Six Attacks on Matrix

`https://nebuchadnezzar-megolm.github.io/`

Martin Albrecht
joint work with Sofía Celi, Benjamin Dowling and Dan Jones

Outline

New phone, who dis?

Cryptography in Matrix

Attacks

Discussion

Getting started on knocking stuff over

Matrix?

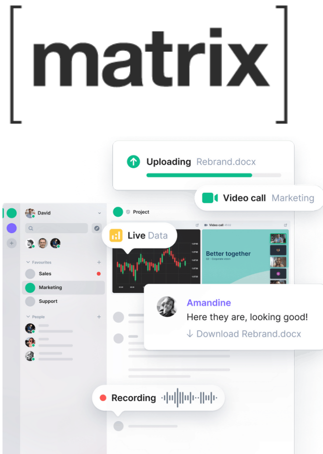
≡ WIRED LONG READS BUSINESS CULTURE GEAR SCIENCE SECURITY VIDEO SUBSCRIBE

GIAN M. VOLPICELLI SECURITY 14.06.2021 06:00 AM

How governments and spies text each other

Matrix has become the messaging app of choice for top-secret communications

Matrix!



- **Matrix** = standard for federated, decentralised, real-time group messaging
- **Element** = glossy flagship client
- End-to-end encryption is enabled by default
  - Adversarial model: servers are the adversary
  - Contrasts with Slack, MS Teams, Zulip, Mattermost, …

Matrix!

Element has over 80 million users. **Matrix**' users include

Matrix and Riot confirmed as the basis for France's Secure Instant Messenger app

2018-04-26 — In the News — Matthew Hodgson

Hi folks,

We're incredibly excited that the Government of France has confirmed it is in the process of deploying a huge private federation of Matrix homeservers spanning the whole government, and developing a fork of Riot.im for use as their official secure communications client! The goal is to replace usage of WhatsApp or Telegram for official purposes.

It's a unbelievably wonderful situation that we're living in a world where governments genuinely care about openness, open source and open-standard based communications - and Matrix's decentralisation and end-to-end encryption is a perfect fit for intra- and inter-governmental communication. Congratulations to France for going decentralised and supporting FOSS! We understand the whole project is going to be released entirely open source (other than the operational bits) - development is well under way and an early proof of concept is already circulating within various government entities.

Germany's national healthcare system adopts Matrix!

2021-07-21 — General, News — Matthew Hodgson

Hi folks,

We're incredibly excited to officially announce that the national agency for the digitalisation of the healthcare system in Germany (gematik) has selected Matrix as the open standard on which to base all its interoperable instant messaging standard - the TI-Messenger.

gematik has released a concept paper that explains the initiative in full.

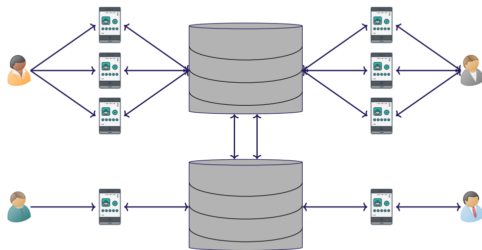
TL;DR

With the TI-Messenger, gematik is creating a nationwide decentralised private communication network - based on Matrix - to support potentially more than **150,000** healthcare organisations within Germany's national healthcare system. It will provide end-to-end encrypted VoIP/Video and messaging for the whole healthcare system, as well as the ability to share healthcare based data, images and files.

Initially every healthcare provider (HCP) with an HBA (HPC ID card) will be able to choose

Architecture

- In **Matrix**, each User account can have many Devices.
- Each User has an account on a particular Homeserver.
- Homeservers maintain the link between a User account and its Devices.
- Messages are distributed by the Homeservers.
- A Room is a collection of Devices that communicate in a single conversation.



Outline

New phone, who dis?

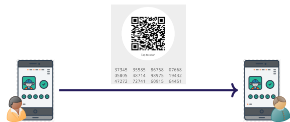
Cryptography in Matrix

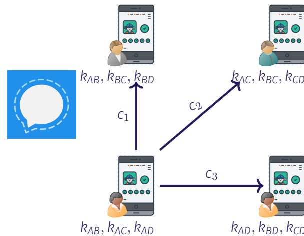
Attacks

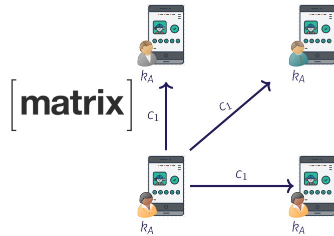
Discussion

Getting started on knocking stuff over

Core Functionalities







Device/Entity Auth
(**Cross-Signing Framework**)

Session Establishment
(pairwise **Olm** channels)

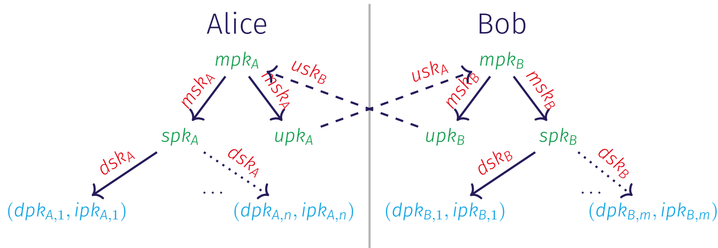
Session Communication
(group **Megolm** channels)

Entity Authentication via Cross-Signing Framework

Each User sets up an account with a particular Homeserver, which allocates a User identifier, A.

The User generates their User Secrets, used to establish ≊ web-of-trust.

- The master key (mpk_A) serves as their long-term identity.
- The user-signing key (upk_A) signs other User's master keys.
- The self-signing key (spk_A) signs a User's own Device keys.



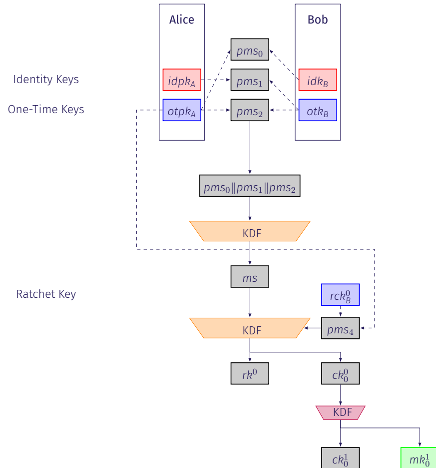
Device Authentication via Cross-Signing Framework

When a new Device logs in with account credentials, Homeserver allocates a device identifier $D^{A,i}$.
The Device then generates keys for this Device and registers it with the Homeserver:

1. Long-term Device Keys, authenticates **Olm Key Bundle**.
2. Olm Key Bundle, used to establish the pairwise channel, **Olm**.

Session Establishment via Olm

- Bob gets Alice's public key from Homeserver
- Bob does triple **D**iffie-**H**ellman (**3DH**) to produce a symmetric **m**aster **s**ecret.
- Bob uses **Double Ratchet** protocol to derive message keys.
- Bob encrypts **Megolm** Session State under these keys, and sends Session State to Alice.



Megolm Session

Megolm Session State allows the Sender to encrypt messages to the **Megolm** channel (resp. a Receiver to decrypt).

A **Megolm** session consists of the current *message index*, the *internal ratchet state*, and the *group signing key*.

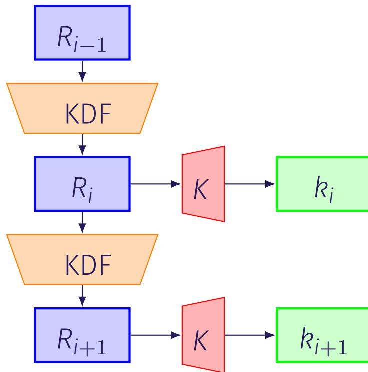
outbound session $\mathfrak{S}_{gsk} = (j, R, gsk)$ is kept in the device and used to encrypt messages for the room.

inbound session $\mathfrak{S}_{gpk} = (j, R, gpk)$ allows other devices in the room to authenticate and decrypt these messages.

Megolm Ratchet

At its core, **Megolm** is a symmetric ratcheting scheme:

- it derives a new key for each message
- so that compromise of the current state cannot be used to recover previous encryption state



Advancing the Megolm Ratchet I

The interesting part of **Megolm**: How does the **Ratchet** advance?

$R_{i,0}$ $R_{i,1}$ $R_{i,2}$ $R_{i,3}$

Makes catching up after a lot of messages quicker. Since you're constantly copying the ratchet state and advancing, do not need to do i HMAC calls to derive current ratchet state.

Advancing the Megolm Ratchet I

The interesting part of **Megolm**: How does the **Ratchet** advance?

Makes catching up after a lot of messages quicker. Since you're constantly copying the ratchet state and advancing, do not need to do i HMAC calls to derive current ratchet state.

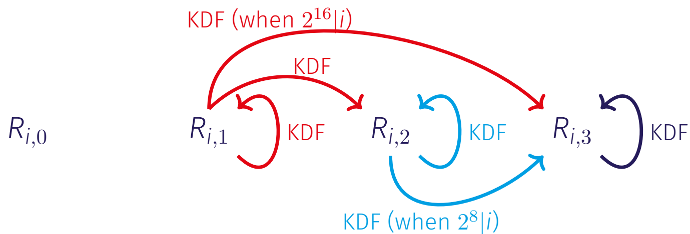
Advancing the Megolm Ratchet I

The interesting part of **Megolm**: How does the **Ratchet** advance?

Makes catching up after a lot of messages quicker. Since you're constantly copying the ratchet state and advancing, do not need to do i HMAC calls to derive current ratchet state.

Advancing the Megolm Ratchet I

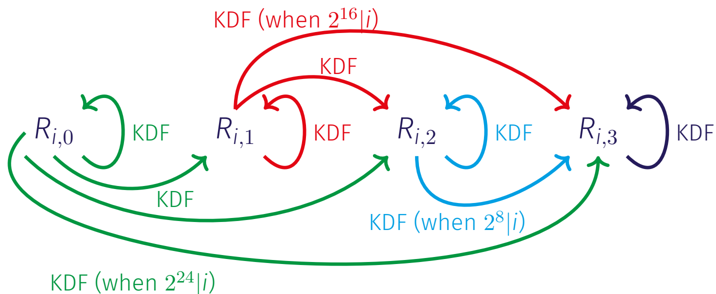
The interesting part of **Megolm**: How does the **Ratchet** advance?



Makes catching up after a lot of messages quicker. Since you're constantly copying the ratchet state and advancing, do not need to do i HMAC calls to derive current ratchet state.

Advancing the Megolm Ratchet I

The interesting part of **Megolm**: How does the **Ratchet** advance?



Makes catching up after a lot of messages quicker. Since you're constantly copying the ratchet state and advancing, do not need to do i HMAC calls to derive current ratchet state.

Advancing the Megolm Ratchet I

We need to advance the **Megolm** Ratchet between encryptions. How?

$$R_{i,0} = \begin{cases} H_0\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n \\ R_{i-1,0} & \text{otherwise} \end{cases}$$

$$R_{i,1} = \begin{cases} H_1\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n \\ H_1\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m \\ R_{i-1,1} & \text{otherwise} \end{cases}$$

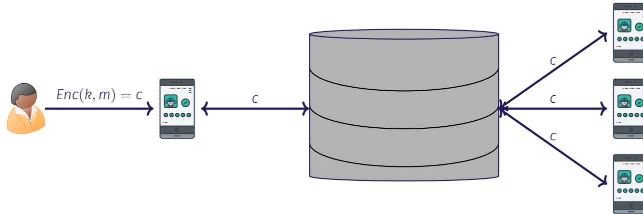
$$R_{i,2} = \begin{cases} H_2\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n \\ H_2\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m \\ H_2\left(R_{2^{8}(p-1),2}\right) & \text{if } \exists p | i = 2^{8}p \\ R_{i-1,2} & \text{otherwise} \end{cases}$$

$$R_{i,3} = \begin{cases} H_3\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n \\ H_3\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m \\ H_3\left(R_{2^{8}(p-1),2}\right) & \text{if } \exists p | i = 2^{8}p \\ H_3\left(R_{i-1,3}\right) & \text{otherwise} \end{cases}$$

$$H_i(x) = \text{HMAC}(\text{key} = x, \text{input} = \texttt{0x0i})$$

Megolm Encryption

1. Sender generates a fresh symmetric key from R,
2. encrypts the message under this key, and
3. signs it to provide authentication.



This ciphertext is distributed by the Homeserver to other devices in the Group.

Megolm Decryption

To decrypt a message, Devices verify the signature with their copy of $gpk_{A,i,G}$, and straightforwardly reverse the process.

Receiver Devices keep the oldest copy of the ratchet state, only advancing their ratchet in a temporary copy. This allows Devices from the same User to share message histories with one another, by sharing old ratchet states.

$\mathbf{Megolm.Dec}(\pi_{\text{mg}}, c)$

$(\text{ver}, i, R, gpk, \sigma_{\text{mg}}) \leftarrow \pi_{\text{mg}}$

$(\text{ver}', i', c', \tau, \sigma) \leftarrow c$

$\textbf{if } !\text{DS.Verify}(gpk, \sigma, (\text{ver}, i', c', \tau)) \textbf{ then}$

$\quad \textbf{return } (\pi_{\text{mg}}, \perp)$

$(i, R) \leftarrow \text{MegolmRatchet.Advance}^{i'-i}(i, R)$

$k_e \parallel k_h \parallel k_{iv} \leftarrow \text{HKDF}(0, R', lbl, 80)$

$\textbf{if } \tau \neq \text{HMAC}(k_h, (\text{ver}, i, c')[0:8] \textbf{ then}$

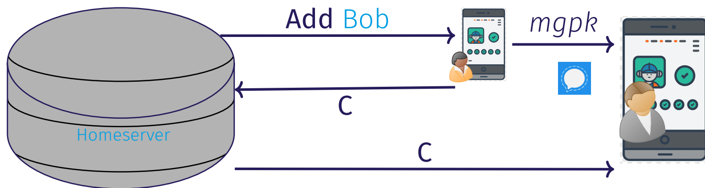
$\quad \textbf{return } (\pi_{\text{mg}}, \perp)$

$m \leftarrow \text{AES-CBC.Dec}(k_e, k_{iv}, c')$

$\textbf{return } (\pi_{\text{mg}}, m)$

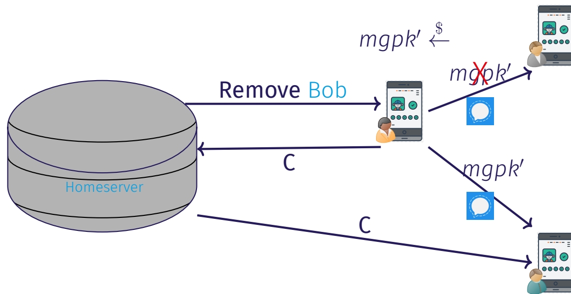
Adding Room Members

- Alice told by her Homeserver that Bob's Device has been added.
- Alice's Device now sends her *current* ratchet state $mgpk = (j, R, gpk)$ to Bob's Device using the **Olm** channel between them
- Bob's Device can now read messages sent by Alice's Device after this point.



Removing Room Members

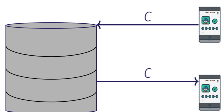
- Alice told by her homeserver that Bob's Device has been removed.
- Alice generates a brand new Megolm ratchet state $mgpk'$
- Alice uses her **Olm** session with all *other* devices in the conversation to send $mgpk'$
- Bob's Device can no longer read messages sent by Alice's Device after this point.

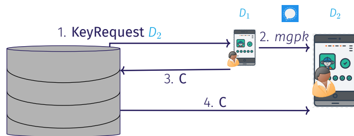


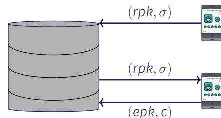
"Pursue your dreams but have a backup plan"

Backup Functionalities:

backup and recover User and Megolm secret values via Homeservers.







User Secret Backups (**Secure Secret Storage and Sharing (SSSS)**)

- backup master (cross-signing) secret keys to server

Online Session Recovery (**KeyRequest** protocol)

- allows a user's devices to share Megolm session information with each other

Offline Session Recovery (**Server-Side Megolm Backups**)

- as a hybrid of both, backup Megolm sessions to server

Outline

New phone, who dis?

Cryptography in Matrix

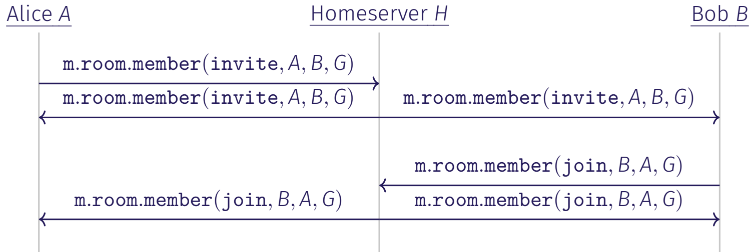
Attacks

Discussion

Getting started on knocking stuff over

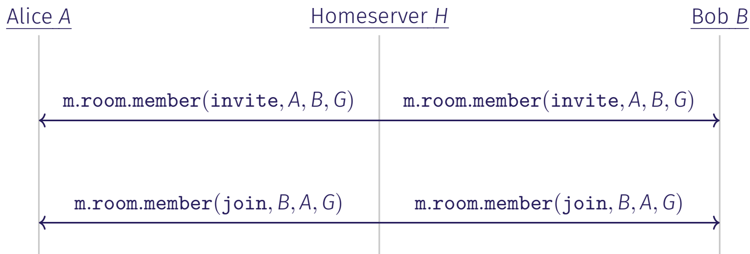
Q: "Who to encrypt to?"

Group membership is managed through events:



A: “Don’t worry, the server will let you know.”

Group membership is managed through unauthenticated events:



Discussion

Matrix knows group administration:

- Matrix knows of roles and permissions in rooms, including admin roles that control room membership
- But there is no cryptographic assurance: encryption, integrity protection, authentication

Element offers some “mitigations”:

- When a user is added to room this is visible as a genuine membership event
- Adding an unverified user to a room, adds a warning to the room that unverified devices are present.
  - But just because you verified Alice, does not mean Alice should have access to all your conversations.

Q: “What are Alice’s devices?” A: “Don’t worry …”

- To send a message to a user, clients need a list of their devices.
- This list is provided by the homeserver and, hence, can be forged.

Discussion

Matrix has a device verification framework:

- the list of devices in a room is maintained independently of that
- the homeserver may also present different device lists to different users/devices, e.g. not show the additional malicious device to the user it is purportedly associated with

Element does offer a "mitigation":

- adding an unverified device to a room will lead to a warning that such a device is present

Damage

Breaks confidentiality: Attackers can eavesdrop on conversations with some indication in (Element's) user interface.

Status

Neither of these two are fixed, but a remediation (signed group membership messages) is in the planning stage.

- Matrix' previous rational: Element client shows list of users for a room, so users can inspect, i.e. burden on users.
- Matrix post-disclosure: "many in the cryptography community consider this a serious misdesign. Eitherway, it's avoidable behaviour and we're ramping up work now to address it by signing room memberships so the clients control membership rather than the server."

Attack on Out-of-Band Verification

How to ensure connection is not being MITM-ed? Out-of-band verification!
Short Authentication String (SAS) protocol ≈

1. Key exchange to generate a shared secret.
2. Compare the shared secret out-of-band
   (using short strings of emojis).
   If they don't match, abort!
3. Send correct cryptographic identities to each other over a secure channel (constructed using the shared secret).

The homeserver tricks devices into sharing a homeserver-controlled identity.

Attack on Out-of-Band Verification

- Two types of verification:
    1. Between two users
    2. Between two devices of the same user
- Each party sends the other a message containing a "key identifier" field:
    1. For two users, this field contains the fingerprint of their master cross-signing key, *mpk*.
    2. For two devices, this field contains their device identifier.

Attack:

- Homeserver assigns the target a device identifier that is also a master cross-signing key fingerprint that the homeserver generated.
- When the target sends a verification request message with their device identifier, the receiving device interprets it as a cross-signing key fingerprint and signs it!

Out-of-band verification is used for:

1. two users' devices verify each other; and,
2. a user verifies a new device.

"Verification methods can be used to verify a user's master key by using the master public key, encoded using unpadded base64, as the device ID, and treating it as a normal device. For example, if Alice and Bob verify each other using SAS, Alice's m.key.verification.mac message to Bob may include "ed25519:alices+master+public+key": "alices+master+public+key" in the mac property. Servers therefore must ensure that device IDs will not collide with cross-signing public keys."

Message Format

```
{"mac": {"ed25519:<mpk'>": SAS.CalcMAC(k, dpk, c || "ed25519:<mpk'>"),
         "ed25519:<mpk>":  SAS.CalcMAC(k, mpk, c || "ed25519:<mpk>"),},
 "keys": SAS.CalcMAC(k, sort("ed25519:<mpk'>", "ed25519:<mpk>"), c || "KEY_IDS")}
```

An `m.key.verification.mac` message generated by a user with cross-signing master verification key *mpk*, long-term device key *dpk* and device identifier *mpk'* (which is also the master verification key of a homeserver controlled cross-signing identity). Whilst the two entries in the `mac` dictionary could be distinguished by the differing second argument given to SAS.CalcMAC, SAS.VerifyMAC interprets the first entry as a device, and then passes it to SAS.SignDevice which interprets it as a cross-signing identity.

Attack i

Alice A and Bob B, each with device $D_{A,1}$ and $D_{B,1}$ respectively. Additionally, they are both registered to a malicious homeserver, whose aim is to compromise their out-of-band verification with the SAS protocol. The attack proceeds as follows:

1. When Alice A registers their account with the homeserver, the homeserver generates a parallel cross-signing identity with verification keys (mpk'_A, spk'_A, upk'_A).
2. When Alice A logs in for the first time, the homeserver sets the device identifier $D_{A,1} \leftarrow mpk'_A$.
3. When Bob B logs in for the first time, the homeserver sets $D_{B,1}$ as normal.

Attack ii

4. Alice and Bob each setup their own cross-signing identities with verification keys (mpk_A, spk_A, upk_A) and (mpk_B, spk_B, upk_B) respectively. They upload these to the homeserver.

Attack iii

5. The homeserver proceeds to present two versions of the cross-signing state:
   5.1 When Alice requests their own cross-signing information, they are presented with the version they uploaded (mpk_A, spk_A, upk_A).
   5.2 When Bob requests Alice's cross-signing information, they are presented with the version generated by the malicious homeserver (mpk'_A, spk'_A, upk'_A).

Attack iv

6. Alice and Bob perform an out-of-band verification using the SAS protocol. At the end, they exchange `m.key.verification.mac` messages containing their cryptographic identity (for signing). $D_{B,1}$ processes it as follows:
   - 6.1 SAS.VerifyMAC interprets the entry for mpk'_A as a request for device verification. It fetches the expected device identity key $dpk_{A,1}$, then calculates a matching MAC. The device identity key pulled from the homeserver is legitimate, and matches the one used by Alice's device to generate the MAC. Thus, the message passes verification.
   - 6.2 SAS.SignDevice interprets the entry for mpk'_A as a request for cross-signing verification. This is because the homeserver has led Bob's client to believe that Alice's cross-signing identity is mpk'_A.

Attack v

7. Bob cross-signs the homeserver controlled identity for Alice, and uploads the signature to the homeserver to distribute to their other devices.

Damage

Breaks confidentiality: Attackers can eavesdrop on conversations

and authentication: Attackers can impersonate users

with no indication in (Element's) user interface!

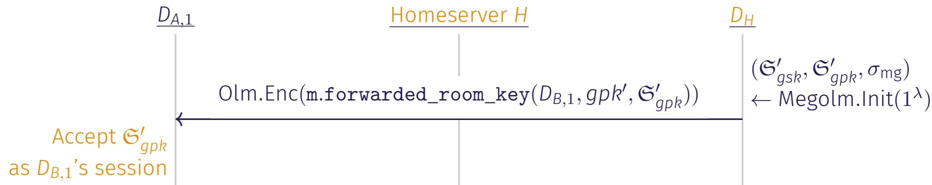
Take Home Message

Alice: ..., Bob: “Here are the keys for Charley”, Alice: “Ta!”

When a user adds a new device, they’d like that device to be able to decrypt messages previously sent to that user via the **KeyRequest** protocol.

Element and other clients limited who they sent secrets to but not who they accepted secrets from.

Attack:



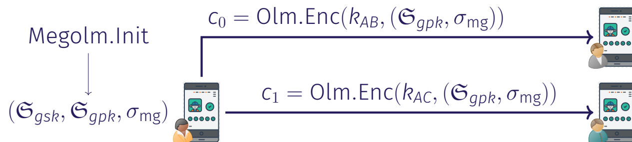
Damage

Semi-trusted Impersonation Attack

Breaks authentication: Attackers can impersonate users

with some indication in (Element's) user interface.

Megolm session setup:

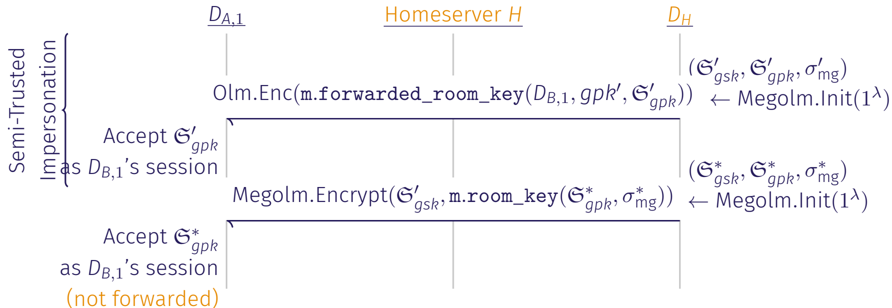


What if we could send $(\mathfrak{S}_{gpk}, \sigma_{\text{mg}})$ over Megolm instead of Olm?

Could we send it over a Megolm session placed via previous impersonation attack?

Layering Attacks for Full Impersonation

Device D_H impersonates $D_{B,1}$ to $D_{A,1}$:



Damage

Semi-trusted Impersonation Attack

Breaks authentication: Attackers can impersonate users

with some indication in (Element's) user interface.

Fully-trusted Impersonation Attack

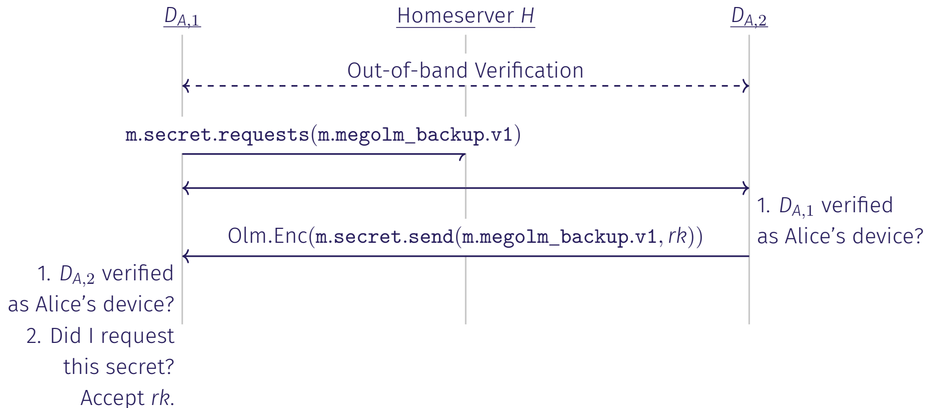
Breaks authentication: Attackers can impersonate users

with no indication in (Element's) user interface.

More Layers: Authentication to Confidentiality Break

When a user verifies their new device, it will use **SSSS** to request User Secrets from the user's existing devices.

This includes the **recovery key** used for **Megolm Backups**, i.e.



Damage

Semi-trusted Impersonation Attack

Breaks authentication: Attackers can impersonate users

with some indication in (Element's) user interface.

Fully-trusted Impersonation Attack

Breaks authentication: Attackers can impersonate users

with no indication in (Element's) user interface.

Authentication to Confidentiality Break

Breaks confidentiality: Attackers can eavesdrop on conversations

with no indication in (Element's) user interface.

Together: complete break of confidentiality and authentication!

Take Home Messages

- There is no confidentiality without authentication.
- Put all cryptographic code in one small core.[1]

[1] Element checked authentication at display time, rather than at receipt time and thus those checks were not run for messages that are not displayed.

IND-CCA Vulnerability

- Matrix uses an encrypt-then-MAC encryption scheme composing AES-CTR with HMAC-SHA-256.
- Recall that AES-CTR proceeds by encrypting a series of blocks $iv, iv \oplus 1, iv \oplus 2, \ldots$ and XORing the result onto the message blocks m_i to produce the ciphertext blocks c_i.
- The full ciphertext is $iv \| c_0 \| c_1 \| \ldots$. However, the *iv* is not covered by the authentication tag produced by HMAC-SHA-256.

IND-CCA Attack

Let $c*$ be some challenge ciphertext for either some message m_0 or m_1 of length 128 bits:

$$c* := iv \| \mathsf{AES}(k_0, iv) \oplus m_b \| \mathsf{HMAC}(k_1, \mathsf{AES}(k_0, iv) \oplus m_b).$$

The adversary requests an encryption of zero from the encryption oracle and receives

$$c := iv' \| \mathsf{AES}(k_0, iv') \oplus 0 \| \mathsf{HMAC}\left(k_1, \mathsf{AES}(k_0, iv') \oplus 0\right)$$

for some iv'. Finally, the adversary requests a decryption of

$$c^+ := iv \| \mathsf{AES}(k_0, iv') \oplus 0 \| \mathsf{HMAC}\left(k_1, \mathsf{AES}(k_0, iv') \oplus 0\right)$$

from the decryption oracle. Note that the MAC verifies correctly, and so the adversary will receive '$t := \mathsf{AES}(k_0, iv') \oplus \mathsf{AES}(k_0, iv) \oplus 0$. Given that the adversary already knows $\mathsf{AES}(k_0, iv')$ it can now recover $\mathsf{AES}(k_0, iv) = t \oplus \mathsf{AES}(k_0, iv')$ and thus decrypt the challenge $c*$.

IND-CCA Attack Limitations

- Forwarding keys to the target and observing the resulting backups on the homeserver provides an encryption oracle.
- Modifying a backup on the homeserver then requesting the resulting key provides a decryption oracle.
- In Matrix, k_0, k_1 are bound to a particular *gpk* and all secret *R*'s associated with a *gpk* can be obtained by the key request protocol.
- Any modified ciphertexts will likely be invalid JSON structures and fail to parse correctly (preventing the decryption oracle from sharing the plaintext with the adversary).

Attachments

- A similar issue exists for attachments which are shared out of band in encrypted form.
- Here the hash shared over Megolm (which takes the role of the MAC) does not include the *iv*.
- Since the *iv* itself is also shared over Megolm and thus implicitly authenticated, we do not see a way to exploit this behaviour.

Take home message:

There is no confidentiality without integrity.[2]

[2] Corollary: The CIA triad – confidentiality, integrity, availability – is nonsense.

Recap & Status

1. Trivial confidentiality breaks not yet fixed
2. Attack on out-of-band verification CVE-2022-39250; reportedly mitigated
3. Impersonation CVE-2022-39246, CVE-2022-39249 and CVE-2022-39257;[3] reportedly mitigated
4. Full impersonation CVE-2022-39248, CVE-2022-39251 and CVE-2022-39255; reportedly mitigated
5. Impersonation to confidentiality break same CVEs as above; reportedly mitigated
6. Theoretical confidentiality attack not yet fixed

[3] In their review of the ecosystem the Matrix developers discovered further clients vulnerable to variants of our attack and assigned CVE-2022-39252, CVE-2022-39254 and CVE-2022-39264.

Outline

New phone, who dis?

Cryptography in Matrix

Attacks

Discussion

Getting started on knocking stuff over

Difficult Problems!

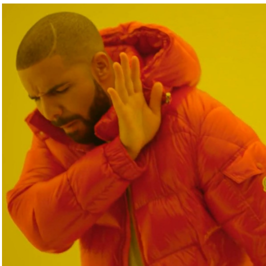
Matrix aims to solve some difficult problems:

1. Secure (Group) Messaging
   …in a multi-device setting,
   …that is scalable to thousands of devices in a single group.
2. Backups and history sharing.
3. Authentication and identity verification
   …cross-signing to reduce user burden of out-of-band verification.
4. Federation.
5. Supporting a variety of clients across many platforms.

Cryptography is not a dark art

"Crypto is hard!"

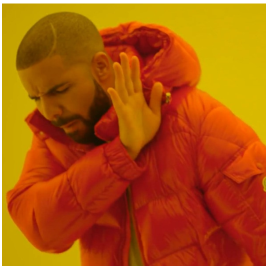
Cryptography is not a dark art



"Crypto is hard!"

Of course, cryptography is hard, so is any other science.

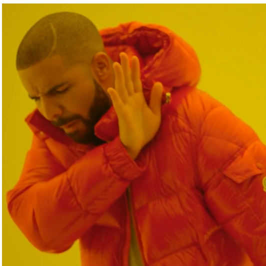
Cryptography is not a dark art



“Crypto is hard!”

Modern cryptography gives us the tools to reason about cryptographic protocols to rule out the sort of issues we found here.

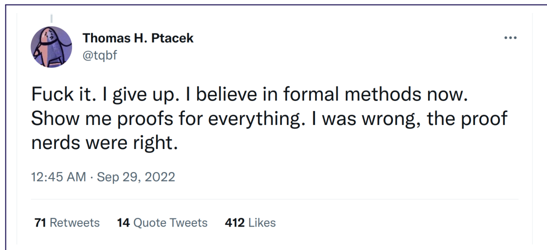
Cryptography is not a dark art



"Crypto is hard!"

"Cryptography needs security models and proofs!"

Cryptography is not a dark art



Outline

New phone, who dis?

Cryptography in Matrix

Attacks

Discussion

Getting started on knocking stuff over

Two approaches

Start from the attack

- Start from a vulnerability (e.g. IV reuse)
- Think about where the relevant primitive is used
- Search for protocols until you find one that is broken

Start from a protocol/scheme

- Start from a protocol you care about
- Read spec/source code
- Identify cryptographic core, focus on that
- Need to have a good idea what is out there in terms of attacks

Cryptographic proof attempts are cryptanalysis

- Our project started with the intention to prove Matrix secure (see Ben's talk)
- We obviously could not make the proof work.
- Eventually the attacks became so many that we decided to write an "attack paper" first

Try to prove more protocols out there secure! If you fail, you get an attack paper, if you succeed you get a proof paper!

Disclosure lessons

- Decide who you have a responsibility to.
  - This does not have to include the developers, the security state or even the main users.
- Typically you will attack a protocol/scheme/product developed by people with no rigorous security or cryptography training.
  - They may plead, threaten, bribe.
  - You can educate them, but you do not owe them this.
- Usually things sour when you move towards public disclosure

  Developer incentive minimise perception of impact

  Your incentive maximise perception of impact
- Security Twitter will be on your side with silly takes, don't join in, don't ridicule, but educate

Thank you! Questions?

`https://nebuchadnezzar-megolm.github.io/`