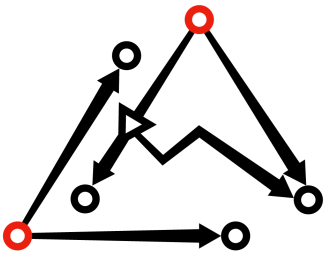

Least Authority

PRIVACY MATTERS

vodozemac

Security Audit Report

Matrix

Final Audit Report: 30 March 2022

Table of Contents

This audit makes no statements or warranties and is for discussion purposes only.

This audit makes no statements or warranties and is for discussion purposes only.

Overview

Background

Matrix has requested that Least Authority perform a security audit of vodozemac, the Rust implementation of the libolm cryptographic library. libolm implements the Olm and Megolm ratchets, and is originally written in C/C++.

Project Dates

- **December 20 - January 25**: Code Review *(Completed)*
- **January 28**: Delivery of Initial Audit Report *(Completed)*
- **March 28 - 29:** Verification *(Completed)*
- **March 30:** Delivery of Final Audit Report *(Completed)*

Review Team

- Anna Kaplan, Cryptography Researcher and Engineer
- Ann-Christine Kycler, Cryptography Researcher and Engineer
- Denis Kolegov, Security Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer
- Rai Yang, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the vodozemac implementation, followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- vodozemac: https://github.com/matrix-org/vodozemac

Specifically, we examined the Git revisions for our initial review:

`7c11a501bc316a0bf92a5fe06fee8582aad24897`

For the verification, we examined the Git revision:

57d8d87a747653d6d7b7a53acb9a8d8f8de48285

For the review, this repository was cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/Matrix_Vodozemac

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third party code, including the Android/Java, Python, and JavaScript bindings, unless specifically included above, are considered out of scope.

Supporting Documentation

The following documentation was available to the review team:

- vodozemac README.md: https://github.com/matrix-org/vodozemac/blob/main/README.md

This audit makes no statements or warranties and is for discussion purposes only.

- Megolm Specification: https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/megolm.md
- Olm Specification: https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/olm.md
- Short Authentication String (SAS) Specification: https://spec.matrix.org/unstable/client-server-api/#short-authentication-string-sas-verification
- Rendered Crates Documentation: https://matrix-org.github.io/vodozemac/vodozemac/index.html
- The Double Ratchet Algorithm: https://signal.org/docs/specifications/doubleratchet/

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation, including cryptographic constructions and primitives;
- Common and case-specific implementation errors;
- Networking and communication with external data;
- Secure key storage and proper management of encryption, ratchet, Diffie-Hellman, and signing keys;
- Performance problems or other potential impacts on performance;
- Data privacy, data leaking, and information integrity;
- Resistance to DDoS (Distributed Denial of Service) and similar attacks;
- Issues resulting from manual memory management;
- Vulnerabilities in the code leading to adversarial actions and other attacks;
- Protection against malicious attacks and other methods of exploitation;
- Performance problems or other potential impacts on performance;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a broad and comprehensive review of vodozemac, a Rust implementation of the libolm cryptographic library that can be used to create an end-to-end encrypted communication channel. The library consists of an implementation of Olm, which is a double ratchet algorithm that is used for peer-to-peer, end-to-end encryption and provides its users the benefit of forward secrecy and post-compromise security. The library also includes an implementation of Megolm, a single ratchet algorithm that is used to secure a group communication channel.

We performed a close investigation of the areas of concern, in addition to possible attack vectors such as Olm and Megolm session management, partial forward secrecy in Megolm, and interaction with higher level applications. Our team reviewed the double ratchet algorithm cryptographic implementation, including the ratchet state change in Olm (active/inactive). In addition, we examined the implementation of key creation, storage, deletion, and verification for the ratchet key, root key, chain key, and message key, in addition to investigating their potential compromise. In general, we found that the vodozemac system design and implementation considers security, as demonstrated by the design choices that are driven by trade-offs between security, availability, and functionality.

System Design

In our review of the system design, we identified a number of issues and suggestions, as detailed below. Resolving the issues and suggestions will result in a more robust implementation and benefit the overall security of the library.

This audit makes no statements or warranties and is for discussion purposes only.

We identified a vulnerability by which an attacker can use a victim's one-time key to create an invalid Olm session, which results in disabling other users from using the same key to establish a valid inbound session with the victim. We recommend implementing a check verifying that the ciphertext in the pre-key message is successfully decrypted before the one-time key is removed when creating inbound sessions (Issue G).

In the current implementation, the compromise of the `initial_ratchet` value in Megolm inbound group sessions would enable an attacker to decrypt all messages encrypted from that ratchet, and in all subsequent ratchets. This would lead to the compromise of historical messages in the inbound group sessions. We recommend enabling the user of Megolm library to set the `initial_ratchet` to a more recent ratchet value (Issue H).

We found that the cleartext keys that are used in the vodozemac library are susceptible to leaks as a result of insufficient safeguards against Swap access and side-channel attacks, which could consequently undermine the confidentiality and authenticity properties of Olm and Megolm. We recommend that more secure use be made of Swap access, and that secret keys be encrypted when not in use (Issue I).

In addition, our team noted that the implementation of the message authentication code (MAC) tag does not adhere to best practices (i.e. it is truncated to 64 bits, while 128 bits is specified as the minimum), which undermines the security assumptions of the authentication. We recommend adhering to best practice recommendations for MAC tag length (Issue J). Furthermore, we found that a strict version of the Ed25519 signature scheme is not used. We recommend performing group malleability checks in order to prevent abuse of signatures and decrease the attack surface (Suggestion 5).

Security/Functionality of Retained Keys

In order to encrypt out-of-order messages, Olm currently keeps a maximum of 5 previous receiving chains, and 40 message keys in each receiving chain. This limit is set as a protection against DoS attacks resulting from an attacker causing a user to store too many skipped message keys. However, any stored keys are vulnerable such that if compromised, the attacker could have the ability to decrypt previous messages for all stored keys. This is a design trade-off of functionality over security.

Given that the maximum number of skipped message keys is set to 40, we identified a concern that it is within reason that a user can receive a sufficient amount of out-of-order messages to fill up that limit, afterwhich the user would no longer be able to decrypt out of order messages because they are no longer able to hold on to additional skipped message keys. As a result, we recommend enabling users of the library to configure the number of chain keys and message keys stored, with the current configuration set as the default (Suggestion 8).

Given that the security of the chain and message key depends on the ratchet advancing, we also identified a concern such that if a party's ratchet is not advancing (as a result of not receiving the new ratchet from the counterparty or no reply), this will cause the receiving chain key to be predictable once the old receiving chain keys are compromised. As a result, this will enable the derivation of the message key. The out-of-order messages received can be decrypted indefinitely. As a result, we recommend that the Matrix team conduct additional research on securing chain keys. Furthermore, we suggest removing previous chain keys as soon as possible, and setting limits for sending messages with the same sending chain key.

Code Quality

The vodozemac codebase is well written and organized, and generally adheres to Rust development best practices. We identified several issues and suggestions in the coded implementation relating to the

This audit makes no statements or warranties and is for discussion purposes only.

deviation from recommended best practices, implementation errors, and input validation that could affect the security and the functionality of the library, as detailed below.

Deviations from Best Practice

We identified two areas where the code deviates from recommended best practices. First, we found that a check is not implemented for the `forget` function. Rust development best practice recommends against the use of the `forget` function for secure environments, in order to reduce the risk of making critical memory resources unreachable and preventing sensitive data being removed from the memory. As a result, we recommend implementing a check to detect and process the use of the `forget` function (Suggestion 2). In addition, we recommend using the struct `ReusableSecret` instead of `StaticSecret` type for the one-time keys, in accordance with the x25519-dalek documentation. This would help prevent the leakage or reuse of secret keys, which could decrease the security level of cryptographic protocol (Suggestion 3).

Implementation Errors

We identified several implementation errors that could impact the security and functionality of the library. In multiple instances, zeroization of sensitive data is not implemented correctly. Ineffective or missing zeroization could make sensitive data, such as cleartext private keys, accessible to an attacker. As a result, we recommend implementing zeroization more consistently and effectively (Issue A). In addition, an implementation error in the function `from_libolm_pickle` exists whereby a missing assertion could cause the system to behave unexpectedly. In particular, a legacy pickle that contains zero `key_id`'s is decrypted using `from_libolm_pickle`. We recommend that the function be corrected to return 0 instead of 1 (Issue C). Furthermore, there are several instances of variables susceptible to integer underflows or overflows in the current implementation of vodozemac, which could result in unexpected behavior and potentially lead to a denial of service. As a result, we recommend implementing appropriate overflow and underflow protections (Issue D; Issue E; Issue F).

Input Validation

Finally, we identified an instance of insufficient input validation, which could result in unintended behavior leading to denial of service. We recommend appropriately validating all function inputs (Issue B). We also recommend verifying that the inner vector length of the implementation block `OlmMessage` is equal to or greater than 8 bytes, in order to prevent the function from panicking and causing the system to behave unexpectedly (Suggestion 4). Finally, we suggest that the inner vector length be validated when computing the starting index of a slice in order to prevent unexpected behavior that may lead to errors and security vulnerabilities (Suggestion 6).

Tests

The vodozemac implementation contains some test coverage. However, there are cases that are not covered by the existing tests. Tests help with the detection of implementation errors and unintended behavior that may lead to security vulnerabilities, in addition to providing users and reviewers of the library a better means to understand the intended functionality of the code. As a result, we recommend expanding the test suite such that coverage is comprehensive and includes success, failure, and edge cases (Suggestion 7).

Documentation

The vodozemac project documentation was accurate and helpful in describing each of the components of the system and the interactions between those components. This aided our team in understanding the system's design and intended behavior, and evaluating the correctness of the implementation. Robust project documentation is an important part of security due diligence.

This audit makes no statements or warranties and is for discussion purposes only.

Code Comments

The documentation within the code is sufficient and clearly describes the intended behavior of each of the components that are critical to the functionality and security of the system.

Scope

The in-scope repository was sufficient and included all the security critical components of vodozemac. However, the code's interactions with high level applications (e.g. Matrix) was out of scope for this review. In addition, the cryptographic design of Olm and Megolm was considered out of scope and presumed to function as intended for this review. We recommend that the Matrix team pursue further cryptographic analysis of these libraries by an independent security auditing team to further strengthen the ability to reason about the security characteristics of vodozemac. This is particularly important since some areas of concern can only be reflected when understanding the larger cryptographic design on which the vodozemac implementation depends.

Use of Dependencies

The vodozemac implementation utilizes several outdated dependencies, which can introduce known and unknown vulnerabilities into the codebase. We recommend using well maintained and up to date dependencies, in addition to closely monitoring dependencies using available tools for updates and security developments (Suggestion 1).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Incorrect Implementation of Zeroing Sensitive Data	Resolved
Issue B: Weak Input Validation in bytes_raw Function	Resolved
Issue C: Erroneous key_id Calculation in from_libolm_pickle if Number of One Time Keys is Zero	Resolved
Issue D: Potential Overflow of OneTimeKeys.key_id	Resolved
Issue E: Potential Integer Underflow in advance in megolm/ratchet.rs	Resolved
Issue F: Potential Integer Underflow in advance_to in megolm/ratchet.rs	Resolved
Issue G: Invalid Inbound Session can be Created Causing Removal of Unused One-Time Keys	Resolved
Issue H: Cannot Permanently and Explicitly Remove Old Ratchet State in the Megolm Inbound Group Session	Resolved
Issue I: Keys in Memory Not Secure Against Swap Access and Side-Channel Attacks	Unresolved

This audit makes no statements or warranties and is for discussion purposes only.

Issue J: MAC Tag Truncated to Insufficient Length	Unresolved
Suggestion 1: Update and Maintain Dependencies	Resolved
Suggestion 2: Use Clippy to Automatically Detect forget	Resolved
Suggestion 3: Use ReusableSecret Instead of StaticSecret	Resolved
Suggestion 4: Validate OlmMessage's inner Vector Length When Extracting Payload	Resolved
Suggestion 5: Use Strict Version of Ed25519 Signature Check	Resolved
Suggestion 6: Validate inner Length in append_mac_bytes	Resolved
Suggestion 7: Increase Test Coverage	Resolved
Suggestion 8: Make Number of Chain and Message Keys Configurable	Unresolved

Issue A: Incorrect Implementation of Zeroing Sensitive Data

Location

Examples (non-exhaustive):

`src/utilities.rs#L106-L109`

`src/olm/session/chain_key.rs#L33-L36`

`src/cipher/key.rs#L54-L60`

`src/megolm/inbound_group_session.rs#L221`

`src/sas.rs#L289`

`src/olm/account/mod.rs#L326`

Synopsis

An attacker that is able to access memory (e.g. accessing core dump, using debuggers, and exploiting vulnerabilities such as Heartbleed) may be able to retrieve non-zeroized sensitive information in cleartext, including, but not limited to, private keys, chain keys, and AES keys.

While the zeroize crate is currently used for the main data structures of the library, zeroization is missing in a number of locations for arrays or is ineffective for types with value semantics.

Impact

The leakage of cryptographic keys could result in loss of security properties such as confidentiality and privacy.

This audit makes no statements or warranties and is for discussion purposes only.

Preconditions

An attacker must be able to read memory regions that contain sensitive data.

Mitigation

We recommend first identifying all instances where sensitive data must be zeroized, and then verifying that the data in each instance is appropriately zeroized. In addition, we recommend that attention be paid to peculiarities in several types in Rust, particularly to stack-allocated values, which require appropriate methods for zeroing data.

Status

The Matrix team has addressed the issue by adding Box wrappers as well as by putting secrets behind a Box to minimize the number of copying.

Verification

Resolved.

Issue B: Weak Input Validation in bytes_raw Function

Location

`src/sas.rs#L247`

Synopsis

The function `bytes_raw` will panic if the value of `count` argument is more than USIZE * 255.

Impact

This could result in a DoS attack.

Preconditions

An attacker must be able to set `count` to a value more than USIZE * 255.

Mitigation

We recommend propagating the error from the HKDF's `expand` function to the caller of the `bytes_raw` function.

Status

The Matrix team has propagated the error from HKDF's `expand` function.

Verification

Resolved.

Issue C: Erroneous key_id Calculation in from_libolm_pickle if Number of One-Time Keys is Zero

Location

`src/olm/account/mod.rs#L398`

Synopsis

An implementation error in the function `from_libolm_pickle` exists such that a missing assertion could cause the system to behave unexpectedly. In particular, a legacy pickle that contains zero `key_id`'s

is decrypted using `from_libolm_pickle`. In the case where there are no one-time keys in a pickle, the `key_id` value will be 1 instead of 0.

Impact

After deserializing an input libolm pickle, an incorrect value of `key_id` is returned in the `one_time_keys` type. This could impact the logic of the system and cause unintended behavior leading to security vulnerabilities.

Mitigation

We recommend implementing an assertion that `key_id` must be equal to 0 if there are no one-time keys in the libolm pickle.

Status

The Matrix team has added an assertion to check if the number of one-time keys in the pickle is zero and increment the `key_id` only if the number of the keys is not zero.

Verification

Resolved.

Issue D: Potential Overflow of OneTimeKeys.key_id

Location

`src/olm/account/one_time_keys.rs#L81`

Synopsis

The `self.key_id` variable of the u64 type is used to store and calculate the identification numbers of one-time-keys. While generating new one-time keys, each next identifier is calculated as `self.key_id += 1`. This operation could potentially cause an overflow.

Impact

An overflow would most likely impact the logic of the system or cause a panic, which could lead to a denial of service.

Preconditions

To be successful, this would require the misuse of the `generate` function or a malicious initial `key_id` set by the untrusted server.

Mitigation

We recommend implementing appropriate safeguards, such as the wrapping addition.

Status

The Matrix team has added the wrapping addition.

Verification

Resolved.

This audit makes no statements or warranties and is for discussion purposes only.

Issue E: Potential Integer Underflow in advance in megolm/ratchet.rs

Location

`src/megolm/ratchet.rs#L131`

Synopsis

If `h == 0` at the starting point of the loop, then unsigned `i` will be underflowed. This could lead to attempts to access unallocated memory.

Impact

An underflow or overflow issue would most likely impact the logic of the system or cause a panic, which could lead to a denial of service.

Preconditions

Any set of circumstances where `self.counter & mask == 0` would result in `h == 0`.

Mitigation

We recommend using the wrapping addition and reimplementing the logic of the loop such that the only allocated parts of memory are accessed.

Status

The Matrix team has reimplemented the loop logic as suggested.

Verification

Resolved.

Issue F: Potential Integer Underflow in advance_to in megolm/ratchet.rs

Location

`src/megolm/ratchet.rs#L194`

Synopsis

In the current implementation, unsigned `k` will be underflowed if the value of the loop variable `j` is equal to `0`, which could lead to attempts to access unallocated memory.

Impact

An overflow would most likely affect the logic of the system or cause a panic, which could cause a denial of service.

Preconditions

The value of variable `j` is equal to `0`.

Mitigation

We recommend using wrapping subtraction and reimplementing the logic of the loop such that the only allocated parts of memory are accessed.

Status

The Matrix team has reimplemented the loop logic as suggested.

This audit makes no statements or warranties and is for discussion purposes only.

Verification

Resolved

Issue G: Invalid Inbound Session can be Created Causing Unused One-Time Keys Removal

Location

src/olm/account/mod.rs#L226-L228

src/olm/account/mod.rs#L207

Synopsis

In the function `create_inbound_session`, an invalid inbound session can be created by sending a pre-key message with an invalid ciphertext. Since authentication of the ciphertext inside the pre-key message is not performed, an attacker can create an invalid inbound session with the victim's one-time key, which results in the removal of the unused one-time key after the session's creation.

Impact

If the victim's one-time key is used by an attacker thus removing it, an unsuspecting user using the same one-time key will not be able to communicate with the victim (e.g. create an inbound session).

Preconditions

An attacker creates a session with a victim's one-time key and sends a pre-key message containing an invalid encrypted ciphertext.

Technical Details

In `create_inbound_session`, the receiving party receives a pre-key message, in which it looks for the private part of its one-time key and decodes the remote one-time key, the remote identity key, from which together with the private part of its one-time key derives a shared secret which computes the root key and chain key. Finally, the one-time key is removed.

However, the ciphertext in the pre-key message is not decrypted in accordance with the Olm specification. Consequently, an attacker can create a pre-key message with an invalid ciphertext (e.g. encrypted with an incorrect message key derived from an incorrect chain key).

This would allow the receiving party to create an invalid inbound session without correctly decrypting the ciphertext (derive the message key from the chain key, and correctly decrypt the ciphertext) and the one-time key would be removed unused. As a result, an unsuspecting user using the same one-time key will not be able to communicate (create an inbound session) with the victim because the one-time key is already removed.

Remediation

We recommend decrypting the ciphertext in the pre-key message when creating an inbound session, and only removing the one-time key if the ciphertext is decrypted successfully.

Status

The Matrix team has resolved the issue by decrypting the pre-key message at session creation and subsequently removing the one-time key that was used to create the inbound session.

This audit makes no statements or warranties and is for discussion purposes only.

Verification

Resolved.

Issue H: Cannot Permanently and Explicitly Remove Old Ratchet State in the Megolm Inbound Group Session

Location

`src/megolm/inbound_group_session.rs`

Synopsis

In the Megolm implementation inbound group session, the initial ratchet value (`InboundGroupSession::initial_ratchet`) can be used to decrypt historical messages (e.g. received past the corresponding point of time). If this value is compromised, an attacker can decrypt past messages, which were encrypted by a key derived from the compromised or subsequent ratchet values, breaking the cryptographic principle of forward secrecy.

There are functions to export and import the `InboundGroupSession` at a given message index, but there is no explicit way to remove the old ratchet value in a session.

Impact

An attacker can decrypt past messages that were encrypted by a key derived from the compromised earliest ratchet value (`initial_ratchet`) or subsequent ratchet values.

Preconditions

An attacker captures the `initial_ratchet` in a Megolm inbound group session.

Remediation

We recommend implementing a permanent and explicit way to remove previous ratchet state values in the Megolm inbound group session. The user of the library should be able to choose to remove or advance the previous initial ratchet value up to a more recent value.

Status

The Matrix team has added a function, `advance_to`, in inbound group sessions to permanently advance the session ratchet value to the given index. This removes the ability to decrypt messages that were encrypted with a lower message index than what is given as the argument.

Verification

Resolved.

Issue I: Keys in Memory Not Secure Against Swap Access and Side-Channel Attacks

Synopsis

If the attacker has access to the user's swap space or can mount side-channel attacks, they may have access (usually unreliable) to the memory of arbitrary processes, including those making use of vodozemac. Since keys are not protected while in memory, this may compromise their security.

This audit makes no statements or warranties and is for discussion purposes only.

Impact

This could result in leakage of secret keys, including identity keys, ephemeral keys (i.e. one-time keys or pre-keys), ratchet keys, as well as encryption and authentication keys. This in turn would undermine the confidentiality and authenticity properties of Olm and Megolm.

Preconditions

The attacker has access to the swap of a user or the machine and operating system (OS) of the user is susceptible to side-channel attacks that undermine process separation.

Feasibility

The attacker would have to find the keys in the pieces of the memory extracted. A successful attack depends on the specific system and the amount of data to be extracted. The attack is not straightforward, but possible in many circumstances.

Technical Details

Swap refers to space on the SSD/HDD reserved to store data that resides in memory while it is not needed. On some systems, it is also used for keeping the memory contents during hibernation (also known as suspend-to-disk), which means that memory contents are written to disk, where an adversary may have access to it.

Side-channel attacks describe a wide range of attacks. In this context, we specifically refer to attacks like Meltdown and Spectre and Rowhammer, which allow one process to access memory regions allocated to another process with moderate accuracy.

Mitigation

We recommend the Matrix team further employ the mitigations implemented by OpenSSH. Here, keys are encrypted while not in use, using a key derived from a 16kB buffer filled with random data, only decrypted when needed and immediately disposed afterwards (i.e. zeroized). This hinders attackers, because they not only need to acquire the keys, but also the 16kB pre-key region in order to decrypt them. Since the probability of a read error increases as the amount of read data grows, having to read significantly more data effectively reduces the success probability of this class of attacks.

In addition, users of applications using vodozemac should ensure that the operating system they use employs all available mitigations against attacks from the Spectre and Meltdown families. Additionally, they should make sure that they either do not use swap at all, or configure their system such that it is encrypted.

Status

The Matrix team has acknowledged the suggested mitigation would provide enhanced security, however, they have stated they will not implement the mitigation until a future date.

Verification

Unresolved.

Issue J: MAC Tag Truncated to Insufficient Length

Location

`src/cipher/mod.rs#L30-L39`

This audit makes no statements or warranties and is for discussion purposes only.

Synopsis

In the vodozemac implementation, HMAC-SHA256 tags are truncated to 8 bytes (i.e. 64 bits). However, according to the HMAC specification, they may only be truncated to half of the length of the underlying hash (128 bits in this case). Thus, the current implementation violates the recommendations from the HMAC specification.

Impact

An insufficient tag length weakens the authentication scheme, which increases the probability of successfully modifying a ciphertext.

Preconditions

The attacker must be able to impersonate a sender on the underlying insecure channel. In the Matrix setting, home servers are able to perform such attacks.

Feasibility

The probability of an attacker guessing a tag for which the verification succeeds is 2^{-64}, which is a relatively high probability, compared to those in other cryptographic settings. However, an attacker only gets one guess per message, so a brute-force attack is not possible with just one message.

Technical Details

A MAC tag allows the receiver to verify that the received data comes from the intended receiver (assuming the two are the only parties with access to the key). If the MAC tag becomes shorter, it becomes easier to guess the tag for which the validation succeeds. Typically, in cryptographic operations the security target of about 128 bits is chosen, and most parts of the system do in fact achieve it.

The truncation of the MAC reduces the security target to only 64 bits, falling short of best practices.

Remediation

We recommend updating the code to truncate to not less than 16 bytes or not at all. We acknowledge that this would require a protocol change, which is not unilaterally possible by the Matrix team in order to maintain compatibility with the larger Matrix ecosystem.

Status

The Matrix team responded that while they agree that modern security targets should be met, the changes in truncation of the MAC are inherited from the libolm implementation and would require a coordinated effort on the Matrix Protocol level to ensure compatibility between implementations. Additionally, the Matrix team assesses the probability of an attack resulting from the truncation of a MAC to 8 bytes as low.

We agree with the assessment by the Matrix team and understand that a change of the Matrix Protocol would be required for remediation of this issue. Nonetheless, we recommend coordinating a protocol change within the Matrix ecosystem and updating the MAC truncation.

Verification

Unresolved.

This audit makes no statements or warranties and is for discussion purposes only.

Suggestions

Suggestion 1: Update and Maintain Dependencies

Synopsis

The `rand 0.7`, `thiserror 1.0.26`, `serde 1.0.126`, `serde_json 1.0.64` and `zeroize 1.2.23` dependencies are outdated. A robust development process includes the regular maintenance and updates of dependencies, in order to minimize the risk of introducing known and unknown vulnerabilities into the codebase.

Mitigation

We recommend updating or replacing the reported dependencies. We suggest updating the relevant upstream package if a dependency is used by an upstream dependency. In addition, we recommend regularly running `cargo audit` and `cargo outdated` tools.

Status

The Matrix team has updated the outdated dependencies.

Verification

Resolved.

Suggestion 2: Use Clippy to Automatically Detect forget

Synopsis

In secure Rust development, the `forget` function of `std::mem (core::mem)` must not be used. Currently, there is no check for the `forget` function in the vodozemac codebase. As a result, the usage of the `forget` function could go undetected. In addition, using the `forget` function may result in not releasing critical resources leading to deadlocks or not erasing sensitive data from the memory.

Mitigation

We recommend using the Clippy function `mem_forget` to automatically detect any future use of `forget`.

Status

The Matrix team has added the recommended `mem_forget` check.

Verification

Resolved.

Suggestion 3: Use ReusableSecret Instead of StaticSecret

Location

`src/olm/shared_secret.rs#L113`

Synopsis

In the current implementation, `StaticSecret` is being used for one-time keys, which does not adhere to recommended best practices and could result in key reuse and leakage. The `X25519-dalek` documentation suggests using a special `ReusableSecret` type key created for protocols such as Noise and X3DH.

This audit makes no statements or warranties and is for discussion purposes only.

Mitigation

We recommend using `ReusableSecret` type one-time keys instead of `StaticSecret` type where possible (e.g. in cases where serialization is not necessary).

Status

The Matrix team has implemented the mitigation by using `ReusableSecret` type for one-time keys.

Verification

Resolved.

Suggestion 4: Validate OlmMessage's Inner Vector Length When Extracting Payload

Location

`src/olm/messages/inner.rs#L42-L45`

Synopsis

If the length of the `inner` vector is less than 8 bytes, this will cause the function to panic when creating the slice `&self.inner[..end - 8]`.

Mitigation

We recommend implementing a check to verify that the inner vector length is greater than or equal to 8 bytes when calculating the end of a slice.

Status

The Matrix team has implemented encoding and decoding functions for Olm Message.

Verification

Resolved.

Suggestion 5: Use Strict Version of Ed25519 Signature Check

Location

`src/megolm/inbound_group_session.rs#L189`

`src/megolm/inbound_group_session.rs#L145`

Synopsis

Different implementations of the Ed25519 signature scheme may or may not use malleability checks and malleable signatures depending on the library's intended use cases. Using non-malleable Ed25519 implementation prevents abuse of signatures and decreases the attack surface.

Mitigation

We recommend using Ed25519's verify_strict() function that provides a group malleability check. Furthermore, we recommend that the group malleability check be made optional and configurable in order for the implementation to conform to the original Ed25519 specification for compatibility.

This audit makes no statements or warranties and is for discussion purposes only.

Status

The Matrix team has implemented the RFC8032 compatible `verify` method by default for group malleability check. Additionally, a feature flag has been added to enable users of the library to configure the recommended strict variant.

Verification

Resolved.

Suggestion 6: Validate inner Length in append_mac_bytes

Location

`src/olm/messages/inner.rs#L58`

Synopsis

The length of the `inner` vector that is less than `Mac::TRUNCATED_LEN` will cause the function to panic.

Mitigation

We recommend implementing a check to verify the `inner` length vector when calculating the starting index of a slice.

Status

The Matrix team has implemented a check to verify that there is enough space for MAC bytes and the length is never too short to panic triggers.

Verification

Resolved.

Suggestion 7: Increase Test Coverage

Location

`src/olm/messages/inner.rs#L220`

`src/sas.rs#L339`

`src/megolm/mod.rs#L147`

Synopsis

The existing test coverage for Olm and Megolm is thorough. However, we identified cases that do not implement tests. Missing tests can result in inconsistencies in the future development and functionality testing process, in addition to leading to errors or security vulnerabilities going unnoticed.

Mitigation

We recommend implementing and maintaining comprehensive test coverage. In particular, we suggest adding tests for the following components:

- Olm: decode Olm message;
- SAS: calculate_mac without input (device key/ed25519 identity key) and info; and
- Megolm: export and import ratchet value test should include success cases.

This audit makes no statements or warranties and is for discussion purposes only.

Status

The Matrix team has implemented tests for the components mentioned in the mitigation. Additionally, they started to implement a fuzz testing setup.

Verification

Resolved.

Suggestion 8: Make Number of Chain and Message Keys Configurable

Location

`src/olm/session/mod.rs#L52`

`src/olm/session/receiver_chain.rs#L25`

Synopsis

The maximum number of stored skipped message keys is set to 40. It is within reason that a user can receive a sufficient amount of out-of-order messages to fill up that limit, afterwhich the user would no longer be able to decrypt out-of-order messages because they are no longer able to hold on to additional skipped message keys.

Mitigation

We recommend enabling users of the library to configure the number of chain keys and message keys stored, with the current configuration set as the default.

Status

The Matrix team has responded that they have decided to not make the number of out-of-order messages configurable at the time of verification. Exposing such an option would increase the probability of misuse among client authors and, from their practical experience, it is extremely unlikely to encounter a larger number of skipped messages.

We acknowledge that the mitigation of this suggestion would require some effort and planning to counteract misuse among client authors. With the practical experience the Matrix team supplied, we agree with the assessment of the Matrix team regarding this suggestion.

Verification

Unresolved.

This audit makes no statements or warranties and is for discussion purposes only.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

This audit makes no statements or warranties and is for discussion purposes only.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.

This audit makes no statements or warranties and is for discussion purposes only.