# Matrix Cryptographic Key Infrastructure


Sumner Evans

21 September 2024

Beeper (Automattic)





Matrix Cryptographic Key Infrastructure

Sumner Evans
21 September 2024
Beeper (Automattic)


1. Hello, my name is Sumner, I'm a software engineer at Automattic working on Beeper.
2. End-to-end encryption is one of the things which **brought me to Matrix**, and I'm sure that it's one of the factors that brought many of you to Matrix as well.
3. However, Matrix's user experience with cryptography is often confusing.
4. I mainly **blame the other chat networks** for their incompetence.
5. Most other chat networks **don't provide** any cryptographically-guaranteed security and privacy.
6. Of the ones that do, most do so in a way that **does not truly leave the user in control of their keys**.
7. Only a few networks, namely Signal, truly leave the user in control, and their UX is arguably worse than Matrix.
8. In this talk, my goal is to discuss the **cryptographic key *infrastructure* in Matrix**.
9. What do I mean by "infrastructure"? I mean all of the features which **support key sharing and identity verification**, but don't actually themselves provide security.
10. You can think of this talk as discussing the "UX layer of cryptography in Matrix". None of the things that I'm going to discuss are strictly necessary for ensuring secure communication, but without them, Matrix' UX would be horrible.

# Why Cryptography?

Matrix uses cryptography for two main purposes:

1. **Message Security** — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. **Identity** — verifying that a user or device is who they say they are.





Why Cryptography?

Matrix uses cryptography for two main purposes:

1. Message Security — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. Identity — verifying that a user or device is who they say they are.

1. Now, let's discuss what Matrix even uses cryptography for. There are two main purposes...
2. The first is **message security**. We only want the people who are part of the conversation to be able to read the messages in the conversation.
3. As an additional benefit of how Matrix achieves this, encrypted messages **cannot be tampered with** by a **man-in-the-middle** actor without the receiving party knowing.
4. The second reason for using cryptography in Matrix is **identity** verification. We want to know that a specific device or user is who they say they are.
5. Note that one of the most important uses for identity verification is verifying your own devices so you can share keys with them.

# Why Cryptography?

Matrix uses cryptography for two main purposes:

1. **Message Security** — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. **Identity** — verifying that a user or device is who they say they are.





Why Cryptography?
Matrix uses cryptography for two main purposes:
1. Message Security — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. Identity — verifying that a user or device is who they say they are.

1. Now, let's discuss what Matrix even uses cryptography for. There are two main purposes…
2. The first is **message security**. We only want the people who are part of the conversation to be able to read the messages in the conversation.
3. As an additional benefit of how Matrix achieves this, encrypted messages **cannot be tampered with** by a **man-in-the-middle** actor without the receiving party knowing.
4. The second reason for using cryptography in Matrix is **identity** verification. We want to know that a specific device or user is who they say they are.
5. Note that one of the most important uses for identity verification is verifying your own devices so you can share keys with them.

# Why Cryptography?

Matrix uses cryptography for two main purposes:

1. **Message Security** — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. **Identity** — verifying that a user or device is who they say they are.





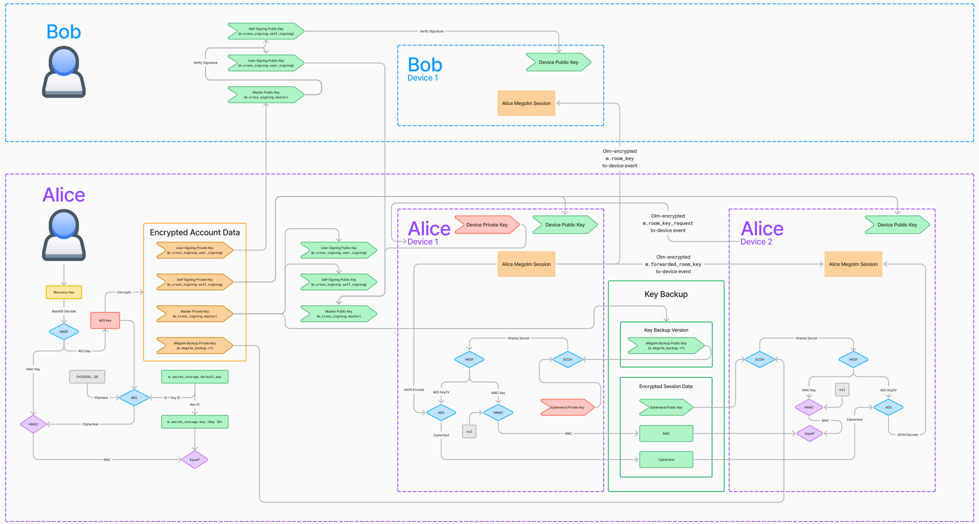
Why Cryptography?

Matrix uses cryptography for two main purposes:

1. Message Security — only the people who are part of the conversation should be allowed to view messages of the conversation.
2. Identity — verifying that a user or device is who they say they are.

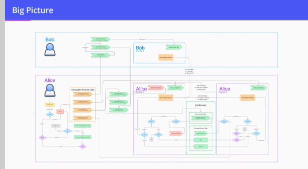
1. Now, let's discuss what Matrix even uses cryptography for. There are two main purposes...
2. The first is **message security**. We only want the people who are part of the conversation to be able to read the messages in the conversation.
3. As an additional benefit of how Matrix achieves this, encrypted messages **cannot be tampered with** by a **man-in-the-middle** actor without the receiving party knowing.
4. The second reason for using cryptography in Matrix is **identity** verification. We want to know that a specific device or user is who they say they are.
5. Note that one of the most important uses for identity verification is verifying your own devices so you can share keys with them.

# Big Picture



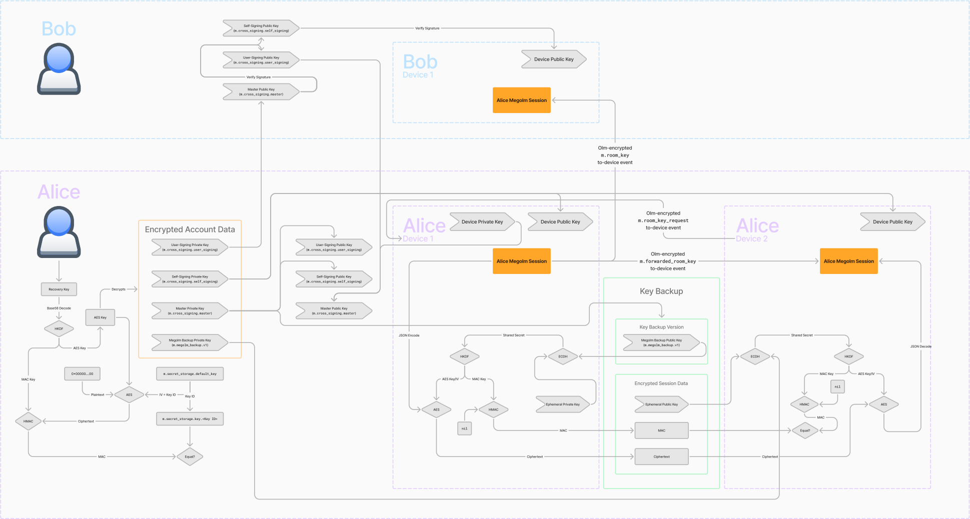




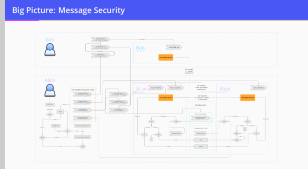
1. Let's take a look at what we are going to talk about today. This diagram shows **how those core features are implemented** as well as the infrastructure that supports them.
2. I know, it's pretty overwhelming. But don't worry, we are **going to go step-by-step** through this. By the end of the talk you should have an understanding of what each part of this diagram means.
3. It's probably too small to read, but I want to start by **orienting ourselves** to what's going on in this diagram, and we will zoom in later.
4. You can see that there are **two users represented** in the diagram: Bob on the top and Alice on the bottom. The diagram is **focused on how** the Megolm session created by Alice Device 1 is **shared** to Bob and to Alice's Device 2.
5. You'll notice that the diagram is color-coded.
   - Red nodes represent data that does not leave the device.
   - Green nodes represent data is public and can be shared with the server and other users.
   - Orange nodes represent data that can be shared with trusted parties, or with members of the same Matrix room.
   - Blue and purple nodes represent cryptographic operations.

# Big Picture: Message Security



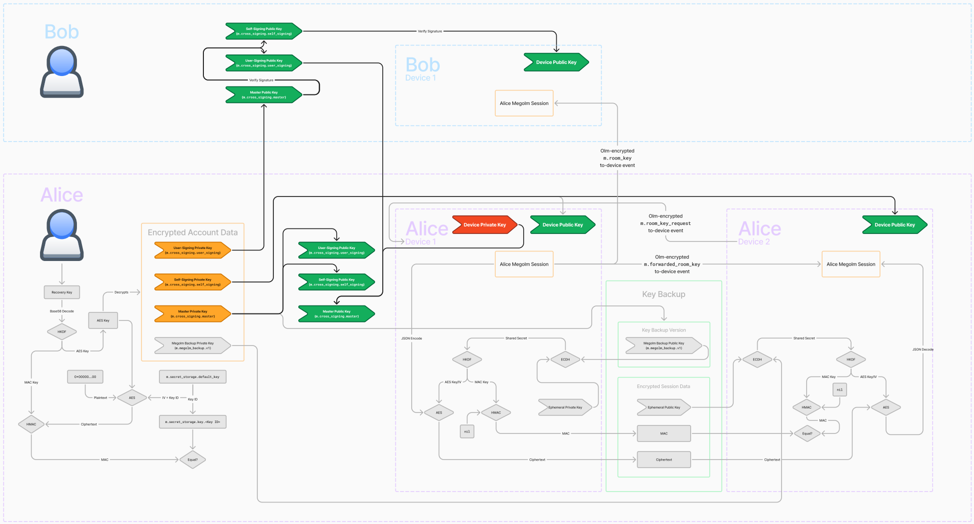




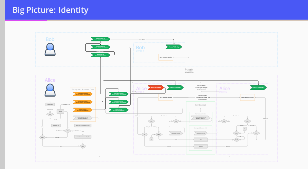
1. It's important that we **don't loose sight** of the reason for all of this infrastructure. Highlighted in **orange**, we have the **core of Matrix security: the Megolm session**.
2. We aren't going to discuss this in detail today. I **wrote an article** about Megolm which you can find on my blog if you want to learn more. I'll provide a link at the end of the talk.
3. For now, the only thing you need to know about it is that it's what is used to **encrypt and decrypt messages**.
4. The Megolm session needs to be **shared** with all the devices that Alice wants to be **able to read her messages**. So it needs to be shared to
   - the devices of **other users** in the Matrix room (in this case Bob)
   - as well as **her other devices**.
5. All of the rest of the infrastructure in this diagram is to facilitate transferring that Megolm session, or verifying that a device should in fact have access to that Megolm session.

# Big Picture: Identity



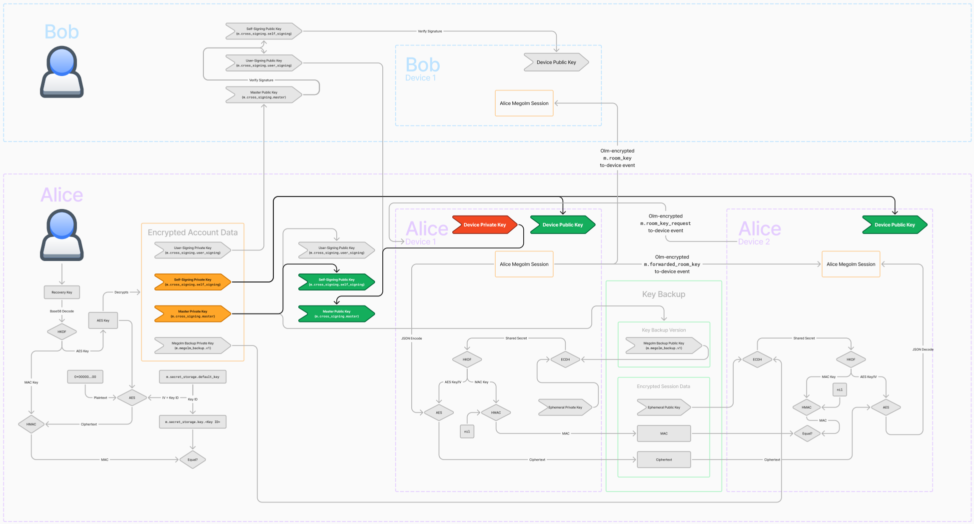




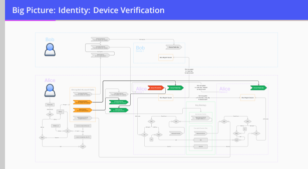

1. Let's move on to identity. The highlighted parts of the diagram provide a cryptographic way to verify that a device belongs to a particular user.
2. There are actually two pieces here...

# Big Picture: Identity: Device Verification



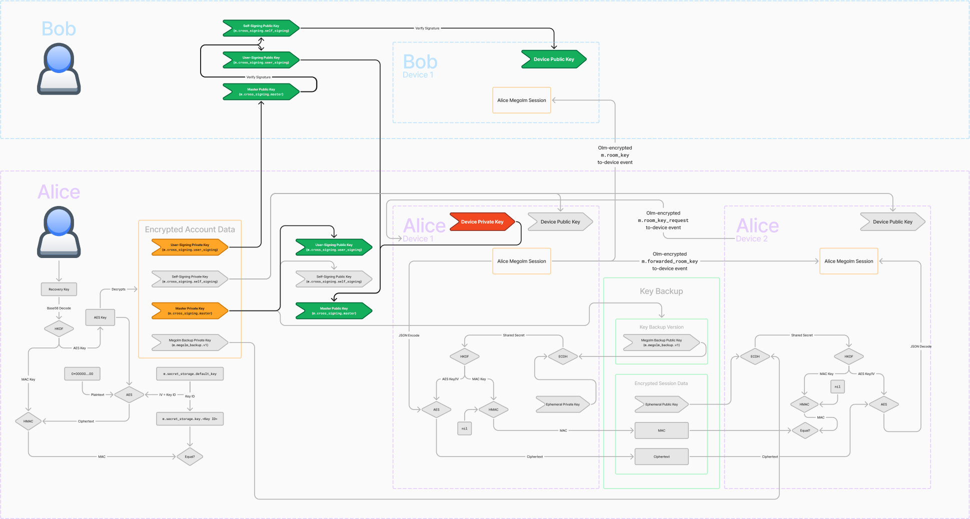




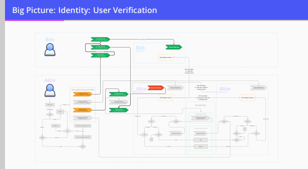




1. Here we have the infrastructure necessary for determining if we **trust another device for our own user**.

# Big Picture: Identity: User Verification

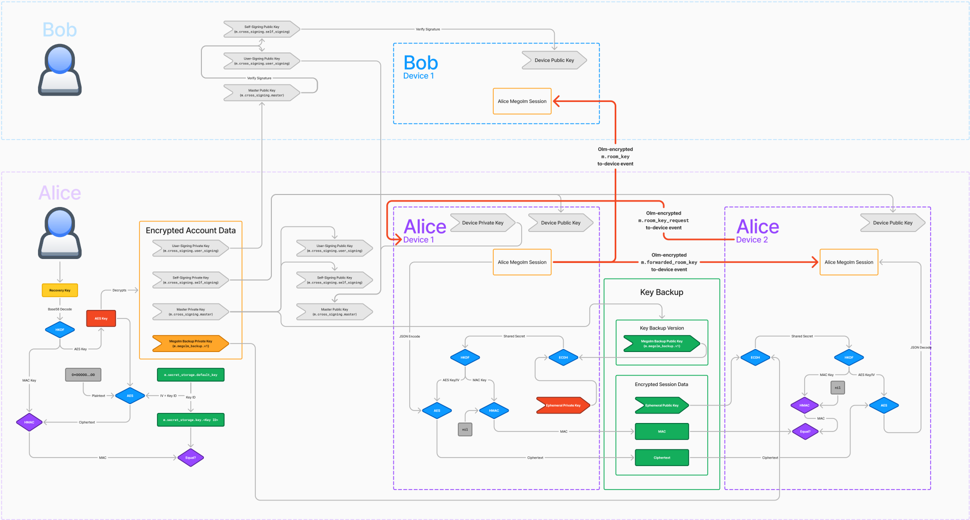




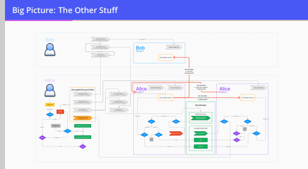
1. And here we have the infrastructure necessary for determining if we **trust another user and their devices**.

# Big Picture: The Other Stuff



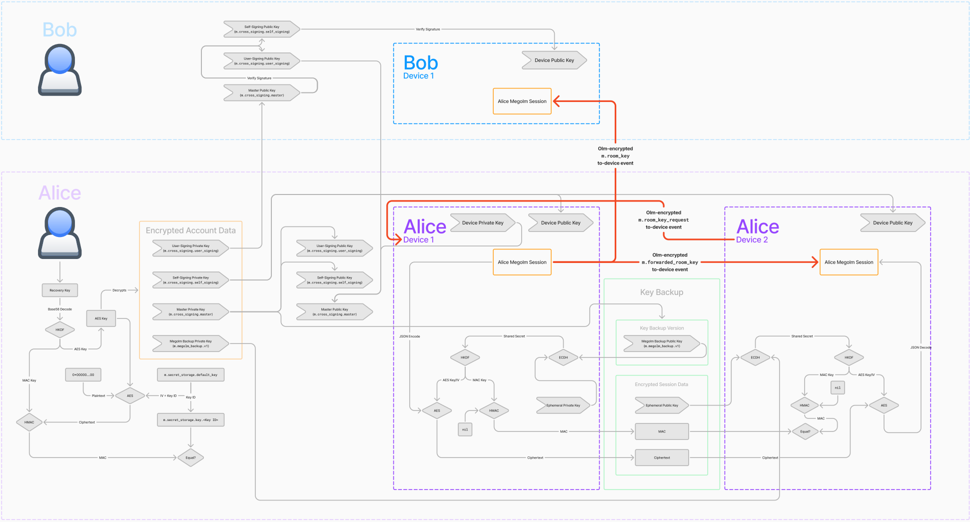




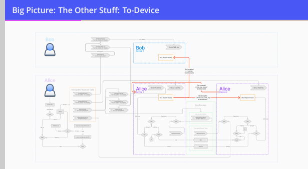

1. So what is all of the other stuff? That is the **infrastructure** for sharing the Megolm session around to other devices and users.

# Big Picture: The Other Stuff: To-Device



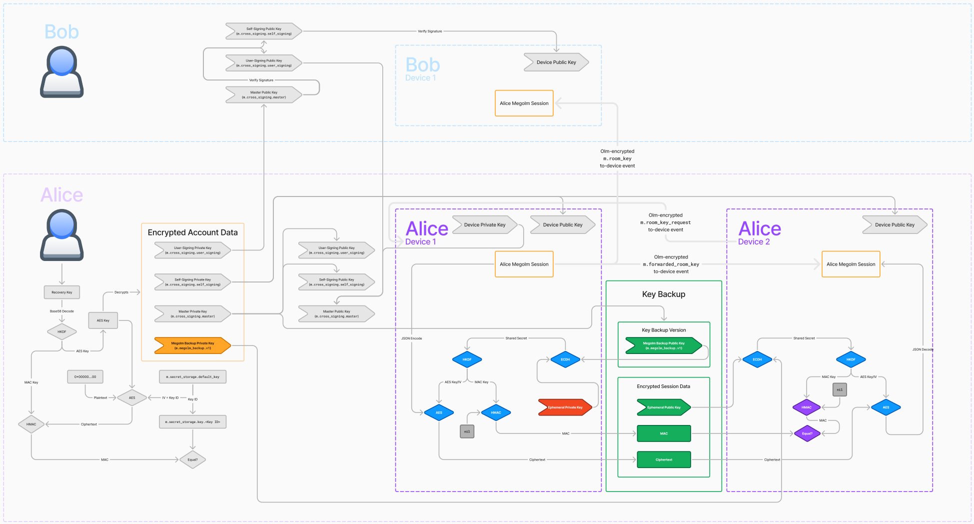




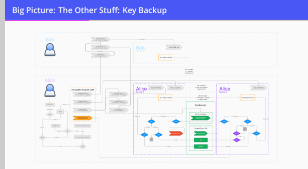
1. For example, in this arrow represents **sending** the Megolm session via **Olm-encrypted to-device messages**.

# Big Picture: The Other Stuff: Key Backup



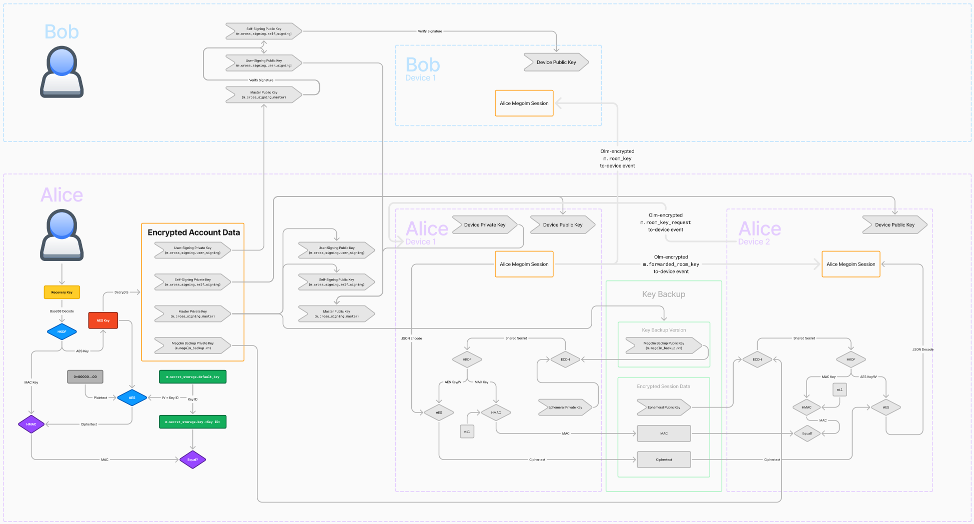




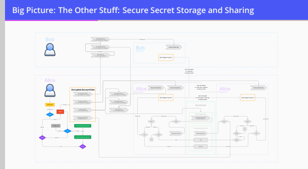

1. This **lower-right section** of the diagram represents **key backup** which allows you to **backup** your keys to the **server** and restore from your other devices.

# Big Picture: The Other Stuff: Secure Secret Storage and Sharing



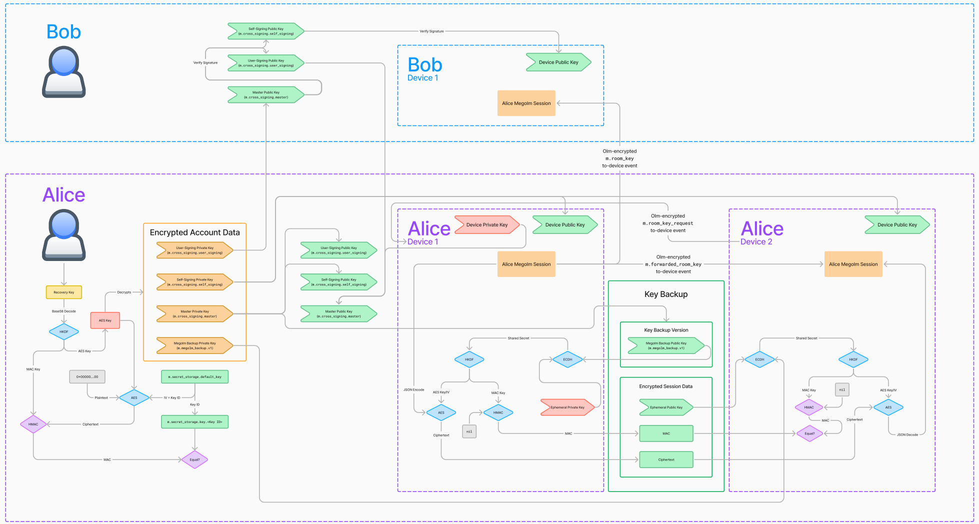




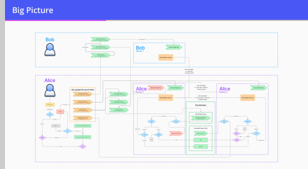
1. And over here on the **left** we have the infrastructure necessary for **storing secrets** on the **server** encrypted by a **recovery code**.

# Big Picture








1. So, that's a **quick overview** of this diagram.
2. **Before** we dive deeper into the **details** of the diagram, we need to **discuss** some basic **cryptography primitives**.
3. Then we will **break down** the diagram into manageable pieces.

# Cryptography Crash Course



Cryptography Crash Course

I will try and explain the cryptography primitives in **simple terms**. It's **not** going to be **mathematically rigorous**, but will focus on the **functionality** that each cryptographic primitive provides.

# Encryption: Symmetric vs Asymmetric

There are two main categories of encryption schemes:

- **Symmetric** — both **the encryptor and the decryptor share the same key** and that key is used in both the encryption and decryption of the message
- **Asymmetric** — **the encryptor needs the public key, and the decryptor needs the private key** and the encryptor encrypts the message with the public key, and the private key is required to decrypt the message





Encryption: Symmetric vs Asymmetric

There are two main categories of encryption schemes:

Symmetric — both the encryptor and the decryptor share the same key and that key is used in both the encryption and decryption of the message

Asymmetric — the encryptor needs the public key, and the decryptor needs the private key and the encryptor encrypts the message with the public key, and the private key is required to decrypt the message

1. Let's start with **encryption**. Encryption allows us to make a message that can only be read by another user who has the key.
2. Let's start with **encryption**. Encryption allows us to make a message that can only be read by another user who has the key.

# Encryption: Symmetric vs Asymmetric

There are two main categories of encryption schemes:

- **Symmetric** — both **the encryptor and the decryptor share the same key** and that key is used in both the encryption and decryption of the message
- **Asymmetric — the encryptor needs the public key, and the decryptor needs the private key** and the encryptor encrypts the message with the public key, and the private key is required to decrypt the message





Encryption: Symmetric vs Asymmetric
There are two main categories of encryption schemes:
- Symmetric — both the encryptor and the decryptor share the same key and that key is used in both the encryption and decryption of the message
- Asymmetric — the encryptor needs the public key, and the decryptor needs the private key and the encryptor encrypts the message with the public key, and the private key is required to decrypt the message

1. Matrix uses Advanced Encryption Scheme or AES for its symmetric encryption needs.

# Encryption: Symmetric vs Asymmetric

There are two main categories of encryption schemes:

- **Symmetric** — both **the encryptor and the decryptor share the same key** and that key is used in both the encryption and decryption of the message
- **Asymmetric** — **the encryptor needs the public key, and the decryptor needs the private key** and the encryptor encrypts the message with the public key, and the private key is required to decrypt the message





Encryption: Symmetric vs Asymmetric
There are two main categories of encryption schemes:
- Symmetric — both the encryptor and the decryptor share the same key and that key is used in both the encryption and decryption of the message
- Asymmetric — the encryptor needs the public key, and the decryptor needs the private key and the encryptor encrypts the message with the public key, and the private key is required to decrypt the message

1. There are a few variants of asymmetric encryption schemes. Matrix uses elliptic-curves for its asymmetric encryption needs.

# Asymmetric Signatures

In addition to providing encryption, asymmetric encryption schemes also provide **signatures**.

Signing uses the *private* key, and anyone who possesses the *public* key can verify the signature.





Asymmetric Signatures
In addition to providing encryption, asymmetric encryption schemes also provide **signatures**.
Signing uses the private key, and anyone who possesses the public key can verify the signature.

1. Only the private key can create a valid signature.
2. This is the opposite of encryption where we use the public key to encrypt, and the private key to decrypt.

# Asymmetric Signatures

In addition to providing encryption, asymmetric encryption schemes also provide **signatures**.

Signing uses the *private* key, and anyone who possesses the *public* key can verify the signature.





Asymmetric Signatures

In addition to providing encryption, asymmetric encryption schemes also provide **signatures**.

Signing uses the *private* key, and anyone who possesses the *public* key can verify the signature.

1. Only the private key can create a valid signature.
2. This is the opposite of encryption where we use the public key to encrypt, and the private key to decrypt.

# Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

**Given the same data, a hash function will always return the same output.**

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.





Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

1. *Read slide to last bullet*
2. If you **hash** the **same message** multiple times, you will receive the **same value**, and an attacker could use this information to deduce the **frequency** of certain **messages** being sent.
3. *Read last bullet*
4. **How** the **key** is **added** is an implementation detail that is **not relevant**. All you need to know is that HMAC prevents metadata attacks.

# Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

**Given the same data, a hash function will always return the same output.**

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.





Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

**Given the same data, a hash function will always return the same output.**

1. *Read slide to last bullet*
2. If you **hash** the **same message** multiple times, you will receive the **same value**, and an attacker could use this information to deduce the **frequency** of certain **messages** being sent.
3. *Read last bullet*
4. **How** the **key** is **added** is an implementation detail that is **not relevant**. All you need to know is that HMAC prevents metadata attacks.

# Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

**Given the same data, a hash function will always return the same output.**

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.





Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

**Given the same data, a hash function will always return the same output.**

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

1. *Read slide to last bullet*
2. If you **hash** the **same message** multiple times, you will receive the **same value**, and an attacker could use this information to deduce the **frequency** of certain **messages** being sent.
3. *Read last bullet*
4. **How** the **key** is **added** is an implementation detail that is **not relevant**. All you need to know is that HMAC prevents metadata attacks.

# Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

**Given the same data, a hash function will always return the same output.**

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.





Hashes and HMAC

A **cryptographic hash function** is a one-directional function which takes an arbitrarily large set of data and produces a unique fixed-size output (called the hash).

**Given the same data, a hash function will always return the same output.**

This allows us to verify that the data did not change in transit (for example, by a malicious actor).

Hashes are vulnerable to **metadata attacks**. To prevent these, we use HMAC which adds a secret key to the hash.

1. *Read slide to last bullet*
2. If you **hash** the **same message** multiple times, you will receive the **same value**, and an attacker could use this information to deduce the **frequency** of certain **messages** being sent.
3. *Read last bullet*
4. **How** the **key** is **added** is an implementation detail that is **not relevant**. All you need to know is that HMAC prevents metadata attacks.

# Key-Derivation Functions (HKDF)

Sometimes, we want to turn a small key into a larger key (or set of larger keys).

**Key-Derivation Functions (KDFs)** are used to do this.

Matrix uses HKDF which uses HMAC for the key derivation process.





Key-Derivation Functions (HKDF)
Sometimes, we want to turn a small key into a larger key (or set of larger keys).
**Key-Derivation Functions (KDFs)** are used to do this.
Matrix uses HKDF which uses HMAC for the key derivation process.

1. *Read first bullet*
2. For example, we might want to **“stretch”** a 32-byte **shared secret** into a key and IV for AES and a key for HMAC which would be 80 bytes in total.
3. *Read rest of slide*

# Key-Derivation Functions (HKDF)

Sometimes, we want to turn a small key into a larger key (or set of larger keys).

**Key-Derivation Functions (KDFs)** are used to do this.

Matrix uses HKDF which uses HMAC for the key derivation process.





Key-Derivation Functions (HKDF)

Sometimes, we want to turn a small key into a larger key (or set of larger keys).
**Key-Derivation Functions (KDFs)** are used to do this.
Matrix uses HKDF which uses HMAC for the key derivation process.

1. *Read first bullet*
2. For example, we might want to **“stretch”** a 32-byte **shared secret** into a key and IV for AES and a key for HMAC which would be 80 bytes in total.
3. *Read rest of slide*

# Diffie-Hellman Key Exchanges

Often, we need a way to share keys with both the sending and receiving parties across an unsecured channel.

**Diffie-Hellman** is a method for using public-key cryptography to facilitate keysharing.

$$\textbf{ECDH}(A_{private}, B_{public}) = \textbf{ECDH}(B_{private}, A_{public}) = K_{shared}.$$





Diffie-Hellman Key Exchanges
Often, we need a way to share keys with both the sending and receiving parties across an unsecured channel.
Diffie-Hellman is a method for using public-key cryptography to facilitate keysharing.
ECDH(Aprivate, Bpublic) = ECDH(Bprivate, Apublic) = Kshared.

1. *Read first bullet...* such as the internet.
2. We **can share** keys securely **in-person**, but that is very **impractical**.
   That's where the **Diffie-Hellman (DH) Key Exchange** method comes in.
3. *Read second bullet slide*
4. Since Matrix uses elliptic-curve cryptography, the **specific variant** of Diffie-Hellman that Matrix uses is ECDH (the elliptic curve variant).
   I'm not going to discuss the actual mathematical mechanism behind ECDH as it's quite complex and not relevant to understanding how Matrix uses ECDH. However, it is essential to understand the main feature it provides:
5. *Read equation*
6. In this equation, we have two public/private keypairs: *A* and *B*.
7. If we have **either one** of the **private** keys and the **other public key**, we can **generate** the same **shared secret**.
8. *Go through example of what to do if you have $A_{private}$ vs having $B_{private}$*
9. We will get the **same value** out of ECDH *regardless of which private key you have.* You only **need** the other **public** key, and those are **public** keys that can be spread around like butter.

# Diffie-Hellman Key Exchanges

Often, we need a way to share keys with both the sending and receiving parties across an unsecured channel.

**Diffie-Hellman** is a method for using public-key cryptography to facilitate keysharing.

$$\textbf{ECDH}(A_{private}, B_{public}) = \textbf{ECDH}(B_{private}, A_{public}) = K_{shared}.$$

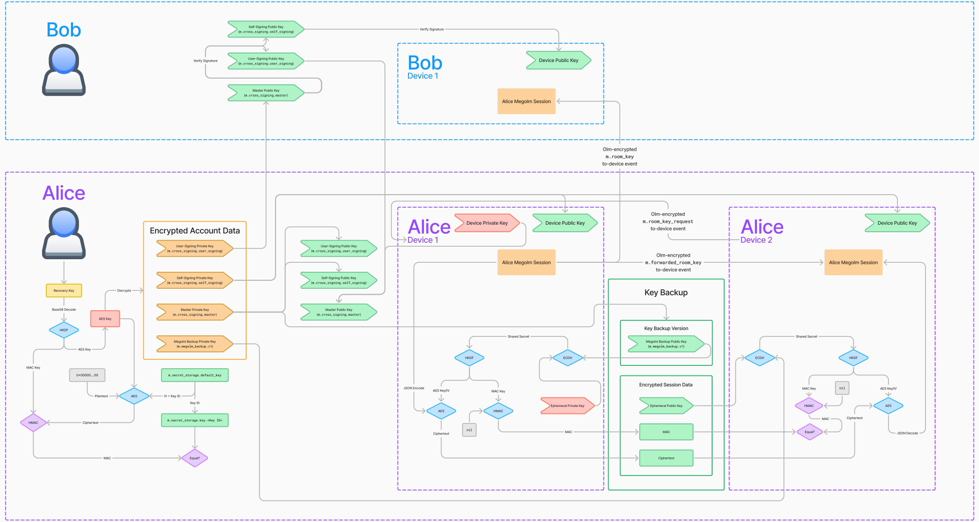




Diffie-Hellman Key Exchanges
Often, we need a way to share keys with both the sending and receiving parties across an unsecured channel.
Diffie-Hellman is a method for using public-key cryptography to facilitate keysharing.
ECDH(A_private, B_public) = ECDH(B_private, A_public) = K_shared.

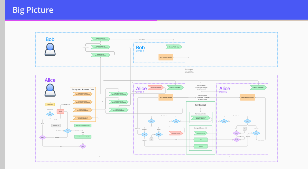
1. *Read first bullet*... such as the internet.
2. We **can share** keys securely **in-person**, but that is very **impractical**.
   That's where the **Diffie-Hellman (DH) Key Exchange** method comes in.
3. *Read second bullet slide*
4. Since Matrix uses elliptic-curve cryptography, the **specific variant** of Diffie-Hellman that Matrix uses is ECDH (the elliptic curve variant).
   I'm not going to discuss the actual mathematical mechanism behind ECDH as it's quite complex and not relevant to understanding how Matrix uses ECDH. However, it is essential to understand the main feature it provides:
5. *Read equation*
6. In this equation, we have two public/private keypairs: *A* and *B*.
7. If we have **either one** of the **private** keys and the **other public key**, we can **generate** the same **shared secret**.
8. *Go through example of what to do if you have* $A_{private}$ *vs having* $B_{private}$
9. We will get the **same value** out of ECDH *regardless of which private key you have*. You only **need** the other **public** key, and those are **public** keys that can be spread around like butter.

# Big Picture







1. Let's go **back to the big picture** now. Recall that the **blue** and **purple** nodes represent **cryptographic operations**.
2. All these **nodes** are one of the operations that we discussed.
3. *Point out a couple of HKDF, AES, ECDH, HMAC*
4. Now, let's discuss **how** these are **composed** together to provide **features** within Matrix.

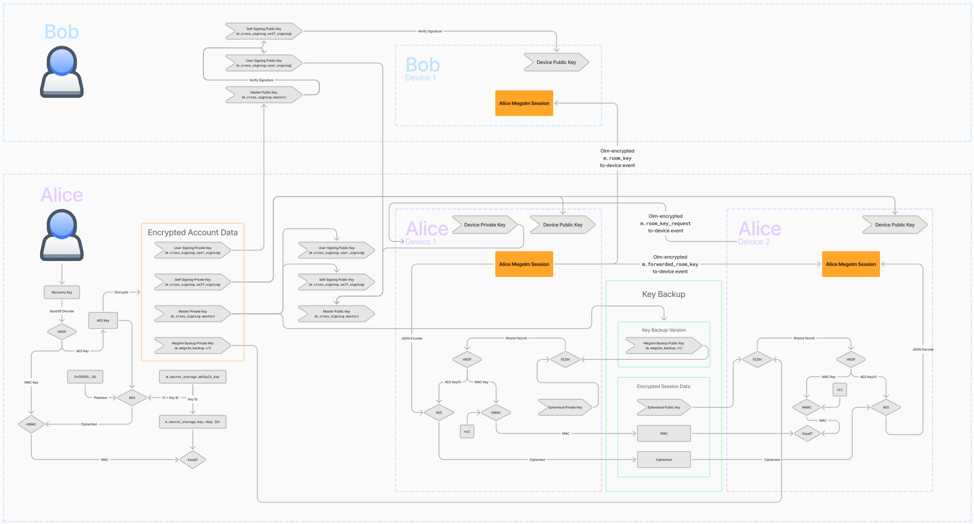
# Sharing Keys



Sharing Keys

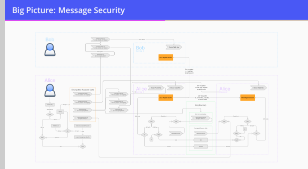
We’re going to start by discussing how we get **keys** from **one device to another**. This process is generally called **“key sharing”**.

# Big Picture: Message Security



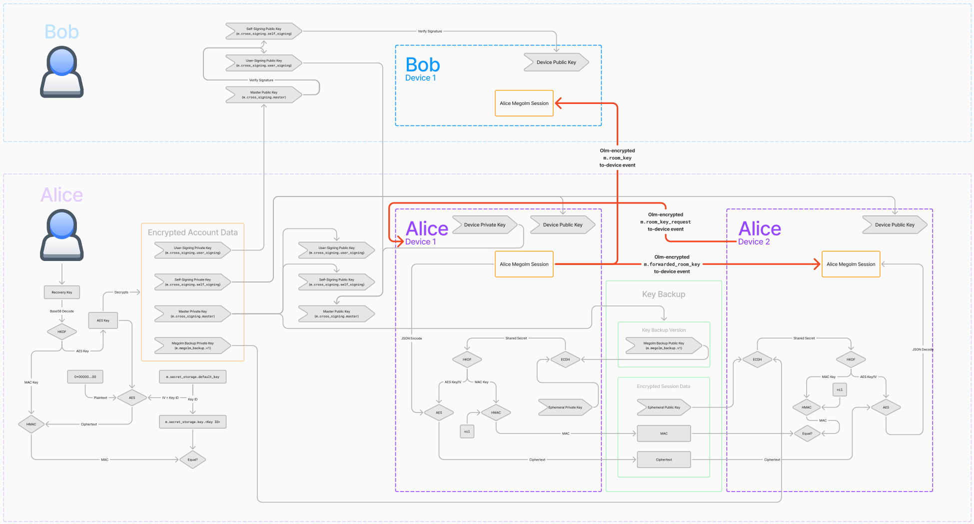





1. Remember, what we are **trying to share** is the **Megolm key** because that's what **encrypts and decrypts** the **messages**.
2. There are **two** ways to share these: **encrypted olm events** and **key backup**.

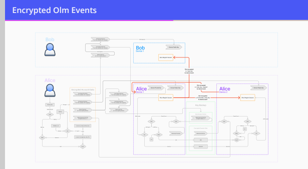
# Encrypted Olm Events





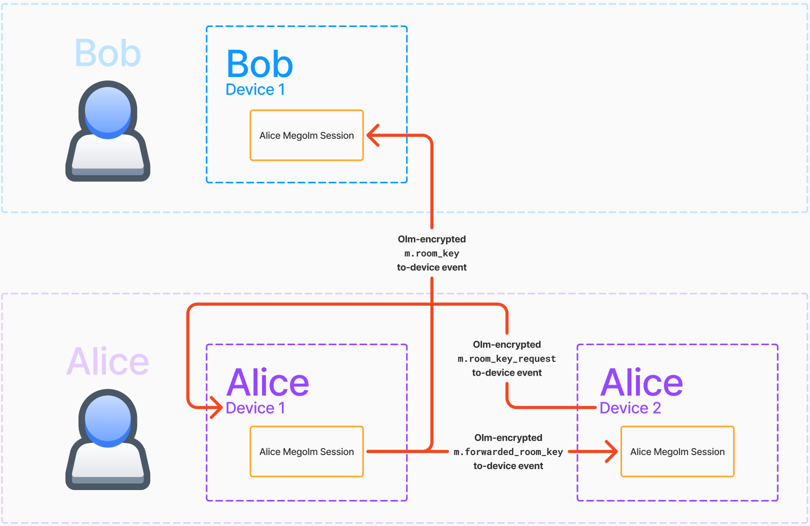

2024-09-21

Matrix Cryptographic Key Infrastructure
└Sharing Keys
└Encrypted Olm Events



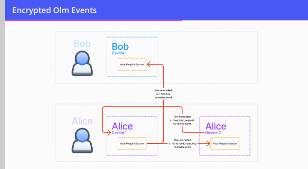
1. **Encrypted Olm events** are represented by the **arrows** highlighted in **red**. They are sent via **to-device** messages which allow you to send messages to specific devices (rather than rooms).
2. Let's zoom in to see what's going on.

# Encrypted Olm Events



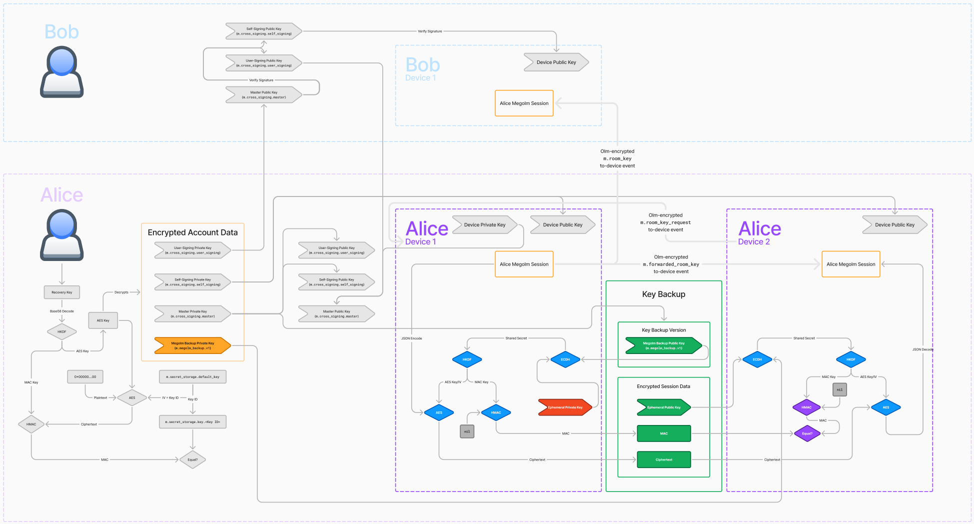




1. I got rid of all the irrelevant nodes
2. You can see that we can **send** keys to **other users' devices** via `m.room_key` events.
3. And actually we use `m.room_key` events to send keys to our **own devices** as well.
4. We can also **request keys** by sending `m.room_key_request` events to our **own verified devices** and the other devices can respond using `m.forwarded_room_key` events. We will talk about how we know a device is verified later.
5. I'm not going to discuss how Olm encryption works. It's already been covered many times since it's basically just the Signal double-ratchet algorithm.
6. For our purposes, it's **sufficient** to know that we can **send keys securely** to **other users' devices** and our **own devices** via these events.
7. This seems great, why do we have anything else?
8. Well, **new logins** are the issue. Say Alice just **logged in** on **Device 2** and finished verification.
   - If Device 1 is *online*, she can send key requests to Device 1 and Device 1 can respond.
     This **works**, but there will likely be a **lot of keys** to request. **Every user** in **every encrypted room** has diffrent keys.
     This will make **Device 1** do a **lot of work** to send back all the keys.
     On **mobile devices**, keysharing **can't** really be done in the **background**, **especially** on **iOS**.
     Even on **desktop** devices, it's still a lot of **work** to **process** a **flood** of key **requests**.
   - But it's **even worse** if Device 1 is *offline*. In that case, Alice's key **requests** will **never be answered**.

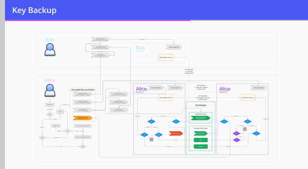
# Key Backup





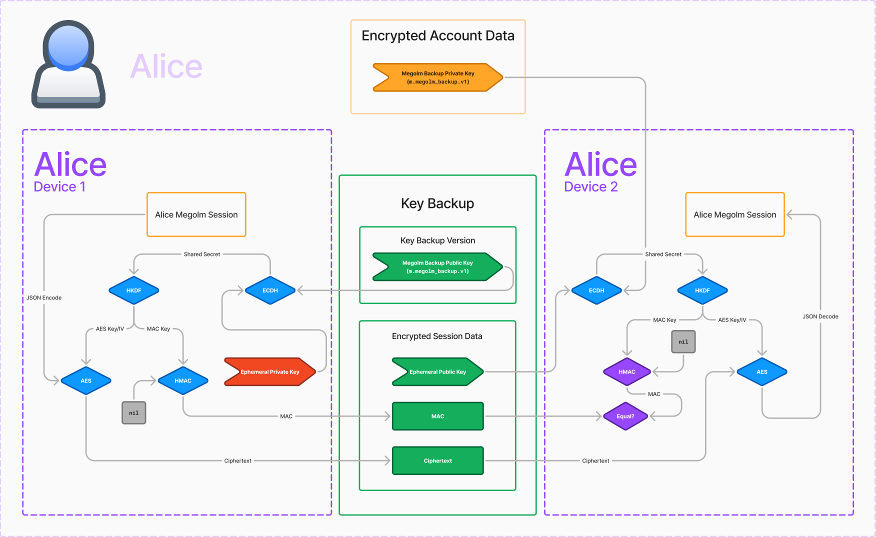

2024-09-21

Matrix Cryptographic Key Infrastructure
└ Sharing Keys
└ Key Backup



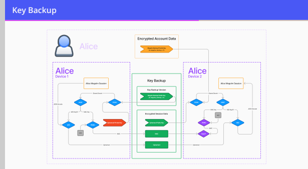

1. This ss where **key backup** comes into play. Key backup allows us to **store keys** on the **server**, and **restore** them from our **other devices** even if your **other devices** are **offline** or **inaccessible**.
2. Let's zoom in and see what's going on.

# Key Backup



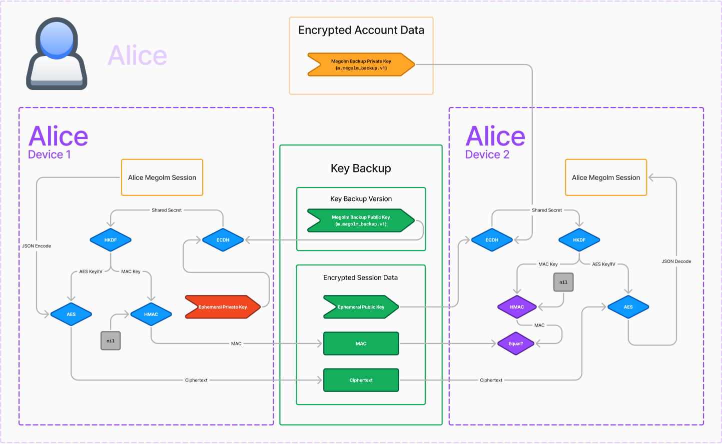




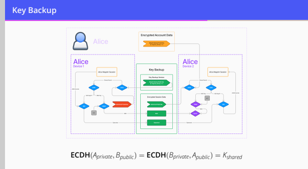
1. In the **middle** here we have the “key backup” in **green**. Key backup is is **stored** on the **server**.
2. In this diagram, we’re trying to get the Megolm key **from** Alice’s **Device 1** to her **Device 2**, so **left to right**.
3. There are **two pieces** to key backup:
   – the **key backup version** which includes the **backup public key**.
   – the **encrypted session data** for **each** of the backed-up Megolm **sessions**.
4. Let’s discuss how this works.
5. The **first thing** to note is that **AES** is used on **both sides** to **encrypt** and **decrypt** the **Megolm session**. **Only** the **encrypted** version is **stored** on the **server**.
6. But, **AES** needs a **key** and **initial vector**. Where do we get that from? Well, we **get it from HKDF**.
7. HKDF **requires a key as well**, so where do we get that from?
8. That comes from a call to ECDH.
9. Note that **everything so far** is **the same on both the encrypting and decrypting sides!**
10. Recall that **ECDH** requires a **private key**, and the **other public key**.

# Key Backup



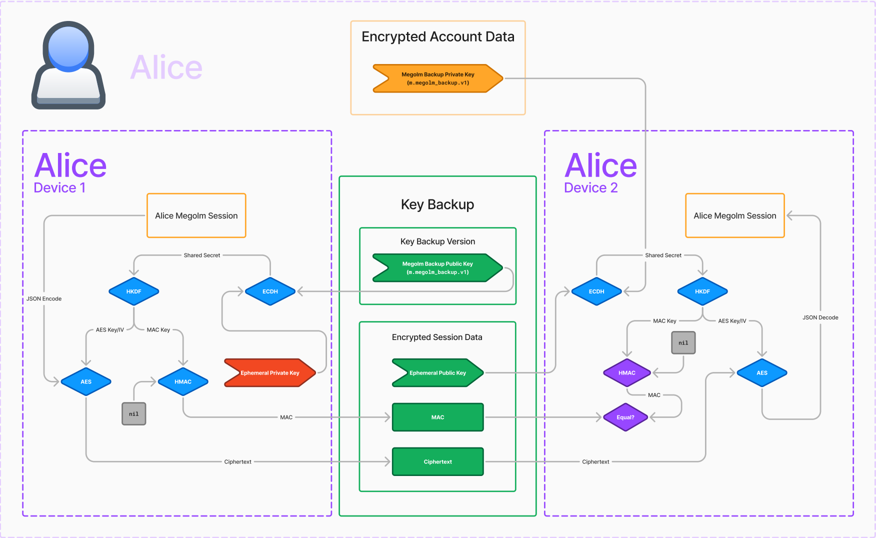

$$\mathbf{ECDH}(A_{private}, B_{public}) = \mathbf{ECDH}(B_{private}, A_{public}) = K_{shared}$$





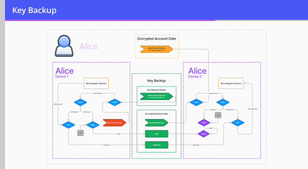
1. Remember this formula? If you have **either private key**, we just **need** the **other public key** to get the same value from ECDH.
2. So where do we get the *A* and *B* keypairs from?
   - The **first keypair** is the **Megolm backup keypair**. The **public key** is stored in the **key backup version**. The **private key** is a secret stored on the **user's device**.
   - The **second keypair** is the **ephemeral keypair**. A **new keypair** gets created for **each** backed up session. It's **ephemeral because** the **private** part is be **discarded** immediately **after the encryption** is done. The **public key** is stored in the **encrypted session data**.
3. This is where the **sides diverge**.
4. 
   - The **encrypting** side gets its **private key** from the *ephemeral keypair*.
     And it uses the **Megolm backup public key** as its public key.
   - The **decrypting** side gets its **private key** from the **Megolm backup private key**.
     And it uses the **ephemeral public key** as its public key.
5. Critically you **must have** the **Megolm backup private key** to **decrypt** the key **backup**.
6. For **each Megolm session** that we **back up** in key backup, we **store** the **ephemeral public key** and the **ciphertext** from AES together in the **encrypted session data** object.

# Key Backup







1. But there's **another item** that we store in this object: the **MAC**.
2. A **MAC** is a **Message Authentication Code**. It's basically just a **hash** of the **ciphertext** that we use to **verify** that it hasn't been **tampered** with by a **malicious** or just straight up **incompetent** party.
3. We use **HMAC** to **generate the MAC** and **avoid metadata attacks**. Recall that HMAC requires a key.
4. Conviniently, we are **already using HKDF** to generate the **AES key and initial vector**, so we can just use the **same key derivation** to get the **HMAC key**.
5. What *should* happen is that we **pass** the **ciphertext** into **HMAC**. However, the original implementation in **libolm failed to do this** correctly and instead just passed an **empty buffer**, and it has been de-facto spec ever since.
6. So, the **MAC** is **not really useful** at all in its **current state**. I'm hoping that a future version of the spec fixes this.

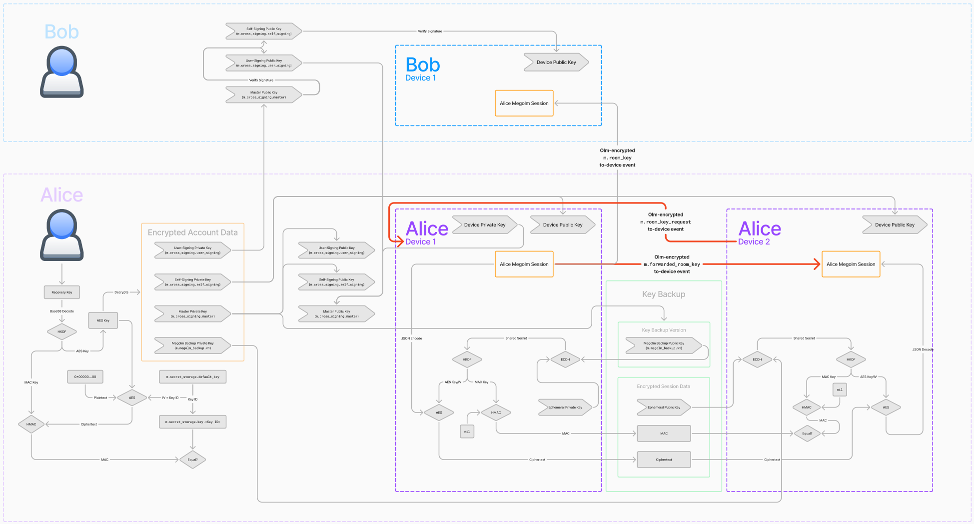
# Device Verification





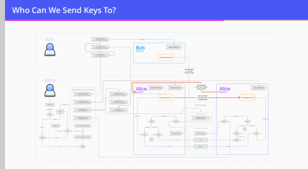
Now, let's discuss **device verification**.

# Who Can We Send Keys To?



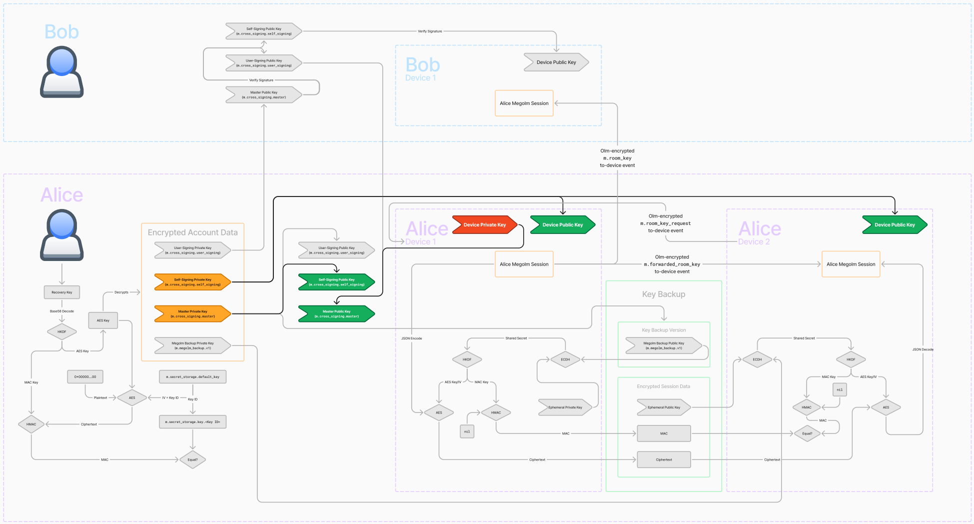




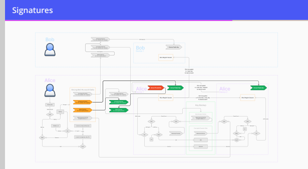
1. Let's go back to the big picture and notice these **arrows** that represent **requesting keys** from our **own devices** and **forwarding** them back.
2. Earlier I said that we **only** want to **forward** keys to our **verified** devices?
3. Now, we are going to discuss how **verification status** is determined.
4. The answer is...

# Signatures



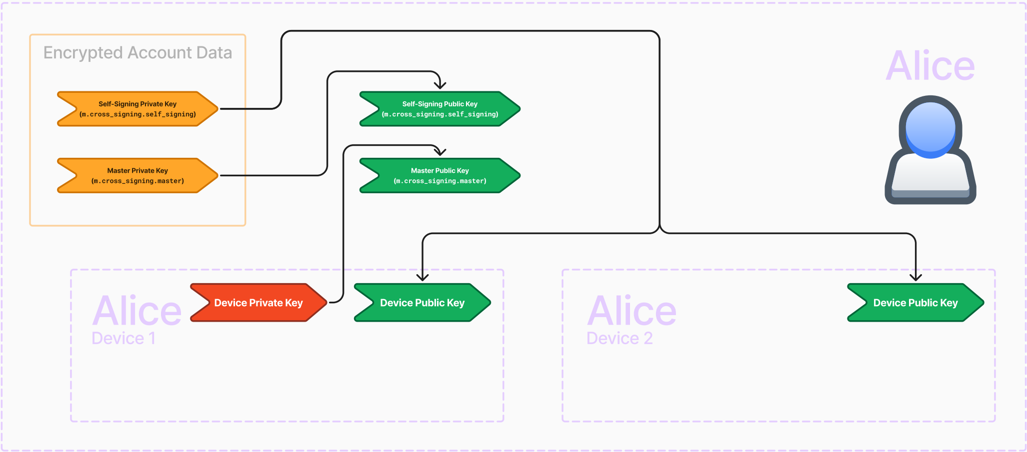





1. **Signatures!**
2. Let's zoom in on this part.

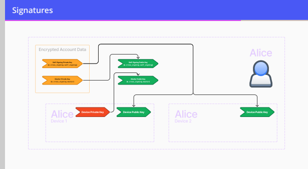
# Signatures






2024-09-21

Matrix Cryptographic Key Infrastructure
└ Device Verification
└ Signatures



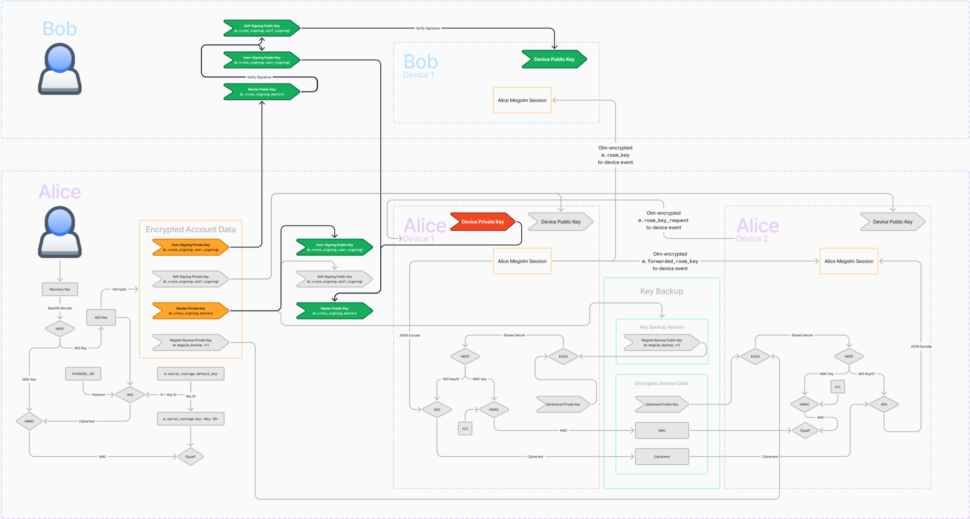
1. Remember, **asymmetric signatures** can only be **created** by the **private** key, and anyone who possesses the **public** key can **verify** the signature.
2. In Matrix, each device has a **device keypair**. The **public** key is an **identifier** for the device.
3. To **verify** a device, we **sign** the **device public key**.
4. Often, we call this process **trusting** a key. We trust the key by **creating a signature** for it.
5. We **can** use our own **device private key** to **directly trust** the **other device key**.
6. But that is **inconvenient**. When we **log in** a **new device**, all our **other devices** will need to **make a signature for the new device**, and the **new device** will have to **make a signature for all the existing devices**!
7. So, we introduce a **new user-wide key** called the **"self-signing key"** because it **signs** our **own devices**.
8. We use the **self-signing** key to **sign the device keys** but **how** do we know if we should **trust** the **self-signing key?**
9. That's where the **master key** comes in. The master key **signs** the **self-signing key**.
10. We then **trust** the master key by **signing** it with our **device private key**.
11. This creates a **chain of trust**.
    The **device private key** signs the **master public key** which corresponds to the **master private key**.
    The **master private key** signs the **self-signing public key** which corresponds to the **self-signing private key**.
    And the **self-signing private key** signs **all** of the **device public keys**.
12. This allows us to **trust** a **single key** (in this case, the **master key**) and then through the **chain of trust**, we can trust **all** of our **own devices**.

# User Verification



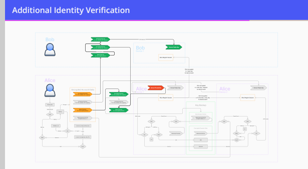
Sometimes, we want to **trust a user** so that we know that all of the **devices on their account** are **under their control**.

# Additional Identity Verification



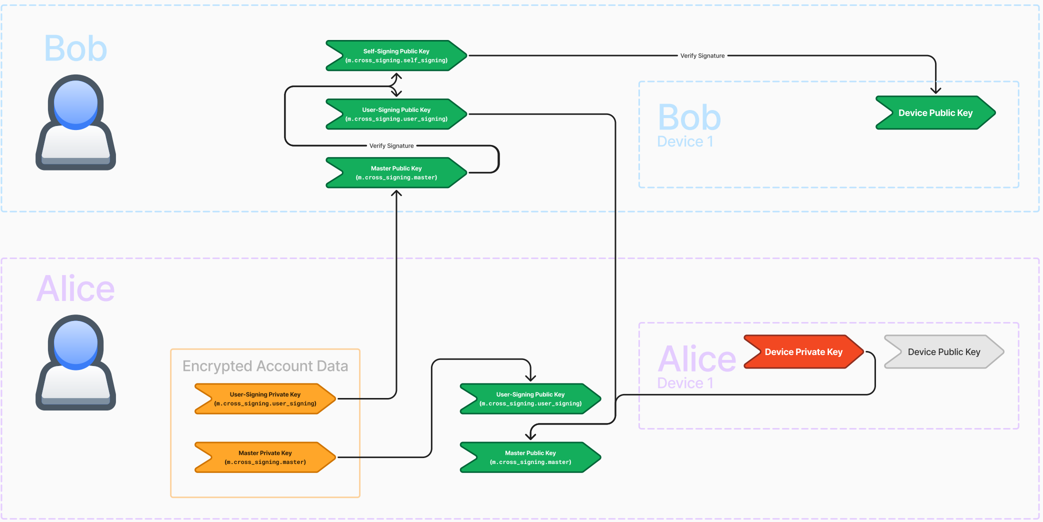




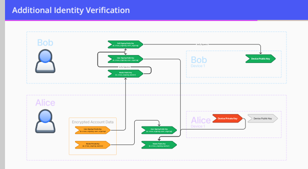
1. If a **new device is logged in**, we will know if they control the device if they have **signed** it.
2. If some **malicious actor** logged in a new device, they would **not** be able to **sign** it, and we would know the **other user** has been **compromised**.

# Additional Identity Verification







1. The user we want to trust has **already signed** their devices with their **self-signing key**, which is itself **signed** by their **master key**. So, if we are able to **trust their master key**, we will have a **chain of trust** to all of their devices.
2. This is where the **user-signing key** comes into play. The user-signing key **signs other users' master keys** and is itself **signed** by our **own master key**.
3. This creates another **chain of trust**.
   The **device private key** signs the **master public key** which corresponds to the **master private key**.
   The **master private key** signs the **user-signing public key** which corresponds to the **user-signing private key**.
   The **user-signing private key** signs other users' **master public keys**.
   We can **verify** the signatures by the other user's **master private key** using their **master public key**.
   Since their **master private key** signed their **self-signing public key**, we can **verify** the signature and **trust** their **self-signing key**.
   Since their **self-signing private key** signed their **device public keys**, we can **verify** the signatures and **trust** their **device keys**.

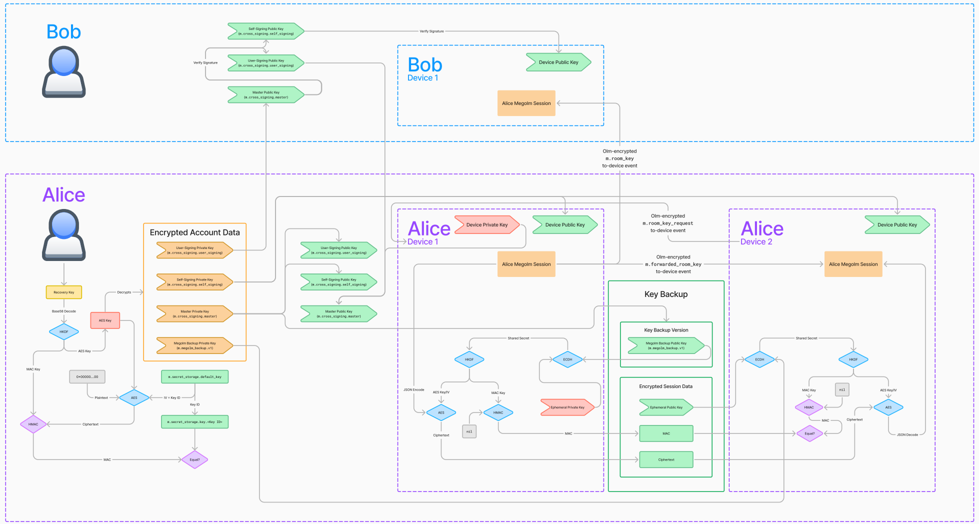
# Secure Secret Storage and Sharing (SSSS)



Secure Secret Storage and Sharing (SSSS)

Wow, that’s a lot of keys! Where are they stored?

# Don’t Forget Your Keys



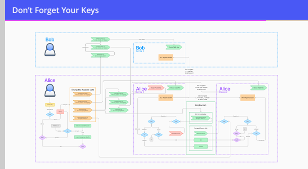



2024-09-21

Matrix Cryptographic Key Infrastructure

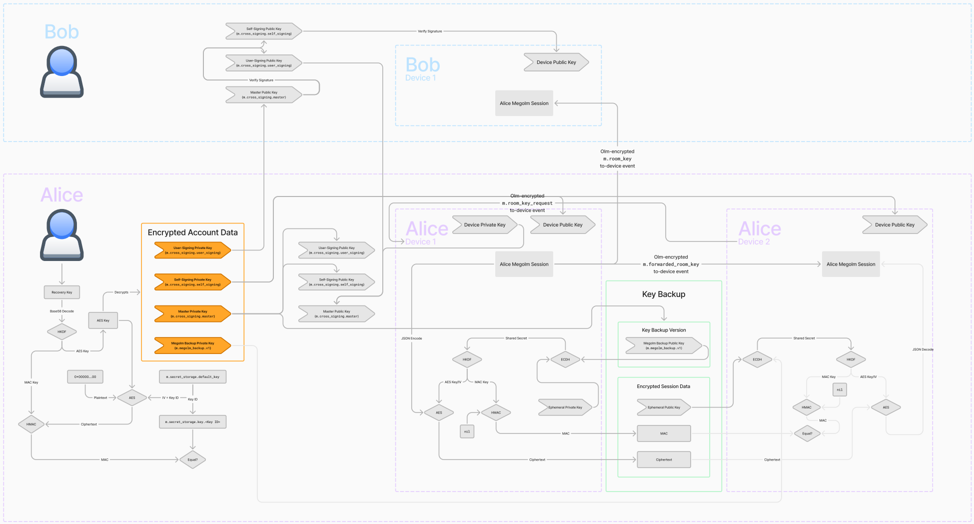
└ Secure Secret Storage and Sharing (SSSS)

└ Don’t Forget Your Keys



1. The **public keys** can be stored on the **server**.
2. However, the **private keys** need to remain in the user’s control.

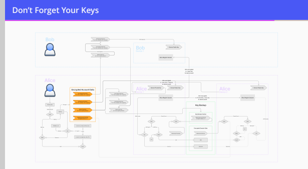
# Don’t Forget Your Keys





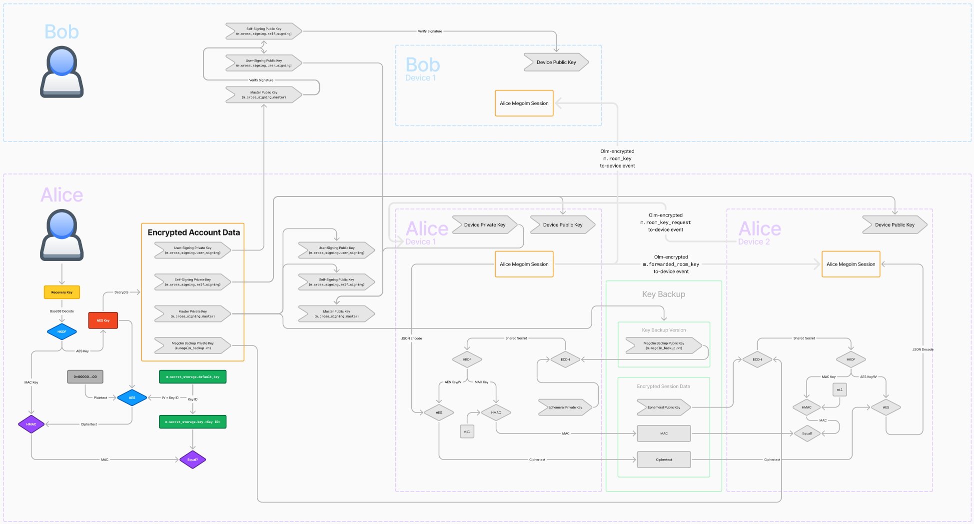

2024-09-21

Matrix Cryptographic Key Infrastructure
└ Secure Secret Storage and Sharing (SSSS)
└ Don’t Forget Your Keys




1. Today, we’ve seen private keys for **key backup**, **user signing**, and **device signing** as well as the **master key**.
2. These keys are **stored** on **each** of **your devices** and can be **shared** with your **other verified devices** using those Olm-encrypted **to-device** events.
3. But what if you **sign out** of all of your devices or **lose access** to them?

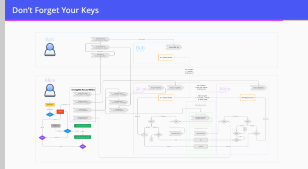
# Don’t Forget Your Keys





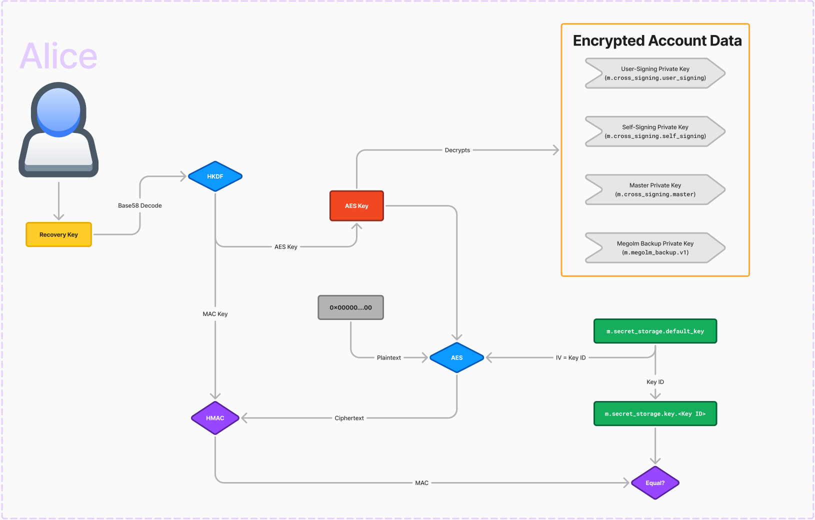

2024-09-21

Matrix Cryptographic Key Infrastructure
└ Secure Secret Storage and Sharing (SSSS)
└ Don’t Forget Your Keys



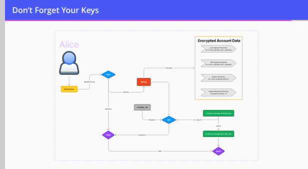
1. That’s where **secure secret storage and sharing** (also known as **SSSS**, or **quadruple S**) comes in.
2. It allows you to **store** your keys **encrypted** within **account data** on the **server**.
3. Let’s zoom in and see what’s going on.

# Don’t Forget Your Keys



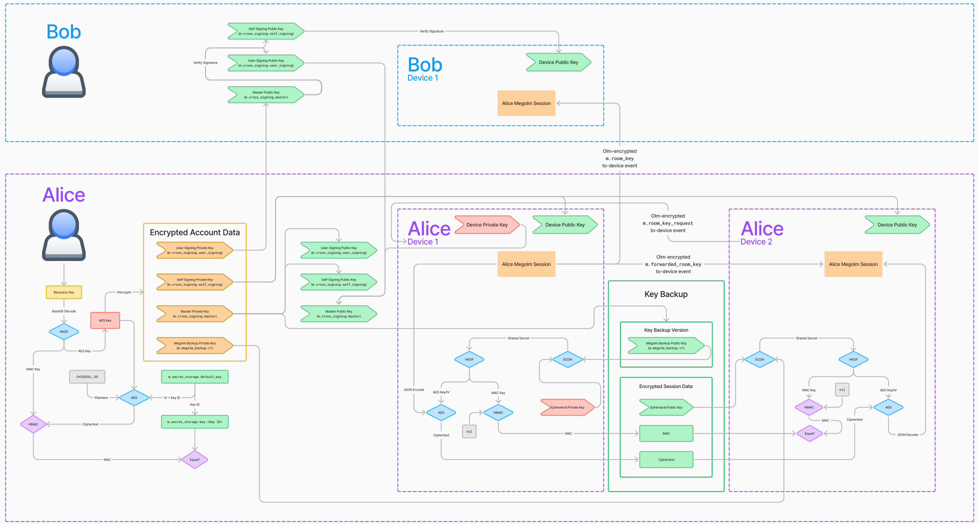




1. The **key** that SSSS uses to **encrypt** the account data is effectively the **recovery key**.
2. There is a **base58-decode** and an **HKDF** transformation which **produces** the **actual key**, but it’s basically just your **recovery key** that **unlocks** the **encrypted account data**.
3. So, you can probably see that if you **lose your recovery key**, and you have **no signed-in devices**, there is **no way to recover the private keys**. This is why it’s important to **store the recovery key in a safe place like a password manager**.
4. So, what’s this **bottom part?** It’s **not** actually **strictly necessary** for the **encryption**, but it allows you to **verify** that your **recovery key** is **correct** before trying to decrypt account data. If you want the details here, you can **read the blog post** associated with this talk.

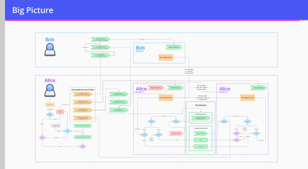
# Big Picture






2024-09-21

Matrix Cryptographic Key Infrastructure
└ Secure Secret Storage and Sharing (SSSS)
└ Big Picture



1. Let's go back once more to the overview.
2. We've talked about each piece of this diagram.
   - We talked about the **Megolm session**
   - We talked about **to-device events**
   - We talked about **key backup**
   - We talked about **self-signing of devices**
   - We talked about **signing of other users**
   - And then we talked about **secure secret storage and sharing**
3. I hope that this presentation has helped you **understand** how it **fits together**.
4. My **goal** is to **convince** people that Matrix **cryptography is not scary**. It's **complex**, but not **inaccessible**.
5. If you have **access** to all of the **underlying cryptography primitives**, all of this is something that a **security-conscious programmer** could **implement**.
6. You almost certainly **should not** implement the **cryptography primitives** yourself, but **composing** them together is doable.

# Thank You for Listening!

Questions?

sumnerevans.com/posts/matrix/cryptographic-key-infrastructure






1. And with that, I'd like to **thank you for listening!**
2. You can **scan the QR code** for the **blog post** associated with this talk. The **slides** are **available to download** there.
3. *If there is time:* I believe we have a few minutes for any questions you may have.
4. *If there is **not** time:* It looks like we don't have time for questions, but I'm happy to talk to you off-stage about any questions you have.