UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Algorithm Substitution Attacks on Matrix

Algorithm Substitution Attacks auf Matrix

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Chris-Benedikt Venn

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Thore Tiemann, M. Sc.
Dr. Sebastian Berndt

Lübeck, den 12. September 2021

Abstract

The subversion of cryptographic applications, known as Algorithm Substitution Attacks, are a dangerous threat. The study has begun with symmetric encryption schemes, later advanced to asymmetric encryption schemes as well as signature schemes. With [BWP+20] Berndt, Wichelmann, Pott, Traving and Eisenbarth extended the study to protocols like TLS, WireGuard and Signal. Government agencies already showed high interest in the customizable and self hostable Matrix network and thus the study for Matrix on this topic is inevitable. We analyzed Megolm, the main communication protocol, on possible Algorithm Substitution Attacks. We used our own local development environment of Element Web (Matrix own client) to conduct the study and build two different subversions. Because Matrix does not sufficiently bind identities to identity keys, we managed to impersonate another user in a room without changing the cryptographic primitives at all. Additionally, we advanced the impersonation to any room the victim was participating in. Furthermore, we also eavesdropped on a conversation happening in another room by leaking the necessary keys. We concluded that Matrix does a way better job than Signal in preventing undetectable and universal compromise of message confidentiality. In contrast to Signal, it was not possible to register a malicious device without the victims' knowledge and read future messages indefinitely.

Zusammenfassung

Die Relevanz und Wichtigkeit von Algorithm Subsitution Attacks wurde schon in aktuellen Forschungsergebnissen gezeigt. Es gibt ein wachsendes Interesse von Regierungen, die gerne verschlüsselten Nachrichtenverkehr überwachen wollen. Eine einfache und sehr wirksame Möglichkeit besteht darin eine Backdoor in das Verschlüsselungssystem einzubauen, da man so unproblematisch und im großen Stil verschlüsselten Datenverkehr mitlesen kann. Die Anfänge zu dem Thema wurden bei symmetrischen Verschlüsselungsverfahren gemacht. Später wurde dies auch auf asymmetrische Verschlüsselungsverfahren und Signaturverfahren ausgeweitet. Mit [BWP+20] haben Berndt, Wichelmann, Pott, Traving und Eisenbarth gezeigt, dass man das Thema auch auf ganze Protokolle wie z.B. TLS, WireGuard und Signal erweitern kann. Regierungen haben außerdem großes Interesse an dem sehr anpassbaren und selbst hostbaren Matrix Netzwerk gezeigt. Die Vorteile sind ganz klar. Es müssen keine teuren Lizenzen erworben werden und man muss nicht die eigenen Daten auf den Servern der Lizenzanbieter hosten. Genau deswegen ist die Analyse sehr wichtig, da man nicht möchte, dass Nachrichtenverkehr von Regierungen in die Hände anderer, potenziell böswilliger Regierungen gelangen. Für die Analyse haben wir uns hauptsächlich auf Megolm beschränkt, da das das Hauptkommunikationsprotokoll von Matrix darstellt. Für die praktische Umsetzung haben wir uns als Client auf Element Web beschränkt. Weil Matrix nicht wirklich die Identitäten mit den Identitätsschlüsseln verbindet, war es uns möglich eine andere Person in einem beliebigen Raum, in dem sie Mitglied ist, zu imitieren ohne etwas an den kryptografischen Werten zu verändern. Außerdem haben wir mit unserer Subversion die Keys für eine Kommunikation in einem anderen Raum geleakt mit denen wir dann alle Nachrichten entschlüsseln konnten. Am Ende sind wir zu dem Ergebnis gekommen, dass Matrix ein wesentlich besseren Job darin macht, zu verhindern, dass ein Angreifer uneingeschränkt und unauffällig jeglichen Nachrichtenverkehr vom Opfer mitlesen kann. Im Vergleich zu Signal war es uns nicht möglich ein böses Angreifer-Device im Namen des Opfers zu registrieren um so uneingeschränkt Nachrichten mitlesen zu können.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 12. September 2021

Acknowledgements

A huge thanks goes to Thore and Sebastian for their great help and continuous support during our weekly meetings. And thank you, Professor Eisenbarth for the supervision and the thrilling topic.

Contents

1 Introduction

The Snowden revelations revealed that intelligence agencies are heavily invested in eavesdropping on its citizens and even foreigners [BBGea13, Gre14, PLS13]. Since most encryption protocols today are engineered in such a way that no third party can read what is sent, a workaround is needed. Circumventing cryptosystems all together seems like an easier and more reliable alternative than installing spyware on each device individually. It requires way less time and a well engineered backdoor guarantees permanent access. A backdoor in a cryptosystem can be thought of another person hiding a key to your house in your garden without your knowledge. They have access to your house at any time and can fully break your privacy without your consent. For a real backdoor in a cryptosystem, you probably do not have to be away from home though in order to offer unnoticed access. And there is still a risk involved that you or other people will find the key which opens up the backdoor to everyone with the possession of the key. A backdoor can be a reliable way to break privacy, but it always weakens the robustness of a cryptosystem. Matrix, in particular, is very appealing because it is open source, decentralized and completely self-hostable. Government agencies already showed great interest in such a highly customizable ecosystem [fre18, bun20]. Apart from that, users with a major desire for privacy are usually the audience for an open-source and decentralized system. Since traffic of government agencies as well as privacy-oriented users should be handled with extra care, the need for analysis on this topic is inevitable.

1.1 Related Work

Originally the topic of putting backdoors in cryptosystems was researched by Young and Yung under the name kleptography [YY96, YY97]. Bellare, Paterson and Rogaway continued the study and introduced the term *Algorithm Substitution Attacks* [BPR14]. In continuation to [BPR14], Bellare, Jaeger and Kane strengthened the result [BJK15]. They made the proposed attacks stateless, resulting in a stronger undetectability. The idea of Bellare, Paterson and Rogaway was also extended to signature schemes by Ateniese, Magri, and Venturi [AMV15]. Berndt and Liśkiewicz interpreted Algorithm Substitution Attacks as steganographic systems, which gave them the possibility to give upper bounds for the number of embeddable bits in one message [BL17]. The study was also advanced to asymmetric encryption schemes by [CHY20]. Last year, Berndt, Wichelmann, Pott, Traving

and Eisenbarth pushed things further and applied the topic to entire protocols including TLS, WireGuard and Signal [BWP+20]. Just recently, Tiemann, Berndt, Eisenbarth and Liśkiewicz introduced a way to communicate via a steganographic channel in the ECDSA algorithm of Bitcoin [TBEL21]. Matrix was previously audited by the NCC Group [BM16]. They analyzed the security of Olm and Megolm and highlighted the weak forward and backward secrecy. Matrix later offered a fix against the weak backward secrecy by periodically resetting the symmetric key used for message encryption. There is also a bachelor thesis by Hendriks which analyzed the key management in Matrix [Hen20].

1.2 Our Contributions

We continue the study from Berndt, Wichelmann, Pott, Traving and Eisenbarth in [BWP+20] by analyzing Matrix. A general overview of Matrix and its protocols is given in the beginning. We subverted Matrix in two different ways by using the IV-replacement attacks from [BPR14]. First, we replaced the randomly initialized room key (later referred to as ratchet value) with a ciphertext reaching a low bandwidth with a high undetectability. And secondly, using the first subversion, we build another one by replacing a deterministic value inside the EdDSA signing algorithm of Megolm resulting in a much higher bandwidth but a lower undetectability. In order to measure the undetectability level, we used the work from [RTYZ16] and introduced different watchdogs. Our subversions enabled impersonation of another person as well as eavesdropping on a communication happening in another room. The impersonation attack reveals that Matrix does not sufficiently bind identity keys to identities since the impersonation was possible without needing any cryptographic keys. This implies that an attacker only needs one information (later introduced as the access token) in order to impersonate someone. Despite that, Matrix shows a higher resistance against Algorithm Substitution Attacks than Signal does. In our study, it was not possible to permanently break future message confidentiality.

2 Preliminaries

In this chapter we introduce what Algorithm Substitution Attacks are, the attacker model, the levels of undetectability, the notation used, the Ed25519 signature algorithm and what a ratchet in cryptography means.

2.1 What are Algorithm Substitution Attacks?

Introduced by [BPR14], when we speak of *Algorithm Substitution Attacks* (ASAs), we refer to the substitution of an algorithm for a different, malicious one. We will apply this definition to symmetric encryption schemes and signature schemes. The substitution, also called *subversion*, has to behave in a very similar way the original one did. Encryption and decryption (or signing and verification) should work perfectly fine, just like before. The only difference being that the subversion now has some kind of backdoor which an attacker can use to *leak secrets* in the form of plaintexts, private keys/signing keys, etc. In order to measure the undetectability, we will give a definition and introduce different levels in the next chapter. An attacker also aims for reliability, meaning that with enough output of the subverted algorithm, the attacker is able to extract secrets.

2.2 Attacker Model

Now that we explained what Algorithm Substitution Attacks are, let us be more specific and model our scenario. We will mostly focus on signature schemes and want to give a formal definition. We have an attacker denoted as $\mathcal{A}$ who has the goal to compromise privacy of a victim $\mathcal{V}$. They have their own distinct attacker key $\widetilde{K}$ and with that $\mathcal{A}$ embeds potential secrets of $\mathcal{V}$ only retrievable by them. Let $\Pi = (\mathcal{K}, \mathcal{S}, \mathsf{Vf})$ be a signature scheme with a key generation function $\mathcal{K}(\sigma) = (pk, sk)$, which generates the key pair (pk, sk) using the seed σ. Moreover, let $\mathcal{S}(M, sk) = S$ be the signing function, which signs a given message M using the private key sk and outputs a signature S. Last but not least, let $\mathsf{Vf}(M, S, pk)$ be the verify function, receiving the message M, a signature S and the public key pk. The function now outputs either 1 or 0 depending on whether the signature is valid or not. Signatures produced by $\mathcal{S}$ are always valid meaning that Vf will evaluate to 1. A subversion of Π would be a tuple $\widetilde{\Pi} = (\mathcal{K}, \widetilde{\mathcal{S}}, \mathsf{Vf}, \mathcal{E})$. In addition to $\mathcal{K}$ and Vf, we now also have the extract function $\mathcal{E}$ and replaced the original signing function

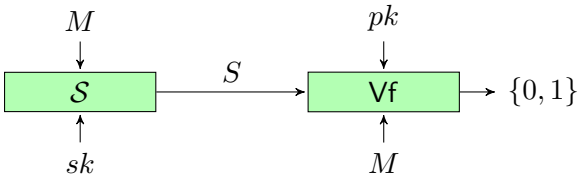

Figure 2.1: Normal behavior of the signing function $\mathcal{S}$

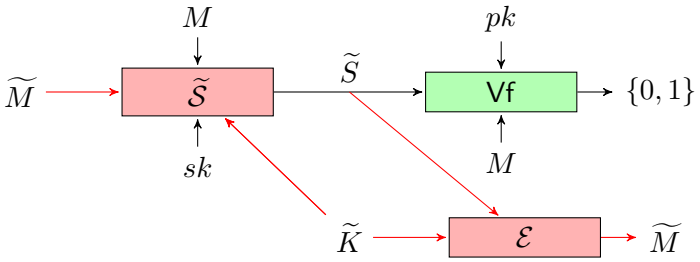

Figure 2.2: Behavior of the subverted signing function $\widetilde{\mathcal{S}}$

with a subverted one $\widetilde{\mathcal{S}}$. The subverted signing function $\widetilde{\mathcal{S}}$ now looks like the following $\widetilde{\mathcal{S}}(M, sk, \widetilde{K}, \widetilde{M}) = \widetilde{S}$. Extended by the attacker key $\widetilde{K}$, as well as the secret message $\widetilde{M}$, the subverted signing function will embed the secret message $\widetilde{M}$ or parts of $\widetilde{M}$ using the attacker key $\widetilde{K}$ resulting in the subverted signature $\widetilde{S}$. Depending on the signature algorithm and the embedding technique chosen, this usually varies. With the extract function $\mathcal{E}$ applied to the subverted signature $\widetilde{S}$ (or multiple signatures, where $o \in \mathbb{N}$ marks the number of signatures), the attacker $\mathcal{A}$ should be able to extract the secret message $\widetilde{M}$ using their attacker key $\widetilde{K}$ by calculating $\mathcal{E}(\widetilde{S_1}, \widetilde{S_2}, ..., \widetilde{S_o}, \widetilde{K}) = \widetilde{M}$. A picture illustrating the different behaviors of $\mathcal{S}$ and $\widetilde{\mathcal{S}}$ can be found in Figure 2.1 and Figure 2.2.

2.3 Levels of Undetectability

Inspired by [RTYZ16], we are looking for a way to formalize the undetectability of our subversions. We introduce a verifier called "watchdog", who either accepts or declines the implementation of the victim $\mathcal{V}$. They accept, if they lose the following security game between a watchdog and the challenger (acting as the bad guy). An illustration is given in Figure 2.3.

1. The challenger picks a random $b \in_R \{0, 1\}$.

2. The watchdog can ask the implementation queries in the form of (M_p, sk_p) where $p \in \mathbb{N}$ marks the index of that pair. They are limited by what the implementation has to offer and in the case of Matrix cannot choose two identical pairs of (M_p, sk_p).

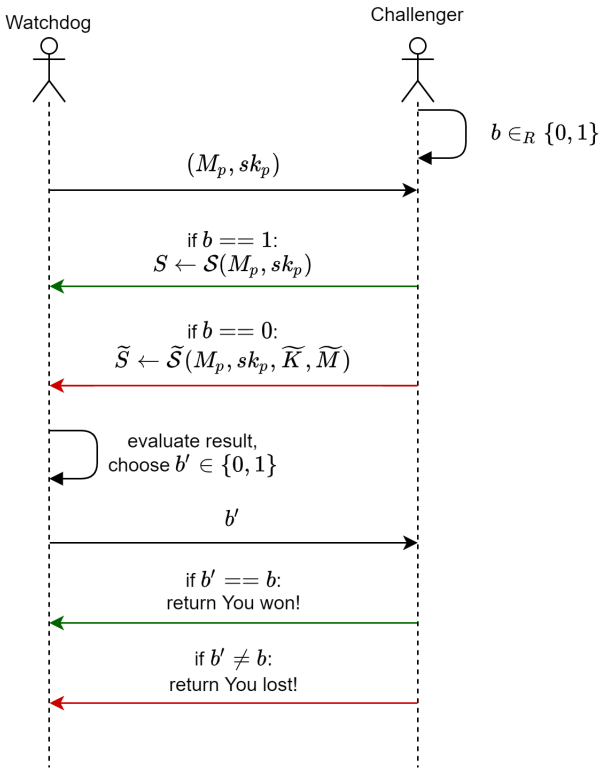

Figure 2.3: The protocol between the watchdog and the challenger

The answer of the query now depends on the secret bit b. If $b = 1$ the challenger returns $S \leftarrow \mathcal{S}(M_p, sk_p)$, otherwise the challenger returns $\widetilde{S} \leftarrow \widetilde{\mathcal{S}}(M_p, sk_p, \widetilde{K}, \widetilde{M})$.

3. The watchdog now has to decide whether they think the implementation is valid or not. They choose a $b' \in \{0, 1\}$ and whenever $b' = b$, they win the security game.

We consider the subversion $\widetilde{\Pi}$ to be undetectable for that particular watchdog if the probability for the watchdog to win is negligible small. Moreover, we differentiate between three watchdogs:

- The **offline watchdog** being the weakest one in the list. They can only do black-box like testing with the implementation of $\mathcal{V}$. Receiving only an input and output, they have to decide whether the implementation of $\mathcal{V}$ is valid or not.
- An advanced version of that, we call an **online watchdog**. They extend the functionality of the offline watchdog to the point where they can monitor the complete communication between $\mathcal{V}$ and the attacker $\mathcal{A}$, and they therefore are also aware of all public keys of $\mathcal{V}$. They also are aware of all room keys (later referred to as ratchet values R_i) inside every room of $\mathcal{V}$.

- Concluding our list, the most powerful watchdog being the **omniscient watchdog**. In addition to possibilities of the online watchdog, the omniscient watchdog also knows all private keys of $\mathcal{V}$.

The watchdogs do not know the attacker key $\widetilde{K}$, and they also cannot verify if a random number is the actual number the implementation produced, but they do analyze the distribution of it.

2.4 Notation

In the following we will talk about the notation used to prevent confusion when reading later chapters.

2.4.1 Concatenation and String Splitting

Whenever $\|$ is used it either refers to string splitting or string concatenation depending on the context. We differentiate between the following cases:

1. $\|$ is on the **left** of an equals sign e.g. $a \parallel b = f(...)$.
   In that case, $\|$ refers to string splitting meaning that the function f returns a value that is split somewhere in between resulting in a and b.

2. $\|$ is on the **right** of an equals sign or **without** any equals sign e.g. $S = a \parallel b$ or $a \parallel b$.
   Here it is used for string concatenation meaning that S is defined as the concatenation of a and b.

2.4.2 Exponents

Whenever a variable is used in an exponent e.g. $R_i^{\mathcal{V}}$ or $sk_{Ed}^{\mathcal{V}}$ it represents the owner of that value. So $R_i^{\mathcal{V}}$ denotes the value R_i of the victim $\mathcal{V}$ and $sk_{Ed}^{\mathcal{V}}$ the value sk_{Ed} of the victim $\mathcal{V}$.

2.4.3 Types of Ciphertexts

In the attacker model, we talked about embedding secrets into signatures using the attacker key $\widetilde{K}$. We normally do this by using AES in a specific mode of operation. We encrypt the secret message $\widetilde{M}$ with an $\widetilde{IV}$, resulting in a ciphertext $\widetilde{C}$. Since we have to transmit the $\widetilde{IV}$ as well, we differentiate between two types of ciphertexts. Whenever $\widetilde{C'}$ is used it explicitly stands for only the ciphertext. The value $\widetilde{C}$ denotes the concatenation of both, i.e. $\widetilde{C} = \widetilde{IV} \parallel \widetilde{C'}$ or $\widetilde{C} = \widetilde{C'} \parallel \widetilde{IV}$.

Table 2.1: Parameters of Ed25519

Parameter	Value
$H(x)$	SHA-512(x)
B	Basepoint in Ed25519 i.e. (0x216936d3cd6e53fec0a4e231fdd6dc5c692cc7609525a7b2c9562d608f25d51a, 0x6666666666666666666666666666666666666666666666666666666666666658)
L	Order of Curve25519 i.e. 0x1000000000000000000000000000000014def9dea2f79cd65812631a5cf5d3ed

2.5 Ed25519 Signature Algorithm

Ed25519 is a special instantiation of the Edwards-Curve Digital Signature Algorithm (EdDSA) on Curve25519 [BDL+12]. Its goal is to offer higher security in contrast to its more common counterpart ECDSA. The specification is based on the RFC 8032 standard which can be found under [edg17]. Table 2.1 defines special parameters of Ed25519 which will be used later on. Let (pk, sk) be the Ed25519 key pair that we want to generate. Both keys are 32 bytes long. The public key pk is used to verify a given signature on a given plaintext while the private key sk is generating such valid signatures. A hex value may be given in decoded form denoted with a "D" (e.g. pk_D, R_D). In that case this value is an elliptic curve point P in the form of coordinates $P = (x, y)$. For further details on how the decoding works, see [edd17].

2.5.1 Key Generation

Algorithm 1: $\mathcal{K}(\sigma)$

1 $sk = \sigma$ // σ contains the 32 byte random data
2 $h = H(sk)$[:32]
3 $s =$ clear_first_bit(h)
4 $s =$ clear_last_three_bits(s)
5 $s =$ set_second_bit(s)
6 $pk_D = s \cdot B$
7 $pk =$ encode(pk_D)
8
9 return (pk, sk)

Generating our Ed25519 key pair (pk, sk) (see Algorithm 1):

1. Initialize the signing key sk with 32 bytes of cryptographically secure random data.

Algorithm 2: $\mathcal{S}(M, sk)$

$h = H(sk)$[:32]
$prefix = H(sk)$[32:]
$s =$ clear_first_bit(h)
$s =$ clear_last_three_bits(s)
$s =$ set_second_bit(s)
$pk_D = s \cdot B$
$pk =$ encode(pk_D)

$r = H(prefix \parallel M)$
$R_D = (r \bmod L) \cdot B$
$R =$ encode(R_D)
$k = H(R \parallel pk \parallel M) \bmod L$
$S' = (r + k \cdot s) \bmod L$

$S = R \parallel S'$
return S

2. Calculate $H(sk)$ and interpret the first 32 bytes of the digest as little-endian number h.

3. Perform the following operations on h:

   - Clear the first bit
   - Clear the last three bits
   - Set the second bit

   The result is the secret scalar s.

4. Multiply s with the basepoint to receive the public key in decoded form $pk_D = sB$.

5. In the final step, encode $pk_D = (x, y)$ by copying the last bit of x to the first bit of y. The result is the 32 byte public key pk in little-endian format.

2.5.2 Signing

Signing a message M with the private key sk (see Algorithm 2):

1. Calculate $H(sk)$ and split the digest in the middle resulting in two 32 byte values. With the first half, calculate s and pk as described in the previous Section. The second half is called $prefix$.

Algorithm 3: $\mathsf{Vf}(M, S, pk)$

$pk_D = \text{decode}(pk)$
$R \parallel S' = S$
$R_D = \text{decode}(R)$
$k = H(R \parallel pk \parallel M) \bmod L$

return $(S' \cdot B) == (R_D + k \cdot pk_D)$

2. Continue by calculating $H(prefix \parallel M)$. Interpret the 64 byte digest as a little-endian number r.
3. Calculate $R_D = (r \bmod L) \cdot B$. Encode $R_D = (x, y)$ with the method from 5 of the previous Section, resulting in a 32 byte value R.
4. Compute $k = H(R \parallel pk \parallel M) \bmod L$ and interpret k as a little-endian number.
5. In the second last step calculate $S' = (r + k \cdot s) \bmod L$.
6. Form the 64 byte signature S with the concatenation of R and the little-endian encoding of S'.

2.5.3 Verification

Verifying S with the public key pk on a given message M (see Algorithm 3):

1. Decode pk back into pk_D.
2. Interpret $R \parallel S' = S$.
3. Decode R back into a point R_D and interpret S' as a little-endian number.
4. Compute $H(R \parallel pk \parallel M) \bmod L$ and interpret the result as a little-endian number k.
5. Verify the equation $S' \cdot B = R_D + k \cdot pk_D$.

2.6 Ratchet Definition

In mechanics, a ratchet is a device that allows rotation only in one direction while blocking the other one. An illustration can be found in Figure 2.4. In cryptography, we make use of this concept by rotating cryptographic keys instead of a mechanical device [BSJ+17]. The idea stays the same. Once a key is rotated to the successor, it is not possible to go back to the old one. This advancement usually can be done indefinitely with the help of cryptographic hash functions. This concept results in an important security property called

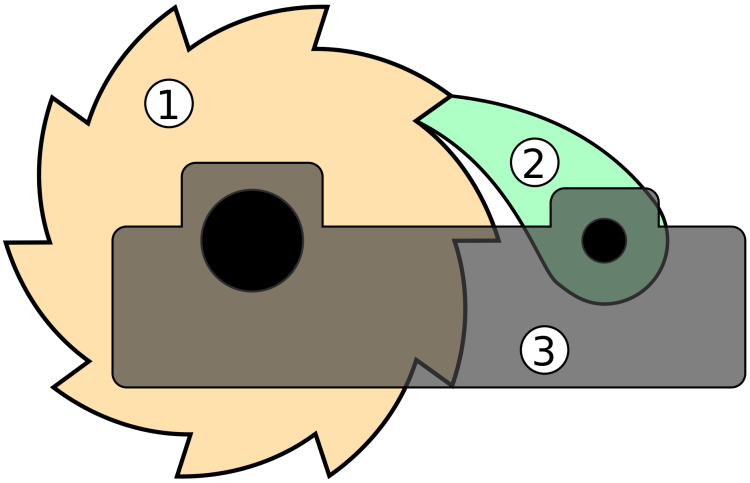

Figure 2.4: A mechanical ratchet [Sch11]

"forward secrecy". If a key is compromised at one point in time, the attacker $\mathcal{A}$ cannot compute any predecessor keys and therefore will not be able to decrypt old messages.

3 High Level Description of Matrix

Matrix is an open source, decentralized, real-time communication network [mat14b]. It is maintained by the non-profit organization "The Matrix.org Foundation" based in the UK and has been around since September 2014. Matrix offers simple HTTP APIs for sharing communication data between users as well as SDKs to give developers the option to implement things themselves. It also claims to provide "state-of-the-art end-to-end encryption" via the cryptographic ratchets of Olm and Megolm. This chapter covers some basic terms of Matrix as well as give some perspective on how the communication structure works.

3.1 Architecture

Let us begin with some terms which are key in order to understand the infrastructure of Matrix. Matrix defines them in its specification [mat].

Homeservers A homeserver is needed for the communication flow between two or more users. Fortunately, a user does not need their own homeservers since Matrix and others provide them with one. Every homeserver stores the communication history and account data for all of their clients and synchronizes the communication history with other homeservers involved in the communication (communication flow illustrated in Figure 3.1). A user may host their own homeserver if they desire.

Users Each user account in Matrix is identified by a unique `user_id` paired with the homeserver on which the account was made on. The user account looks like `@testuser:matrix.org` where `testuser` is the `user_id` and `matrix.org` the homeserver used.

Clients A client is needed for a user to communicate with the homeservers and therefore with other users. Since Matrix is open source everyone can develop their own client. There is an overview of available clients on the Matrix website [cli] for various different platforms. In addition to that, the company behind Matrix also develops their own client called Element (formerly Riot). We recommend to use Element since it is the most advanced client in terms of features and security.

Devices In Matrix, there is a special meaning for the term "device". A user may have several devices like a desktop client, a mobile client, browser clients etc. They are

primarily used for key management in the end-to-end encryption process since every device has their unique set of keys. Upon successful login on a new device, the device registers itself as a "new device" which should be verified by other devices from the user. Otherwise, it is marked as an unverified device and is deemed to be untrusted. The longevity of devices is dependent on the type. A browser client, which tends to be more volatile, registers as a new device upon every new login whereas for a mobile client, it might be acceptable to reuse the device after session timeout. A device is identified by a `device_id` which is unique for every device within a user account.

Events Everything exchanged via Matrix is realized as an event and normally each client action correlates with one event. In Matrix, events are realized as extensible JSON objects. Events can be sending messages, inviting people to a room, updating the room name etc.

Rooms Rooms are used between two or more people to exchange messages. Matrix describes it as "a conceptual place where users can send and receive events" [roo]. Rooms have different properties like room name, room topic, members etc. and are identified by a unique `room_id`.

3.2 Keys

For direct message encryption Matrix uses a symmetric key exchanged via a secure channel (Matrix uses Olm for that purpose, check Section 4.1). Besides that, four more different key pairs are involved for the end-to-end encryption process [key]. For us only two of those four will be relevant which will be explained in the following.

Ed25519 Megolm signing key pair In Matrix, conversations are encrypted via Megolm (see Section 4) and whenever a new Megolm session is created, a new Ed25519 key pair (pk_{Ed}, sk_{Ed}) is generated. They are used to sign outgoing messages within that session. The public key is shared between other room members and the private key never leaves the device.

Curve25519 identity key pair The Curve25519 identity key pair, denoted as (pk_{Cu}, sk_{Cu}) is mainly used to establish an Olm session and also acts as an identifier for that Olm session. Olm is needed to securely exchange the symmetric session keys used in Megolm. Again, the public key is published to the network whereas the private key never leaves the device.

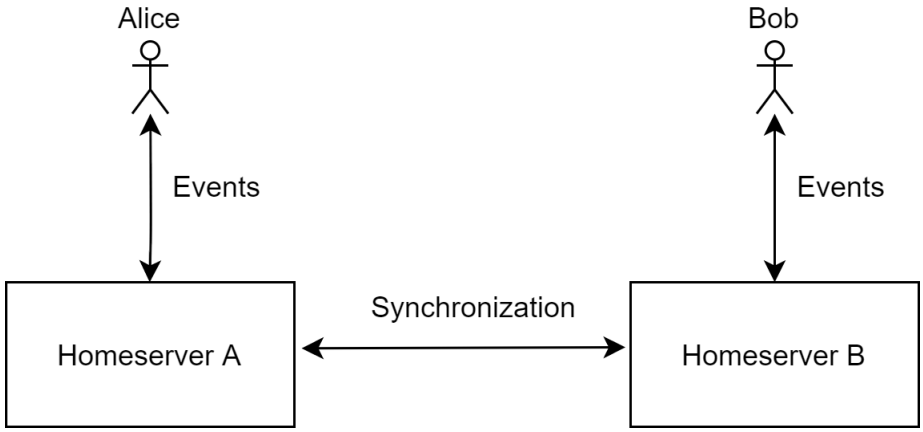

Figure 3.1: General communication flow between two clients through two different homeservers

3.3 How the communication works

If Alice and Bob want to communicate with each other they first have to create a room and invite the other person. In Matrix terms, Alice's client sends a room creation event via an HTTP POST request to her homeserver. The homeserver acknowledges the event of Alice and returns the unique `room_id` and the room is created. Next she sends another event indicating the invitation of Bob. Again, Alice's homeserver acknowledges the event and sends it over to Bob's homeserver. His homeserver verifies the event and Bob can now send a GET request to it, receiving the invite. If Bob decides to accept, the process repeats. The general communication flow is illustrated in Figure 3.1.

4 Protocols Used in Matrix

The messaging protocol used in Matrix is divided into two sub protocols called Olm and Megolm. Originally, Olm was intended for one-to-one conversations and Megolm for group conversations. In practice Matrix uses Megolm even for one-to-one conversations because other people may be added to a room at any time and by that Olm would be incompatible. This chapter is mostly inspired by [meg19].

4.1 Olm

Olm [olm19] is Matrix own implementation of the double cryptographic ratchet used in the Signal Protocol [sig16]. It provides a secure end-to-end encrypted channel between two parties. In Matrix, it is only used to securely exchange the ratchet value R_i for the Megolm protocol. Due to the fact that Signal has already been analyzed on possible Algorithm Substitution Attacks (see [BWP+20]) and Olm does not bring any fundamental changes to the table, our focus will be Megolm.

4.2 Megolm

Megolm [meg19] is the main communication protocol used in Matrix for private and group chats. It consists of a single ratchet (as explained in Section 2.6) which advances after every message, and it requires a secure peer-to-peer channel (such as Olm) to exchange its ratchet value R_i. In this chapter we will take a detailed look on how Megolm works, what security properties it provides and what is missing.

4.2.1 Session setup

Suppose Alice, Bob, and Charlie want to communicate in a secure group chat realized as a room in Matrix. In the first step, an Olm session is created between each of the members which in this case results in three sessions. Based on that, each member of the room now creates their own Megolm session $\Omega = \{i, (pk_{Ed}, sk_{Ed}), R_i\}$, where Ω_P refers to the public part of the Megolm session without the private key sk_{Ed}. Each Ω consists of:

- a four byte counter i
- an Ed25519 signing key pair (pk_{Ed}, sk_{Ed}),

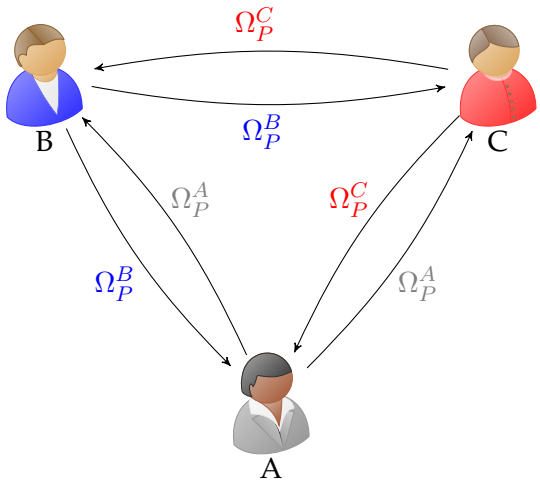

Figure 4.1: Session exchange between room members

- a ratchet value R_i, which consists of four 32 byte values, $R_{i,j}$ for $j \in \{0, 1, 2, 3\}$

The counter i is initialized to 0. A new Ed25519 signing key pair (pk_{Ed}, sk_{Ed}) is generated, and the ratchet R_i is initialized with 128 bytes of cryptographically-secure random data. A Megolm session Ω_P is identified by a unique `session_id` which changes upon every reinitialization. Let Ω_P^A, Ω_P^B, and Ω_P^C be their individual public Megolm session. Alice now sends Ω_P^A to Bob and Charlie. Bob sends Ω_P^B to Alice and Charlie and Charlie sends Ω_P^C to Alice and Bob (illustrated in Figure 4.1). They all do this via their encrypted one-to-one Olm sessions. Every member of the room now stores the other parties public Megolm session Ω_P. If Bob sends his first message to the room in the form of an event (as illustrated in Figure 3.1), Alice and Charlie will access Ω_P^B and use R_0^B to decrypt it. After decryption, Alice and Charlie will increase the counter i for Bobs Ω_P by one and advance his ratchet value from R_0^B to R_1^B. Bob will advance his own ratchet as well and use the new value R_1^B to encrypt his next message. Therefore, only the initial ratchet R_0 is transmitted to the other room members. This scheme continues until Bob's ratchet resets (e.g. after 100 messages, after one week, someone leaves the room) and will be initialized with 128 bytes at random again. The same goes for the ratchets of Alice and Charlie.

4.2.2 Advancing the ratchet

After every message of a user, their ratchet has to be advanced in order to generate the next message encryption keys. The advancement works as follows:

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

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

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

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

And the hash functions being defined as:

$$H_0(A) = \mathrm{HMAC}(A, \text{"\textbackslash x00"})$$
$$H_1(A) = \mathrm{HMAC}(A, \text{"\textbackslash x01"})$$
$$H_2(A) = \mathrm{HMAC}(A, \text{"\textbackslash x02"})$$
$$H_3(A) = \mathrm{HMAC}(A, \text{"\textbackslash x03"})$$

The operator HMAC(A, K_H) represents HMAC-SHA256 of A using K_H as the key, a hash based Message Authentication Code (MAC) using SHA256 [hma97]. The ratchet was originally designed to be advanced indefinitely but after a security audit of the NCC Group [BM16] concluding the lack of backward secrecy, it is now only advanced 100 times or at most used one week before getting reset again. Therefore, only the last 32 bytes $R_{i,3}$ of the ratchet value R_i are ever advanced during those 100 messages. In the later chapters, we will only focus on the whole 128 byte ratchet value R_i and not differentiate between each $R_{i,j}$.

4.2.3 Sharing session data

We just discussed how R_i is advanced, and we now take a look at how it is transferred between two room members via their Olm session. To share the public Megolm session Ω_P whenever Ω is initialized, Megolm uses a specific session-sharing format illustrated in

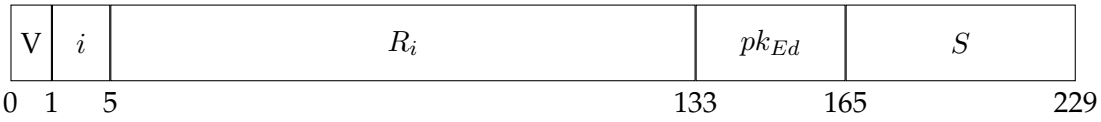

Figure 4.2: Session-sharing format in Megolm

Figure 4.2. The byte V denotes the version number which is currently 0x02, followed by the four byte counter value i in big-endian encoding, continuing with the ratchet value R_i, the Ed25519 public key pk_{Ed} of the sender and ending with an Ed25519 signature S (described in Section 2.5) signing every byte preceding.

4.2.4 Message encryption

The ratchet value R_i of every Ω is the basis for encryption. Although it is not used directly for encryption, the critical values used for the encryption are derived from it. Moreover, Megolm uses AES-256 in CBC mode with PKCS#7 padding together with HMAC-SHA-256 (truncated to 8 bytes). The ratchet value R_i is used to derive the 32 byte AES key, the 32 byte HMAC key and the 16 byte AES IV. The index i corresponds to the same i of the ratchet value.

$$AES\_KEY_i \parallel HMAC\_KEY_i \parallel AES\_IV_i = \text{HKDF}\,(0,\, R_i, \text{"MEGOLM\_KEYS"},\, 80)$$

The operator $\text{HKDF}\,(salt,\, IKM,\, info,\, z)$ describes the HMAC-based extract-and-expand key derivation function [hkd10] using SHA-256 as the hash function with a salt value of $salt$, input key material of IKM, context string $info$, and output material length of z bytes.

4.2.5 Message format

After the encryption process $C_i = \text{AES-CBC}^{AES\_IV_i}_{AES\_KEY_i}(M_i)$, messages have to be formatted properly in order to transmit them. The Megolm documentation defines the following message format illustrated in Figure 4.3. Starting with another version byte at 0x03 followed by the variable length payload based on [var20] which consists of the two important values i and the ciphertext C_i. A MAC, also based on HMAC-SHA256, is appended to ensure integrity for the version byte and payload. The ending marks an Ed25519 Signature S (described in Section 2.5) signing all preceding bytes including the MAC.

4.2.6 Limitations

With Megolm and Olm Matrix provides the three most important security properties confidentiality, integrity and authenticity [BM16]. The confidentiality is guaranteed through

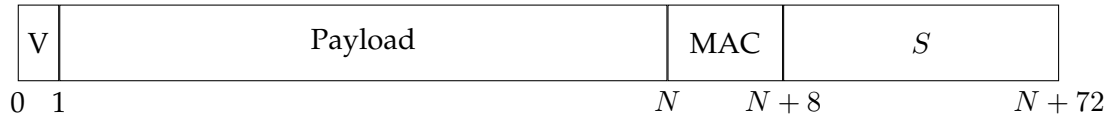

Figure 4.3: Message format for transmission in Megolm

the encryption, while the integrity and authenticity are provided by the MAC and signature appended to every message (as seen in Figure 4.3). But Megolm in particular also has some pretty substantial limitations we will discuss in the following.

Missing Backward Secrecy

One of its big weaknesses is the lack of backward secrecy. Backward secrecy describes the property that if current private keys are compromised, an attacker $\mathcal{A}$ is not able to decrypt any future messages. In Megolm, the ratchet value R_i is used to derive the message keys and whenever $\mathcal{A}$ gets hold of it, they can compute corresponding message keys and future values of R_i and therefore is able to decrypt future messages. As mentioned earlier, Matrix tries to minimize the damage by periodically reinitializing R_i and therefore $\mathcal{A}$ will not be able to decrypt messages like this indefinitely.

Partial Forward Secrecy

Another big weakness is the partial forward secrecy. Forward secrecy describes the opposite of backward secrecy. Whenever current private keys are compromised, an attacker $\mathcal{A}$ is not able to decrypt past messages. Although the ratchet in Megolm is not reversible, the clients keep a copy of the first ratchet value R_0 in case another person is added to a room and wants to read conversation history. So whenever R_0 is compromised, $\mathcal{A}$ can decrypt past messages. Matrix also tries to mitigate this issue by emphasizing application creators to give the user the option to discard historical conversations and not keep a copy of R_0 or to drop entire sessions.

Replay Attacks

A received message can be successfully decrypted several times and therefore $\mathcal{A}$ can resend an old message and the recipient will treat it as a fresh one. It is the application's responsibility to keep track of the counter value i and flag messages sent by the same i it has already seen before. During our testing at the time of writing most clients that offered end-to-end encryption successfully identified message replays and notified the user that something was wrong. However, Element Android did not. It was possible to send all

messages with the same i and the decryption was successful every time. Some research shows that the protection was disabled on purpose for Android and iOS to circumvent a problem with the app on iOS devices. We only verified that the protection was disabled on Android and most likely is still disabled on iOS as well. We were not the first to notice [git20] and opened another GitHub issue but did not receive any feedback.

5 Subverting Matrix

Now after discussing everything we need in order to understand possible attacks, we can begin with our main part. This chapter focuses on creating channels to make use of in the next chapter. Additionally, we will talk about the development environment used as well as some background to why we chose which attack vectors. We will be subverting Megolm in two different ways. The first subversion involves the Megolm ratchet R_i while the second subversion concentrates on the Ed25519 signatures S in Megolm. In Chapter 6 we will analyze possible attacks with those channels.

5.1 The development environment

For all our testing we set up a local development environment of Element Web using [dev20]. It contains the Element Web client, the matrix-js-sdk and matrix-react-sdk. We had to download and link the libolm library [lib16] to our local development environment in order to manipulate the cryptographic implementation of Megolm. For our attacks we only manipulated code inside libolm and the matrix-js-sdk. Client-wise, we mainly focussed on the Element Client since it is the official client and probably what most users will end up using. Third party clients were only involved to analyze different behavior between them. Later, we also used python-olm [pyt15] together with the Matrix Client-Server API [mat14a] to manually encrypt and send customized message events.

5.2 Background

In the last chapters, we discussed how Matrix and the protocol behind it works. We now want to take a look on possible *Algorithm Substitution Attacks* later referred to as ASAs. From [BWP+20] we know that if a protocol provides forward secrecy, it is vulnerable to ASAs. Since Megolm provides partial forward secrecy, we should be able to find something. Based on that, we can analyze two different possibilities for an Algorithm Substitution Attack given in [BPR14]. The paper mentions IV-replacement attacks, where the basic idea is replacing a random nonce a server needs, with the encryption of a secret (e.g. a session key) using the attacker key $\widetilde{K}$. Here, we are limited by the length of the random nonce the server needs. The second attack mentioned, is called a biased-ciphertext attack. The short version here is to compute a ciphertext C using a random nonce but only use

Algorithm 4: $\widetilde{\mathcal{S}}_1(M, R_0, \widetilde{K}, \widetilde{M})$

```
1 M̃ = 0x00 * 112                  // Write zero byte 112 times
2 ĨV = rand(16)                    // Generating 16 bytes random data
3 K̃ = rand(32)
4 C̃ = AES-CBC_{K̃}^{ĨV}(M̃)          // Encrypting secret message of length 112
5
6 R_0 = C̃                          // Replace original R_0
7
8 return R_0
```

a C that embeds one bit of a previously chosen secret $\widetilde{M}$ using the attacker key $\widetilde{K}$. If it does not, we just generate a new nonce and try again. With every C received, the attacker $\mathcal{A}$ can decrypt the secret, bit by bit. This type of leakage is rather slow and needs more ciphertexts the longer $\widetilde{M}$ gets. Now if we take a look a Matrix and in particular Megolm, we can quickly come to the conclusion that a biased-ciphertext attack is not feasible here, because we are missing any kind of continuous randomness in the encryption scheme. In Megolm, only the first ratchet value R_0 is chosen at random. After that, all subsequent ratchet values R_i and their respective AES_KEY_i and AES_IV_i are calculated deterministically. Therefore, we are only being able to leak 1 bit every 100 messages which is not very efficient. For an IV-replacement attack to work, the attacker $\mathcal{A}$ and the victim $\mathcal{V}$ have to be in the same room, because the initial ratchet value R_0 is encrypted and transmitted via an Olm session and only room participants will receive a copy of it. The next section evolves around this idea, and we will create our first subversion.

5.3 Megolm Ratchet Subversion

Beginning with our first subversion, we want to manipulate part of the Megolm algorithm to leak some kind of secret $\widetilde{M}$ of the victim $\mathcal{V}$ to an attacker $\mathcal{A}$, both being members of the same room. Due to the fact that a biased-ciphertext attack [BPR14] is not feasible here as described in Section 5.2, we try to make use of the IV-replacement attacks also described in the same paper [BPR14]. The goal for the subversion is to be undetectable and yet secure in the sense that only $\mathcal{A}$ is able to retrieve the leaked secrets. We first highlight the theoretical description of the subversion, following a look on the implementation and finally summarizing the results.

Algorithm 5: $\mathcal{E}_1(R_0^{\mathcal{V}}, \widetilde{K})$

$\widetilde{IV} \parallel \widetilde{C'} = R_0^{\mathcal{V}}$

$\widetilde{M} = \text{AES-CBC}_{\widetilde{K}}^{\widetilde{IV}}(\widetilde{C'})$ `// Decrypting secret message`

return $\widetilde{M}$

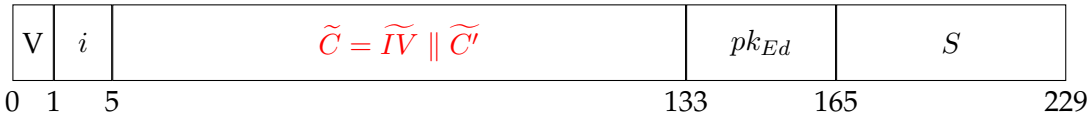

Figure 5.1: Subverted session-sharing format (changes in red)

Our first subversion

Since AES_KEY_i, $HMAC\_KEY_i$ and AES_IV_i from Section 4.2.4 are all deterministically derived from our ratchet value R_i we cannot alter them if we want to maintain functionality. But looking at the ratchet value R_i is promising. Even though it is deterministically expanded after every message by every user, it is initialized randomly with 128 bytes. Based on the work of IV-replacement attacks described in [BPR14] we can make use of that and replace the initial ratchet value R_0 with a ciphertext $\widetilde{C}$ using the unique attacker key $\widetilde{K}$. For the encryption we choose an algorithm that produces ciphertexts indistinguishable from random data to maintain a high level of undetectability. A good candidate for this is AES-CBC since it meets our requirement according to [Rog11] and to our convenience, it is already present in libolm. To ensure the indistinguishability from random data the attacker chosen $\widetilde{IV}$ for AES-CBC has to be randomly chosen for every encryption process of our message $\widetilde{M}$. For now, $\widetilde{M}$ will be test message to ensure everything works. The encryption looks like this $\widetilde{C} = \widetilde{IV} \parallel \widetilde{C'} = \text{AES-CBC}_{\widetilde{K}}^{\widetilde{IV}}(\widetilde{M})$. In the next section, we will use this channel to leak a private key. For our $\widetilde{IV}$, we choose 16 bytes secure random data. After encryption, we substitute the ratchet value by $R_0 = \widetilde{C} = \widetilde{IV} \parallel \widetilde{C'}$. In a slight abuse of notation, we will also use the symbols from Section 2.2 for the pseudocode given in Algorithm 4 and Algorithm 5 marking our subversion and extraction function. In contrast to the normal message format in Figure 4.3, the manipulated session setup is illustrated in Figure 5.1.

Implementation

To transfer our approach into actual code, we went into libolm, where Megolm resides and only had to change one function. Inside `olm/src/outbound_group_session.c`,

we inspect the function `olm_init_outbound_group_session` which initializes the ratchet R_i upon creating an outbound group session. The function receives `session` which is the session object, `random` which is the random data, and `random_length` containing the length of `random`. The parameter `random` is always 160 bytes long and consists of $R_0 \parallel sk$. The first 128 bytes of `random` containing the cryptographically-secure random data required for the initialization of the Megolm ratchet and is therefore copied into the session object representing R_0. The remaining 32 bytes are the secret key sk described in Section 2.5 acting as the initializer for the Ed25519 key pair used for signing in message transmission described in Section 4.2.1. To perform the attack, the attacker $\mathcal{A}$ has an attacker key $\widetilde{K}$ which has to be a 32 byte random byte sequence in order to match the security requirements of AES-CBC. Next we define $\widetilde{IV}$ to be the 16 bytes of secure random data and choose our input $\widetilde{M}$ to be a test message of size 112 bytes since we have to leave room for $\widetilde{IV}$. Now we encrypt $\widetilde{M}$ by calling `_olm_crypto_aes_encrypt_cbc` with our just defined values and replace the first 128 bytes of `random` with $\widetilde{C} = \widetilde{IV} \parallel \widetilde{C'}$. Instead of the original random data, $\widetilde{C}$ is now copied as R_0 into the session object.

Conclusion

We successfully substituted $R_0^{\mathcal{V}}$ with a ciphertext $\widetilde{C}$. The attacker $\mathcal{A}$ and the victim $\mathcal{V}$ are in the same room and whenever $\mathcal{V}$ initializes their Ω, the victim $\mathcal{V}$ will send $R_0^{\mathcal{V}} = \widetilde{C} = \widetilde{IV} \parallel \widetilde{C'}$ to $\mathcal{A}$. Upon receiving R_0 the attacker $\mathcal{A}$ can now decrypt $\widetilde{C'}$ using $\widetilde{IV}$ and their unique attacker key $\widetilde{K}$ back into the secret message $\widetilde{M}$. In [BPR14], Theorem 1 argues the undetectability for a symmetric encryption scheme using an IV-replacement attack, but the Theorem covers a weaker case, where encryption function used for the IV replacement is not an ind$-secure function. For our case, we have an ind$-secure function, namely AES-CBC and we replace R_0 with $\widetilde{C} = \text{AES-CBC}_{\widetilde{K}}^{\widetilde{IV}}(\widetilde{M})$ using a unique $\widetilde{IV}$ every time. Now, the normal R_0 is random data and $\widetilde{C}$, the output of our ind$-secure function is also random data (or rather indistinguishable from random data), we can conclude that Theorem 1 from [BPR14] still holds (if the watchdog does not know $\widetilde{K}$), and the replacement of R_0 has to be undetectable up to the omniscient watchdog.

5.4 Megolm Signature Subversion

After our first subversion from the previous chapter we managed to leak 112 bytes every 100 messages. Our goal for this chapter is to improve the channel bandwidth and thus further expand the subversion. Since R_0 is the only random part involved in the raw encryption process of Megolm we have to look elsewhere to make improvements. Because we want to expand, we are looking for something sent frequently. Inspecting the message

format, we see each message is appended a MAC and a signature S. The MAC does not sound promising because it does not involve randomness but signatures usually do. The signature algorithm used for signing messages is EdDSA more specific Ed25519. Unfortunately, EdDSA does not involve randomness in contrast to its more common counterpart ECDSA. The value r (seen in Section 2.5) that is multiplied by the basepoint B is deterministically derived from a hash function in contrast to ECDSA where it is randomly chosen. Even though r will be the same value for the same Ed25519 signing key pair (pk_{Ed}, sk_{Ed}) and same plaintext M, this case does not occur in Megolm. Megolm uses a message format, where the index i is part of the variable length payload as seen in Section 4.2.5. Because i is incremented after every message, the plaintext M is going to be different and the produced signature S will always be unique. Due to this fact, we can safely assume that only someone who can actually calculate r is able to verify if it was calculated according to specification or not. And in order to be able to calculate r, the private key sk_{Ed} is needed. As seen in the next section, secret scalar s is needed in order to calculate r and s can only be calculated with the private key sk_{Ed}. An omniscient watchdog is in possession of all private keys, so by taking a lower level undetectability into account, we can make another subversion possible described in the following.

Expanding the channel

The objective is to replace r with some ciphertext $\widetilde{C}$ as we did in Section 5.3, but the finished signature S does not involve r, so how do we extract it? As explained in Section 2.5 the value S' is calculated by $S' = (r + k \cdot s) \bmod L$. Rewriting that to r is fairly simple $r = (S' - k \cdot s) \bmod L$. The value S' are the last 32 bytes of our finished signature S, and we can calculate $k = H(R \parallel pk \parallel M)$ because R, pk, and M are all public values. The last remaining parameter we need is s, but s can only be calculated via the private key sk_{Ed}. Fortunately, we already have a successful subversion from before, where we can just leak the private key sk_{Ed} and be able to calculate s and thus recover r. We have one more thing to consider though. In the signature process R_D is calculated via $R_D = (r \bmod L) \cdot B$. During this calculation r is reduced, and therefore we have to make sure that does not occur. Otherwise, the recovery of r will be pointless as long as it is not a decryptable ciphertext $\widetilde{C}$ anymore. During reduction, the length of r went from 64 bytes to 32 bytes. Due to this huge reduction, it is not affordable to brute force multiples of L. To circumvent this we have to make sure that $r < L$ since a calculation with a smaller number than the modulus L does not change the input. Due to the control over the ciphertext $\widetilde{C'}$ and $\widetilde{IV}$ we choose a format where $r = \widetilde{C} = \widetilde{C'} \parallel \widetilde{IV}$. The value r is interpreted as a little-endian number and by having control over $\widetilde{IV}$, we can freely determine the most significant bytes of r. The most significant byte of our modulus L (in Table 2.1) is 0x10. Therefore, the most

Algorithm 6: $\widetilde{\mathcal{S}}_2(M, sk_{Ed}^{\mathcal{V}}, \widetilde{K}, \widetilde{M})$

$h = sk_{Ed}^{\mathcal{V}}$[:32] // In Matrix $H(sk) == sk_{Ed}^{\mathcal{V}}$
$prefix = sk_{Ed}^{\mathcal{V}}$[32:]
$s = \text{clear\_first\_bit}(h)$
$s = \text{clear\_last\_three\_bits}(s)$
$s = \text{set\_second\_bit}(s)$
$pk_D = s \cdot B$
$pk = \text{encode}(pk_D)$

$\widetilde{M} = 0\text{x}00 * 16$ // Write zero byte 16 times
$\widetilde{IV} = \text{rand}(15) \parallel \text{rand}(0\text{x}00, 0\text{x}0\text{F})$ // Last byte only between 0x00 and 0x0F
$\widetilde{K} = \text{rand}(32)$
$\widetilde{C} = \text{AES-CBC}_{\widetilde{K}}^{\widetilde{IV}}(\widetilde{M})$ // Encrypting secret message of length 16
$r = \widetilde{C}$ // Replace original r

$R_D = (r \bmod L) \cdot B$ // $r \bmod L$ does nothing because $r < L$
$R = \text{encode}(R_D)$
$k = H(R \parallel pk \parallel M) \bmod L$
$S' = (r + k \cdot s) \bmod L$

$\widetilde{S} = R \parallel S'$

return $\widetilde{S}$

significant byte has to be smaller than 0x10 to ensure that no reduction is happening. By choosing the first 15 bytes of $\widetilde{IV}$ completely at random we have to make sure that the most significant byte is only chosen between 0x00 and 0x0F. Following this constraint our ciphertext $\widetilde{C}$ is always recoverable. One could ask themselves if that does not change the distribution of R_D because we calculate $r \cdot B$ to get R_D and for that reason be detectable. And in fact we are changing the distribution but let us do a simple calculation.

$$L = 0\text{x}1000000000000000000000000000000014\text{def9dea2f79cd65812631a5cf5d3ed}$$

$$c = 0\text{x}14\text{def9dea2f79cd65812631a5cf5d3ed} + 1$$

$$P(\text{"Number with} \leq 0\text{x}0\text{F"}) = 1 - \frac{c}{L} \geq 1 - \frac{2^{125}}{2^{252}} = 1 - 2^{-127}$$

As we can see, the probability for the correct implementation to choose a number with a most significant byte of 0x0F or smaller is basically 1. The amount of numbers we cut off

Algorithm 7: $\mathcal{E}_2(\widetilde{S}, \widetilde{K}, sk_{Ed}^{\mathcal{V}})$

$R = \widetilde{S}$[:32]
$R_D = \text{decode}(R)$
$S' = \widetilde{S}$[32:]

$h = sk_{Ed}^{\mathcal{V}}$[:32] // In Matrix $H(sk) == sk_{Ed}$
$prefix = sk_{Ed}^{\mathcal{V}}$[32::]
$s = \text{clear\_first\_bit}(h)$
$s = \text{clear\_last\_three\_bits}(s)$
$s = \text{set\_second\_bit}(s)$
$pk_D = s \cdot B$
$pk = \text{encode}(pk_D)$

$k = H(R \parallel pk \parallel M) \bmod L$
$r = (S' - k \cdot s) \bmod L$ // Recover r

$\widetilde{C'} \parallel \widetilde{IV} = r$
$\widetilde{M} = \text{AES-CBC}_{\widetilde{K}}^{\widetilde{IV}}(\widetilde{C'})$ // Decrypting secret message

return $\widetilde{M}$

is microscopic in contrast to the total number of $L-1$ possibilities. Since r is 32 bytes long, we will be able to leak 16 bytes of secrets $\widetilde{M}$ every message $\mathcal{V}$ sends. A pseudocode for the subversion and extraction algorithm is also given in Algorithm 6 and Algorithm 7.

Implementation

For the implementation, we went back into libolm again. The signing process is realized in `olm/lib/ed25519/sign.c`. To start our subversion, we first make a copy of the `ed25519_sign` function because we only want it to be called during the encryption process. Next we define the 32 byte attacker key $\widetilde{K}$ and the 16 byte $\widetilde{M}$. As discussed, for $\widetilde{IV}$ we choose the first 15 bytes completely at random and the last byte only between 0x00 and 0x0F. We perform the encryption and replace r with $\widetilde{C} = \widetilde{C'} \parallel \widetilde{IV}$. For the final step we go inside `olm/src/outbound_group_session.c` into the `_encrypt` function and make sure the subverted signing function $\widetilde{\mathcal{S}}$ is called.

Eliminating the IV

The replacement of r still contains 16 bytes of $\widetilde{IV}$ and in this section we want to diminish it as much as possible to improve the channel bandwidth. To achieve this we have to

Algorithm 8: $\widetilde{\mathcal{S}}_3(M, R_i^{\mathcal{V}}, sk_{Ed}^{\mathcal{V}}, \widetilde{K}, \widetilde{IV})$

$h = sk_{Ed}^{\mathcal{V}}$[:32] // In Matrix $H(sk) == sk_{Ed}^{\mathcal{V}}$
$prefix = sk_{Ed}^{\mathcal{V}}$[32:]
$s = \text{clear\_first\_bit}(h)$
$s = \text{clear\_last\_three\_bits}(s)$
$s = \text{set\_second\_bit}(s)$
$pk_D = s \cdot B$
$pk = \text{encode}(pk_D)$

$\widetilde{M} = 0\text{x}00 * 31$ // Write zero byte 31 times
$\widetilde{IV} = H(R_i^{\mathcal{V}} \parallel \widetilde{K})$[:16] // Ensuring $\widetilde{IV}$ is fresh and secret
$\widetilde{K} = \text{rand}(32)$
$\widetilde{C} = \text{AES-CTR}_{\widetilde{K}}^{\widetilde{IV}}(\widetilde{M})$ // Encrypting secret message of length 31
$r = \widetilde{C'} \parallel \text{rand}(0\text{x}00, 0\text{x}0\text{F})$ // Now r contains 31 ciphertext bytes

$R_D = (r \bmod L) \cdot B$ // $r \bmod L$ does nothing because $r < L$
$R = \text{encode}(R_D)$
$k = H(R \parallel pk \parallel M) \bmod L$
$S' = (r + k \cdot s) \bmod L$

$\widetilde{S} = R \parallel S'$

return $\widetilde{S}$

replace $\widetilde{IV}$ with something $\mathcal{V}$ and $\mathcal{A}$ already know and therefore does not need to be transmitted. The best candidate for this is the ratchet value $R_i^{\mathcal{V}}$ because it is updated after every message and is known to both parties. Remember our goal is that $r = \widetilde{C'}$, but we have to keep in mind that $r < L$ is met otherwise the reduction inside $R_D = (r \bmod L) \cdot B$ will destroy $\widetilde{C'}$. The easiest and most reliable way to circumvent this is to use the same method from before but this time we have to sacrifice the last byte of $\widetilde{C'}$ instead of $\widetilde{IV}$. The last byte should only be a random number between 0x00 and 0x0F. Now the remaining 31 bytes of the r replacement can even be all 0xFF and $r < L$ still holds and therefore $\widetilde{C'}$ is always recoverable. Since AES-CBC is a block cipher it can only encrypt blocks of 16 bytes, meaning that it would pad our 31 byte $\widetilde{C'}$ to 32 bytes. For our approach to work, we have to change the encryption mode to something like AES-CTR [DH79] because 31 bytes of $\widetilde{M}$ will result in 31 bytes of $\widetilde{C'}$. The same paper [Rog11] that provided the ind$-security property for AES-CBC also covered the counter mode of AES concluding that AES-CTR also meets the ind$-security property when using a random nonce. Under the notion of the random oracle model [KL14], we assume that output of a

Algorithm 9: $\mathcal{E}_3(\widetilde{S}, R_i^{\mathcal{V}}, \widetilde{K}, sk_{Ed}^{\mathcal{V}})$

$R = \widetilde{S}$[:32]
$R_D = \text{decode}(R)$
$S' = \widetilde{S}$[32:]

$h = sk_{Ed}^{\mathcal{V}}$[:32] // In Matrix $H(sk) == sk_{Ed}^{\mathcal{V}}$
$prefix = sk_{Ed}^{\mathcal{V}}$[32::]
$s = \text{clear\_first\_bit}(h)$
$s = \text{clear\_last\_three\_bits}(s)$
$s = \text{set\_second\_bit}(s)$
$pk_D = s \cdot B$
$pk = \text{encode}(pk_D)$

$k = H(R \parallel pk \parallel M) \bmod L$
$r = (S' - k \cdot s) \bmod L$ // Recover r

$\widetilde{C'} = r$[:31] // Now r is 31 bytes ciphertext only
$\widetilde{IV} = H(R_i^{\mathcal{V}} \parallel \widetilde{K})$[:16] // can be calculated the same way
$\widetilde{M} = \text{AES-CBC}_{\widetilde{K}}^{\widetilde{IV}}(\widetilde{C'})$ // Decrypting secret message

return $\widetilde{M}$

hash function is indistinguishable from random data if the input is unknown. The basis for our $\widetilde{IV}$ is the ratchet value $R_i^{\mathcal{V}}$ paired with $\widetilde{K}$. Calculating $H(R_i^{\mathcal{V}} \parallel \widetilde{K})$ ensures that the input of H is unknown, because none of the watchdogs know $\widetilde{K}$. Since $R_i^{\mathcal{V}}$ advances after every message, we ensure a fresh and random $\widetilde{IV}$ for each leak to fulfill the ind$-security property. The CTR mode of AES is not implemented in libolm, so we had to implement it ourselves. Switching over to the implementation and going back into `ed25519_sign`, we increased our $\widetilde{M}$ to 31 bytes and chose the 32nd byte randomly between 0x00 and 0x0F to fit our constraint. Finally instead of AES-CBC we called our implementation of AES-CTR and replaced r with $\widetilde{C'} \parallel \text{rand}$(0x00, 0x0F). For the ind$-security, we can make the same argument as we did in Section 4 considering the Theorem 1 from [BPR14]. In conclusion, we increased our channel bandwidth from 16 bytes every message to 31 bytes every message. A pseudocode for the subversion and extraction algorithm is once again given in Algorithm 8 and Algorithm 9.

Conclusion

Our goal was to further expand the channel bandwidth of our subversion. From Section 5.3 we were able to leak 112 bytes whenever an Ω is initialized (i.e. $i \bmod 100 = 0$). In addition to that, we now made it possible to leak 31 bytes every message by substituting r with another ciphertext $\widetilde{C'}$ and choosing the 32nd byte with rand(0x00, 0x0F) to preserve $r < L$. In order for r to be retrievable we had to leak $sk_{Ed}^{\mathcal{V}}$ via the Megolm ratchet subversion from Section 5.3. Unfortunately this type of subversion is less discreet because we are replacing the deterministic value r. Therefore, an omniscient watchdog will now be alarmed by our subversion because with their knowledge of all private keys, they can verify that our replacement of r is not what it is originally supposed to be by the Ed25519 specification in [edg17]. The undetectability level now decreases down to the online watchdog with the following three arguments. The most important one being that the online watchdog cannot directly compute r without the private key sk_{Ed}. In addition to that, they are not able to encrypt the same plaintext M twice, because as we explained earlier, the implementation prevents it due to the message index i. They also cannot verify that we changed the distribution because the probability of reaching a value we cut off $P(\text{"Number with} \leq \text{0x0F"})$, is microscopic, as we calculated earlier. A similar argument was used in Lemma 1 from [TBEL21].

6 Attacking Matrix

We have established two working channels, which we can use to leak secrets from $\mathcal{V}$ to $\mathcal{A}$. In this chapter we want to make use of those channels and see what we can do with them. In the first attack, we try to impersonate another user in a given room of people using the Megolm ratchet subversion. The second attack involves the attacker $\mathcal{A}$ eavesdropping on a communication between two distinct parties using the Ed25519 signature subversion.

6.1 Impersonation with signature keys

In the last two chapters we first subverted the Megolm Ratchet with the replacement of $R_0^{\mathcal{V}}$ and with that subversion we leaked the private Ed25519 signature key $sk_{Ed}^{\mathcal{V}}$ to make another subversion possible which improves upon the channel bandwidth. Until now, we did not really utilize them except for the Megolm signature subversion. In this chapter we want to change that and use $sk_{Ed}^{\mathcal{V}}$ to impersonate the victim $\mathcal{V}$ in a given room. Our goal is to send room messages from $\mathcal{A}$ in the name of $\mathcal{V}$. For our testing we set up a room with the attacker $\mathcal{A}$, the victim $\mathcal{V}$ and a legitimate third party denoted as $\mathcal{T}$. From the client of $\mathcal{T}$ we can evaluate if $\mathcal{A}$ sent a message displayed as $\mathcal{V}$.

Starting off

For our first attempt, let us (in the role of $\mathcal{A}$) try sending a message encrypted with R_i of $\mathcal{V}$ denoted as $R_i^{\mathcal{V}}$ and signed by the private Ed25519 key of $\mathcal{V}$ denoted as $sk_{Ed}^{\mathcal{V}}$. With the Megolm Ratchet Subversion from before, we first leak $sk_{Ed}^{\mathcal{V}}$ via $R_0^{\mathcal{V}}$. After the setup of $\Omega^{\mathcal{A}}, \Omega^{\mathcal{V}}, \Omega^{\mathcal{T}}$ and all three parties exchanging their public Megolm sessions $\Omega_P^{\mathcal{A}}, \Omega_P^{\mathcal{V}}, \Omega_P^{\mathcal{T}}$ including $R_0^{\mathcal{A}}, R_0^{\mathcal{V}}, R_0^{\mathcal{T}}$ the attacker $\mathcal{A}$ is now in possession of $R_0^{\mathcal{V}}$ and $sk_{Ed}^{\mathcal{V}}$. The initial ratchet value $R_0^{\mathcal{V}}$ is enough because the attacker $\mathcal{A}$ can compute all successors of $R_0^{\mathcal{V}}$ themselves. We rewrote the client of $\mathcal{A}$ to encrypt and sign their first message with $R_0^{\mathcal{V}}$ and $sk_{Ed}^{\mathcal{V}}$. Switching to the client of the third party $\mathcal{T}$ reveals a `Bad Signature` error. Analyzing the console output we can quickly find the reasoning behind it. Because the Curve25519 public key pk_{Cu} and the `session_id` are still those from $\mathcal{A}$, party $\mathcal{T}$ associates the message with $\mathcal{A}$ and therefore takes their public key $pk_{Ed}^{\mathcal{A}}$ to verify the signature S which obviously results in a verification error. We now have to swap to the Curve25519 public key $pk_{Cu}^{\mathcal{V}}$ and `session_id` of victim $\mathcal{V}$. Via the GUI we can use the view source function on a given room message from $\mathcal{V}$ to retrieve them. Instead of making changes

```
1  PUT /_matrix/client/r0/rooms/!DdTtQuElCLOpMmAmMU%3Amatrix.org/send/
2  m.room.encrypted/m1629982623109.31 HTTP/2
3  Host: matrix-client.matrix.org
4  Content-Length: 447
5  Sec-Ch-Ua: " Not A;Brand";v="99", "Chromium";v="90"
6  Accept: application/json
7  Authorization: Bearer syt_YmxhYmxheGR4ZA_yAqzSkffIXCtoxFLXxCJ_4O6NKM
8  Sec-Ch-Ua-Mobile: ?0
9  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
10 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.93 Safari/537.36
11 Content-Type: application/json
12 Origin: http://127.0.0.1:8080
13 Sec-Fetch-Site: cross-site
14 Sec-Fetch-Mode: cors
15 Sec-Fetch-Dest: empty
16 Accept-Encoding: gzip, deflate
17 Accept-Language: en-US,en;q=0.9
18 Connection: close
19
20
21
22 {
23     "algorithm": "m.megolm.v1.aes-sha2",
24     "sender_key": "OPKhMwL76FmTsco88rOTuoja0OsfKQfOWsoC7MnoOms",
25     "ciphertext": "AwgAEnC3mDjvGBitnPWlkfrnxyGJr7JA8NNN3p4tQM9IWsNSh2ing++d7
26         gE+Y9A1Pb0HYM3e6thEonj8DsKzejXqudHoHNlgrGJkSyWyCB1si3qA174NGQxjhf5Ps
27         CrzY8tku25FfInik6BBGI0EYlAPmmBTOjNnXogokGE3/Os1mgxi+XMvfX9lCutXKlarI
28         4eWn03UyEOOuT/j5XDgVzb9kw8EvYIC1ZdRwaVWFglVFs6UTJ24laRGRPMM",
29     "session_id": "CWG1TXV+Zs+XqlaYJCMOW/5Cc7pTMcqXSUd+n613xNM",
30     "device_id": "NTLNZCKSKP"
31 }
```

Figure 6.1: PUT request of a message event captured in Burp

to the client of $\mathcal{A}$ every time we test something, we switched to python-olm [pyt15] using the Client-Server API to manually send message events to improve our workflow and be less error-prone. In Figure 6.1 you see an example message event. From the second line we can see an `m.room.encrypted` event [mro18] being sent to a room with `room_id="!GTOVdzziBcpXDhxwaT:matrix.org"`. The mostly base64-encoded JSON data from line 23 to 30 reveals the usage of Megolm, the Curve25519 sender key pk_{Cu} we mentioned before, the ciphertext C_i in the format of Section 4.2.5, the `session_id` and `device_id`. Remember $\mathcal{V}$ already sent a message and from that, we got $\texttt{session\_id}^{\mathcal{V}}$ and $\texttt{sender\_key}^{\mathcal{V}}$. With python, we recreated the request as well as manually encrypted a test message using python-olm with $R_1^{\mathcal{V}}$ and $sk_{Ed}^{\mathcal{V}}$. In addition to that, we now also used $\texttt{session\_id}^{\mathcal{V}}$ and $\texttt{sender\_key}^{\mathcal{V}}$. We should also swap to $\texttt{device\_id}^{\mathcal{V}}$ for undetectability even though it is not necessary function-wise because otherwise they would be identical for a legitimate message from $\mathcal{A}$ and an impersonation attempt in the name

of $\mathcal{V}$. An observant user may detect the equality of them in an impersonation attempt by using the view source function in the GUI. The result is surprising, third party $\mathcal{T}$ can successfully decrypt and display the message of $\mathcal{A}$. However, the GUI still shows the name of $\mathcal{A}$. More research reveals that internally $\mathcal{T}$ believes it is a message of $\mathcal{V}$ because $\mathcal{T}$ uses $R_1^{\mathcal{V}}$ for decryption as well as $pk_{Ed}^{\mathcal{V}}$ for signature verification. Whenever $\mathcal{V}$ now wants to send a message, $\mathcal{T}$ flags it as a possible replay attack because $\mathcal{V}$ is sending a message with the same index i that $\mathcal{A}$ already used. As we explained in Section 4.2.6 the application is responsible for flagging replay attacks. In our case, we used Element Web, which implements such a replay check. After $\mathcal{V}$ sent enough unreadable messages with old message indices i, party $\mathcal{T}$ can successfully decrypt and display legitimate messages of $\mathcal{V}$ again because $\mathcal{V}$ caught up with the message indices $\mathcal{A}$ already depleted and is now sending messages with fresh i again (unreadable because we only see a replay warning). An illustration can be found in Figure 6.2.

Conclusion

The third party $\mathcal{T}$ used $\Omega_P^{\mathcal{V}}$ to decrypt and verify a message from $\mathcal{A}$ but the client of $\mathcal{T}$ did not display it as such. This seemed to be the closest we can get for now and even after swapping to $\texttt{sender\_id}^{\mathcal{V}}$ and $\texttt{device\_id}^{\mathcal{V}}$ there was no difference in the result. The problem seems to lie with the homeservers because they still know the message originates from $\mathcal{A}$. Looking back at the Matrix specification there must be another parameter that is needed for authentication which turns out to be something called the access token, which we will cover in the next section.

6.2 Impersonation with the access token

In the last chapter we tried to impersonate $\mathcal{V}$ but were not successful because it seems like swapping to $R_i^{\mathcal{V}}, sk_{Ed}^{\mathcal{V}}, \texttt{session\_id}^{\mathcal{V}}$ and $\texttt{sender\_key}^{\mathcal{V}}$ is not enough. After a successful login by a user, the corresponding homeserver issues an access token t to authenticate that user. It is similar to a cookie which is transferred inside the authorization header of an HTTP request (see Figure 6.1). The Matrix specification does not mandate any special format for t. Homeservers are free to choose their own format, but they should make sure only one t correlates to a `device_id` [mat17]. As long as the device is "alive", the access token t does not change. Upon logout a device usually "dies" (i.e. the device gets discarded) and whenever a new device registers, usually at login, a new t is generated for that device. If we can leak t we should definitely be able to impersonate $\mathcal{V}$ because it is basically the same as being in possession of their account. Unfortunately t belongs to high level part of Matrix namely the matrix-js-sdk. Since it is generated and processed in the

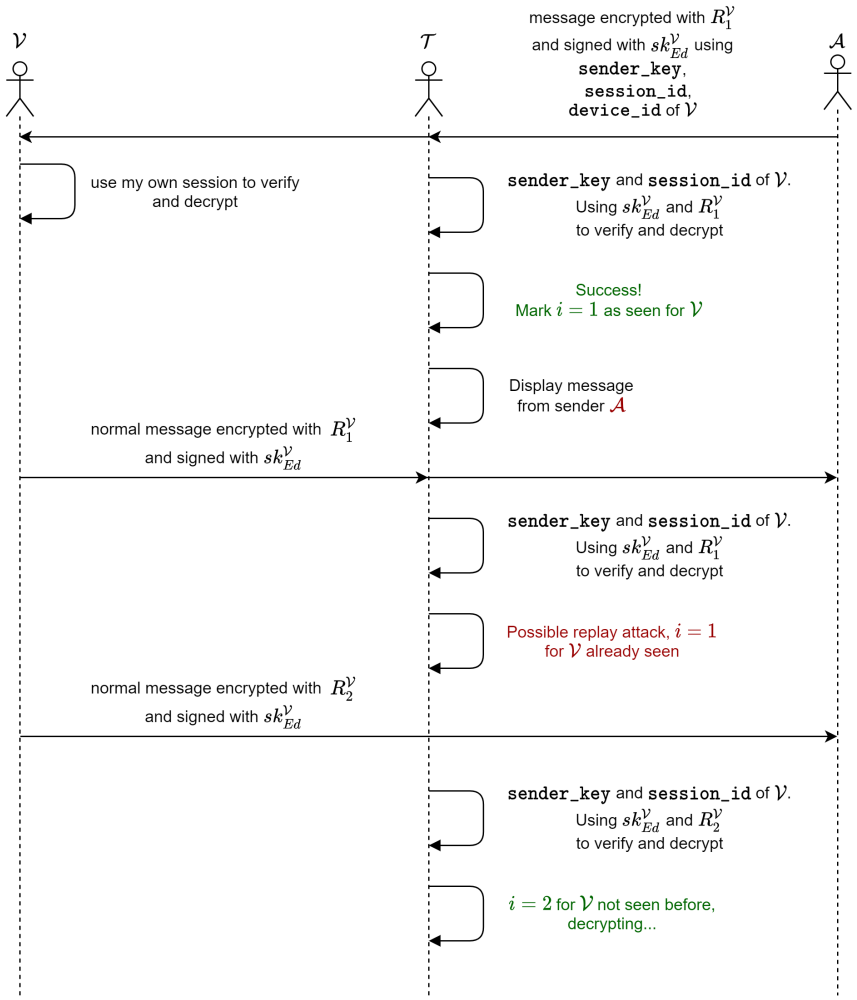

Figure 6.2: Behavior of different parties when trying to impersonate $\mathcal{V}$

application layer only, we have to forward it to the libolm library in order to leak it via the subversion. Let $t^{\mathcal{V}}$ be the access token of $\mathcal{V}$.

Leaking the access token

Now that we do not want to manipulate the specification of the libolm library, because our goal is to make as little changes as possible, we cannot change their function footprints, and we have to find a way to forward $t^{\mathcal{V}}$ into the library without an extra function parameter. To achieve this, we can utilize the array parsing the random data for $R_0^{\mathcal{V}}$ because it is passed via the application layer. Moreover, we do not really need the original $R_0^{\mathcal{V}}$ because it is replaced with $\widetilde{C}$ during the subversion anyway. After we forwarded $t^{\mathcal{V}}$ to the libolm library via the random data for $R_0^{\mathcal{V}}$, we can use Section 5.3 to

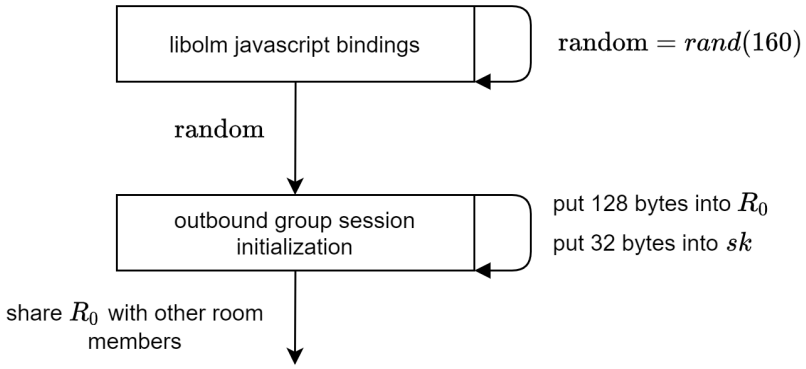

Figure 6.3: Normal behavior of libolm

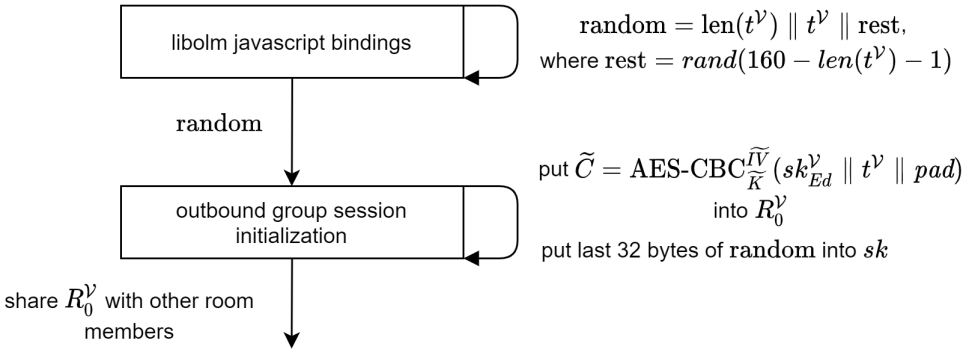

Figure 6.4: Transferring $t^{\mathcal{V}}$ from JavaScript to C

leak it together with $sk^{\mathcal{V}}_{Ed}$ looking like this $R^{\mathcal{V}}_0 = \widetilde{C} = \text{AES-CBC}^{\widetilde{IV}}_{\widetilde{K}}(sk^{\mathcal{V}}_{Ed} \parallel t^{\mathcal{V}} \parallel pad)$ where $|pad| = 112 - len(sk^{\mathcal{V}}_{Ed}) - len(t^{\mathcal{V}})$. An illustration can be found in Figure 6.3 and Figure 6.4. We replace the original random data with $\text{len}(t^{\mathcal{V}}) \parallel t^{\mathcal{V}} \parallel \text{rest}$, where $\text{rest} = rand(160 - len(t^{\mathcal{V}}) - 1)$, and after we forwarded it to the initialization of the outbound group session in libolm, we encrypt and leak it via $R^{\mathcal{V}}_0$.

Implementation

We start by locating $t^{\mathcal{V}}$ in `matrix-js-sdk/src/base-apis.js`, where we define a global variable and copy $t^{\mathcal{V}}$ to that variable. Next we switch into `crypto/OlmDevice.js` where we can find the `createOutboundGroupSession` function. Inside, a session is created, and we add another parameter to transfer $t^{\mathcal{V}}$. The function being called now resides in libolm, so we switch to `olm/js/olm_outbound_group_session.js` where we can find the `create` function, which now receives $t^{\mathcal{V}}$. Moreover, a function named `random_stack` is called which allocates storage on the stack as well as writes the ran-

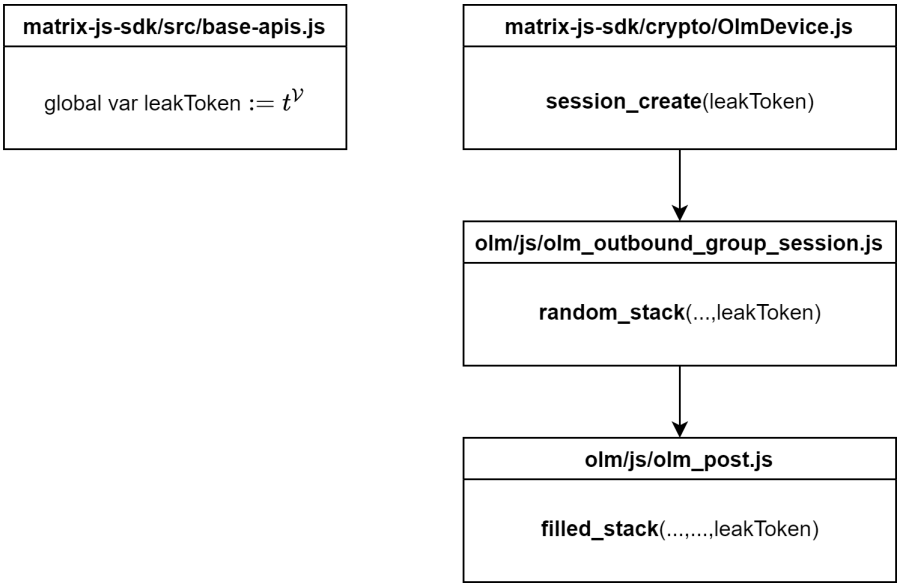

Figure 6.5: Different method calls for passing $t^{\mathcal{V}}$ from JavaScript to C

dom bytes to that address. We now pass that function our token $t^{\mathcal{V}}$ and switch to the file `olm_post.js` where it is defined. Following the trail we land in `filled_stack` which now receives our token $t^{\mathcal{V}}$. It turns out that $t^{\mathcal{V}}$ has a variable length, so in order for C to know how many bytes it has to copy, we had to transfer $\text{len}(t^{\mathcal{V}})$ as well. After the location on the stack is filled with random data, we convert $t^{\mathcal{V}}$ to an uint8 array. The first byte denoting $\text{len}(t^{\mathcal{V}})$ following by $t^{\mathcal{V}}$, we copy the whole byte array to the location on the stack. An illustration for the passing of $t^{\mathcal{V}}$ and all files involved can be found in Figure 6.5 Now the C part of libolm receives $\text{len}(t^{\mathcal{V}})$, $t^{\mathcal{V}}$ and the remaining random data, and we can continue to process $t^{\mathcal{V}}$ there. For that we once again go in the initialization function of the outbound group session inside `olm/src/outbound_group_session.c`, and we now copy $t^{\mathcal{V}}$ from the parameter `random` into an array. The function also holds our subversion from Section 5.3, so we just redefine our input to be $\widetilde{M} = sk_{Ed}^{\mathcal{V}} \parallel t$. Now because in Section 5.3 we defined $\widetilde{IV}$ to be the first 16 bytes of our random data we have to make an adjustment here because the first 16 bytes are always part of $t^{\mathcal{V}}$ and therefore only rarely change. Unfortunately, we cannot use the last 16 bytes of our random data because we replaced almost all random data with $sk_{Ed}^{\mathcal{V}} \parallel t$. The remaining number of bytes is not enough for $\widetilde{IV}$. To keep fulfilling our ind$-security property, we generate fresh randomness before every encryption. Since we are using the Megolm ratchet subversion, we automatically write the $\widetilde{IV}$ to our ciphertext $\widetilde{C}$ and do not need to take care of anything else.

Our original scenario

After successfully leaking the access token $t^{\mathcal{V}}$ to $\mathcal{A}$, we can now get back to our original scenario with the attacker $\mathcal{A}$, the victim $\mathcal{V}$ and a third party $\mathcal{T}$ being in a room where $\mathcal{A}$ wants to impersonate $\mathcal{V}$. Without changing anything except the access token, $\mathcal{T}$ now displays a message of $\mathcal{A}$ in the name of $\mathcal{V}$. After some investigation we see a similar behavior like in Section 6.1. This time, $\mathcal{T}$ displays the message in the name of $\mathcal{V}$ but internally still processing it as a message from $\mathcal{A}$ because we kept `session_id`$^{\mathcal{A}}$ and `sender_key`$^{\mathcal{A}}$. This also results in a neat side effect being that future messages from $\mathcal{V}$ will not get flagged as replay attacks because $\mathcal{T}$ still advances $R_i^{\mathcal{A}}$. The observed behavior seemed to be consistent with the different clients we tested (Element Web [ewe16], Element Android [ean16], Element Desktop [ede16], Nheko Desktop [nhe17], Mirage Desktop [mir19], SchildiChat Android [sch18]). If we do the same thing again but now swapping to `session_id`$^{\mathcal{V}}$ and `sender_key`$^{\mathcal{V}}$ (seen in Figure 6.1) we get the same result but this time $\mathcal{T}$ completely believes that the message of $\mathcal{A}$ came from $\mathcal{V}$ and thus future messages of $\mathcal{V}$ will be flagged as replay attacks by $\mathcal{T}$.

Preventing replay attacks

As stated before, if the attacker $\mathcal{A}$ sends messages in the name of the victim $\mathcal{V}$ using $pk_{Cu}^{\mathcal{V}}$, `session_id`$^{\mathcal{V}}$, $R_i^{\mathcal{V}}$, $(pk_{Ed}^{\mathcal{V}}, sk_{Ed}^{\mathcal{V}})$ and $t^{\mathcal{V}}$ third party $\mathcal{T}$ advances the ratchet $R_i^{\mathcal{V}}$ and therefore $\mathcal{V}$ will send messages with old message indices already used by $\mathcal{A}$. To prevent this, we can further manipulate the implementation of $\mathcal{V}$ by comparing pk_{Ed} of the outbound group session of $\mathcal{V}$ with every inbound group session of $\mathcal{V}$. Whenever they match and the message index of the inbound group session i_{in} is higher than our local ratchet index of our outbound group session i_{out}, we need to advance $R_i^{\mathcal{V}}$ before encrypting the next outgoing message. By doing this, $\mathcal{V}$ now always uses fresh message indices after an impersonation attempt from $\mathcal{A}$ and $\mathcal{T}$ will not flag legitimate messages of $\mathcal{V}$ anymore. An illustration can be found in Figure 6.6.

Pushing things further

After impersonating $\mathcal{V}$ in the same room with $\mathcal{A}$, we tried to advance our attack scenario to any room. Together with $t^{\mathcal{V}}$ and minimal changes, it was also possible to send messages in rooms of $\mathcal{V}$ where $\mathcal{A}$ was not part of. On the implementation side of $\mathcal{A}$, in `matrix-js-sdk/src/client.js`, we swapped to `user_id`$^{\mathcal{V}}$. This step was necessary in order to make the web interface usable and bug free. We head over to our local instance of Element Web and login as $\mathcal{A}$. Now we use a browser add-on to permanently replace $t^{\mathcal{A}}$ with $t^{\mathcal{V}}$ inside the HTTP authorization header. In the web interface of Element, one

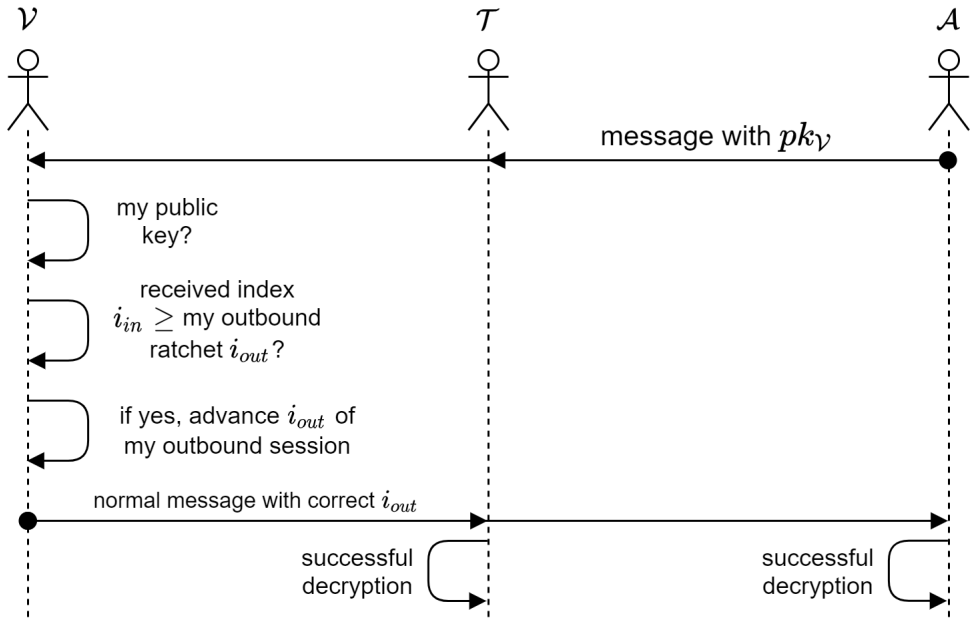

Figure 6.6: Illustration of syncing between in and outbound group sessions

last step is necessary. We head to Settings → Help & About and click the "Clear cache and reload" button. Now $\mathcal{A}$ has the same web interface $\mathcal{V}$ has and $\mathcal{A}$ now fully controls the account of $\mathcal{V}$ using the same device authorized by $t^{\mathcal{V}}$. We are basically mirroring the real device $t^{\mathcal{V}}$ belongs to without adding a new device and can now write messages in every room of $\mathcal{V}$. In addition to that, we (in the role of $\mathcal{A}$) can read all metadata e.g. all rooms from $\mathcal{V}$, who sent a message at what time, the room participants etc. Since Matrix is only using API calls, you could also do all of this manually using $t^{\mathcal{V}}$. The other parties will treat us as a normal device from $\mathcal{V}$ with a different `sender_key` and `device_id`. When $\mathcal{A}$ sends a message, the recipient receives the Olm session of $\mathcal{A}$ authorized by the homeserver via $t^{\mathcal{V}}$ and will accept it without question. In the current state, this seems to be undetectable to any other user, because $\mathcal{A}$ sends their own Olm session and the receiver believes the homeserver that the Olm session of $\mathcal{A}$ came from a legit device of $\mathcal{V}$. The homeserver maintains a device list for each of its users and only devices on that list will be eligible to receive any Olm sessions and therefore keys from other people or other devices of $\mathcal{V}$. For our scenario, we can neither read any inbound nor outbound messages of $\mathcal{V}$. Before an outbound message is sent, the corresponding encrypted key has to be transmitted via an Olm session and since the homeserver believes $\mathcal{A}$ and $\mathcal{V}$ are using the same device according to $t^{\mathcal{V}}$, the victim $\mathcal{V}$ would need to send the key to themselves to make it retrievable by $\mathcal{A}$, which obviously does not happen. An exception may be if $\mathcal{V}$ uses 2 different devices and $\mathcal{A}$ steals $t^{\mathcal{V}}$ from one of the devices while $\mathcal{V}$ messages from the

other device. Then of course, we are able to retrieve the encrypted keys of $\mathcal{V}$, as long as $\mathcal{V}$ uses the other device for messaging. For inbound traffic, we can retrieve the encrypted keys of $\mathcal{V}$ from the homeserver using $t^{\mathcal{V}}$ since $t^{\mathcal{V}}$ represents a valid device. We cannot decrypt any of the decrypted keys inside the Olm sessions though, because we are missing private key material of $\mathcal{V}$. We will refer to this in Section 7.4.

Conclusion

This time we were successful in impersonating $\mathcal{V}$, and we even made it possible to impersonate $\mathcal{V}$ in any room they are in. For this to work, we had to get access to the user account of $\mathcal{V}$ in the form of $t^{\mathcal{V}}$. We also tried to only change the sender id to $\texttt{sender\_id}^{\mathcal{V}}$, but the displayed sender is only controlled by $t^{\mathcal{V}}$. It looks like there is still a level of trust on the homeserver because regarding the sender of a message, it does not matter if $\mathcal{V}$ signed a message as long as $t^{\mathcal{V}}$ is used. The users or homeservers do not verify if the Curve25519 sender key pk_{Cu} actually belongs to the acclaimed sender. If the homeserver tells the client the message came from $\mathcal{V}$, then it came from $\mathcal{V}$. Since everyone can host a homeserver themselves, this might pose a potential security risk.

6.3 Eavesdropping on conversations in other rooms

In this section we want to go into another direction. Let us say, we have two distinct rooms. The members of room 1 denoted as $room_{\mathcal{V},\mathcal{T}}$ are the victim $\mathcal{V}$ and a third party $\mathcal{T}$. They maintain a private conversation with end-to-end encryption. Room 2 denoted as $room_{\mathcal{V},\mathcal{A}}$ consists of $\mathcal{V}$ and the attacker $\mathcal{A}$ also end-to-end encrypted. Our goal now is to eavesdrop on the conversation between $\mathcal{V}$ and $\mathcal{T}$ happening in $room_{\mathcal{V},\mathcal{T}}$ using the direct channel to the attacker $\mathcal{A}$ in $room_{\mathcal{V},\mathcal{A}}$.

How we do it

Remember from Section 5.4, we are able to leak 31 bytes every message $\mathcal{V}$ sends to $\mathcal{A}$ in $room_{\mathcal{V},\mathcal{A}}$. That will be our main leaking channel. Now let us talk about what we are going to leak. Leaking plaintexts is not very efficient because they can be really long and therefore many 31 byte message chunks between $\mathcal{V}$ and $\mathcal{A}$ are needed. Leaking the derived AES_KEY_i and AES_IV_i is also not very meaningful. Even though they are only 48 bytes in length together, they can only decrypt one message because the ratchet value R_i is advanced after every message. Once again the ratchet value R_i sounds the most promising. Leaking the first ratchet value R_0 ensures that $\mathcal{A}$ will be able to decrypt the following 100 messages because $\mathcal{A}$ can compute any successors themselves. Because

Algorithm 10: Outbound Session with leak array size of 4 keys

```
if message_to_attacker then
    if array[index] == empty then // Skip current entry if empty
        index = (index + 1) mod 4
        counter = 0
    S̃(M, sk_Ed, K̃, array[index], counter)
    counter++
    if counter == 5 then // if we leaked 5 chunks, remove the key
        counter = 0
        array.remove[index]
        index++
    if index ≥ 4 then // Reset, if we hit end of array
        index = 0
else
    if !in_array(R_0) then
        array.add(R_0)
    S(M, sk)
```

Algorithm 11: Inbound Session

```
if !in_array(R_0) then // If inbound key not in array, add it
    array.add(R_0)
```

we want to eavesdrop in $room_{\mathcal{V},\mathcal{T}}$, we only have to leak the ratchet values $R_0^{\mathcal{V}}, R_0^{\mathcal{T}}$ from $room_{\mathcal{V},\mathcal{T}}$. Our attack scenario will be the following. Whenever $\mathcal{V}$ messages $\mathcal{A}$ in $room_{\mathcal{V},\mathcal{A}}$, victim $\mathcal{V}$ leaks a 31 byte chunk of either $R_0^{\mathcal{V}}$ or $R_0^{\mathcal{T}}$ from $room_{\mathcal{V},\mathcal{T}}$. An illustration can be found in Figure 6.7. Every ratchet value R_0 is 128 bytes long, so we need 5 chunks to fully leak one ratchet value. Therefore, we need 10 messages to leak both of them.
Now after we discussed how $\mathcal{A}$ will get the ratchet values, let us talk about how they can retrieve the encrypted messages between $\mathcal{T}$ and $\mathcal{V}$ in $room_{\mathcal{V},\mathcal{T}}$. In order to get them, the attacker $\mathcal{A}$ can use the access token $t^{\mathcal{V}}$ that we leaked in Section 6.2. Via a `/sync` API Call using the Client-Server API [mat14a], the attacker $\mathcal{A}$ uses $t^{\mathcal{V}}$ to retrieve all the encrypted messages sent in $room_{\mathcal{V},\mathcal{T}}$ and with $R_0^{\mathcal{V}}, R_0^{\mathcal{T}}$ from $room_{\mathcal{V},\mathcal{T}}$, the attacker $\mathcal{A}$ can successfully decrypt them. A pseudocode for the behavior of the inbound and outbound session is given in Algorithm 10 and Algorithm 11.

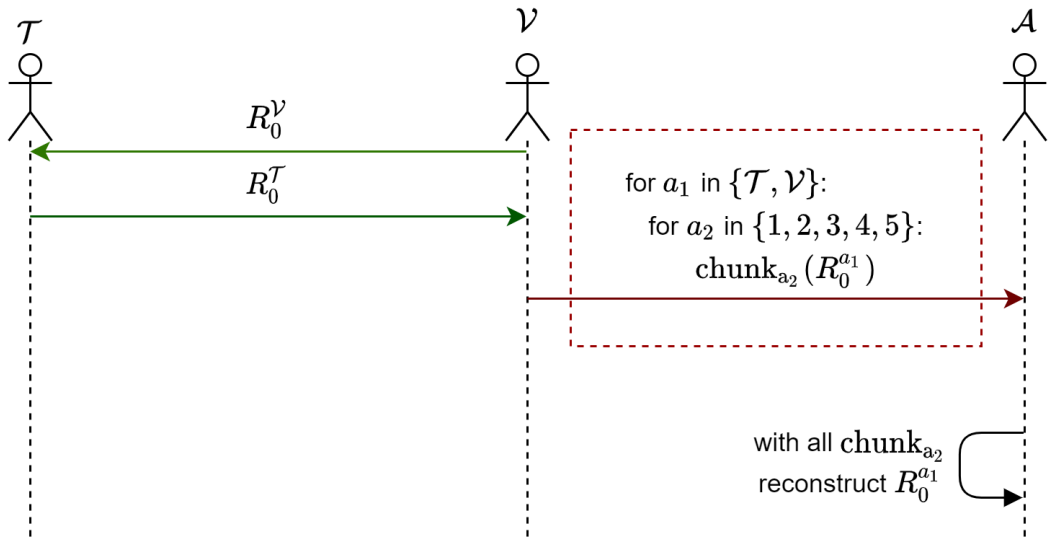

Figure 6.7: Leaking keys from another room to $\mathcal{A}$

Implementation

Going over to the implementation, the biggest issue we had to fight, was linking a message to $\mathcal{A}$. Since the receiver of a message event is always a room, we had no way of distinguishing between $\mathcal{V}$ and $\mathcal{T}$. Fortunately, the initial ratchet value R_0 is sent to each person directly via their Olm session and by that we could do the differentiation. We start in `olm/src/olm.cpp` inside `olm_encrypt` where we can scan every plaintext before encryption to check whether it is going to $\mathcal{A}$. The transmission of a ratchet value a. k. a. `session_key` is called an `m.room_key` event. An example can be found in Figure 6.8. The base64-encoded ratchet value R_0 can be found inside the `session_key` entry. We also see $\texttt{user\_id}^{\mathcal{A}}$ under the recipient entry. With that information we can make a copy of the ratchet value $R_0^{\mathcal{V}}$ going to $\mathcal{A}$. Now, in `olm/src/outbound_group_session.c` we can compare ratchet values every time a message is encrypted. If the ratchet value $R_0^{\mathcal{V}}$ is used, we now know that we are communicating with $\mathcal{A}$. Additionally, we have to advance the copy we made. Otherwise, all subsequent comparisons will fail because $R_0^{\mathcal{V}}$ was advanced to $R_1^{\mathcal{V}}$. Now that, we know when we sent something to $\mathcal{A}$, we can start leaking the ratchet values of $room_{\mathcal{V},\mathcal{T}}$. For our test scenario, we made an array of 4 possible entries. Every time we are messaging in $room_{\mathcal{V},\mathcal{T}}$, we add the initial ratchet values for that person to the array. Whenever we send a message to $\mathcal{A}$ in $room_{\mathcal{V},\mathcal{A}}$, we sequentially leak 31 bytes per message of the first ratchet value in the array. As soon as the first entry is fully transmitted ($\mathcal{V}$ sent 5 messages to $\mathcal{A}$), the ratchet value gets removed from the array and the next entry is going to be leaked. When the current array entry is empty, it is skipped and whenever it hits the end, it just restarts with entry 0. To prevent adding mul-

```
{
    "sender": "@victim:matrix.org",
    "sender_device": "ZGXAUVPQJO",
    "keys": {
        "ed25519": "M++tCeX/D5f0c5yZDODaQEJ+XfcF7o5M3Rgal7+HJGI"
    },
    "recipient": "@attacker:matrix.org",
    "recipient_keys": {
        "ed25519": "T+QDHAQfQEVKQnia1osF3B3FQBxenYwGnYg7CT1T0Xg"
    },
    "type": "m.room_key",
    "content": {
        "algorithm": "m.megolm.v1.aes-sha2",
        "room_id": "!GTOVdzziBcpXDhxwaT:matrix.org",
        "session_id": "HRA/xryLwyAt3j6ihJkv5t5cSOEXQdf/zFZsWYoFyVI",
        "session_key": "AgAAAAiZ//5ruoLCDTYwXWT6Vjblzlmrx PHppgLzDu7RF7Lo2
            Sy1p54NU+vPQwTuk+kmTJDDD0j4MddXm1xL4bg6w+k2lN/9drEC+xHippE1L0m
            9Md0TOr6/zrDqCk9ATdW0zMvqCUf+hLHZPHjW6Ue8keGhn5ZOBYs2Nglry7b5z
            vIVB0QP8a8i8MgLd4+ooSZL+beXEjhF0HX/8xWbFmKBclS6J4AkCLk3tdwkNam
            3VW4hgBFNetPZe1+dFKRAL0zaz20D2wKz2NCnVgtTwj1oT2V0Tx5pd6iej2Ree
            R8Ep2WAA",
        "chain_index": 0,
        "org.matrix.msc3061.shared_history": true
    }
}
```

Figure 6.8: Transmission of a ratchet value R_0 as an m.room_key event

tiple versions of the same ratchet value $(R_0, R_1, R_2, \ldots)$ we only add ratchet values with an $i = 0$, because the initial ratchet value R_0 is enough for decrypting all subsequent messages. For the actual leaking process we call our subverted signing function $\widetilde{\mathcal{S}}$ described in Section 5.4. After the leak, $\mathcal{A}$ can now rebuild the ratchet values from the chunks and via the `/sync` API they can use $t^{\mathcal{V}}$ to retrieve all the messages sent in $room_{\mathcal{V},\mathcal{T}}$. In the end $\mathcal{A}$ can decrypt all of them via python-olm [pyt15].

Conclusion

We successfully leaked the two ratchet values of $room_{\mathcal{V},\mathcal{T}}$ through $room_{\mathcal{V},\mathcal{A}}$ to the attacker $\mathcal{A}$. We are limited by the size of the array and the amount of messages sent from $\mathcal{V}$ to $\mathcal{A}$. Five messages from $\mathcal{V}$ to $\mathcal{A}$ are needed for every ratchet value we want to leak. If $\mathcal{V}$ rarely messages $\mathcal{A}$ and has a lot of chat rooms, the array might be filled at some point without room for new ratchet values and therefore losing access to messages encrypted by these ratchet values.

7 Conclusion

This chapter marks our conclusion to the subject. We will give a short summary as well as discuss the things we discovered. We will give an evaluation of the result as well as discuss countermeasures for the identity binding problem related to the impersonation. In the last section, we show some ideas on how the attacks discovered can be advanced.

7.1 Summary

In the first step we subverted the Megolm Ratchet. With that, we gained a channel with a bandwidth of 112 bytes every 100 message. Even though the bandwidth was not that high we managed to withstand the analysis of an omniscient watchdog giving us the highest possible undetectability level on our list. We used that channel to leak the private signature key of the victim $sk_{Ed}^{\mathcal{V}}$. With that key we could make it possible to subvert the Ed25519 signature scheme and appended subverted signatures $\widetilde{S}$ to every message. Even though we did not reach the highest undetectability level, we made a fair trade off, which brought us down to the online watchdog. With the disposal of the transmission of $\widetilde{IV}$, this signature subversion gave us a much higher bandwidth of 31 bytes with every message. Utilizing those two subversions, we made two attacks possible. We managed to impersonate $\mathcal{V}$ in every room they are part of and sent messages in their name and also eavesdropped on the communication happening between $\mathcal{V}$ and $\mathcal{T}$ inside another room we were not part of by leaking the ratchet values $R_i^{\mathcal{V}}, R_i^{\mathcal{T}}$ used for the encryption.

7.2 Discussion

In an end-to-end encrypted service there should not be trust on any third parties in between. The Matrix website does not reveal whether the homeservers are trusted entities or not but there is a strong guess that they are not because in end-to-end encryption particularly the Signal protocol, the servers are considered untrustworthy. Whenever a user receives a new Olm session, the user does not verify the sender of this session. They accept the session no matter what and solely rely on what the homeserver told them about the sender. Due to this fact, solely the homeserver dictates the sender of a message. The following assumption was not tested and is subject to future work but in theory if two or more homeservers are used, a homeserver cannot choose a sender outside its database

because it did not issue the access token t. However, a homeserver can choose any of its own users as the sender as long as they are part of the room. The other homeservers will acknowledge the event and forward it to the receiver. Despite that, Matrix showed a high defense against Algorithm Substitution Attacks. Due to its design it was impossible to read old messages without specifically leaking the necessary keys R_i during communication. For that kind of security a major trade off has been made in the usability department. Before logging out, you always have to export all values for R_i for every room you are in, otherwise they will be wiped and messages cannot be decrypted anymore. For that, Matrix offers an encrypted key backup on the homeserver to retrieve them later on. The only way to retrieve them without exporting, is via a key request to a currently active session (e.g. your smartphone) that still has the requested R_i values. From an attacker's perspective that means $\mathcal{V}$ has to explicitly accept a key request in order for $\mathcal{A}$ to get the keys. Most likely, this will not succeed and if so, it will be really suspicious. Imagine you will get a popup from another browser session requesting values of R_i that you do not know of. Eavesdropping on future communication was not possible either. In contrast to Signal, which offers no backward secrecy, because once an attacker $\mathcal{A}$ registered a malicious device, they are able to read all future messages [WBPE21], Matrix is an entirely different story. Even after leaking t, which basically gives you access to the account, you cannot read any future messages. The only thing you can learn is some metadata e.g. the different rooms $\mathcal{V}$ is in, the members of the room, timestamps of messages etc. As we discussed in Section 6.2 the access token $t^{\mathcal{V}}$ marks exactly one device and when using the same $t^{\mathcal{V}}$ on 2 different physical devices, the victim $\mathcal{V}$ would have to send their own keys to their own current device in order for $\mathcal{A}$ to be retrievable. The only potentially message-confidentiality-breaking information $\mathcal{A}$ can retrieve are the encrypted keys inside the Olm session from any person messaging $\mathcal{V}$. Apart from that, in order for $\mathcal{A}$ to retrieve all keys and fully break message confidentiality, they would have to register a new device resulting in their own t. The only way to register a legitimate device and read future messages is by doing a login via username and password. This will immediately notify $\mathcal{V}$, and the attacker $\mathcal{A}$ will appear on their list of active devices. In addition to that, every message sent by $\mathcal{A}$ will be flagged as "sent by an unverified device" until $\mathcal{V}$ manually verifies it. As one can probably imagine, this will be more than alarming. Overall, Matrix does not offer a way to undetectably and universally compromise confidentiality of messages.

7.3 Countermeasures

We want to suggest a countermeasure for the impersonation problem we discovered. Remember, solely the access token $t^{\mathcal{V}}$ was enough for $\mathcal{A}$ to impersonate $\mathcal{V}$ in any room they

are in. A receiver of an Olm session does not verify if that session actually came from the acclaimed sender. The receiver solely trusts on what the homeserver told them the sender would be. Every user maintains an updated list of public keys on their homeserver including their public Curve25519 identity key pk_{Cu}. Now either the homeserver or the receiver could verify if the pk_{Cu} inside the Olm session belongs to any devices of $\mathcal{V}$. If not, then this message obviously not came from $\mathcal{V}$ and has to be handled accordingly.

7.4 Future Work

We want to give some ideas on how to advance the things we discovered. In Section 6.2 we came to the conclusion that we are missing key material in order to decrypt the Olm session of a third party. This key material involves the private Curve25519 identity key sk_{Cu} as well as one of the private one-time keys mentioned in [key]. It should be no challenge to leak them as well and decrypt that Olm session resulting in the compromise of R_i, which would break confidentiality. The Olm session is probably reinitialized on a regular basis and therefore a new one-time key is needed every time, which limits the use case of this leak. An attacker $\mathcal{A}$ has to maintain regular conversation with $\mathcal{V}$ in order to keep receiving them. If they manage to do that, they can still only compromise messages from a third party to $\mathcal{V}$, which brings us to our next attack vector. The homeservers are definitely somewhat trusted entities. Since they maintain the device list for each of their users, it may be possible for $\mathcal{A}$ to host a homeserver themselves and add their own device to the device list of every user. If they can do this undetectably, a total compromise of confidentiality may be possible. Being on the device list effectively means, that the malicious device receives ratchet values R_i from all parties and therefore is able to decrypt messages. Furthermore, we have seen that t is enough to impersonate another user and the homeserver is the sole administrator of these tokens. If $\mathcal{A}$ compromises the homeserver, where them and $\mathcal{V}$ are registered on, they may be able to manipulate the sender of a message without any additional knowledge. With the leak of t, it may also be possible to register a malicious device without compromising a homeserver resulting in the same effect as before. Even though Matrix prevents multiple devices being registered on the same t, this restriction may or may not be circumvented.

References

[AMV15] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signature schemes. In *CCS*, pages 364–375. ACM, 2015.

[BBGea13] James Ball, Julian Borger, Glenn Greenwald, and et al. Revealed: how US and UK spy agencies defeat internet privacy and security, 2013.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptogr. Eng.*, 2(2):77–89, 2012.

[BJK15] Mihir Bellare, Joseph Jaeger, and Daniel Kane. Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In *CCS*, pages 1431–1440. ACM, 2015.

[BL17] Sebastian Berndt and Maciej Liśkiewicz. Algorithm substitution attacks from a steganographic perspective. In *CCS*, pages 1649–1660. ACM, 2017.

[BM16] Alex Balducci and Jake Meredith. Olm Cryptographic Review. Technical report, NCC Group, 2016. URL: `https://www.nccgroup.com/globalassets/our-research/us/public-reports/2016/november/ncc_group_olm_cryptogrpahic_review_2016_11_01.pdf`.

[BPR14] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In *CRYPTO (1)*, volume 8616 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2014.

[BSJ+17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 619–650. Springer, 2017.

[bun20] Bundeswehr will komplett auf Matrix-Chat wechseln, 2020. Accessed 2021-04-22. URL: `https://www.golem.de/news/messenger-bundeswehr-will-komplett-auf-matrix-chat-wechseln-2005-148407.html`.

[BWP+20] Sebastian Berndt, Jan Wichelmann, Claudius Pott, Tim-Henrik Traving, and Thomas Eisenbarth. ASAP: algorithm substitution attacks on cryptographic

protocols. *IACR Cryptol. ePrint Arch.*, 2020:1452, 2020. URL: https://eprint.iacr.org/2020/1452.

[CHY20] Rongmao Chen, Xinyi Huang, and Moti Yung. Subvert KEM to break DEM: practical algorithm-substitution attacks on public-key encryption. In *ASIACRYPT (2)*, volume 12492 of *Lecture Notes in Computer Science*, pages 98–128. Springer, 2020.

[cli] Matrix Different Clients Overview. Accessed 2021-06-14. URL: https://matrix.org/clients/.

[dev20] Element Web GitHub Development Environment, 2020. Accessed 2021-05-04. URL: https://github.com/vector-im/element-web#setting-up-a-dev-environment.

[DH79] Whitfield Diffie and Martin Hellman. Privacy and Authentication: An Introduction to Cryptography. In *Proceedings of the IEEE*, volume 67, pages 397–427, 1979.

[ean16] Element Android GitHub, 2016. Accessed 2021-05-04. URL: https://github.com/vector-im/element-android.

[edd17] Edwards-Curve Digital Signature Algorithm (EdDSA) RFC 8032 Chapter 5.2.3, 2017. Accessed 2021-06-05. URL: https://datatracker.ietf.org/doc/html/rfc8032#section-5.2.3.

[ede16] Element Desktop GitHub, 2016. Accessed 2021-05-04. URL: https://github.com/vector-im/element-desktop.

[edg17] Edwards-Curve Digital Signature Algorithm (EdDSA) RFC 8032, 2017. Accessed 2021-06-05. URL: https://datatracker.ietf.org/doc/html/rfc8032.

[ewe16] Element Web GitHub, 2016. Accessed 2021-05-04. URL: https://github.com/vector-im/element-web.

[fre18] French government Matrix, 2018. Accessed 2021-04-22. URL: https://matrix.org/blog/2018/04/26/matrix-and-riot-confirmed-as-the-basis-for-frances-secure-instant-messenger-app.

[git20] GitHub Issue Missing Replay Protection, 2020. Accessed 2021-08-10. URL: https://github.com/vector-im/element-android/issues/1990.

[Gre14] Glenn Greenwald. No place to hide: Edward Snowden, the NSA, and the US surveillance state, 2014.

[Hen20] Floris Hendriks. Analysis of key management in Matrix, 2020. URL: https://www.cs.ru.nl/bachelors-theses/2020/Floris_Hendriks___4749294___Analysis_of_key_management_in_Matrix.pdf.

[hkd10] HMAC-based Extract-and-Expand Key Derivation Function (HKDF), 2010. Accessed 2021-08-13. URL: https://datatracker.ietf.org/doc/html/rfc5869.

[hma97] HMAC: Keyed-Hashing for Message Authentication, 1997. Accessed 2021-08-14. URL: https://datatracker.ietf.org/doc/html/rfc2104.

[key] End-to-End Encryption Implementation Guide. Accessed on 2021-07-12. URL: https://matrix.org/docs/guides/end-to-end-encryption-implementation-guide.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[lib16] Libolm Library (Matrix GitLab), 2016. Accessed 2021-05-08. URL: https://gitlab.matrix.org/matrix-org/olm/-/tree/master.

[mat] Matrix Specification. Accessed 2021-08-14. URL: https://matrix.org/docs/spec/.

[mat14a] Matrix Client-Server API, 2014. Accessed on 2021-07-12. URL: https://matrix.org/docs/api/client-server/.

[mat14b] Matrix Website, 2014. Accessed on 2021-08-14. URL: https://matrix.org/.

[mat17] Client-Server API — Relationship between access tokens and devices, 2017. Accessed 2021-08-13. URL: https://matrix.org/docs/spec/client_server/latest#relationship-between-access-tokens-and-devices.

[meg19] Megolm Protocol (Matrix GitLab), 2019. Accessed 2021-04-22. URL: https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/megolm.md.

[mir19] Mirage Desktop Client GitHub, 2019. Accessed 2021-05-04. URL: https://github.com/mirukana/mirage.

[mro18] Matrix Client-Server API m.room.encrypted event, 2018. Accessed on 2021-07-12. URL: https://matrix.org/docs/spec/client_server/latest#m-room-encrypted.

[nhe17] Nheko Desktop Client GitHub, 2017. Accessed 2021-05-04. URL: https://github.com/Nheko-Reborn/nheko.

[olm19] Olm Protocol (Matrix GitLab), 2019. Accessed 2021-04-22. URL: https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/olm.md.

[PLS13] Nicole Perlroth, Jeff Larson, and Scott Shane. NSA able to foil basic safeguards of privacy on web, 2013.

[pyt15] Python-Olm: Python bindings for Olm, 2015. Accessed on 2021-07-12. URL: https://gitlab.matrix.org/matrix-org/olm/-/tree/master/python.

[Rog11] Phillip Rogaway. Evaluation of Some Blockcipher Modes of Operation. University of California, Davis, 2011.

[roo] Room Definition. Accessed 2021-08-01. URL: https://matrix.org/docs/spec/#id15.

[RTYZ16] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Cliptography: Clipping the power of kleptographic attacks. In *ASIACRYPT (2)*, volume 10032 of *Lecture Notes in Computer Science*, pages 34–64, 2016.

[Sch11] Dr. Schorsch. A mechanical ratchet, 2011. URL: https://creativecommons.org/licenses/by-sa/3.0/.

[sch18] SchildiChat Android GitHub, 2018. Accessed 2021-05-04. URL: https://github.com/SchildiChat/SchildiChat-android.

[sig16] Signal Documentation, 2016. Accessed on 2021-04-22. URL: https://signal.org/docs/specifications/doubleratchet/.

[TBEL21] Thore Tiemann, Sebastian Berndt, Thomas Eisenbarth, and Maciej Liśkiewicz. "Act natural!": Having a private chat on a public blockchain. 2021. URL: https://eprint.iacr.org/2021/1073.

[var20] Protocol Buffers encoding, 2020. Accessed on 2021-07-12. URL: `https://developers.google.com/protocol-buffers/docs/encoding`.

[WBPE21] Jan Wichelmann, Sebastian Berndt, Claudius Pott, and Thomas Eisenbarth. Help, my signal has bad device! - breaking the signal messenger's post-compromise security through a malicious device. In *DIMVA*, volume 12756 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2021.

[YY96] Adam L. Young and Moti Yung. The dark side of "black-box" cryptography, or: Should we trust capstone? In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 1996.

[YY97] Adam L. Young and Moti Yung. Kleptography: Using cryptography against cryptography. In *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1997.