Encryption and Day One Sync: Technical Details

This document assumes an understanding of the overall Day One encryption design found here. It also assumes a basic understanding of how non-encrypted Day One Sync works.

Encryption Design

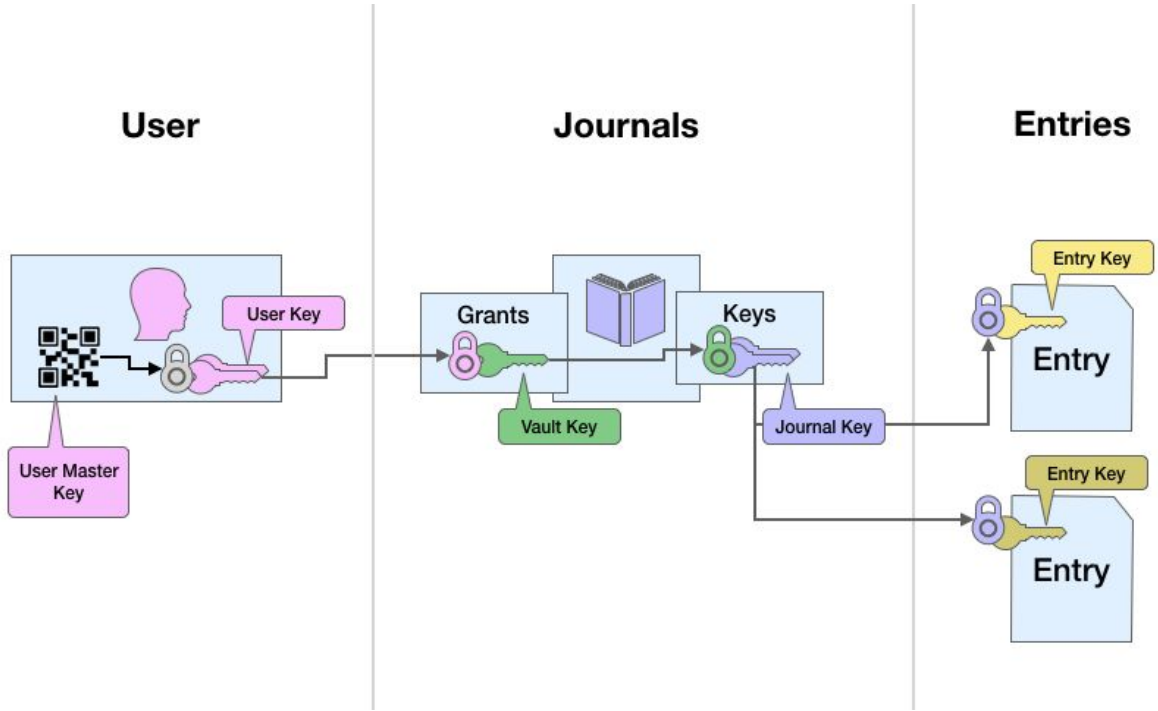
Encryption Keys:

There are five types of encryption keys used in our design:

- Symmetric **content keys**, which encrypt user content (entries, photos, etc.)
- Asymmetric **journal keys**, which protect the content keys
- Symmetric **journal vault keys**, which encrypt the journal keys.
- Asymmetric **user keys**, which protect journal vault keys.
- Symmetric **user master keys**, which protect user keys.

There is a pattern of alternating symmetric and asymmetric keys used in our design. This is because while asymmetric keys have nice security properties, they are also slower, and can only encrypt small amounts of data at a time[1].

Thus, we generate random symmetric keys for encrypting content, and then *protect* the symmetric key with the corresponding asymmetric key. This pattern is widely used in the industry, and means that asymmetric keys will only ever need to encrypt a small amount of data at a time, but can still protect large quantities of data.



[1] The 2048-bit asymmetric keys we are using can encrypt only 256 bytes at a time

Encryption Algorithms:

For **symmetric** encryption, we use **GCM-AES256**. This encryption mode produces both encrypted content and an *authentication tag*. The authentication tag is provided to the encryption engine when starting decryption, and will cause decryption to fail if the wrong key is used.

For asymmetric encryption, we use **2048-bit RSA** keys, configured with **PKCS1 OAEP padding**. (OAEP padding using SHA-1 for MGF1).

Overview of Encryption changes

Entries

- Uploading an entry with the `v2` entry API uses a multipart form encoding. That remains unchanged with encryption. However, in an encrypted journal, the portion representing the entry body is no longer a JSON object; instead, it is an encrypted binary blob in the D1 data format, described below.

- In the multipart encoding, the body of the entry has previously been given the name "`content`", with content type "`application/json`". For encrypted entries, the name is instead "`encrypted-content`", with content type "`vnd/day-one-encrypted`"

- Thumbnails are likewise encrypted in the `v2` entry upload API. However, their multipart form name remains unchanged; whether encrypted or not, the name is "`thumbnail.<momentID>`", and the content type is still "`application/octet-stream`".

- The `v2` entry APIs return entries in a two-part format: the "envelope" and the "entry content", separated by a newline character. When encryption is not enabled, both the envelope and the entry content are JSON objects. When encryption is enabled, the entry content is instead an encrypted binary blob, in the D1 data format.

- Each entry and thumbnail has its own randomly generated[2] 256-bit symmetric key, which is itself protected by the journal key and stored in the D1 blob, as described later in this document. The symmetric key for an entry changes every time a new revision is saved.

[2] Make sure to use a secure random number generator so that keys cannot be guessed.

Images

- Each image is encrypted with a random 256-bit symmetric key, stored in the D1 blob format and protected by the journal, much like thumbnails.

- The md5 of an image given in the envelope's `moments` array is the hash of the *encrypted* image content. The md5 present in the encrypted entry body itself is the hash of the original content.

Journals

- `Journal` objects have an `encryption` key, which contains information about their encrypted state. For non-encrypted journals, the value is the string **"**`plaintext`**"**. For encrypted journals, this is instead JSON object containing a single `vault` object, which is a journal vault, documented below.

- The journal name is encrypted along with the journal itself. It is then base64-encoded and stored inline in place of the plaintext journal name. The key to decrypt the name is the symmetric journal vault key, described below.

- The journal matching API (`POST /api/sync/journals/matches`) supports a `?includeEncrypted=true` query parameter. Clients that support encrypted journals should include that query parameter when matching journals.

Journal Vault

- Each encrypted journal contains a journal vault, which contains two things:
    - Journal **Keys**, which are used to encrypt journal content
    - Journal **Grants**, which give users access to the journal keys

- Each journal **key** is an RSA key pair consisting of a public key and private key. The public key is used to encrypt the entry symmetric keys, and the private key is used to decrypt it. The first key in the the list is the active key, but previous keys are kept in the journal vault to allow decrypting older content created with those keys.

- While public keys are stored in plaintext, the private journal keys are stored in an encrypted format to prevent unauthorized access to journal content. These keys are all encrypted with a symmetric key known as the **journal vault key**. Every time the list of journal keys changes, a new secure-random journal vault key is generated, and all the journal private keys are re-encrypted.

- Our encryption design supports a feature we plan to develop in the future: sharing journals between multiple users. Thus, each journal contains a list of **journal grants**. A journal grant contains a copy of the journal vault key for each user granted access, encrypted with that user's public key. The journal grants are the only canonical copy of

the journal vault key; without a grant to a particular user, there is no way to access journal keys.

Account Keys

- The User object in the API has been augmented with a `keyFingerprint` value. This contains the fingerprint of the user's current key pair, if one exists.

- Two new endpoints have been created, to support managing user keys. In these APIs, the "lockedPrivateKey" is the user's private key, encrypted with the user master key.
    - `PUT /api/users/key`
        - `Request:` **ProposedUserKey**
    - `GET /api/users/key`
        - Response**:** **UserKey**

User Master Key

- The user master key is **never sent to the server**. It is kept only on-device, and is the only key the user ever directly sees.

- The key is an alphanumeric string that looks something like this:
  `D1-123-XXXXXX-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX`

- The D1 portion identifies the format of the key, and is currently always "`D1`"

- The following numeric section ("`123`", above) is the user's account ID. Each user master key is tied to a single user account.

- The remaining 31 characters (shown as `X`'s above) constitute the secret portion of the key. They are randomly drawn from the following pool of characters, selected to be visually unambiguous:
  `ABCDEFGHJKLMNPQRTUVWXYZ2346789`

- To use the master key for encryption, it needs to be converted into a 256-bit symmetric key. This is done using the PBKDF2 algorithm, with the following configuration:
    - Number of rounds: 100,000
    - Hash function: sha256
    - Salt: AccountID
    - Source material: The secret portion of the key (the `X`'s above), ASCII encoded, with all hyphens removed.

- The user master key is short enough that it can be typed, but it is still somewhat cumbersome. To facilitate easier code entry, we also support encoding the key in a QR code. The QR code content is a URL with the following format:

```
dayone2://masterkey?userString=D1-123-XXXXXX-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX&hostname=dayone.me
```

- Note that although it's not present in the user-visible text, we also store the hostname associated with the key. This allows us to use the correct keys when working with staging servers, etc.

API JSON Data Types

- **UserKey**: Object
  - `publicKey`: PEM Encoded key
  - `fingerprint`: Fingerprint
  - `encryptedPrivateKey`: D1Blob(privateKey PEM data, binary format = 0)

- **ProposedUserKey**: Object
  - `publicKey`: PEM Encoded public key
  - `encryptedPrivateKey`: D1Blob(privateKey PEM data, binary format = 0)
  - `nonce`[3]: Base64(16 random bytes)
  - `signature`: Signature(keyPair, nonceBytes)

- **JournalKey**: Object
  - `fingerprint`: Fingerprint(publicKey)
  - `publicKey`: PEM Encoded key data
  - `lockedPrivateKey`: D1Blob(privateKey, binaryFormat: 0)
  - `updated`: SignedUpdate(user_key, publicKey + lockedPrivateKey)
  - `journalSignature`: Signature(journal_key, lockedPrivateKey)

  Note: `journalSignature` is only included on upload; it's not present in responses.

- **JournalGrant**: Object
  - `userId`: Long
  - `fingerprint`: Fingerprint of used public key from userId above.
  - `lockedKey`: RSAEncrypted(symmetric KeyVault key)
  - `updated`: SignedUpdate

- **KeyVault**: Object
  - `vaultKeyFingerprint`: Fingerprint(vault key)
  - `keys`: Array[JournalKey] (The first key is the active one)
  - `grants`: Array[JournalGrant]

- **SignedUpdate**(key, target): Object
  - `userId`: Long

[3] A nonce is a random value with no particular meaning. But by providing a signature of the nonce, we prove we have access to the private key.

    - `fingerprint`: Fingerprint(key)
    - `signature`: Signature(key, target)
    - `at`: DateTime
    -

- **Fingerprint**(asymmetric key): String
    - `sha256(key DER-encoded bytes)` as lowercase hex string

- **Fingerprint**(symmetric key): String
    - `sha256(key)` as lowercase hex string

- **RSAEncrypted**(key, value): String
    - `Base64(RSA(public_key, value_bytes))`

- **Signature**(key, value): String
    - `Base64(RSA_SIGN_SHA256withRSA(key, value))`

- **DateTime**: String
    - Date, encoded as follows:
      `2017-10-13T10:24:19+00:00`

D1 Encrypted Data Format

Day One data encrypted with a symmetric key is always stored in a structured format, known as the **D1 data format.** This allows us to easily distinguish encrypted content from random bytes, and provides some level of internal integrity.

In many cases, a unique symmetric key is generated for each piece of encrypted content, so this format supports storing a locked version of that symmetric key along with the data, encrypted with an asymmetric key. For example, each revision of an entry has its own symmetric key, which is protected by encrypting it with the journal key pair. The D1 data format allows us to store both the encrypted entry content and the associated protected symmetric key together in the same file. If you have a D1 blob and the associated journal key, you are able to extract the symmetric key and decrypt the entry content.

This design allows us to store the symmetric key alongside the encrypted content, so there is no chance of them getting out of sync.

D1 Data Format		
Size (bytes)	Field Name	Description

2	Magic Header	Always ASCII “D1” (0x44, 0x31)
1	Crypto schema version	Currently always 0x1, which signifies GCM_AES256.
1	Binary format version	0-2 are recognized formats: 0 - Encrypted content 1 - Encrypted content + locked key used to decrypt the content 2 - Like format 1, but content is gzip’d before encryption
Begin locked key info, present in binary formats 1 & 2		
32	Fingerprint	Binary encoded fingerprint of the key used protect the encryption key
2	Signature length	(Big-endian Int16) Length of the following ‘Signature’ field. Generally either 0 or 256.
?	Signature	Signature of the Locked Key bytes below. May not be present; if not present, indicates that the creator of this content had access to only the public key and not the private key.
256	Locked Key	Symmetric key used to encrypt ‘Ciphertext’ below, encrypted using the public key referenced by ‘Fingerprint’.
End locked key info, present in binary formats 1 & 2		
12	IV	12-byte random IV used in encryption
?	Ciphertext	The actual encrypted content. Calculated as aes_256_gcm(iv, key, plaintext). Length unspecified; it’s the rest of the file, minus 32 bytes. If binary_format == 0x2, plaintext is first gzipped.
16	GCM tag	16-byte authentication tag produced by aes_256_gcm
16	Checksum	MD5 of all preceding bytes

Common use cases:

The three binary formats are used in the following cases:

- Format 0 - Encrypted content, no locked key embedded
  Used for content encrypted with an existing known symmetric key
    - Journal names (encrypted with journal vault key)
    - Journal private keys (encrypted with journal vault key)
    - User private key (encrypted with user master key

- Format 1 - Encrypted content, locked symmetric key embedded:
  Used for non-plaintext encrypted user content
    - Encrypted thumbnails
    - Encrypted images

- Format 2 - Encrypted gzipped content, locked symmetric key embedded:
  Used for plaintext encrypted user content
    - Encrypted JSON content (specifically, journal entries)