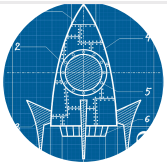
Signal



The Sesame Algorithm: Session Management for Asynchronous Message Encryption

Revision 2, 2017-04-14 [PDF]

Moxie Marlinspike, Trevor Perrin (editor)

Table of Contents

Signal

1. Introduction

This document describes the Sesame algorithm for managing message encryption sessions in an asynchronous and multi-device setting.

Sesame was designed to manage Double Ratchet sessions created with X3DH key agreement [1], [2]. However, Sesame is a generic algorithm that works with any session-based message encryption algorithm that meets certain conditions.

2. Preliminaries

2.1. Overview

Asynchronous key agreement protocols like X3DH enable one party to create a message encryption session and use that session to encrypt an initial message to a recipient, even if the recipient is offline [1]. The recipient can later retrieve the message and use it to calculate a matching session which is used to decrypt the message.

Ratcheting algorithms like the Double Ratchet allow these two parties to update session keys as they communicate, for forward secrecy [2].

Combining these algorithms in a practical context introduces some new concerns:

- Alice and Bob might each have several devices, so encrypting a message from Alice to Bob might require creating sessions from Alice's sending device to all of Bob's

- Alice and Bob might add and remove devices, so they will have to add and delete sessions to handle these changes.
- Alice and Bob might simultaneously initiate a new session with each other, so that two new sessions are created. For the Double Ratchet to be maximally effective Alice and Bob must send and receive messages using matching sessions, so somehow they must agree on which matching sessions to use.
- Alice might choose to erase her device's session state, or restore from a backup, thus causing either her or Bob to possess orphaned sessions which no longer match one of the other party's sessions.

Any solutions to the above must consider that messages might be lost or arrive out-of-order, that clock synchronization is not reliable, and that attackers might compromise devices and/or interfere with communication.

The Sesame algorithm manages the creation, deletion, and use of sessions to support these requirements. A central idea is for each device to keep track of an "active" session for each other device it is communicating with, and use the active session when sending to that device. When a message is received on an "inactive" session, that becomes the new active session. By this process each device converges on using a single session for each remote device it communicates with.

2.2. Assumptions

Sesame is based on the following assumptions:

Server

- There is a server which stores the current record of all users and devices.
- The server temporarily stores the messages that devices send to each other, until the messages are fetched.

Users

- At any point in time there is a set of users.

Signal

- After a user is deleted, its *UserID* can be taken by a new user.

Devices

- At any point in time each user has a nonempty set of devices.
- Users can add or delete devices at any time.
- Each device has a ***DeviceID*** which is unique for the *UserID*.
- Devices can ask the server for information about other users and devices.
- Devices can maintain state, but at any point in time this state might be deleted in whole or part (e.g. by hardware failure or a user action), or rolled back to an earlier state (e.g. by restoring a backup).
- Devices have clocks that can measure elapsed time, but are not synchronized.

Mailboxes

- The server stores a mailbox for each device.
- A mailbox holds a set of messages that were sent to the device.
- Messages are removed from the mailbox when fetched.
- Devices can send messages to other devices' mailboxes. The server stores the sending device's *UserID* and *DeviceID* alongside the message.
- Devices can fetch messages from their own mailbox. The recipient device fetches the message and the sender's *UserID* and *DeviceID* from the server.
- Messages sent to a mailbox are unreliable in normal operation - they may be corrupted, deleted, reordered, delayed, or duplicated before arriving at the mailbox.
- In normal operation, a message that hasn't arrived at a mailbox within some time interval (***MAXLATENCY***) has been lost.

delete, reorder, duplicate, or forge messages before they arrive at the mailbox.

Sessions

- Messages can be encrypted and decrypted using a session, which is some secret data stored by a device.
- Decrypting a message might fail (e.g. if the ciphertext has been tampered with or forged, and thus fails an authentication check).
- An encrypted message can only be decrypted using a **matching session**.
- There is some ***SessionID*** which uniquely identifies each session.
- A session might contain different data after encrypting or decrypting a message (e.g. keys might be deleted after they are used, for forward secrecy).

Session creation for senders

- Each device has an **identity key pair** consisting of a **public key** and **private key**.
- A device can create a new **initiating session** at any time.
- Creating an initiating session requires specifying the identity public key for the intended recipient device which will receive the matching session.
- Creating a new session might fail (e.g. the sending device may have to fetch and cryptographically authenticate parameters associated with the recipient device, such as **prekeys** [1]).

Session creation for recipients

- All messages encrypted by an initiating session are **initiation messages**.
- All initiation messages contain the sending device's identity public key in an unencrypted header.

decrypt the initiation message.

- Creating a matching session may fail (e.g. if cryptographic authentication of the initiation message fails).
- Upon decrypting a message for the first time, an initiating session becomes a regular (non-initiating) session, and thus stops producing initiation messages.

3. Sesame

The Sesame algorithm defines the state that each device stores, and the algorithms that use this state to send and receive encrypted messages.

3.1. Device state

Each device stores a set of ***UserRecords*** for its correspondents, indexed by *UserID*.

Each *UserRecord* contains a set of ***DeviceRecords***, indexed by *DeviceID*.

Each *DeviceRecord* may contain an **active session** and/or an ordered list of **inactive sessions**.

Each device stores a *UserRecord* for its own *UserID*, but does not store a *DeviceRecord* for its own *DeviceID*. This *UserRecord* enables a device to send a copy of each outbound message to the user's other devices.

A *UserRecord* or *DeviceRecord* might be marked **stale**, meaning the record corresponds to a deleted user or device but is being kept around to decrypt delayed messages.

A stale record may be deleted by the sending device at any time (including immediately upon being marked stale). To handle delayed messages, the recommended deletion policy is for a stale record to contain a timestamp recording when it was first marked stale. Once the timestamp is older than *MAXLATENCY*, the stale record can be safely deleted (without fear of message loss) after the next time the device fetches and processes all messages from its mailbox.

always have the same *DeviceID* and identity key pair (to change these for some physical device the device must be logically deleted and then added with new values).

Sesame supports two different models for key pairs: With **per-user identity keys**, all devices under a user share the same key pair. With **per-device identity keys**, each device may have a different key pair.

With per-user identity keys, identity public keys for other devices are stored in *UserRecords*. With per-device identity keys, identity public keys for other devices are stored in *DeviceRecords*.

3.2. Updating device state

Devices can modify their local state in several ways:

Devices can **delete** *UserRecords*, *DeviceRecords*, and sessions. If the last session in a *DeviceRecord* is deleted, then the *DeviceRecord* is deleted. If the last *DeviceRecord* in a *UserRecord* is deleted, then the *UserRecord* is deleted.

Devices can **insert** new sessions into a *DeviceRecord*. An inserted session always becomes the *DeviceRecord's* active session, and the previously active session (if any) is moved to the head of the *DeviceRecord's* inactive sessions list. If the inactive sessions list grows too large, sessions may be deleted from the tail end.

Devices can **activate** an inactive session in a *DeviceRecord*, which moves the inactive session to the *DeviceRecord's* active session, and moves the previously active session (if any) to the head of the *DeviceRecord's* inactive sessions list.

Devices can **mark** *UserRecords* or *DeviceRecords* as stale.

Devices can **conditionally update** their records based on a (*UserID*, *DeviceID*, public key) tuple:

1. If a relevant *UserRecord* does not exist or stores an identity public key that doesn't equal the input public key, then an empty *UserRecord* is added for this *UserID* (replacing the previous *UserRecord* if one exists). With per-user identity

2. If a relevant *DeviceRecord* does not exist or stores an identity public key that doesn't equal the input public key, then an empty *DeviceRecord* is added for this *DeviceID* (replacing the previous *DeviceRecord* if one exists). With per-device identity public keys, the input public key is stored in the empty record. If the *UserID* and *DeviceID* equal the device's own values, then a *DeviceRecord* is not added (a device doesn't add a *DeviceRecord* for itself).

Devices can **prep for encrypting** to a (*UserID*, *DeviceID*, public key) tuple:

1. The device deletes the relevant *UserRecord* and/or *DeviceRecord* if they are stale.
2. The device conditionally updates its records based on the tuple.
3. If the relevant *DeviceRecord* doesn't have an active session, then the device creates a new initiating session using the relevant public key for the *DeviceRecord*. The new session is inserted into the *DeviceRecord*.

3.3. Sending messages

The input to the Sesame sending process is some plaintext and a set of recipient *UserIDs*. The recipient set includes the device's own *UserID*.

The plaintext is encrypted and sent by the sending device using the following process for each recipient *UserID*:

1. If a relevant non-stale *UserRecord* exists for the recipient *UserID*, then for each non-stale *DeviceRecord* in the *UserRecord* that contains an active session, the sending device encrypts the plaintext using that active session.
2. The recipient *UserID* is sent to the server, along with the list of encrypted messages and a corresponding list of *DeviceIDs* indicating the recipient mailbox for each message. These lists will be empty if no relevant active sessions exist.

server accepts the messages and the messages are sent to the relevant mailboxes. This process then terminates for the recipient *UserID*, returning to step 1 for the next recipient *UserID*.

4. Otherwise the server rejects the messages and either informs the sending device if the recipient *UserID* does not exist; or informs the sending device of the old *DeviceIDs* and new *DeviceIDs* needed to make the sending device's records current, and the identity public keys corresponding to any new *DeviceIDs*.
5. If the server indicates that the recipient *UserID* does not exist, then the sending device marks the relevant *UserRecord* (if any) as stale. The sending device then terminates this process for the recipient *UserID*, returning to step 1 for the next recipient *UserID*.
6. For each old *DeviceID*, the sending device marks the relevant *DeviceRecord* as stale.
7. For each new *DeviceID*, the sending device preps for encrypting to the tuple (*UserID*, *DeviceID*, relevant public key).
8. This process is restarted from step 1 for the current recipient *UserID*.

If any error occurs in encrypting to a particular user (e.g. an invalid server response, or a failure during session creation), then the sending device shall discard any changes to the relevant *UserRecord*. This avoids leaving records in an inconsistent state. The sending device may choose to continue encrypting and sending to other users, or may terminate the entire sending process.

To avoid excessive looping in case of a malicious or buggy server, devices should impose some limit on the number of times they're willing to repeat the message sending loop for a recipient user.

3.4. Receiving messages

were fetched from the server.

How the message is fetched is out of scope for this document. Devices might periodically poll the server, or they might receive some notification when new messages are available to be fetched.

An encrypted message is decrypted by a recipient device using the following process:

1. If the encrypted message is an initiation message and the recipient device does not have a relevant *DeviceRecord* containing a session that can decrypt the message, then the following steps are performed:
    a. The relevant public key is extracted from the message header.
    b. The device conditionally updates its records based on the (sender's *UserID*, sender's *DeviceID*, relevant public key) tuple.
    c. The device creates a new session using the initiating message and inserts the new session into the relevant *DeviceRecord*.
2. If no session in the relevant *DeviceRecord* can decrypt the encrypted message, then the encrypted message is discarded, all changes to device state are discarded, and this process terminates.
3. Otherwise, the message is decrypted with the relevant session.
4. If the relevant session is not active it is activated.

If any error occurs in parsing or processing the message, including cryptographic errors in session creation or decrypting the message, then the device shall discard all state changes, discard the encrypted message, and terminate the decryption process.

4. Optional features

Signal

receipts

If the sender or recipient device's state has been rolled back, or the recipient device's state has been deleted, then it is possible for the recipient device to receive a valid message that it can't decrypt.

To handle this without message loss the sending device may store a set of ***MessageRecords***, indexed by some ***MessageID*** which is unique for each encrypted message. If a single message is encrypted to several recipient devices the sender will store a separate *MessageRecord* for each recipient device, each with a unique *MessageID*. Each *MessageRecord* stores the following values:

- The plaintext of the encrypted message.
- The *UserID* for the recipient device.
- The *SessionID* for the session the message was encrypted with.

When the recipient device receives an undecryptable message, the recipient device sends an unencrypted **retry request** message to the original sending device's mailbox, containing the undecryptable message's *MessageID*.

When the original sending device fetches a retry request along with the relevant *UserID* and *DeviceID* of the device that sent the retry request, the original sending device executes the following **resending** process:

1. If the *MessageID* doesn't refer to a current *MessageRecord*, then the retry request is discarded and this process terminates.
2. If the relevant *MessageRecord* doesn't contain a *UserID* that equals the *UserID* used to send the retry request, then the retry request is discarded and this process terminates. (Note that there is no similar check for *DeviceID*; for flexibility, the retry request is allowed from a different *DeviceID* than was originally sent to.)
3. If a non-stale *UserRecord* and non-stale *DeviceRecord* with an active session do not exist for the relevant *UserID* and

   a. The resending device queries the server for the identity public key corresponding to the relevant *UserID* and *DeviceID*.

   b. If the server indicates that the recipient *UserID* or *DeviceID* do not exist, then the relevant records are marked stale, the retry request is discarded, and this process terminates.

   c. The sender then preps for encrypting to the tuple (*UserID*, *DeviceID*, public key).

   d. If the *DeviceRecord's* active session matches the *SessionID* from the relevant *MessageRecord*, then the sending device creates a new initiating session using the relevant public key for the *DeviceRecord*. The new session is inserted into the *DeviceRecord*. This prevents the sending device from repeatedly sending a message using an orphaned session which doesn't match any recipient session.

4. The resending device encrypts the plaintext from the *MessageRecord* using the active session from the relevant *DeviceRecord*.

5. The resending device sends the encrypted message to the server, along with the *UserID* and *DeviceID* indicating the recipient mailbox.

6. If the server indicates that the recipient *UserID* or *DeviceID* do not exist, then the relevant records are marked stale, the retry request is discarded, and this process terminates.

7. Otherwise, the server accepts the message and the message is sent to the relevant mailbox. The resending device deletes the old *MessageRecord* and adds a new *MessageRecord* for the new encrypted message.

MessageRecords might be deleted after some time has elapsed, or if the plaintext they refer to has been deleted from the sending device by the user. Devices might also send **delivery receipts** upon successful message decryption. Delivery receipts refer to some *MessageID* and notify the sender that the *MessageRecord* may be deleted. Delivery receipts may be

To avoid excessive resending, devices should impose some limit on the number of times they're willing to resend a message. If any other error occurs, then the resending device shall discard any state changes and terminate the process.

4.2. Session expiration

It may be desirable for devices to periodically replace old sessions with new sessions, for security purposes. One approach is to give each session a timestamp. The timestamp is set to the current time when an initiating session is created. When initiation messages are fetched, the server tells the recipient device the time difference between when the message arrived at the mailbox and the current time. The recipient device sets the timestamp for any initiated session to the current time minus this difference.

Time constants ***MAXSEND*** and ***MAXRECV*** are defined, where *MAXRECV* must be greater than *MAXSEND + 2(MAXLATENCY)*. At time *MAXSEND* past its timestamp a session must no longer be used for encryption, and shall be moved to the head of the inactive sessions list if active. Attempts to activate such a session have no effect. At time *MAXRECV* past a session's timestamp the session may be deleted, after first fetching and processing all mailbox messages.

5. Implementation considerations

5.1. Server

For simplicity of presentation this document discusses a single server that handles all user and device records, and all messages. In a real system these functions might be distributed across multiple entities.

For example, different servers might handle different groups of users. In this case, Alice would send a message to Bob by

For another example, the server(s) that handle user and device records might be separate from the server(s) that handle mailboxes.

Other divisions of labor might be possible; analyzing all the possibilities is outside the scope of this document.

5.2. X3DH and the Double Ratchet

Sesame was designed for use with Double Ratchet sessions [2] created via X3DH key agreement [1].

In this instantiation, devices will publish **one-time prekeys** and **signed prekeys** to the server, alongside their identity public key.

To create an initiating session, a sending device will contact the server and fetch a **prekey bundle** containing the recipient device's identity public key, signed prekey, and a one-time prekey (if one is available). These values will be used by the X3DH algorithm to create both a secret key that initializes a Double Ratchet session, and an X3DH initial message.

The X3DH initial message is attached to every initiation message, so that the recipient can use it to create a matching Double Ratchet session. Once a response to an initiation message is received, the original sender ceases attaching the X3DH initial message to future messages, so the devices henceforth communicate using only the Double Ratchet.

6. Security considerations

6.1. Authentication

Sesame relies on users to authenticate identity public keys with their correspondents. For example, a pair of users might compare public key fingerprints for all their devices manually, or by scanning a QR code. The details of authentication methods are outside the scope of this document.

with.

A Sesame device might encounter a changed identity public key (or keys) for some remote user when sending, receiving, or resending messages.

This might indicate that a new user is using the *UserID*, that the user re-installed the application, or that the user added new devices (if per-device identity keys are being used). It might also indicate an attempt to impersonate the remote user by a malicious server, or by a malicious user who hijacked the original user's *UserID*.

Whenever a change in identity keys is detected users must repeat the authentication process or they will receive no cryptographic guarantee as to who they are communicating with. A device may wish to pause (or abort) the sending, receiving, or resending processes if a key change occurs, and only continue (or restart) the process if the user affirmatively acknowledges the key change. In this case, the conditional update operation will pause (or generate an error) on detecting a key change.

6.2. Device compromise

Security is catastrophically compromised if an attacker learns a device's secret values, such as the identity private key and session state. Recovery from a device compromise requires the user to replace the compromised device and compromised identity key pair and notify all correspondents of the new public key.

An attacker can leverage a compromise in many ways:

- An attacker with a device's identity private key can impersonate the compromised device to other devices.
- An attacker who can compromise a device may be able to keep persistent backdoor access to the device, or tamper with it to reduce its future security (e.g. weakening the random number generator).
- An attacker who can steal a device's secret keys can probably also steal the plaintext of locally archived

Signal

decryption. For example, the attacker might try to retroactively decrypt old communications, or stealthily decrypt future communications. The attacker might also use some cryptanalytic or forensic attack to reveal a device's state at some time in the past, then try to use this state to decrypt as much old traffic as possible.

Sesame's resistance to passive decryption is inherited from the session creation and message encryption algorithms it is instantiated with. For example, when Sesame uses X3DH and the Double Ratchet Algorithm, passive decryption of pre-compromise and post-compromise messages is tightly bounded by the use of ephemeral keys, prekeys, and ratcheting ([1], [2]).

The only caveat occurs when security is improved by message exchange with matching sessions, as in the Diffie-Hellman ratchet [2]. In that case, and in the rare instance that two parties simultaneously initiate new sessions with each other, then it may take a few exchanged messages before Sesame converges on a single pair of matching sessions.

- Prior to compromising the target device, an attacker might manipulate communications so as to increase the attacker's ability to decrypt older messages once the compromise occurs. For example, the server could make each message received by the target device use a new X3DH initial message without a one-time prekey (by forging retry requests, or by repeatedly deleting and re-adding devices). In this case, messages sent to the target during the lifetime of a signed prekey's private key would be decryptable if the attacker compromises that private key.
- An attacker might try to perform **key-compromise impersonation**, where the attacker impersonates other parties to the compromised party (which is different than impersonating the compromised party to other parties). For example, when Sesame uses X3DH and the Double Ratchet, the server can use a compromised signed prekey with X3DH to create sessions with the target that appear to match arbitrary third parties. This attacker capability would continue until the target deletes their compromised signed

using any sessions the attacker had possession of prior to signed prekey deletion. This attacker capability would continue unless (and until) the target deletes the attacker-controlled sessions through some means such as session expiration.

To mitigate the last two points, Sesame devices using X3DH and the Double Ratchet, or similar algorithms, should delete their signed prekeys on a regular basis, without allowing a malicious server to inhibit deletion. Session expiration (Section 4.2) would help mitigate the final point.

Despite the various mitigations, none of these threats can be eliminated completely. None of the mitigations change the catastrophic nature of a device compromise, or the necessity of completely replacing a compromised device and any compromised keys.

6.3. Protecting server communications

Communication between devices and servers should be encrypted and authenticated. This limits the amount of metadata that is exposed to eavesdroppers, and makes it harder for third-party attackers to perform active or passive attacks on device-to-device communications.

If an attacker is able to impersonate a victim device when authenticating to the server, the attacker could fetch messages being sent to this device. The attacker would not be able to decrypt these messages, but would learn sender *UserIDs* and *DeviceIDs*.

This could also occur if the attacker registers with the server using the same *UserID*, *DeviceID*, and identity public key that had previously belonged to a victim device. To mitigate this, the server could assign random *DeviceIDs*, to prevent reuse, or could require the registering user to prove possession of the identity private key (e.g. by signing a nonce during registration).

6.4. Deleting old data

should also be deleted.

Sesame uses time thresholds to determine when it is safe to delete older sessions (i.e. stale *UserRecords* and *DeviceRecords* as in Section 3.1, and session expiration as in Section 4.2). The idea is for a recipient to wait for a *MAXLATENCY* time period after the last messages might have been sent that require the older sessions, at which point these messages have either arrived in the recipient's mailbox or been lost. After the recipient next fetches and processes all mailbox messages, the recipient is assured there are no more outstanding messages that require the older sessions, so the older sessions may be deleted.

If clock errors result in the recipient deleting the sessions too early, then there is a risk that undecryptable delayed messages may arrive. If clock errors prevent the recipient's clock from advancing, these older sessions might never be deleted.

To mitigate these risks clients should use secure and reliable clocks that cannot be manipulated by an attacker. Clients might also wish to combine time checks with other checks (e.g. counting some number of message round-trips or other events), to provide sanity checks so that brief clock glitches don't delete needed data.

6.5. Bounded loops and bounded storage

The message sending loop in Section 3.3 will repeatedly attempt to adjust the device's records to match the server's records. The resending process in Section 4.1 could be repeatedly triggered to resend a message to a recipient.

To avoid excessive looping in either case, devices are recommended to use some counter with each process (e.g. stored in memory during sending, and stored in a *MessageRecord* for resending). Devices would trigger an error after an excessive number of attempts to send, or resend, a message.

A device might also be triggered to create an excessive number of *DeviceRecords* or sessions for some *UserID*. A device might

also choose to set some limit on the number of sessions it will store for any *DeviceRecord*. Extra sessions will be deleted from the tail of the inactive sessions list if this limit is exceeded.

6.7. Error handling

Care should be taken that any error in sending, receiving, or resending messages terminates the relevant process, and discards any changes that would leave the device in an inconsistent state.

Errors in message sending or resending might result in a message being delivered to some but not all of its intended recipient mailboxes.

Higher-level protocols built on Sesame must be prepared to handle partial delivery of messages to groups of recipients. One option would be to add transaction semantics so that messages are only committed to mailboxes if sending succeeds for all users, and to mandate support for resending so that messages are likely to be delivered.

7. IPR

8. Acknowledgements

Thanks to Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Douglas Stebila, Nik Kinkel, and Tom Ritter for helpful discussions.

Thanks to Michael Kirk, Lilia Kai, and Tom Ritter for editorial feedback.

9. References

[1] M. Marlinspike and T. Perrin, “The X3DH Key Agreement Protocol,” 2016. https://whispersystems.org/docs/specifications/x3dh/

Signal

https://whispersystems.org/docs/specifications/doubleratchet/

Want to get involved with Open Whisper Systems? We're hiring!