The Labyrinth Encrypted Message Storage Protocol

December 6, 2023
Version 1

Meta

TABLE OF CONTENTS

Abstract

End-to-end encrypted messaging creates significant new challenges regarding data storage and accessing message history between devices. Labyrinth is a novel storage system, currently being developed and rolled out in Messenger, which aims to serve this purpose while maintaining a high bar for message content privacy. It is designed to protect messages against non-members (devices and entities which are not enrolled in a user's Labyrinth mailbox) , including preventing new messages from being decryptable on revoked devices which may have previously had access to earlier messages, while achieving low operational overheads and high reliability.

This document describes the Labyrinth protocol as designed, and as it is intended to operate. Details of Meta's implementation may differ in places and will likely evolve over time. This white paper should not be read as making any assurances or commitments to users on Meta's products or services.

Introduction

End-to-end encryption carries a number of significant challenges in the context of consumer messaging. One of them is mailbox storage: plaintexts of end-to-end encrypted messages are not accessible on the server-side (except when explicitly shared by the user), making many common product goals difficult to achieve, such as restoring messaging history on new devices.

Labyrinth - a novel encrypted message storage protocol - aims to address a number of these challenges by enabling users to store their messages server-side, while also maintaining strong privacy. It was designed primarily with Meta's Messenger application and user base in mind, although we hope to contribute to a broader discussion around how cloud storage can be built for privacy and security.

This paper describes the product and privacy goals of Labyrinth, along with its detailed protocol design. It is intended to drive broader discussion and awareness on how privacy and security can be designed into message storage solutions. Currently, Labyrinth is being tested and rolled out in Messenger, meaning that certain features are in progress, and details are subject to change. This paper will touch on Messenger's progress in the rollout and implementation as of the time of publication.

We anticipate that Labyrinth will evolve over time, and that this paper will be updated over time with its implementation; although we cannot guarantee that they will always be fully synchronized.

Product Goals

Labyrinth is intended to function as secure storage for messaging systems structured around time-ordered conversations. Such a messaging system will typically consist of an inbox listing all conversations available, and allow the user to navigate history using a reverse-chronological view of messages.

In this setting, Labyrinth is designed to serve three primary functions:

1. Allow messaging history to be accessed on new devices.
2. Provide an efficiently queryable and writeable storage mechanism, so devices need not store their entire message history.
3. Prevent Meta from accessing data stored in Labyrinth (further elaborated under “Privacy Goals” below).

We aim to ensure that these functions can be achieved as seamlessly as possible. Usage must be easy for people who want to achieve these outcomes to avoid requiring users to have high technical literacy.

These functions should also permit close-to-seamless operation across multiple devices, apps and platforms, in order to enable the product to fill the same role for users as it did prior to end-to-end encryption. Some people will use devices long-term (e.g., mobile devices), some will primarily use ephemeral devices (e.g., web browsers), and others will use a mix. Requiring users to restore a cryptographic key before they access their messages on a new device will add unwanted friction for most users, but the goal is to make it accessible and user-friendly so we can deliver enhanced messaging security to the greatest possible number of users.

Privacy Goals

Labyrinth carries a number of privacy goals, each with its own threat model. These differences broadly reflect a ranking of the importance of each goal, and take into account product goals, usability and implementation feasibility. As Labyrinth has been designed with Messenger in mind, its goals also reflect an application of Meta's Security Principles for Private Messaging[1].

1. Baseline Message Secrecy

The most important privacy goal of Labyrinth is to protect the content of messages as securely as if they were encrypted and authenticated on the client via a strong symmetric key under the user's control. This baseline level of message secrecy aims to protect message content from the server, from anybody who can extract server-side data, and from anybody who can observe data on the wire. It similarly aims to prevent message forgery from any actor without access to a valid encryption key.

Labyrinth aims to achieve this goal while depending only on the security of authenticated symmetric encryption, key derivation, and randomness generation; although other goals below do rely on additional primitives. The varying methods of transferring keys between devices also each rely on different assumptions.

2. Post-Revocation Message Secrecy

Labyrinth is designed for use across multiple devices. This means that we anticipate devices being both added and removed from accounts. When a device is removed, the expectation is typically that it should not have access to new messages received by that account. Labyrinth aims to enforce this cryptographically, by ensuring that the ciphertexts of new messages cannot be decrypted by an entity which is no longer enrolled into a mailbox.

This goal should be achieved even if the server is dishonest at the time of device revocation, from the point that each device connected to the Labyrinth mailbox is aware of the revocation.

[1]https://engineering.fb.com/2022/07/28/security/five-security-principles-for-billions-of-messages-across-metas-apps/

3. Attachment Unlinkability

Attachments in Labyrinth are stored separately from mailboxes. We aim for a strong guarantee that these cannot be connected to their associated mailboxes or messages. This property is only intended to apply with respect to stored data, and not while attachments are being accessed, as we must check eligibility to load them at this point.

4. Labyrinth in Messenger

Messenger is in the process of testing and continued development of Labyrinth, and does not yet commit to all of the above goals. We consider Baseline Message Secrecy to be the most critical goal that should not be violated, and are actively testing the mechanism for Post-Revocation Message Secrecy. We are also implementing the necessary mechanisms for attachment unlinkability, but will require a migration to ensure that the relevant keys are secret before it can be enforced. During the development and testing process, we are ensuring that reliability and debuggability are prioritized, as the successful storage and recovery of messages is the most important goal for most people who use Messenger.

Database Abstraction

Labyrinth presumes client knowledge of the threads they are accessing. This data is stored elsewhere in Meta's infrastructure, and is used to provide a thread list to all clients on an account — including those not enrolled in Labyrinth.

This means that Labyrinth must provide a database that stores all messages. These messages should each have a unique identifier, exist within a specific thread, which itself sits within a given inbox. They also each have a specific timestamp. Data should be possible to look up by inbox, by thread, directly by message, or queried based on ranges.

This abstraction could be represented in SQL syntax as:

```
CREATE TABLE messages (
  inbox_id VARCHAR(INBOX_ID_LENGTH),
  thread_id VARCHAR(THREAD_ID_LENGTH),
  message_id VARCHAR(MESSAGE_ID_LENGTH),
  timestamp BIGINT,
  message_data BLOB,

  KEY inbox_id USING HASH,
  KEY thread_id USING HASH,
  KEY message_id USING HASH,
  INDEX (inbox_id, thread_id, timestamp) USING BTREE,
);
```

Cryptographic Primitives

Labyrinth uses a number of cryptographic primitives, most relatively standard. We consider the overall protocol design agnostic to the details of a primitive, and will refer to algorithms abstractly. This section will note the primitives, their interfaces, as well as the specific algorithms used within Messenger's current implementation.

Primitive	Interface	Implementation used

Authenticated symmetric encryption	`encrypt(key, aad, plaintext)`
	`  -> ciphertext`
	
	`decrypt(key, aad, ciphertext)`
	`  -> plaintext`
		AES-GCM-Extended (detailed below)

Public key encryption	`pk_enc_keygen()`
	`  -> (`
	`    priv_key_enc,`
	`    pub_key_enc)`
	
	`pk_auth_keygen()`
	`  -> (`
	`    priv_key_auth,`
	`    pub_key_auth)`
	
	`pk_encrypt(`
	`  pub_key_enc,`
	`  pub_key_auth,`
	`  priv_key_auth,`
	`  pre_shared_key,`
	`  aad,`
	`  plaintext)`
	`  -> ciphertext`
	
	`pk_decrypt(`
	`  pub_key_enc,`
	`  priv_key_enc,`
	`  pub_key_auth,`
	`  pre_shared_key,`
	`  aad,`
	`  ciphertext)`
	`  -> plaintext`
		Labyrinth HPKE (detailed below)

Primitive	Interface	Implementation used

Signing	`pk_sig_keygen()`
	`  -> (priv_key, pub_key)`
	
	`pk_sign(`
	`  priv_key,`
	`  use_case_byte,`
	`  data)`
	`  -> signature`
	
	`pk_verify(`
	`  pub_key,`
	`  signature`
	`  use_case_byte,`
	`  data)`
	`  -> is_valid`
		XEdDSA[2]

Message authentication	`mac(data, key)` `  -> mac_value`	HMAC-SHA256
Key derivation	`kdf(ikm, salt, info)` `  -> (key1, key2, …)`	HKDF-SHA256
Randomness	`random(bytes)` `  -> random_bytes`	Any available cryptographically strong random number generator.

[2] https://signal.org/docs/specifications/xeddsa/

Primitive	Interface	Implementation used
Oblivious revocable function	orf_client_init() -> client_state orf_server_init() -> server_state orf_client_map(client_state, client_scope, id) -> oblivious_id orf_server_map(server_state, server_scope, oblivious_id) -> mapped_id orf_client_evolve(client_state) -> (new_client_state, evolve_token) orf_server_evolve(server_state, evolve_token) -> new_server_state	See below

1. AES-GCM-Extended

AES-GCM-Extended is the approach that Labyrinth follows to avoid nonce reuse concerns. It takes inspiration from XChaCha20-Poly1305 in its approach, but uses AES-GCM-256 as its underlying cipher, as this is already present and used elsewhere in Messenger's codebase.

AES-GCM-Extended takes a 32-byte key and a 28-byte nonce. From these it generates a subkey and subnonce, as follows:

```
subkey := hchacha20(nonce[0:16], key)
subnonce := nonce[16:28]
```

These are then used directly within AES-GCM-256 to encrypt/decrypt the data. The 28 byte nonce is prepended to the ciphertext.

2. Padding

When Labyrinth encrypts messages using AES-GCM-Extended, padding is also applied to protect ciphertext lengths. Padding involves a tradeoff between privacy and storage overhead: longer padding hides message lengths better, but uses more storage space. We use the PADMÉ scheme[3], which provides a good balance.

For messages with fewer than 10 characters, the plaintext is simply padded to 10 characters. Otherwise, the PADMÉ scheme is applied. This scheme bounds leakage to no more than `O(log log N)` bits for messages of length `N`, and uses at most around 12% overhead.

Each message plaintext is prefixed with a 4 byte unsigned big-endian integer indicating the length of the unpadded plaintext.

[3] https://nikirill.com/files/purbs.pdf

3. Labyrinth HPKE

Labyrinth HPKE is a Hybrid Public Key Encryption scheme, built out of existing primitives included within the Messenger app. It is designed to provide authentication to the sender's authentication public key, the recipient's encryption public key, and to a pre-shared key. The pre-shared key also contributes to the secrecy of the output.

All keypairs used within Labyrinth HPKE are X25519 keypairs, although we segregate them for encryption and authentication use cases. The algorithm takes as inputs the recipient's public encryption key, the sender's authentication keypair, a pre-shared key, additional authenticated data, and the plaintext to send. Encryption is as follows:

```
function labyrinth_hpke_encrypt(
  recipient_enc_pub,
  sender_auth_pub,
  sender_auth_priv,
  psk,
  aad,
  plaintext
) {
  assert_valid_curve_point(recipient_enc_pub)
  assert(length(psk) <= 32 bytes)

  (pub_ephem, priv_ephem) := generate_x25519_keypair()
  id_id := x25519(sender_auth_priv, recipient_enc_pub)
  id_ephem := x25519(priv_ephem, recipient_enc_pub)
  fresh_secret := id_id || id_ephem
  inner_aad :=
    0x01 || sender_auth_pub || recipient_enc_pub || pub_ephem || aad
  subkey := hkdf(fresh_secret, psk, inner_aad)
  nonce := 0x000000000000000000000000
  ciphertext := aes_gcm_256_encrypt(subkey, nonce, aad, plaintext)
  return 0x01 || pub_ephem || ciphertext
}
```

4. Oblivious Revocable Function

The Oblivious Revocable Function (ORF) is a novel construction that we introduce to reduce the linkability between attachments and their associated mailboxes. We have published security proofs[4], but will briefly summarize the structure here.

ORF consists of two pseudo-random functions (PRF) that are intended to be chained - one running over the output of the other. One should run on the client side and one on the server. Each entity has its own secret scalar key, but the structure enables these keys to be re-randomised into a new client-server key pairing, such that the output of chaining the two PRFs over the second key pairing, matches those of the first key pairing, for a given input.

This structure allows for an overall mapping function in which the client is oblivious to the overall output, the server is oblivious to the overall input; and the mapping can remain consistent across multiple clients with different keys.

Within this structure, the server can revoke a client's ability to provide it with inputs that can be mapped to these outputs by deleting the server-side secret.

Our ORF is built around the Ristretto 255 group[5], and constitutes the following methods:

```
function hash_to_curve(input) {
  return ristretto_from_hash(sha512(input))
}

function orf_client_init() {
  return random_scalar()
}

function orf_server_init() {
  return random_scalar()
}
```

[4] https://eprint.iacr.org/2022/1044
[5] https://ristretto.group/

```
function orf_client_map(client_state, client_domain, input) {
  scoped_input := hmac(input, client_domain)
  curve_point := hash_to_curve(scoped_input)
  return ristretto_scalar_multiply(curve_point, client_state)
}

function orf_server_map(server_state, server_domain, input_point) {
  point := ristretto_scalar_multiply(input_point, server_state)
  return hmac(point, server_domain)
}

function orf_client_evolve(client_state) {
  rotation_token := random_scalar()
  new_client_state := client_state * rotation_token
  return (new_client_state, rotation_token)
}

function orf_server_evolve(server_state, rotation_token) {
  new_server_state := server_state / rotation_token
  return new_server_state
}
```

Protocol

1. Components

Overall Labyrinth instance

Labyrinth's backend consists of two components: one containing operational protocol data (the mailbox metastore), and one containing message ciphertexts in a structured database (the mailbox).

This distinction, and most of the protocol complexity, arises from the goal of treating revoked devices as threat actors. This necessitates a protocol which supports key rotation, while supporting devices remaining offline for long periods of time.

Every Labyrinth instance has the following global values associated with it:

- `labyrinthID`: a unique identifier, assigned by the server, associated with the mailbox metastore and known to devices.
- `mailboxRootSalt`: a random value, global to the Labyrinth instance, but non-secret.

Devices

Every entity with access to a Labyrinth mailbox is termed a device. Most commonly these will be physical devices such as smartphones, but they may equally be a "virtual device" - which is a collection of cryptographic keys that are treated by the protocol as a device, but are not associated with a physical device. Virtual devices are used for restoring access to Labyrinth, and will be covered in more detail below.

Each Labyrinth device has the following keys:

- `deviceKeyPriv`, `deviceKeyPub`: asymmetric signing key pair.
- `epochStorageKeyPriv`, `epochStorageKeyPub`: asymmetric encryption key pair.
- `epochStorageAuthKeyPriv`, `epochStorageAuthKeyPub`: asymmetric authentication key pair.

Within the mailbox metastore, the server stores, per-device, the following:

- `deviceKeyPub`.
- `epochStorageKeyPub`, `signature(deviceKeyPriv, epochStorageKeyPub)`.

- `epochStorageAuthKeyPub`, `signature(deviceKeyPriv, epochStorageAuthKeyPub)`.

Epochs

To support key rotation, Labyrinth uses the concept of an “epoch”. This is a period of time during which no device is revoked from Labyrinth (although devices can be added during an epoch). Each epoch is associated with the following values:

- `epochRootKey`: a secret, shared among devices, but unknown to the server.
- `epochID`: a value assigned by the Labyrinth metadata server, uniquely identifying the epoch.
- `epochSequenceID`: a value assigned by clients, used within the mailbox. These are predictable, but are explicitly non-unique between users.
- `epochMetadata`: encrypted secrets of at least one previous epoch.
- `epochDeviceMac[]`: list of proofs of epoch membership for each device.

2. Phases

Initialization

During Labyrinth initialization, a client generates its own secrets, the overall Labyrinth secrets, and the initial epoch. It registers the relevant components with the server, which in turn generates its ORF server state for that client.

a. Client secret generation

```
(deviceKeyPriv, deviceKeyPub)
  := pk_sig_keygen()
(epochStorageKeyPriv, epochStorageKeyPub)
  := pk_enc_keygen()
epochStorageKeySig
  := pk_sign(deviceKeyPriv, 0x30, epochStorageKeyPub)
(epochStorageAuthKeyPriv, epochStorageAuthKeyPub)
  := pk_auth_keygen()
epochStorageAuthKeySig
  := pk_sign(deviceKeyPriv, 0x31, epochStorageAuthKeyPub)()
orfClientState
  := orf_client_init()
```

b. Backup salt generation on the client

```
mailboxRootSalt
  := random(32)
```

c. Server initialization

The server registers a new Labyrinth instance, with a new `labyrinthID`.

The client uploads the following to the server:

- `deviceKeyPub`.
- `epochStorageKeyPub`.
- `epochStorageKeySig`.
- `epochStorageAuthKeyPub`.
- `epochStorageAuthKeySig`.

The server saves these as a representation of the device, associated with `labyrinthID` and generates:

```
orfServerState
```

```
  := orf_server_init()
```

This is also saved to the server's representation of the device.

Note that, while `orfClientState` and `orfServerState` are per-device, their overall combined mapping is now fixed for this Labyrinth instance.

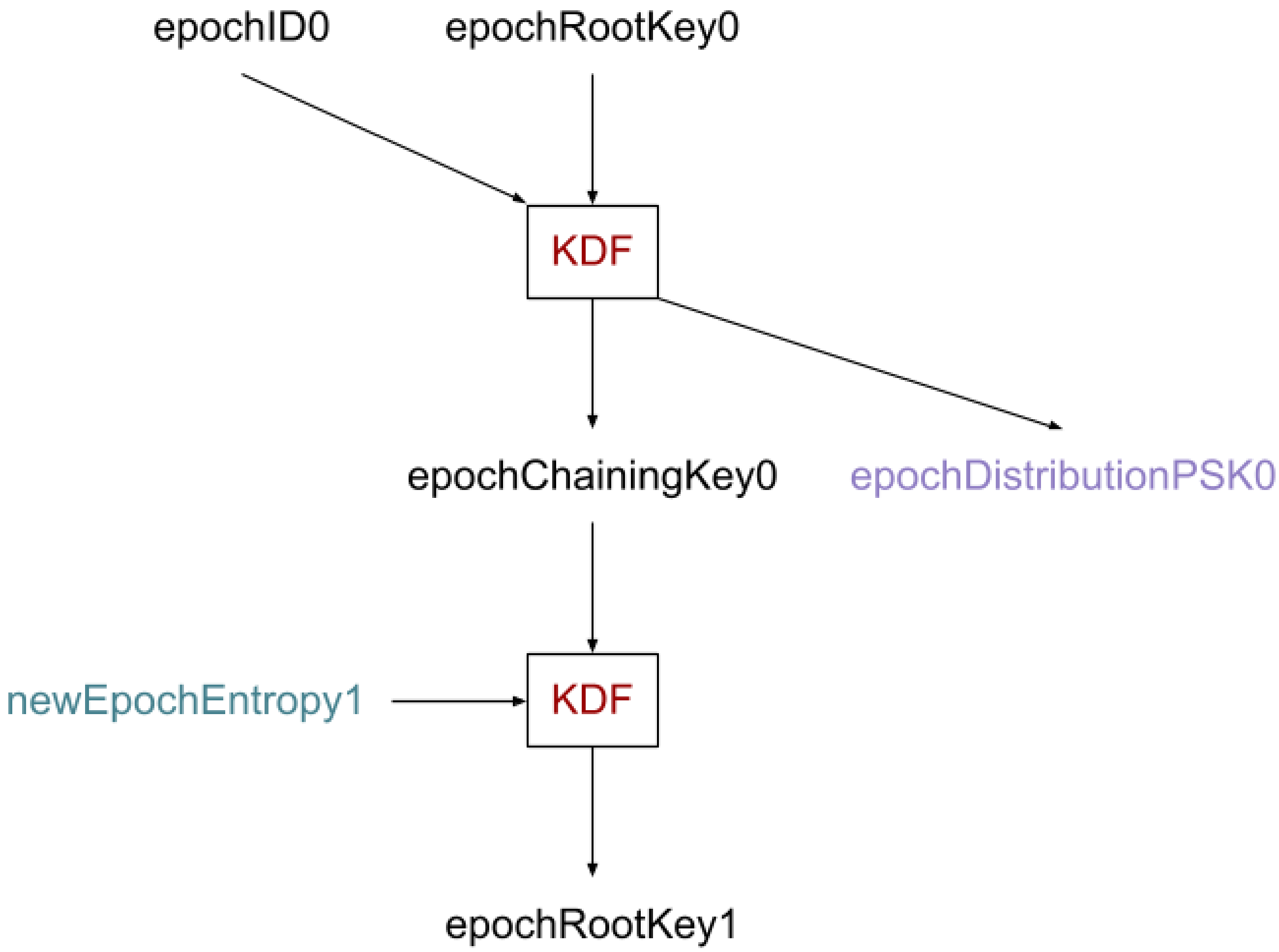
Opening an Epoch

a. Generate core epoch values

Each epoch contains a root key. This is either completely random (generated as 32 bytes) if it is the first epoch, or it is derived. New derived epoch root keys are generated from previous epoch and new entropy as follows:

```
newEpochEntropy
  := random(32)
(epochChainingKey, epochDistributionPreSharedKey)
  := kdf(
    previousEpochRootKey,
    null,
    concat(
      "epoch_chaining_",
      previousEpochSequenceID,
      "_",
      previousEpochID
    )
  )
epochRootKey
  := kdf(
    newEpochEntropy,
    epochChainingKey,
    "epoch_root_key"
  )
```

Epoch root keys can be visualized as chaining via the following diagram:



The `epochSequenceID` is an integer counter, beginning at 0. It is therefore initialized to 0, or calculated as follows:

```
epochSequenceID
  := previousEpochSequenceID + 1
```

b. Store epoch metadata

Each epoch must authenticate its member devices, as well as containing encrypted data for accessing its previous epoch. To do this, we derive two further keys:

```
epochDataStorageKey
  := kdf(
    epochRootKey,
    null,
    concat("epoch_data_storage_", base64_encode(epochSequenceID))
  )
epochDeviceMacKey
  := kdf(
    epochRootKey,
    null,
    concat("epoch_devices_", base64_encode(epochSequenceID))
  )
```

We then encrypt and upload the data for the previous epoch:

```
encryptedPreviousEpochSequenceID
  := encrypt(
    epochDataStorageKey,
    "epoch_data_metadata",
    previousEpochSequenceID
  )
encryptedPreviousEpochRootKey
  := encrypt(
    epochDataStorageKey,
    "epoch_data_metadata",
    previousEpochRootKey
  )
```

Note that clients must validate the format of the previousEpochSequenceID when decrypting this later on. Specifically, they must enforce that it is an integer encoded into a decimal string of at most 11 characters. This serves as implicit domain separation between the values.

For every device in the epoch, Labyrinth first verifies its existing MAC, then authenticates its membership to the new epoch with a new MAC.

```
expectedMac
  := mac(
    previousEpochDeviceMacKey,
    deviceKeyPub
  )
isValidEpochMember
  := expected_mac == previousEpochDeviceMac
epochDeviceMac
  := mac(
    epochDeviceMacKey,
    deviceKeyPub
  )
```

c. Distribute new epoch entropy

Labyrinth doesn’t directly send the new epoch root key to other devices, but rather just shares the new epoch entropy. This helps to enforce an invariant that an epoch can only ever be created containing devices that had been members of the previous epoch (although new devices can be added after this moment).

The new epoch entropy is encrypted to each other member of the epoch, using public key encryption - contingent on the above MAC check having succeeded.

```
isValidEpochStorageKey
  := pk_verify(
    deviceKeyPub_recipient,
    epochStorageKeySig_recipient,
    0x30,
    epochStorageKeyPub_recipient
  )
epochEntropyEncrypted
  := pk_encrypt(
    epochStorageKeyPub_recipient,
    epochStorageAuthKeyPub_self,
    epochStorageAuthKeyPriv_self,
    epochDistributionPreSharedKey,
    concat("epoch__",epochSequenceID),
    newEpochEntropy
  )
```

Receiving devices can verify the sender's `epochStorageAuthKeyPub`, decrypt the `epochEntropy`, then use this to derive the new `epochRootKey` based on the above calculations.

Recovery Codes

New devices are enrolled using 40-character "recovery codes". As Labyrinth involves a number of distinct classes of keys, recovery codes are not used to directly decrypt any data in Labyrinth, but rather to decrypt a bundle of keys that can be used for enrolling a new device. These key bundles are referred to as "Virtual Devices" as they are treated similarly to true devices in most aspects of the protocol.

a. Format

Recovery codes consist of 40 characters, chosen out of the following 32-character alphabet, chosen to avoid homoglyphs:

```
ACDEFHJKLMNPQRSTUVWXYZ0123456789
```

If the characters B, G, I or O are entered, they are internally converted to 8, C, 1 and 0 respectively.

The 40 characters are structured as follows:

- 1 character as a version (currently fixed as 2).
- 1 character as an identifier (currently unused and always 0).
- 34 characters of entropy.
- 4 characters of error correction.

b. Creation

Recovery codes are created by generating 34 random characters from the above alphabet, followed by an error correcting code.

c. Error correction

The last four characters of a recovery code are the error correction code (ECC). Up to three incorrect digits anywhere in the code can be corrected, except for errors in the first two characters (the version and identifier).

To compute the ECC, the 34 entropy characters are interpreted as a column vector `entropy` over the finite field GF(32). The code is then computed such that the following holds, using a 4x38 generator matrix `G`:

```
G * (entropy || code) = 0
```

Where `(entropy || code)` denotes the 38x1 column vector formed by concatenating the entropy and ECC.

The generator matrix `G` is a hardcoded, fixed value for the protocol. It is selected such that any four columns, when concatenated side-by-side, form a 4x4 invertible matrix. Hence, any four missing characters can be recovered by solving a linear system to find the values that satisfy the same above equation.

This code is not a true error correction code, but rather an optimal *erasure* code. The incorrect characters are identified by brute force. We iterate each possible combination of three erasures, compute the resultant recovery code, and check if it derives a valid virtual device ID associated with the mailbox

d. Usage

Using a user's recovery code alongside their Facebook account ID (a 64-bit integer, encoded into an ASCII string), two values can be derived as follows:

```
(vdeviceId, vdeviceDecryptionKey)
  := kdf(
    entropy,
    NULL,
    concat(
      "BackupRecoveryCode_v",
      version,
      "_",
      identifier,
      "_",
      user_id,
      )
  )
```

Of which the `vdeviceId` is 16 bytes long, and `vdeviceDecryptionKey` is 32 bytes long.

The `vdeviceId` is used to identify a specific virtual device on the server; and the `vdeviceDecryptionKey` is then used as a symmetric key for encrypting its data.

Virtual Devices

A virtual device is treated within the protocol as a read-only device. It must be created from a device which is already a member of this Labyrinth instance.

a. Creation

First, Labyrinth generates the device and epoch storage keys, as with a normal device. Note that for virtual devices there is no need for an `epochStorageAuthKey`.

```
(deviceKeyPriv, deviceKeyPub)
  := pk_sig_keygen()
(epochStorageKeyPriv, epochStorageKeyPub)
  := pk_enc_keygen()
epochStorageKeySig
  := pk_sign(deviceKeyPriv, 0x30, epochStorageKeyPub)
```

Next it generates the ORF state for this virtual device by evolving the current device's ORF state.

```
(vdeviceClientState, evolveToken)
  := orf_client_evolve(clientState)
```

The virtual device’s secrets, alongside the global secrets and the latest epoch secrets, can now be encrypted and saved to the server; alongside the various public values.

```
encryptedDeviceKeyPriv
  := encrypt(
    vdeviceDecryptionKey,
    “virtual_device:virtual_device_private_key”,
    deviceKeyPriv
  )
encryptedEpochStorageKeyPriv
  := encrypt(
    vdeviceDecryptionKey,
    “virtual_device:epoch_storage_private_key”,
    epochStorageKeyPriv
  )
encryptedOrfClientState
  := encrypt(
    vdeviceDecryptionKey,
    “virtual_device:ocmf_client_state”,
    vdeviceClientState
  )
encryptedMailboxRootSalt
  := encrypt(
    vdeviceDecryptionKey,
    “virtual_device:mailbox_root_key”,
    mailboxRootSalt
  )
encryptedEpochRootKey
  := encrypt(
    vdeviceDecryptionKey,
    “virtual_device:epoch_root_key”,
    epochRootKey
  )
encryptedEpochSequenceID
  := encrypt(
    vdeviceDecryptionKey,
    “virtual_device:epoch_anon_id”,
    base64_encode(epochSequenceID)
  )
```

The membership proof for the virtual device in the epoch is also generated at this point, to ensure that any new epoch entropy will be shared with the virtual device:

```
epochDeviceMac
  := mac(epochDeviceMacKey, deviceKeyPub)
```

The encrypted values, the two public keys, the `vdeviceId`, the `epochStorageKeySig`, the `evolveToken` and the `epochDeviceMac` are uploaded to the server; which can then create its representation of the virtual device. Uploaded values are stored verbatim, aside from the `evolveToken`, which is used alongside the current physical device's server-side ORF state, to produce a new server state for the virtual device.

```
vdeviceOrfServerState
  := orf_server_evolve(
    evolveToken,
    currentDeviceORFServerState
  )
```

b. Cleanup

We note at this point that the privacy goals around device revocation can be damaged if the device maintains the virtual device secrets or recovery code, unless the virtual device itself is revoked. We accept this risk, as recovery codes are intended to be durable and reliable over the long-term. The alternative would be to disable any recovery code a device has ever accessed at the point that this device is revoked.

A device must delete all virtual device values, and the recovery code, after the creation flow is completed.

Adding a Device

The process of adding a device from a recovery code is very similar to the process of creating a virtual device.

a. Client secret generation

The device initially generates its own secrets, similarly to the initialisation phase.

```
(deviceKeyPriv, deviceKeyPub)
  := pk_sig_keygen()
(epochStorageKeyPriv, epochStorageKeyPub)
  := pk_enc_keygen()
epochStorageKeySig
  := pk_sign(deviceKeyPriv, 0x30, epochStorageKeyPub)
(epochStorageAuthKeyPriv, epochStorageAuthKeyPub)
  := pk_auth_keygen()
epochStorageAuthKeySig
  := pk_sign(deviceKeyPriv, 0x31, epochStorageAuthKeyPub)
```

Having parsed the recovery code, the device can fetch all of the virtual device's values from the server, and decrypt those which are encrypted.

b. Enrolling

The device then generates the values necessary to enroll itself into Labyrinth and the current epoch, similarly to Virtual Device creation:

```
(clientState, evolveToken)
  := orf_client_evolve(vdeviceClientState)
epochDeviceMac
  := mac(epochDeviceMacKey, deviceKeyPub)
```

The public keys, signatures, `evolveToken` and `epochDeviceMac` are registered to the server. The server then calculates the device's new server-side ORF state:

```
orfServerState
  := orf_server_evolve(evolveToken, vdeviceOrfServerState)
```

c. Epoch chaining

At this point, the device knows a single epoch anon ID and epoch root key. It must now perform backward- and forward-chaining.

Backward chaining is the process of rehydrating its cache of epochs created prior to the virtual device's epoch. This can be achieved by simply iterating backwards - as each epoch contains an `encryptedPreviousEpochSequenceID` and `encryptedPreviousEpochRootKey` encrypted under its `epochDataStorageKey`.

Forward chaining is the process of using each epoch's `newEpochEntropy` to calculate the new `epochRootKey`. This requires fetching all of the `epochEntropyEncrypted` values, encrypted under the virtual device's `epochStorageKeyPub`, and using its `epochStorageKeyPriv` to decrypt them. Using these, the new device can derive all new epoch data.

d. Cleanup

As with creation, we require that the recovery code and all virtual device secrets are immediately deleted once the device has been enrolled in the Labyrinth instance.

Device Revocation

Physical and virtual devices can both be revoked from Labyrinth. Revocation aims to provide, under the assumption that the revoked device does not delete any of the keys it knows, cryptographic guarantee that they cannot read any new messages uploaded to Labyrinth.

Device revocation involves two steps. First is a device triggering a new epoch rotation (subject to the above caveat that this has not yet been rolled out). The new epoch will not contain the revoked device, and its new entropy will not be shared with this device.

Second is a server-side step. The server must delete this device's `orfServerState`. It may also delete the device's public keys, and its previous `epochDeviceMacs` although this is not a cryptographic requirement.

Saving Messages

As noted above, the storage abstraction that we wish to provide for messages can be represented by the following SQL:

```
CREATE TABLE messages (
  inbox_id VARCHAR(INBOX_ID_LENGTH),
  thread_id VARCHAR(THREAD_ID_LENGTH),
```

```
  message_id VARCHAR(MESSAGE_ID_LENGTH),
  timestamp BIGINT,
  message_data BLOB,

  KEY inbox_id USING HASH,
  KEY thread_id USING HASH,
  KEY message_id USING HASH,
  INDEX (inbox_id, thread_id, timestamp) USING BTREE,
);
```

Labyrinth aims to protect the message content field within this abstraction from being directly visible to the server using symmetric encryption. This schema can be modeled via the following SQL schema:

```
CREATE TABLE encrypted_messages (
  inbox_id VARCHAR(INBOX_ID_LENGTH),
  thread_id VARCHAR(THREAD_ID_LENGTH),
  message_id VARCHAR(MESSAGE_ID_LENGTH),
  timestamp BIGINT,
  encrypted_message_data BLOB,
  epoch_id BIGINT,
  version BIGINT,

  KEY inbox_id USING HASH,
  KEY thread_id USING HASH,
  KEY message_id USING HASH,
  INDEX (inbox_id, thread_id, timestamp) USING BTREE,
);
```

First, the client derives the key used for encrypting the message from the current `epochRootKey`. Note that these keys are both scoped to the individual thread for domain separation.

```
messageKey
  := kdf(
    epochRootKey,
    NULL,
```

```
    concat(
      “message_key_in_epoch_”,
      epochSequenceID,
      “_cipher_version_3_thread_”,
      threadId
    )
  )
```

It then encrypts the message and uploads it to the server:

```
encryptedMessageData
  := encrypt(
    messageKey,
    concat(“message_thread_”, thread_id),
    messageData
  )
```

Note that this illustrates cipher version 3. Further versions are listed in “implementation notes”.

Loading Messages

Messages within a mailbox can be queried either by mailbox, thread and timestamp range, or by specific message IDs. Decryption mirrors the encryption process.

Attachments

Attachments in Labyrinth, such as media, are encrypted and stored the same as they are during message transmission - in a separate data store from messages. When a message containing an attachment is saved to Labyrinth, the client signals to the server that this must be persisted. Labyrinth aims to store attachments in such a way that they are unconnected either to a user account or to a particular message, except at the point that they are accessed .

However, with attachments being stored separately in this way, the server must ensure that a revoked device which has cached an attachment ID and decryption key cannot still download and access it.

Further, an attachment may be shared between multiple users’ Labyrinth instances (for example if it was sent in a group message), and it is possible that its parent message has been deleted in some mailboxes and not in others.

Labyrinth must therefore ensure that:

1. The device accessing the attachment is still allowed to do so.
2. The message which this attachment is attached to still exists.

The former is achieved via indirection: each attachment is represented by a reference object per mailbox that it lives in. These objects are indexed via ORF - ensuring that only a valid device can identify them in the first instance.

The second is achieved by storing a MAC on this indirection object over the thread and message that this attachment belongs to. This MAC is also keyed with ORF, to ensure that it is opaque while it is not actively being checked.

Between these two values, we can ensure that the user can only index to an attachment via an enrolled device; and that we can prevent access in the case of message erasure, without us ever storing the connection between the user, attachment and mailbox.

It is worth noting that attachments are accessed via CDN URLs, which means that the access control is used to protect access to a CDN URL, rather than to the attachment itself. CDN URLs have TTLs associated with them, so will eventually expire.

Data recovery

In Messenger's deployment of Labyrinth, we offer a number of recovery methods for accessing data on a new device. While these are largely approaches to storing recovery codes, rather than modifications to the core protocol, we include them here for clarity on our usage of the protocol in practice.

1. Recovery Codes

Recovery codes, as described above, can be provided directly to the user. While this places a high burden on those who choose to do this, they may manage them directly. For this purpose, we have included a number of usability features within the design of recovery codes, including a constrained alphabet, and error correction.

It is worth noting that a mailbox can have multiple recovery codes associated with it - each mapping to a different virtual device. This approach is used to enable different recovery mechanisms (detailed below) to be managed independently.

2. User-Chosen PINs

Commonly, users will prefer not to manage their own high entropy secrets, so we also provide an option for users to choose their own authentication codes. Naturally, PINs typically come with a high risk of brute forcing - due to their low key space. To prevent this, we do not use PINs directly to derive cryptographic secrets, but rather use a technology called "Backup Key Vault"[6] to store recovery codes for users - simply using PINs to authenticate and retrieve the recovery codes. Backup Key Vault limits each user to 10 incorrect guesses, ensuring that a reliable brute force attack is not achievable, even for a low entropy key.

3. Third-Party Cloud Storage

To ensure that users are not entirely reliant on self-managed codes, they may also store a recovery code in either iCloud Drive or Google Drive - depending on their mobile platform. These are inaccessible to Meta, but Meta's end-to-end encryption does not protect them from their cloud drive providers. If somebody using this mechanism is able to authenticate to both their Facebook account and their platform's cloud account, they will be able to restore their data, so long as they have not deleted or revoked the recovery code.

Recovery codes stored within these Drives will be located within app-specific hidden folders, so as to simplify the product experience.

4. Device Keychain

When a user sets up Labyrinth on a given device, a recovery code will be generated and saved to their device's local keychain. These should not leave the device, but will persist across uninstalls. This ensures that the significant population of users who uninstall and reinstall the app on the same device are able to maintain access to their messages without having had to set up or track a recovery mechanism.

[6] https://engineering.fb.com/2021/09/10/security/whatsapp-e2ee-backups/

5. One-Time Codes

If a user has physical access to a device that is enrolled in their Labyrinth mailbox, they may enroll a new device using a one-time code, instead of using a recovery code.

To do so, they can request a one-time code using their new device, and consent to generating one on their existing device. The existing device will generate a random 6-digit code locally and display it, for the user to enter on their new device. To limit the feasibility of brute force attacks, we allow up to three attempts at guessing the code before a new one must be generated, and require consent on the existing device each time a new code is generated.

Messenger uses the CPace PAKE to generate a shared secret `otc_secret` between the two-devices, which is then used to communicate the Labyrinth secrets. This enrolment is slightly modified from the enrolment process used with a recovery code.

Before transmitting over the secrets to enroll the new device in the backup, the existing device generates a temporary ORF client state, so as to avoid revealing its own to another device.

```
(temporaryClientState, evolveToken)
  := orf_client_evolve(clientState)
```

It shares the `evolveToken` with the server, which can creating a corresponding temporary ORF server state:

```
temporaryOrfServerState
  := orf_server_evolve(evolveToken, orfServerState)
```

It serializes `temporaryClientState`, `mailboxRootSalt`, `epochRootKey`, `epochID` and `epochSequenceID` into a blob, and encrypts these using `otc_secret`. This ciphertext is then transmitted to the new device, which can use these values to enroll itself into Labyrinth similarly to as if it had received a recovery code.

Known Limitations

During formal verification of Labyrinth, two notable limitations arose which have not yet been addressed in the above protocol design. Labyrinth is intended to evolve and develop over time, and Meta aims to improve on these limitations in due course.

1. Epoch Integrity

When a device restores using a virtual device, the intention is that it receives all epoch entropy for epochs created after that virtual device was. A malicious server may choose to truncate the epochs that it serves in this situation. This would mean that the new device cannot decrypt any messages sent in newer epochs than those it has received; however it may believe that it has received the most up-to-date epoch. If the server is colluding with a revoked device, it could ensure that the new device has received an epoch that was known to this revoked device.

In this situation, the new device could begin writing any new messages that it receives to the epoch that it believes is most current. This would result in the messages being encrypted using keys known to the revoked device.

This is an attack against post-revocation message secrecy. While we have not yet fully explored the space of solutions, we believe that it can be mitigated via a mechanism which informs new devices about the latest epoch that it should have access to.

2. Membership Enumeration

Labyrinth's protocol as it exists today provides an assurance that new epoch entropy will not be transmitted to unauthorized devices. It does not, however, guarantee that every device is aware of each other device in the current epoch. The server is in a position to only share a subset of devices to each other device. This may not strictly manifest as a vulnerability to Labyrinth's stated privacy goals, but nonetheless runs counter to intuitions of the properties which Labyrinth might provide.

Implementation Caveats in Messenger

We note that Labyrinth is a new and complex system. While much of the scheme described above has been implemented in Messenger, we note that - as of the date of publication - the following gaps exist - consistent with the existing privacy goals noted above:

1. Epoch rotation has not yet been implemented on device removal. While the server-side revocation components still apply, this means that we currently only provide the revocation guarantees that rely on an honest server during revocation.
2. Virtual device revocation has not yet been exposed in the product in all cases. Therefore, any recovery code generated may remain viable until this is implemented. However Virtual Devices backing PINs or Third-Party Cloud storage can generally be revoked already.
3. Server-side data deletion is subject to constraints around reliability of the underlying storage system. Meta's backend storage includes backup systems that can be used for a limited time period after data deletion, to protect users from bugs resulting in accidental data loss[7].
4. Client ORF keys are currently accessible to the server. These are only used today for strong attachment unlinkability. This means that we must currently consider all usage of ORF, with respect to the server, as a pseudo-random function that Meta is capable of calculating. Due to our current approach to storing attachments, this does not lead to efficient enumerability, meaning that Meta does not have a mechanism to query the messages associated with a given attachment, nor the attachments associated with a given message; but rather that we could theoretically calculate a boolean point query of whether a specific attachment is associated with a specific message.

Given that Labyrinth is bleeding edge technology, without wide-scale production usage so far, we expect changes to be required over time to ensure it operates correctly for our entire user base. We therefore will be taking care when expanding our cryptographic guarantees beyond baseline message protection. It is critical to note that widespread adoption of end-to-end encryption depends on widespread usability of such tools.

---

[7] DELF: Safeguarding deletion correctness - Engineering at Meta." Engineering at Meta. https://engineering.fb.com/2020/08/12/security/delf/

We note that, while Labyrinth aims to provide novel revocation properties for a storage system, it aims - even with the above caveats - to never regress its protection to be weaker than a naive encrypted storage system protected by a single un-rotatable user secret. As such, any known weaknesses should be constrained to attacks involving revoked devices and recovery methods.

Appendix

Implementation notes

To aid in white hat and cryptographic analysis of our implementation of the above protocol, we present here further information on the details of our implementation.

Encodings

Where we present a string as a parameter to a method, these will typically be encoded as UTF-8.

Integers presented within, or concatenated to, strings are printed in their decimal form without leading zeros. Epoch Sequence IDs, specifically, are sometimes subsequently encoded into Base64 after their decimal serialization. For example, epoch sequence ID 3 may be encoded as “Mw==”. We attempt to indicate above where this is the case.

Secret sizes

Where unspecified, Labyrinth typically uses 32 byte secrets for symmetric cryptographic operations.

Thread IDs

In Messenger, threads may either be “canonical” threads between two users, or group threads.

For canonical threads, the thread ID from each user’s perspective is the user ID of the other user. So user X will see the thread ID for their thread with user Y as Y; whereas user Y would view the same thread with ID X.

Group threads, on the other hand, have their own unique ID. It is possible for a thread between just two users to be treated as a group thread if all other users have been removed.

Message encryption versions

As of the date of publication, there are four possible values for message encryption versions within Messenger's Labyrinth implementation. These only differ in their generation of the messageKey.

Version 0:

```
messageKey
  := kdf(
    epochRootKey,
    threadId,
    concat(
      "message_key_in_epoch_",
      epochSequenceID
    )
  )
```

Version 2:

```
messageKey
  := kdf(
    epochRootKey,
    threadId,
    concat(
      "message_key_in_epoch_",
      epochSequenceID,
      "cipher_version_2"
    )
  )
```

Version 3:

```
messageKey
  := kdf(
    epochRootKey,
    NULL,
    concat(
      "message_key_in_epoch_",
      epochSequenceID,
      "_cipher_version_3_thread_",
      threadId
    )
  )
```

Version 4:

```
messageKey
  := kdf(
    epochRootKey,
    NULL,
    concat(
      “message_key_in_epoch_”,
      epochSequenceID,
      “_cipher_version_4_thread_”,
      threadId
    )
  )
```

Note that versions 3 and 4 are identical aside from their version number. This distinction exists for testing versioning.

Version 1 is no longer supported in clients, and only ever existed in early development.