Where Are My Keys?

A Survey of Matrix Cryptographic Key Infrastructure

Sumner Evans

21 September 2024

Beeper (Automattic)

Where Are My Keys?

A Survey of 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 and today I'm going to be talking about cryptographic key infrastructure in Matrix.
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. I mainly blame the other chat networks for their incompetence. Most other chat networks don't even provide any cryptographically-guaranteed security and privacy. Some networks provide encryption in a way that does not truly leave the user in control of their keys. Only a few networks (Signal) truly leave the user in control, and their UX is arguably worse than Matrix.
4. In this talk, my goal is to discuss the cryptographic key *infrastructure* in Matrix. 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. 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 E2EE 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.

1. 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.
2. Note that one of the most important parts of identity this is verifying that our own devices are under our own control and we should allow our own clients to share keys to it.

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. 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.
2. Note that one of the most important parts of identity this is verifying that our own devices are under our own control and we should allow our own clients to share keys to it.

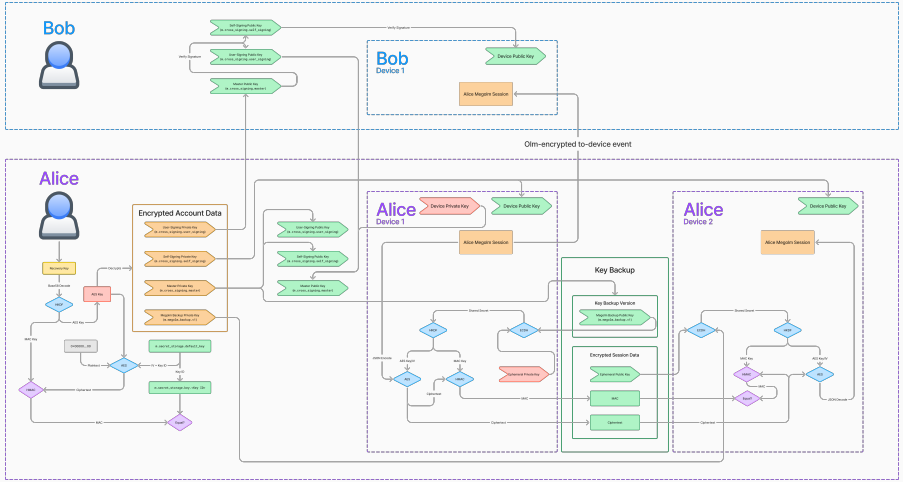
Why Cryptography?

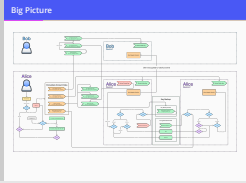
Matrix uses cryptography for two main purposes:

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. 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.
2. Note that one of the most important parts of identity this is verifying that our own devices are under our own control and we should allow our own clients to share keys to it.

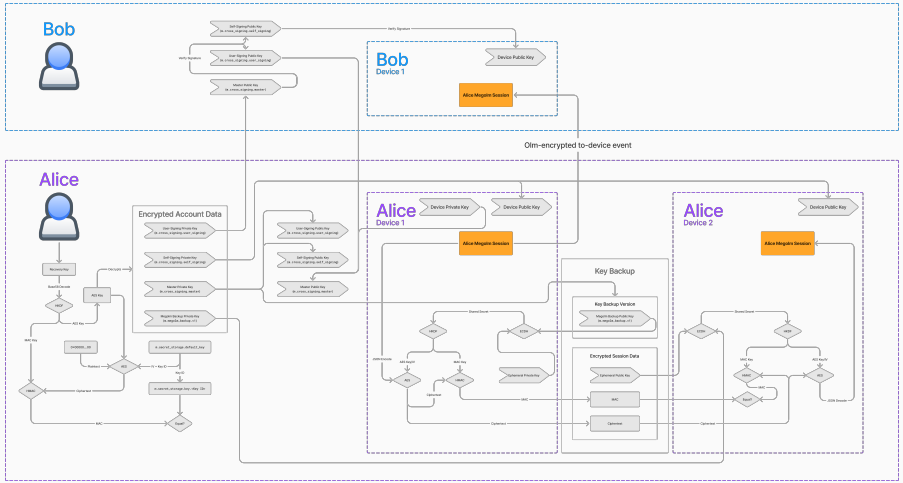
Big Picture

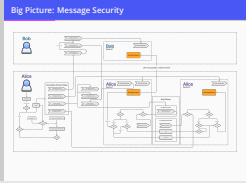




1. This diagram represents all of the infrastructure in Matrix for providing those core features of message security and identity verification.
2. I know, it's pretty overwhelming. But don't worry, we are going to go step-by-step through this, at the end of the talk you should have an understanding of what each part of this diagram means.
3. Let's start by orienting ourselves to the big picture of this diagram, then we will take a short detour into a few core cryptography concepts required to understand the diagram, and then we will break down the diagram into manageable pieces. And at the end of the talk hopefully you have a complete understanding of Matrix cryptographic key infrastructure.
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. 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.

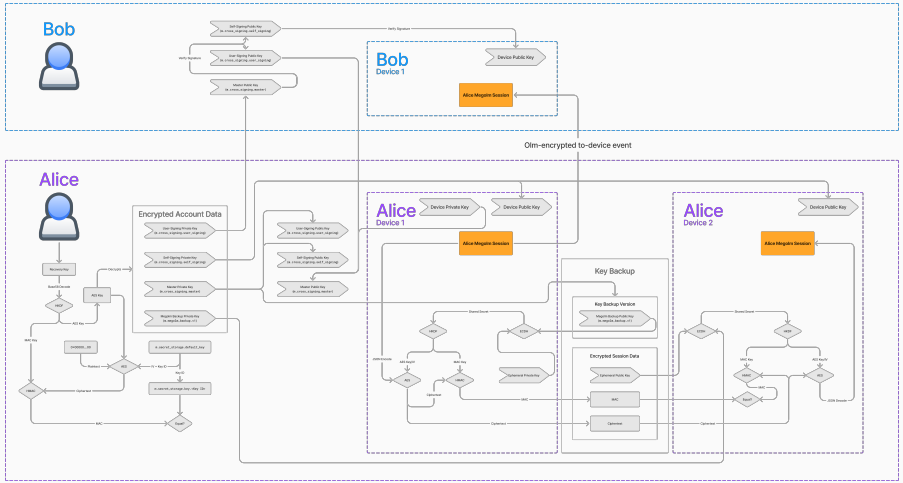
Big Picture: Message Security

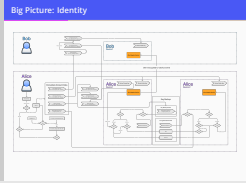




1. It's important that we don't loose sight of the reason for all of this infrastructure. In orange, we have the core of Matrix security: the Megolm session.
2. We aren't going to discuss this in detail today. I wrote an article about Megolm which you can find on my blog if you want to learn more. I'll provide a link at the end of the talk.
3. But for now, the only thing you need to know about it is that it provides end-to-end encryption for messages and needs to be shared with 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 same Matrix room (in this case Bob)
   - but her device also needs to share the session with her other devices.
4. 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.

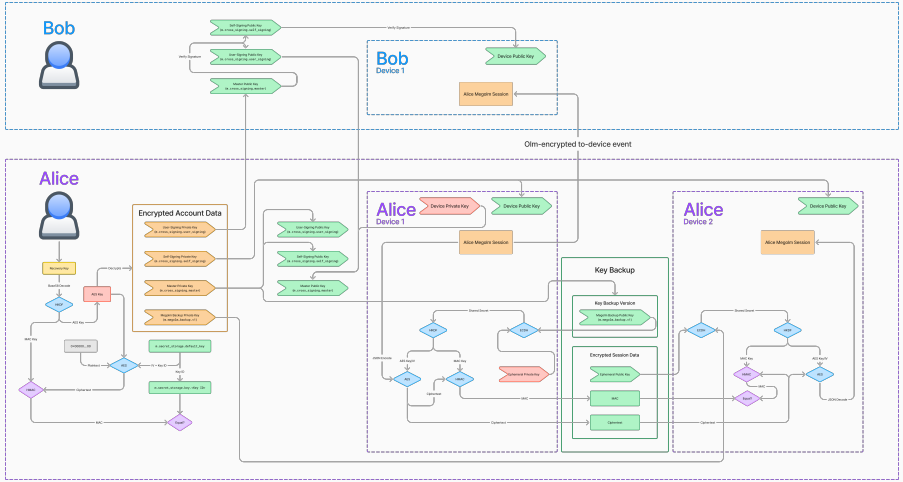
Big Picture: Identity

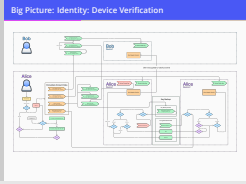




1. Let's move on to identity. The highlighted parts of the diagram provide a cryptographic way to verify that a device belongs to a particular user.
2. There are actually two pieces here...

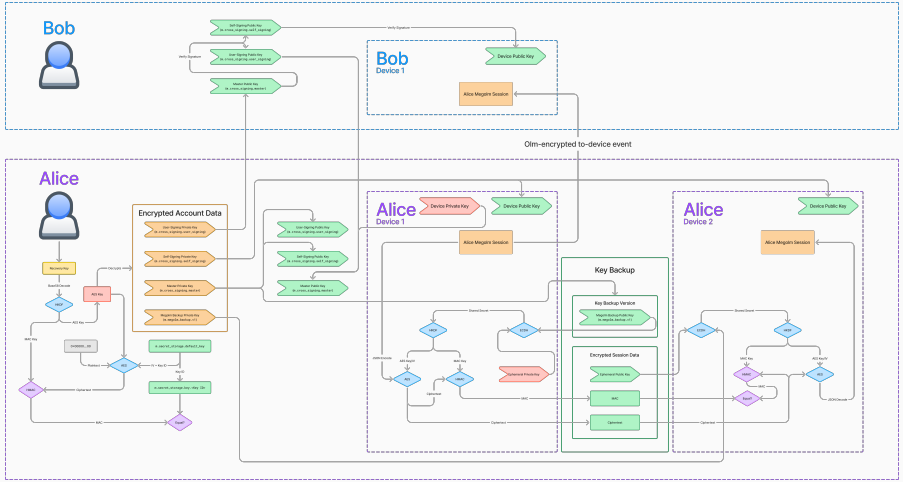
Big Picture: Identity: Device Verification

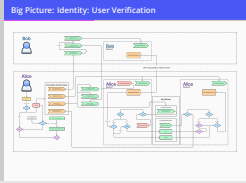




1. And here we have the infrastructure necessary for determining if we trust another device for our own user.

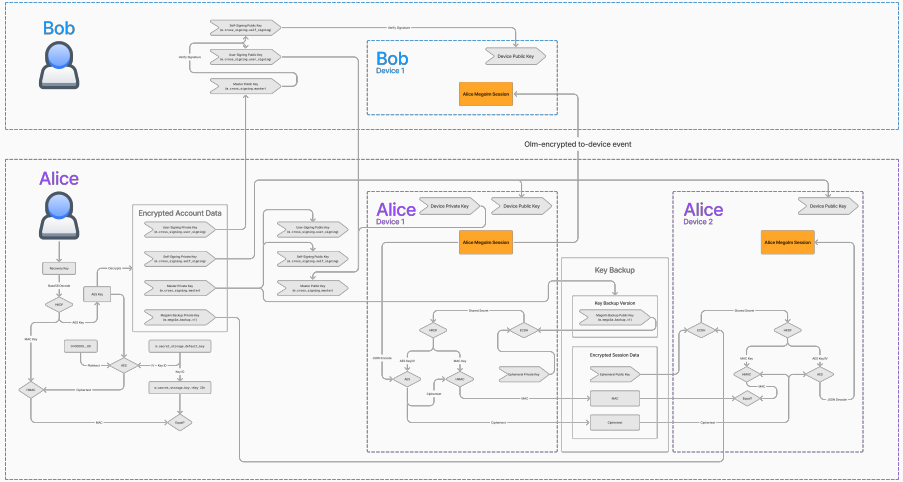
Big Picture: Identity: User Verification

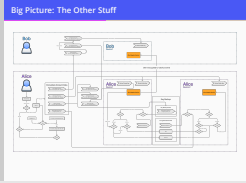




1. And here we have the infrastructure necessary for determining if we trust another user and their devices.

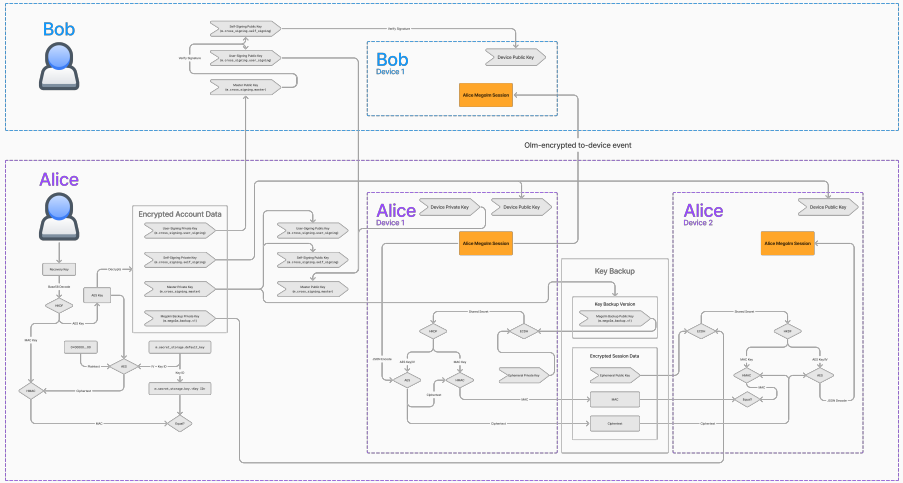
Big Picture: The Other Stuff

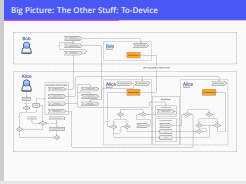




1. So what is all of the other stuff? That is the **infrastructure** for sharing the Megolm session around to other devices and users.

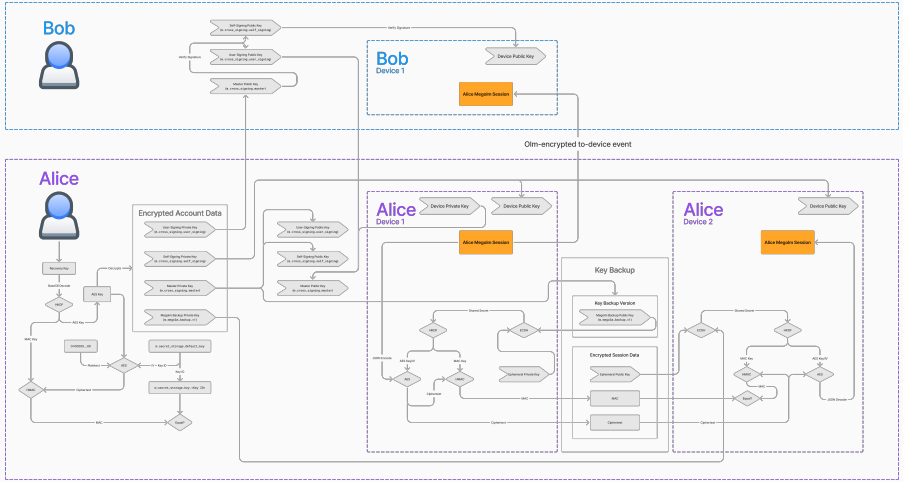
Big Picture: The Other Stuff: To-Device

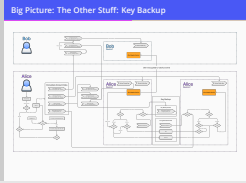




1. For example, in this arrow represents sending the Megolm session via Olm-encrypted to-device messages.

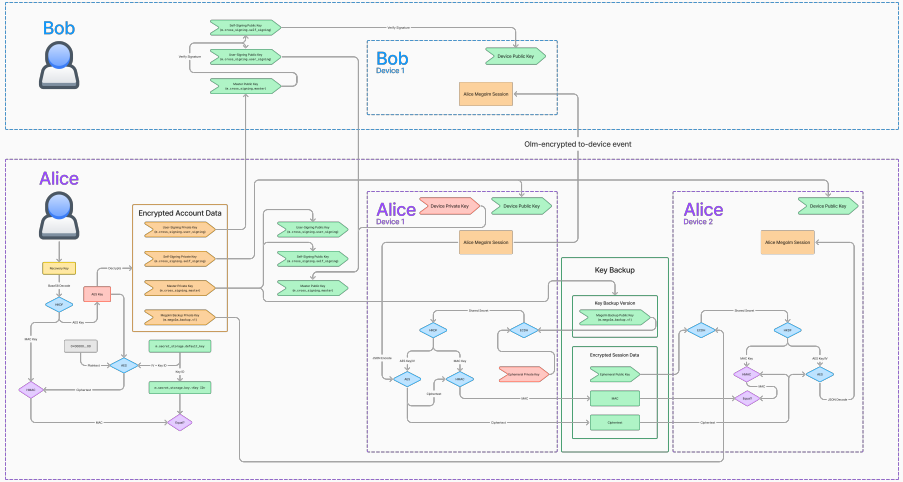
Big Picture: The Other Stuff: Key Backup

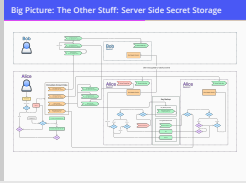




1. This lower section of the diagram represents key backup which allows you to backup your keys to the server and restore from your other devices.

Big Picture: The Other Stuff: Server Side Secret Storage





1. Lastly, on the left we have the infrastructure necessary for storing secrets on the server encrypted by a recovery code.

Cryptography Crash Course

Cryptography Crash Course

Before we dive deeper into the details of the diagram, we need to discuss some basic cryptography primitives.

I will try and explain these 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

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

Asymmetric Signatures

In asymmetric encryption schemes, there are two main operations: **encrypt** and **sign**.

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

Asymmetric Signatures
In asymmetric encryption schemes, there are two main operations: **encrypt** and **sign**.
Signing uses the *private* key, and anyone who possesses the *public* key can verify the signature.

1. We discussed encryption already.
2. You use the public key to encrypt a message, and the private key to decrypt, but...

Asymmetric Signatures

In asymmetric encryption schemes, there are two main operations: **encrypt** and **sign**.

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

Asymmetric Signatures

In asymmetric encryption schemes, there are two main operations: **encrypt** and **sign**.

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

1. We discussed encryption already.
2. You use the public key to encrypt a message, and the private key to decrypt, but...

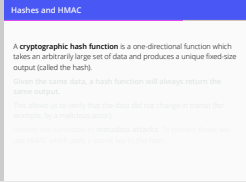
Hashes and HMAC

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. 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.
2. How the key is added is an implementation detail that is not relevant to your understanding of what functionality HMAC provides.

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. 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.
2. How the key is added is an implementation detail that is not relevant to your understanding of what functionality HMAC provides.

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. 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.
2. How the key is added is an implementation detail that is not relevant to your understanding of what functionality HMAC provides.

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. 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.
2. How the key is added is an implementation detail that is not relevant to your understanding of what functionality HMAC provides.

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. For example, we might want to “stretch” a 32-byte shared secret into a 32-byte AES key, a 16-bit AES IV, and a 32-byte HMAC key.

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.

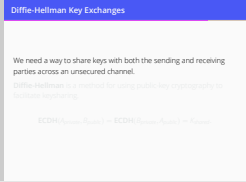
1. For example, we might want to "stretch" a 32-byte shared secret into a 32-byte AES key, a 16-bit AES IV, and a 32-byte HMAC key.

Diffie-Hellman Key Exchanges

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.

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



1. For example, we may need to share a MAC or AES key. This could be done in-person, but that is very impractical.
   That's where the Diffie-Hellman (DH) Key Exchange method comes in.
2. 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:
3. If A_{public} and $B_{private}$ are both available, then K_{shared} can be recovered. This is true even if $A_{private}$ has been discarded.

Diffie-Hellman Key Exchanges

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.

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

1. For example, we may need to share a MAC or AES key. This could be done in-person, but that is very impractical.
   That's where the Diffie-Hellman (DH) Key Exchange method comes in.
2. 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:
3. If A_{public} and $B_{private}$ are both available, then K_{shared} can be recovered. This is true even if $A_{private}$ has been discarded.

Sharing Keys

Sharing Keys

Olm Events

Olm Events

Key Backup

Key Backup

Device Verification

Device Verification

User Verification

Server Side Secret Storage (SSSS)

Server Side Secret Storage (SSSS)

Thank You for Listening!

I've been Sumner Evans.

TODO slide link

Thank You for Listening!

I've been Sumner Evans.
TODO slide link

Questions?

Questions?