapsulate the message key;
3. sign the context/public values and the encrypted message;
4. send everything necessary;
5. wait for the reactions to your awesome joke.

# Public-key cryptography 101

Quick reminder:

**symmetric cryptography** (e.g. AES) the same key is used to encrypt and decrypt

**asymmetric cryptography** (e.g. RSA), we have two keys: the *public* key is used to encrypt, and the *private* key to decrypt; anyone can encrypt a message but only one person can decrypt it

# Public-key cryptography 101

Quick reminder:

**symmetric cryptography** (e.g. AES) the same key is used to encrypt and decrypt

**asymmetric cryptography** (e.g. RSA), we have two keys: the *public* key is used to encrypt, and the *private* key to decrypt; anyone can encrypt a message but only one person can decrypt it

**digital signatures** same idea, other way around: only the person who has the secret/signing key can sign, and everyone can use the public/verification key to check

# Public-key cryptography 101

Quick reminder:

**symmetric cryptography** (e.g. AES) the same key is used to encrypt and decrypt

**asymmetric cryptography** (e.g. RSA), we have two keys: the *public* key is used to encrypt, and the *private* key to decrypt; anyone can encrypt a message but only one person can decrypt it

**digital signatures** same idea, other way around: only the person who has the secret/signing key can sign, and everyone can use the public/verification key to check

## Best of both worlds

Public-key cryptography has a high computational cost, so we usually encrypt the message with a symmetric key (much more efficient), then use public-key cryptography to encrypt that key: this is called *hybrid* cryptography.

# Diffie-Hellman

## Keys

- Adrien has private key $d$, public key $g^d$, Angèle has $(n, g^n)$;
- Adrien sends $g^d$, Angèle sends $g^n$;
- Adrien compute $k = (g^n)^d$, Angèle computes $k = (g^d)^n$;
- we can each use $k$ to encrypt/decrypt a message

# Diffie-Hellman

**Keys**

- Adrien has private key $d$, public key $g^d$, Angèle has $(n, g^n)$;
- Adrien sends $g^d$, Angèle sends $g^n$;
- Adrien compute $k = (g^n)^d$, Angèle computes $k = (g^d)^n$;
- we can each use $k$ to encrypt/decrypt a message

In the *ephemeral* setting, Angèle will generate a new $n$ randomly for each message that she *sends*, but always use Adrien's long-term $g^d$.

# Encryption

For each recipient, we have a public ECDH key (P-256[2]).

- Randomly generate AES128-CTR key+IV, and encrypt the message

[2] Because we use WebCrypto to have non-extractable keys.

# Encryption

For each recipient, we have a public ECDH key (P-256[2]).

- Randomly generate AES128-CTR key+IV, and encrypt the message
- Key is encapsulated with AES-KW
  - the sender generates *one* ephemeral ECDH key,
  - computes the shared keys between that key and the recipients' ECDH keys,
  - the shared key is hashed with SHA256 and used to encapsulate the message key

[2] Because we use WebCrypto to have non-extractable keys.

# Encryption

For each recipient, we have a public ECDH key (P-256[2]).

- Randomly generate AES128-CTR key+IV, and encrypt the message
- Key is encapsulated with AES-KW

In the end, the encypted message structure contains:

$$[\texttt{IV},\texttt{pubECDHE},[\texttt{wrappedKey}_0,\ldots,\texttt{wrappedKey}_n],\texttt{encryptedMsg},\textit{signature}]$$

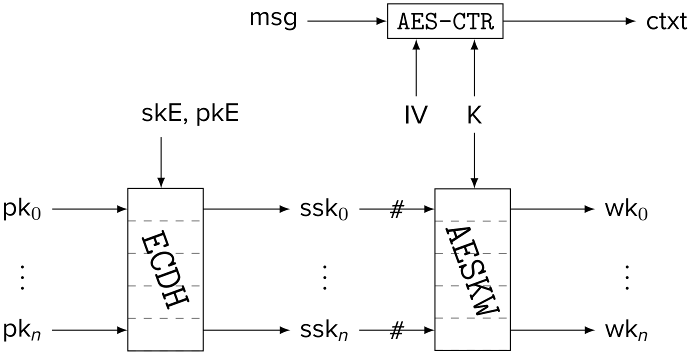
[2] Because we use WebCrypto to have non-extractable keys.

# Signature

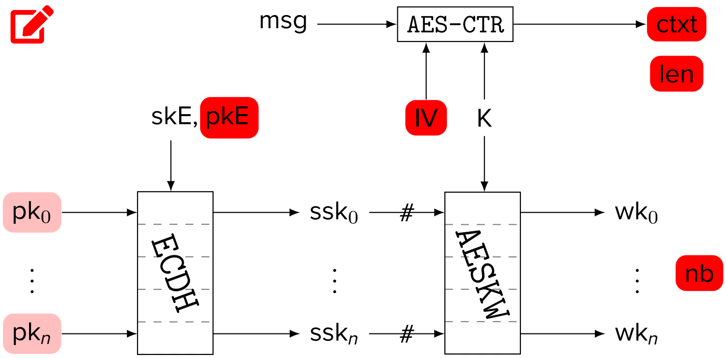
Each recipient knows the verification ECDSA key (P-256) of the sender, who signs:

- IV and public ECDHE key;
- number of recipients and *ordered* public key IDs;
- length of the message, and the encrypted message

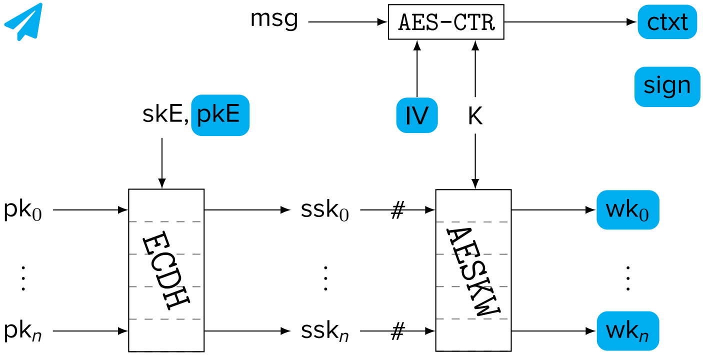
# Visual summary



# Visual summary



# Visual summary



# A quick note on security

Here is what the attacker can do depending on the known secrets:

| secret | implication |
|---|---|
| $K$ | can only decrypt current message |
| *skECDHE* | can only decrypt current message |
| *skECDH* | can decrypt all messages *received* by user |
| *skECDSA* | can impersonate user and send messages |

tl;dr cryptanalysis on messages is useless, need to compromise users' devices

# Table of Contents

# Table of Contents

# The webapp integrity problem

## An attack scenario

In the attack model where the **server isn't trusted / is compromised**:

- **Modified Javascript** can be shipped to a **targeted end user**
- That javascript could leak the original, unencrypted message to a third party
- $\Rightarrow$ **Defeats** the whole end-to-end encryption system

## Who else has this problem?

- Every webapp doing client-side cryptography (e.g. cryptpad (!), protonmail, ...)
- Signal: one of the reasons why there's only an Electron app [3]
- Whatsapp: browser plugin to verify code for `web.whatsapp.com` [4]

[3] https://mobile.twitter.com/moxie/status/1347351631420014592
[4] https://engineering.fb.com/2022/03/10/security/code-verify/

# The webapp integrity problem

## Subresource integrity?

```
<script type="text/javascript" src="main.js" integrity="sha384-oqVuAfXRKap7fdg...">
```

- The browser validates all sub resources (CSS/JS) against known hashes
- Originally design to be able to load these resources from untrusted CDNs

## Searching for a root-of-trust

- Even with SRI on all subresources, we need to trust the overall HTML...
- ...so we need a Root-of-Trust!
- How?

# The webapp integrity problem: service workers?

## TOFU with Service Workers

- Service workers can intercept requests client-side before they are **interpreted** by the browser
- We could ship a service worker with an **embedded Root-of-Trust** (TOFU), and verify HTML pages
  - We can even enforce SRI for all subresources from the service worker itself

## How to trust the service worker?

- **Problem**: a service worker can't intercept request to gather service workers themselves
  - **But it could work** with **SRI for Service Workers**!

```
navigator.serviceWorker.register('sw.js', { integrity: 'sha384-XXXX' })
```

# Conclusion

Some takeaways:

- we built a plugin for Mattermost to ensure more security
- we provide privacy, authenticity, and integrity of the messages
- works very well, but we still faced some limitations
- Apache license

You can find more details on our **blog**.quarkslab.com and on the **github**.com/quarkslab/mattermost-plugin-e2ee ☺

# Links

**Blogposts on secure messaging:**

https://blog.quarkslab.com/secure-messaging-apps-and-group-protocols-part-1.html
https://blog.quarkslab.com/secure-messaging-apps-and-group-protocols-part-2.html

**Blogpost on the plug-in:**

https://blog.quarkslab.com/mattermost-end-to-end-encryption-plugin.html

**GitHub of the plugin:**

https://github.com/quarkslab/mattermost-plugin-e2ee/blob/main/docs/design.md

# Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: https://www.quarkslab.com

Quarkslab