Covata Delta Python SDK Documentation

Release 0.0.1-alpha

Covata

Mar 29, 2017

Contents:

Covata Delta provides an easy to use framework for sharing secrets across networks, and organisations.

CHAPTER 1

Quick Start

Requirements

- Python 2.7 + or 3.3 +
- pip 9.0.1 +

Setting up Virtualenv

```
sudo pip install virtualenv
virtualenv venv
source venv/bin/activate
```

The project can then be installed directly using pip or built from source.

Installation

- Using `pip` directly from Github (Note: This will install the current master branch):

```
pip install git+git://github.com/Covata/delta-sdk-python.git@master
```

Building from Source

Building the project

- Install PyBuilder:

```
pip install pybuilder
```

- Check out the project:

```
git clone https://github.com/Covata/delta-sdk-python.git
cd delta-sdk-python
```

- Build the project:

```
pyb
```

Installing the binary distribution

- Using PyBuilder:

```
pyb install
```

- Using Distutils, where *x.y.z* is the version number:

```
cd target/dist/delta-sdk-python-x.y.z
python setup.py install
```

CHAPTER 2

Example Usage

These examples assume a folder called `~/keystore` is present with `passPhrase` as the password. Each example code-snippet is self-contained and runnable.

Initialisation

- Initialising the client

```
from covata.delta import Client, FileSystemKeyStore

key_store = FileSystemKeyStore("~/keystore/", "passPhrase")
client = Client(key_store)
```

Identity

- Creating an identity

```
from covata.delta import Client, FileSystemKeyStore

key_store = FileSystemKeyStore("~/keystore/", "passPhrase")
client = Client(key_store)

client.create_identity()
```

- Getting your own identity

```
from covata.delta import Client, FileSystemKeyStore

key_store = FileSystemKeyStore("~/keystore/", "passPhrase")
client = Client(key_store)
```

```

identity = client.get_identity("8e91cb8c-1ea5-4b69-bedf-9a14940cce44")
```

- Getting a different identity

```
from covata.delta import Client, FileSystemKeyStore

key_store = FileSystemKeyStore("~/keystore/", "passPhrase")
client = Client(key_store)

identity = client.get_identity("8e91cb8c-1ea5-4b69-bedf-9a14940cce44",
                               "1cb9375f-329c-405a-9b0c-b1659d9c66a4")
```

Secret

- Creating a base secret

```
from covata.delta import Client, FileSystemKeyStore

key_store = FileSystemKeyStore("~/keystore/", "passPhrase")
client = Client(key_store)

# option 1: via identity object
identity = client.get_identity("8e91cb8c-1ea5-4b69-bedf-9a14940cce44")
secret_1 = identity.create_secret("here is my secret")

# option 2: via client object
secret_2 = client.create_secret("8e91cb8c-1ea5-4b69-bedf-9a14940cce44",
                                "here is my secret")
```

- Getting a base secret and the contents

```
from covata.delta import Client, FileSystemKeyStore

key_store = FileSystemKeyStore("~/keystore/", "passPhrase")
client = Client(key_store)

# option 1: via identity object
identity = client.get_identity("8e91cb8c-1ea5-4b69-bedf-9a14940cce44")
secret = identity.get_secret("a9724dd3-8fa1-4ecd-bbda-331748410cf8")

# option 2: via client object
secret = client.get_secret("8e91cb8c-1ea5-4b69-bedf-9a14940cce44",
                           "a9724dd3-8fa1-4ecd-bbda-331748410cf8")

# it's all the same secret
content = secret.get_content()
```

- Deleting a secret

```
from covata.delta import Client, FileSystemKeyStore

key_store = FileSystemKeyStore("~/keystore/", "passPhrase")
client = Client(key_store)

```

```
# option 1: via secret object
identity = client.get_identity("8e91cb8c-1ea5-4b69-bedf-9a14940cce44")
identity.delete_secret("a9724dd3-8fa1-4ecd-bbda-331748410cf8")

# option 2: via client object
secret = client.delete_secret("8e91cb8c-1ea5-4b69-bedf-9a14940cce44",
                              "cb684cfe-11d1-47da-8433-436ca5e6efb0")
```

CHAPTER 3

Client

class covata.delta.**Client**(*key_store*, *api_client_factory=<class covata.delta.apiclient.ApiClient>*)

The main entry point for the Delta SDK.

An instance of this class will provide an interface to work and interact with the Delta API. The core domain objects (Identity, Secret and Event) are returned from method calls to this class, and themselves provide fluent interface that can be used to continue interactive with the Delta API. Consumers of this SDK can therefore choose whether they wish to construct all the calls from base values (i.e. id strings such as identity_id, secret_id, etc) or via the fluent interfaces (or a mixture of both).

Creates a new DeltaClient instance from the provided configuration.

Parameters

- **key_store** (*DeltaKeyStore*) – the key store
- **api_client_factory** ((*DeltaKeyStore*) -> *ApiClient*) – the API client factory

add_secret_metadata(*identity_id*, *secret_id*, *metadata*)

Adds metadata to the given secret. The version number is required for optimistic locking on concurrent updates. An attempt to update metadata with outdated version will be rejected by the server. Passing in an empty metadata map will result in no changes to the metadata or version number.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id
- **metadata** (*dict[str, str]*) – a map of metadata key and value pairs

create_identity(*external_id=None*, *metadata=None*)

Creates a new identity in Delta.

Parameters

- **external_id** (*str | None*) – the external id to associate with the identity
- **metadata** (*dict[str, str] | None*) – the metadata to associate with the identity

Returns the identity

Return type *Identity*

create_secret(*identity_id*, *content*)
Creates a new secret in Delta with the given byte contents.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **content** (*bytes*) – the secret contents

Returns the secret

Return type *Secret*

delete_secret(*identity_id*, *secret_id*)
Deletes the secret with the given secret id.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id

get_events(*identity_id*, *secret_id=None*, *rsa_key_owner_id=None*)
Gets a list of events associated filtered by secret id or RSA key owner or both secret id and RSA key owner.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **secret_id** (*str | None*) – the secret id of interest
- **rsa_key_owner_id** (*str | None*) – the rsa key owner id of interest

Returns a generator of audit events

Return type generator of *Event*

get_identities_by_metadata(*identity_id*, *metadata*, *page=None*, *page_size=None*)
Gets a list of identities matching the given metadata key and value pairs, bound by the pagination parameters.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **metadata** (*dict[str, str]*) – the metadata key and value pairs to filter
- **page** (*int | None*) – the page number
- **page_size** (*int | None*) – the page size

Returns a generator of *Identity* satisfying the request

Return type generator of *Identity*

get_identity(*identity_id*, *identity_to_retrieve=None*)
Gets the identity matching the given identity id.

Parameters **identity_id** (*str*) – the authenticating identity id

Returns the identity

Return type *Identity*

get_secret(*identity_id*, *secret_id*)
Gets the given secret by id.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the id of the secret to retrieve

Returns the secret

Return type *Secret*

get_secret_content(*identity_id*, *secret_id*, *symmetric_key*, *initialisation_vector*)
Gets the plaintext content, given the symmetric key and initialisation vector used for encryption.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id
- **symmetric_key** (*str*) – the symmetric key used for encryption encoded in base64
- **initialisation_vector** (*str*) – the initialisation vector encoded in base64

Returns the plaintext content of the secret

Return type bytes

get_secret_content_encrypted(*identity_id*, *secret_id*)
Gets the base64 encoded encrypted content given the secret id.

Note that the returned encrypted content when decoded from base64 has a trailing 16 byte GCM authentication tag appended (i.e. the cipher text is the byte range [:-16] and the authentication tag is the remaining [-16:] bytes).

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id

Returns the encrypted content encoded in base64

Return type str

get_secret_metadata(*identity_id*, *secret_id*)
Gets the metadata key and value pairs for the given secret.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id to be retrieved

Returns the retrieved secret metadata dictionary and version tuple

Return type (dict[str, str], int)

get_secrets(*identity_id*, *base_secret_id=None*, *created_by=None*, *rsa_key_owner_id=None*, *metadata=None*, *lookup_type=<SecretLookupType.any: 3>*, *page=None*, *page_size=None*)
Gets a list of secrets based on the query parameters, bound by the pagination parameters.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **base_secret_id** (*str | None*) – the id of the base secret
- **created_by** (*str | None*) – the id of the secret creator
- **rsa_key_owner_id** (*str | None*) – the id of the RSA key owner

- **metadata** (*dict[str, str] | None*) – the metadata associated with the secret
- **lookup_type** (*SecretLookupType*) – the type of the lookup query
- **page** (*int | None*) – the page number
- **page_size** (*int | None*) – the page size

Returns a generator of secrets satisfying the search criteria

Return type generator of *Secret*

share_secret(*identity_id*, *recipient_id*, *secret_id*)

Shares the base secret with the specified recipient. The contents will be encrypted with the public encryption key of the RSA key owner, and a new secret key and initialisation vector will be generated. This call will result in a new derived secret being created and returned.

Parameters

- **identity_id** (*str*) – the authenticating identity id
- **recipient_id** (*str*) – the target identity id to share the base secret
- **secret_id** (*str*) – the base secret id

Returns the derived secret

Return type *Secret*

CHAPTER 4

Identity

An Identity is an entity (such as user, device, or another service) registered with Delta and is comprised of a number of attributes, of which two rely on cryptographic primitives. These are the long-lived key pairs:

- Encryption key pair - An asymmetric key pair, associated with an identity for the purposes of encrypting and decrypting secret encryption keys:
  - Public encryption key - The public key that functions as a key encryption key, to encrypt a secret encryption key. The public encryption key is stored in Delta as part of the identity creation process.
  - Private decryption key - The private key used to decrypt a secret

    encryption key. The private decryption key must be managed outside of Delta.
- Signing key pair - An asymmetric key pair, associated with an identity for the purpose of request signing and authentication:
  - Public signing verification key - The public key used to verify request authenticity and ownership. The public signing verification key is stored in Delta as part of the identity creation process and is not publicly visible (unlike the public encryption key).
  - Private signing key - The private key used to sign requests as required by Delta so that the requests can be verified. The private signing key must be managed outside of Delta.

class covata.delta.**Identity**(*parent*, *identity_id*, *public_encryption_key*, *external_id*, *metadata*)

An instance of this class encapsulates an identity in Covata Delta. An identity can be a user, application, device or any other identifiable entity that can create secrets and/or be target recipient of a secret.

An has two sets of asymmetric keys, for encryption and for signing of requests. Identities may also have optional, public, searchable metadata and a reference to an identifier in an external system.

Creates a new identity in Delta with the provided metadata and external id.

Parameters

- **parent** (*Client*) – the Delta client that constructed this instance
- **identity_id** – the id of the identity

- **public_encryption_key** (*str*) – the public signing key of the identity
- **external_id** (*str | None*) – the external id of the identity
- **metadata** (*dict[str, str] | None*) – the metadata belonging to the identity

create_secret (*content*)
Creates a new secret in Delta with the given contents.

Parameters **content** (*bytes*) – the secret content

Returns the secret

Return type *Secret*

delete_secret (*secret_id*)
Deletes the secret with the given secret id.

Parameters **secret_id** (*str*) – the secret id

get_events (*secret_id=None*, *rsa_key_owner_id=None*)
Gets a list of events associated filtered by secret id or RSA key owner or both secret id and RSA key owner.

Parameters

- **secret_id** (*str | None*) – the secret id of interest
- **rsa_key_owner_id** (*str | None*) – the rsa key owner id of interest

Returns a generator of audit events

Return type generator of *Event*

get_identities_by_metadata (*metadata*, *page=None*, *page_size=None*)
Gets a list of identities matching the given metadata key and value pairs, bound by the pagination parameters.

Parameters

- **metadata** (*dict[str, str]*) – the metadata key and value pairs to filter
- **page** (*int | None*) – the page number
- **page_size** (*int | None*) – the page size

Returns a generator of *Identity* satisfying the request

Return type generator of [*Identity*]

get_identity (*identity_to_retrieve=None*)
Gets the identity matching the given identity id.

Returns the identity

Return type *Identity*

get_secrets (*base_secret_id=None*, *created_by=None*, *rsa_key_owner_id=None*, *metadata=None*, *lookup_type=<SecretLookupType.any: 3>*, *page=None*, *page_size=None*)
Gets a list of secrets based on the query parameters, bound by the pagination parameters.

Parameters

- **base_secret_id** (*str | None*) – the id of the base secret
- **created_by** (*str | None*) – the id of the secret creator
- **rsa_key_owner_id** (*str | None*) – the id of the RSA key owner
- **metadata** (*dict[str, str] | None*) – the metadata associated with the secret

- **lookup_type** (*SecretLookupType*) – the type of the lookup query
- **page** (*int | None*) – the page number
- **page_size** (*int | None*) – the page size

Returns a generator of secrets satisfying the search criteria

Return type generator of *Secret*

retrieve_secret(*secret_id*)

Retrieves a secret with this identity.

Parameters **secret_id** (*str*) – the secret id

Returns the secret

Return type *Secret*

CHAPTER 5

Secret

class covata.delta.**Secret**(*parent*, *secret_id*, *created*, *rsa_key_owner*, *created_by*, *encryption_details*, *base_secret_id=None*)

An instance of this class encapsulates a secret in Covata Delta. A secret has contents, which is encrypted by a symmetric key algorithm as defined in the immutable EncryptionDetails class, holding information such as the symmetric (secret) key, initialisation vector and algorithm. The symmetric key is encrypted with the public encryption key of the RSA key owner. This class will return the decrypted contents and symmetric key if returned as a result of Client.

Creates a new secret with the given parameters.

Parameters

- **parent** (*Client*) – the Delta client that constructed this instance
- **secret_id** (*str*) – the id of the secret
- **created** (*str*) – the created date
- **rsa_key_owner** (*str*) – the identity id of the RSA key owner
- **created_by** (*str*) – the identity id of the secret creator
- **encryption_details** (*EncryptionDetails*) – the encryption details of the secret

add_metadata(*metadata*)

Adds the key and value pairs in the provided map as metadata for this secret. If the metadata previously contained a mapping for the key, the old value is replaced by the specified value.

Parameters **metadata** (*dict[str, str]*) – a map of metadata key and value pairs

get_content()

Gets the content of a secret, encrypted with the details defined in the encryption_details of this secret and encoded in base64.

Returns the content of the secret encoded in base64

Return type str

get_derived_secrets(*page=None*, *page_size=None*)
Gets a list of secrets derived from this secret, bound by the pagination parameters.

The credentials of the secret creator be present in the local key store.

Parameters

- **page** (*int | None*) – the page number
- **page_size** (*int | None*) – the page size

Returns a generator of secrets

Return type generator of *Secret*

get_events(*rsa_key_owner_id=None*)
Gets a list of events associated filtered by this secret id or both this secret id and RSA key owner.

The credentials of the secret creator must be present in the local key store.

Parameters **rsa_key_owner_id** (*str | None*) – the rsa key owner id of interest

Returns a generator of audit events

Return type generator of *Event*

get_metadata()
Gets the metadata for this secret. Metadata are key-value pairs of strings that can be added to a secret to facilitate description and lookup. Secrets can support any number of metadata elements, but each key or value has a limit of 256 characters.

Returns the metadata for this secret

Return type dict[str, str]

share_with(*identity_id*)
Shares this secret with the target recipient identity. This action will create a new (derived) secret in Covata Delta, and the new secret will be returned to the caller.

The credentials of the RSA key owner must be present in the local key store.

Parameters **identity_id** (*str*) – the recipient identity id

Returns the derived secret

Return type *Secret*

CHAPTER 6

Encryption Details

class covata.delta.**EncryptionDetails**(*symmetric_key*, *initialisation_vector*)

This class holds the necessary key materials required to decrypt a particular secret. The symmetric key itself is protected by a public encryption key belonging to an identity.

Creates a new encryption details with the given parameters.

Parameters

- **symmetric_key** (*str*) – the symmetric key
- **initialisation_vector** (*str*) – the initialisation vector

CHAPTER 7

Event

class covata.delta.**Event**(*event_details*, *host*, *event_id*, *source_ip*, *timestamp*, *event_type*)

An instance of this class encapsulates an event in Covata Delta. An event is an audit entry representing an action undertaken by an identity on a secret.

Creates a new *Event* with the given parameters.

Parameters

- **event_details** (*EventDetails*) – details of the audit event.
- **host** (*str*) – the host address
- **event_id** (*str*) – the identifier of the event object
- **source_ip** (*str*) – the source IP address
- **timestamp** (*datetime*) – the timestamp of the event
- **event_type** (*str*) – the type of the event

class covata.delta.**EventDetails**(*base_secret_id*, *requestor_id*, *rsa_key_owner_id*, *secret_id*, *secret_owner_id*)

This class describes the details of an event related to a secret. Information includes the secret id, the owner identity id of the secret, and the identity id triggering the event.

Additional information such as base secret id and RSA key owner id are also available for derived secrets.

Creates an instance of event details.

Parameters

- **base_secret_id** (*str*) – the id of the base secret
- **requestor_id** (*str*) – the id of the requesting identity
- **rsa_key_owner_id** (*str*) – the id of the RSA key owner
- **secret_id** (*str*) – the id of the secret
- **secret_owner_id** (*str*) – the id of the secret owner

CHAPTER 8

API Client

class covata.delta.**ApiClient**(*key_store*)

The Delta API Client is an abstraction over the Delta API for execution of requests and responses.

Constructs a new Delta API client with the given configuration.

Parameters **key_store** (*DeltaKeyStore*) – the DeltaKeyStore object

create_secret(*requestor_id*, *content*, *encryption_details*)

Creates a new secret in Delta. The key used for encryption should be encrypted with the key of the authenticating identity.

It is the responsibility of the caller to ensure that the contents and key material in the encryption details are properly represented in a suitable string encoding (such as base64).

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **content** (*str*) – the contents of the secret
- **encryption_details** (*dict[str, str]*) – the encryption details

Returns the created base secret

Return type dict[str, str]

delete_secret(*requestor_id*, *secret_id*)

Deletes the secret with the given secret id.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id to be deleted

get_events(*requestor_id*, *secret_id=None*, *rsa_key_owner_id=None*)

Gets a list of events associated filtered by secret id or RSA key owner or both secret id and RSA key owner.

Parameters

- **requestor_id** (*str*) – the authenticating identity id

- **secret_id** (*str | None*) – the secret id of interest
- **rsa_key_owner_id** (*str | None*) – the rsa key owner id of interest

Returns a list of audit events

Return type list[dict[str, *any*]]

get_identities_by_metadata(*requestor_id*, *metadata*, *page=None*, *page_size=None*)

Gets a list of identities matching the given metadata key and value pairs, bound by the pagination parameters.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **metadata** (*dict[str, str]*) – the metadata key and value pairs to filter
- **page** (*int | None*) – the page number
- **page_size** (*int | None*) – the page size

Returns a list of identities satisfying the request

Return type list[dict[str, *any*]]

get_identity(*requestor_id*, *identity_id*)

Gets the identity matching the given identity id.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **identity_id** (*str*) – the identity id to retrieve

Returns the retrieved identity

Return type dict[str, *any*]

get_secret(*requestor_id*, *secret_id*)

Gets the given secret. This does not include the metadata and contents, they need to be made as separate requests, `get_secret_metadata()` and `get_secret_content()` respectively.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id to be retrieved

Returns the retrieved secret

Return type dict[str, *any*]

get_secret_content(*requestor_id*, *secret_id*)

Gets the contents of the given secret.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id to be retrieved

Returns the retrieved secret

Return type str

get_secret_metadata(*requestor_id*, *secret_id*)

Gets the metadata key and value pairs for the given secret.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id to be retrieved

Returns the retrieved secret metadata dictionary and version tuple

Return type (dict[str, str], int)

get_secrets(*requestor_id*, *base_secret_id=None*, *created_by=None*, *rsa_key_owner_id=None*, *metadata=None*, *lookup_type=<SecretLookupType.any: 3>*, *page=None*, *page_size=None*)

Gets a list of secrets based on the query parameters, bound by the pagination parameters.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **base_secret_id** (*str | None*) – the id of the base secret
- **created_by** (*str | None*) – the id of the secret creator
- **rsa_key_owner_id** (*str | None*) – the id of the RSA key owner
- **metadata** (*dict[str, str] | None*) – the metadata associated with the secret
- **lookup_type** (*SecretLookupType*) – the type of the lookup query
- **page** (*int | None*) – the page number
- **page_size** (*int | None*) – the page size

Returns a list of secrets satisfying the search criteria

Return type list[dict[str, *any*]]

register_identity(*public_encryption_key*, *public_signing_key*, *external_id=None*, *metadata=None*)

Creates a new identity in Delta with the provided metadata and external id.

Parameters

- **public_encryption_key** (*str*) – the public encryption key to associate with the identity
- **public_signing_key** (*str*) – the public signing key to associate with the identity
- **external_id** (*str | None*) – the external id to associate with the identity
- **metadata** (*dict[str, str] | None*) – the metadata to associate with the identity

Returns the id of the newly created identity

Return type str

share_secret(*requestor_id*, *content*, *encryption_details*, *base_secret_id*, *rsa_key_owner_id*)

Shares the base secret with the specified target RSA key owner. The contents must be encrypted with the public encryption key of the RSA key owner, and the encrypted key and initialisation vector must be provided. This call will result in a new derived secret being created and returned as a response.

It is the responsibility of the caller to ensure that the contents and key material in the encryption details are properly represented in a suitable string encoding (such as base64).

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **content** (*str*) – the contents of the secret

- **encryption_details** (*dict[str, str]*) – the encryption details
- **base_secret_id** (*str*) – the id of the base secret
- **rsa_key_owner_id** (*str*) – the id of the rsa key owner

Returns the created derived secret

Return type dict[str, str]

signer(*identity_id*)

Generates a request signer function for the the authorizing identity.

```
>>> signer = api_client.signer(authorizing_identity)
```

Parameters **identity_id** (*str*) – the authorizing identity id

Returns the request signer function

Return type (PreparedRequest) -> PreparedRequest

update_identity_metadata(*requestor_id*, *identity_id*, *metadata*, *version*)

Updates the metadata of the given identity given the version number. The version of an identity's metadata can be obtained by calling `get_identity()`.

An identity has an initial metadata version of 1.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **identity_id** (*str*) – the identity id to be updated
- **metadata** (*dict[str, str]*) – metadata dictionary
- **version** (*int*) – metadata version, required for optimistic locking

update_secret_metadata(*requestor_id*, *secret_id*, *metadata*, *version*)

Updates the metadata of the given secret given the version number. The version of a secret's metadata can be obtained by calling `get_secret()`.

A newly created base secret has a metadata version of 1.

Parameters

- **requestor_id** (*str*) – the authenticating identity id
- **secret_id** (*str*) – the secret id to be updated
- **metadata** (*dict[str, str]*) – metadata dictionary
- **version** (*int*) – metadata version, required for optimistic locking

class covata.delta.**SecretLookupType**

Enumerates the applicable secret lookup types.

any = 3

Perform lookup on both base and derived secrets.

base = 1

Restricts lookup to base secrets.

derived = 2

Restricts lookup to derived secrets.

CHAPTER 9

Cryptography

The Delta Crypto package provides functionality for client-side cryptography.

covata.delta.crypto.**generate_private_key**()
Generates a RSAPrivateKey object. The public key object can be extracted by calling public_key() method on the generated key object.

Returns the generated private key object

Return type RSAPrivateKey

covata.delta.crypto.**serialize_public_key**(*public_key*)
Serializes the provided public key object as base-64-encoded DER format using X.509 SubjectPublicKeyInfo with PKCS1.

Parameters **public_key** (RSAPublicKey) – the public key object

Returns the key as base64 encoded unicode string

Return type str

covata.delta.crypto.**calculate_sha256hex**(*payload*)
Calculates the SHA256 hex digest of the given payload.

Parameters **payload** (*str*) – the payload to be calculated

Returns SHA256 hex digest

Return type bytes

covata.delta.crypto.**generate_secret_key**()
Generates a 256 bits secret key.

Uses /dev/urandom on UNIX platforms, and CryptGenRandom on Windows.

Returns the 256 bits secret key

Return type bytes

covata.delta.crypto.**generate_initialisation_vector**()
Generates a 128 bits initialisation vector.

Uses `/dev/urandom` on UNIX platforms, and `CryptGenRandom` on Windows.

Returns the 128 bits initialisation vector

Return type bytes

`covata.delta.crypto.`**`encrypt`**(*data*, *secret_key*, *initialisation_vector*)
Encrypts data using the given secret key and initialisation vector.

Parameters

- **`data`** (*bytes*) – the plaintext bytes to be encrypted
- **`secret_key`** (*bytes*) – the key to be used for encryption
- **`initialisation_vector`** (*bytes*) – the initialisation vector

Returns the cipher text and GCM authentication tag tuple

Return type (bytes, bytes)

`covata.delta.crypto.`**`decrypt`**(*ciphertext*, *tag*, *secret_key*, *initialisation_vector*)
Decrypts a cipher text using the given GCM authentication tag, secret key and initialisation vector.

Parameters

- **`ciphertext`** (*bytes*) – the cipher text to be decrypted
- **`tag`** (*bytes*) – the GCM authentication tag
- **`secret_key`** (*bytes*) – the key to be used for encryption
- **`initialisation_vector`** (*bytes*) – the initialisation vector

Returns the decrypted plaintext

Return type bytes

`covata.delta.crypto.`**`encrypt_key_with_public_key`**(*secret_key*, *public_encryption_key*)
Encrypts the given secret key with the public key.

Parameters

- **`secret_key`** (*bytes*) – the key to encrypt
- **`public_encryption_key`** (`RSAPublicKey`) – the public encryption key

Returns the encrypted key

Return type bytes

`covata.delta.crypto.`**`decrypt_with_private_key`**(*secret_key*, *private_encryption_key*)
Decrypts the given secret key with the private key.

Parameters

- **`secret_key`** (*bytes*) – the secret key to decrypt
- **`private_encryption_key`** (`RSAPrivateKey`) – the private encryption key

Returns the decrypted key

Return type bytes

`covata.delta.crypto.`**`deserialize_public_key`**(*b64_encoded_public_key*)
loads a `RSAPublicKey` object from a serialized public key.

Parameters **`b64_encoded_public_key`** (*str*) – the key as base64 encoded string

Returns the public key object

Return type `RSAPublicKey`

CHAPTER 10

Key Store

The management and storage of private keys is the responsibility of the client. The `DeltaKeyStore` provides the interface for a key-storage implementation. The `FileSystemKeyStore` is an implementation to store keys in PEM formats on the file system.

Retrieval and usage of these keys is required in the following use cases:

- Request Signing - all endpoints requiring authentication will require the private signing key of the requesting identity as part of the CVT1 request signing process.
- Retrieving Secret Content - to retrieve secret content, a client will need access to the secret encryption key, which can only be decrypted with their private decryption key.

The Delta framework does not dictate or impose restrictions on how a client should manage and store private keys. It is therefore up to the implementation on whether to develop a custom solution or use pre-existing solutions, as long as the keys are accessible in the above use cases.

class `covata.delta.keystore.`**DeltaKeyStore**

get_private_encryption_key(*identity_id*)
Loads a private encryption key instance for the given identity id.

Parameters **identity_id** (*str*) – the identity id of the key owner

Returns the cryptographic private key object

get_private_signing_key(*identity_id*)
Loads a private signing key instance for the given identity id.

Parameters **identity_id** (*str*) – the identity id of the key owner

Returns the signing private key object

store_keys(*identity_id*, *private_signing_key*, *private_encryption_key*)
Stores the signing and encryption key pairs under a given identity id.

Parameters

- **identity_id** (*str*) – the identity id of the key owner

- **private_signing_key** (RSAPrivateKey) – the private signing key object
- **private_encryption_key** (RSAPrivateKey) – the private cryptographic key object

File-System Key Store

Implementation of the DeltaKeyStore abstract base class using the file system. Private keys are saved in the file system as encrypted PEM formats and are only decrypted in memory on read.

class covata.delta.keystore.**FileSystemKeyStore**(*key_store_path*, *key_store_passphrase*)

Bases: *covata.delta.keystore.DeltaKeyStore*

Constructs a new Filesystem-backed *DeltaKeyStore* with the given configuration.

Parameters

- **key_store_path** (*str*) – the path to the private key store
- **key_store_passphrase** (*str*) – the passphrase to decrypt the keys

CHAPTER 11

Signer

The Delta Signer package implements the CVT1 request signing scheme.

covata.delta.signer.**get_updated_headers**(*identity_id*, *method*, *url*, *headers*, *payload*, *private_signing_key*)

Gets an updated header dictionary with an authorization header signed using the CVT1 request signing scheme.

Parameters

- **identity_id** (*str*) – the authorizing identity id
- **method** (*str*) – the HTTP request method
- **url** (*str*) – the delta url
- **headers** (*dict[str, str]*) – the request headers
- **payload** (*bytes*) – the request payload
- **private_signing_key** – the private signing key object

Returns the original headers with additional Cvt-Date, Host, and Authorization headers.

Return type dict[str, str]

CHAPTER 12

CVT1 Request Signing

All requests to the Delta service (with the exception of the Create Identity request) must be signed using the CVT1 request signing scheme, which is similar to other request signing schemes such as those implemented by Amazon AWS.

At a high level, signing a request using the CVT1 request signing scheme involves the following 4 steps (noting that each step uses output from the previous stage):

- Create a canonical request (a digital fingerprint unique to the request), consisting of: - The HTTP request method - The canonical path - The canonical query string - The canonical headers - The signed headers - The hashed payload
- Create a string to sign, using the canonical request
- Calculate the signature, using the string to sign
- Add the authorization header to the request, using the signature

Each of these steps are described below.

Creating a canonical request

The canonical request is a representation of the request in a standardised (canonical) format that can be procedurally constructed (and reconstructed on the server) to be used as part of the signature calculation and verification process. The canonical request is constructed with the following request elements:

- The HTTP request method
- The canonical path
- The HTTP query string, in a canonical format (the canonical query string)
- The HTTP header names and values, in a canonical format (the canonical headers)
- The HTTP header names, in the order they are appear in the canonical headers (the signed headers)
- The payload, ordered and hashed using SHA-256 (the hashed payload)

These elements are joined together as a single string, delimited by newline (‘n’) character. The following example shows the pseudocode to create a canonical request:

```
CanonicalRequest = HTTPRequestMethod + '\n'
 + CanonicalURI + '\n'
 + CanonicalQueryString + '\n'
 + CanonicalHeaders + '\n'
 + SignedHeaders + '\n'
 + HashedPayload
```

To construct each of these elements, follow the steps below.

Constructing the HTTP request method

This is the HTTP request method (GET, PUT, POST, etc.) in uppercase.

Example for a POST request:

```
POST
```

Constructing the canonical path

The canonical path is the absolute path component of the entire URI - that is, everything in the URI from the end of the HTTP host component through to the question mark character (”?”) that begins the query string parameters. Each such path-segment should be URI-encoded and normalised according to RFC 3986. The absolute path component should be enclosed by an opening and trailing “/”.

A request to the identities endpoint `https://delta.covata.io/v1/identities` has the following canonical path:

```
/identities/
```

If the absolute path is empty, simply represent this as a forward slash (/):

```
/
```

If the canonical path requires encoding, this should be present in the string too:

```
/my%20secrets/
```

Constructing the canonical query string

The canonical query string consists of the query string, sorted, URI-encoded and normalised according to RFC 3986. If the request does not include a query string, use an empty string so that the delimited canonical header will include a blank line between the canonical request and the canonical headers:

```
POST
/identities/

content-type:application/json; charset=utf-8
...
```

To create the canonical query string:

1. Sort the parameter names by character code in ascending order (ASCII order). For example, a parameter name that begins with the uppercase letter F (ASCII code 70) precedes a parameter name that begins with a lowercase letter b (ASCII code 98).
2. URI-encode each parameter name and value according to the following rules:
   - Do not URI-encode any of the unreserved characters that RFC 3986 defines: A-Z, a-z, 0-9, hyphen (-), underscore (_), period (.), and tilde (~).
   - Percent-encode all other characters with %XY, where X and Y are hexadecimal characters (0-9 and uppercase A-F). For example, the space character must be encoded as %20. Do not include plus symbols ('+'), as some encoding schemes do.
   - Extended UTF-8 characters must be in the form %XY%ZA%BC.
3. Build the canonical query string by starting with the first parameter name in the sorted list.
4. For each parameter, append the URI-encoded parameter name, followed by the character '=' (ASCII code 61), followed by the URI-encoded parameter value. Use an empty string for parameters that have no value.
5. Append the character '&' (ASCII code 38) after each parameter value, except for the last value in the list.

Example of a canonical query string containing a single parameter:

```
sampleQueryParamName=sampleQueryParamValue
```

Example of a canonical query string containing 2 parameters where the first parameter has no value:

exampleQueryParamName=&sampleQueryParamName=sampleQueryParamValue

Constructing the canonical headers

The canonical headers consist of a list of all the HTTP headers that are included with the signed request in a form that Delta can interpret. At a minimum, the date header `Cvt-Date` must be included. Standard headers like Content-Type are optional. Be aware that different Delta API endpoints may require other headers.

To create the list of canonical headers:

1. Convert all header names to lowercase and remove leading and trailing spaces.
2. Convert sequential/consecutive spaces in the header value to a single space (and remove leading and trailing spaces from these segments).
3. Append the lowercase header name with a colon, followed immediately by the value itself. This concatenated string is the canonical header entry.
4. Lexicographically, sort all the canonical header entries.
5. Join all the canonical header entries, where each entry is delimited by a newline character (' n') followed by a space.

The following pseudocode describes how to construct the list of canonical headers:

```
CanonicalHeaderEntry = Lowercase(HeaderName)
 + ':'
 + Trimall(HeaderValue)
 + (FinalHeader ? '' : ' \n')

CanonicalHeaders = CanonicalHeaderEntry0
 + CanonicalHeaderEntry1
 + ...
 + CanonicalHeaderEntryN
```

Note:

- Lowercase() represents a function that converts all characters in its argument to lowercase.
- Trimall() represents a function that converts all sets of consecutive spaces in its argument's value (including any quoted content) to single spaces.

The following example shows the original, complex set of headers:

```
Host: delta.covata.io
Content-Type:application/json; charset=utf-8
My-header1:    a   b   c
Cvt-Date:20150830T123600Z
My-Header2:    "a   b   c"
```

And the list of headers in their canonical form:

```
content-type:application/json; charset=utf-8
 cvt-date:20150830T123600Z
 host:delta.covata.io
 my-header1:a b c
 my-header2:"a b c"
```

Constructing the signed headers

The signed headers is the list of headers which are included in the list canonical headers (above). The purpose of the signed headers list is to instruct Delta about which headers in the request have been included in the signing process and which headers can be ignored. Such additional headers may be those which are added after the client application has completed the signing process (for instance by a proxy) and therefore, these additional headers would be unknown to the client application making the request.

Hence, the signed headers list must represent every header included in the list of canonical headers (above).

To create the list of signed headers:

- Convert all header names to lowercase.
- Sort the lowercase header names lexicographically.
- Join all the sorted, lowercase headers delimited by a semicolon.

The following pseudocode describes how to construct the list of signed headers:

```
SignedHeaders = Lowercase(HeaderName0) + ';'
 + Lowercase(HeaderName1) + ";"
 + ...
 + Lowercase(HeaderNameN)
```

The following example shows a signed header string:

```
content-type;cvt-date;host;my-header1;my-header2
```

Constructing the hashed payload

Use a hash (digest) function like SHA256 to create a hashed value from the body of the request (i.e. the payload). The hashed payload must be represented as a lowercase hexadecimal string.

All such payloads are contained in a JSON object, whose contents should be sorted and compacted.

- Sorting is performed by member name and must be conducted at all levels of the entire JSON object (i.e. if any member has a value which itself is another JSON object, the contents of this nested JSON object need to be sorted too).
- The compacting process involves removing all spaces outside of everything contained outside quoted strings in the JSON object.

In summary, to construct the hashed payload:

1. Sort all members of every level in the JSON object payload by member name.
2. Compact the entire JSON object payload.
3. Run a SHA256 function on the compacted payload.
4. Encode the results of this function as hexadecimal.
5. Ensure that all alphabetical characters in the hexadecimal-encoded result are lowercase.

For requests with empty payloads (i.e. all GET requests), use the empty JSON object ("{}") as the payload. This should always result in a hashed value of `44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a`.

The following example shows a request's JSON object payload, which is unsorted and uncompacted:

```
{
    "signingPublicKey":
↪"E021472BCF554198752798A956DCB5065126D578CCCF632A6BB2BA1EEF7EE685",
    "cryptoPublicKey":
↪"220418D56A32B5B747EF301E57FA1466C229F03B1B11CC5B7900A996ACF360E8"
}
```

This is what the JSON object payload looks like after sorting and compacting:

```
{"cryptoPublicKey":"220418D56A32B5B747EF301E57FA1466C229F03B1B11CC5B7900A996ACF360E8",
↪"signingPublicKey":"E021472BCF554198752798A956DCB5065126D578CCCF632A6BB2BA1EEF7EE685
↪"}
```

And this is what the payload looks like after hashing with SHA256 and encoding as a lowercase hexadecimal string:

```
daadd72c2e2f5b63ad67e2131a598e4a6edcd75d6bc70c36e7e3f3ec5de95417
```

Example canonical request

To construct the completed canonical request, combine all the components from each step as a single string. As noted (above), each component ends with a newline character.

An example canonical request string is shown below:

```
POST
/identities/
sampleQueryParamName=sampleQueryParamValue
content-type:application/json; charset=utf-8
 cvt-date:20150830T123600Z
 host:delta.covata.io
 my-header1:a b c
 my-header2:"a b c"
content-type;cvt-date;host;my-header1;my-header2
daadd72c2e2f5b63ad67e2131a598e4a6edcd75d6bc70c36e7e3f3ec5de95417
```

Creating a string to sign

The string to sign is a set of strings representing meta information about the entire request.

The following pseudocode describes how to create the string to sign, which is a concatenation of the algorithm (representing this CVT1 request signing scheme), the date of the request (which must match the date in the header) and the digest of the canonical request (using SHA256), delimited with the newline ('n') character, as shown:

```
StringToSign = Algorithm
       + '\n' + RequestDate
       + '\n' + HashedCanonicalRequest
```

Algorithm

The designation for the CVT1 request signing scheme algorithm is:

```
CVT1-RSA4096-SHA256
```

Request Date

The request date is the value of the cvt-date header (which is in ISO8601 format YYYYMMDD'T'HHMMSS'Z'). The date/time must be in UTC and does not include milliseconds. This value must match the value you used in relevant previous stages.

Hashed Canonical Request

The canonical request (see Example canonical request and Stage 1 description above) whose content has been hashed using the SHA256 algorithm.

The hashed canonical request must be hex-encoded and be lowercase, as defined by Section 8 of RFC 4648.

Example string to sign

The following is a completed string to sign, with a date of 31st January, 2017 at 12:34.56pm:

```
CVT1-RSA4096-SHA256
20170131T123456Z
cc113fcef267dbbdce8416b1a9a8bcb09a32460142449c3289bc093598a9eef0
```

Calculating the signature

The final signature is calculated according to the following steps:

1. Calculate a SHA256 digest of the string to sign from the previous stage (above). The output must be hex-encoded and be lowercase, as defined by Section 8 of RFC 4648.
2. Obtain the private signing key of the identity that is making the request. This should be in base64-encoded DER format.

3. Create an RSA signature of the string to sign using RSASSA-PSS and the private signing key, with the following parameters: - SHA256 as the digest function - MGF1 with SHA256 as the mask generator function - 32 bytes as the salt value for MGF1 - base64 encoding

Note that RSASSA-PSS will generate a different result each time due to the salt.

The following shows an example SHA256 digest output of the string to sign:

```
725890633d7210c079a408d521c6545ffefc6c8e1fa8843052a047a1568a5912
```

Next, using the private signing key, create an RSA signature of the String to Sign. An example of a private signing key in base64-encoded DER format is shown below:

```
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQCk2gVVEjuQEUFKvysoJS8i3hjc0cJ9OJtAZHqz0QsUbjQz0jvu
↪YUr3IGd1LFFDfezXbChUGh+TcptCGr97BQuMEAiP1kPT+YtS8QYtfwTq13DvP4WZ9ql129m8dfrBPXO/
↪eBd0dSV3NLUiG1YIEnPWREJRAmV+FDWtxQYSBCa+JeUGRz3iRagL6oqDPpc2mcdU4o7gvjfoYNgTtcJw5Qnn6vRsu1oFgs7GgAt
↪yz/jvuB76jbgRZnctehxzQVVk/9Xb3GOFcOj4jpkEZX9VAgMBAAECggEAB9FfF125/
↪WcUFZtjTJAW4CxwOWipNI8OrcsWFpj/UYS4bQy3UuZc9GJF2KiuAV3eb5miWK46d2TsYqa/
↪XZcjEb2XuLU9wJPZPPk4qH2mayVf8zQP0xqsCajt7ywIg1psqzTP/Sl0YH6/lKqBA5Dzr5HVjwuE/
↪VrOwxTqntPSTWumhd2tXc434QdfEWXsVW7H6xKLPTZTK1jWYzQxYZmvf/
↪td5NKKXmhfY3TMamRHb2x5XDnnCE6ktOs83CffBISzhucSe/w5/1DChRy3Xuri442nIxKtQI1Ad21aI7C/
↪yoiUqPYpt0TkDJ53KFKgsPGvyY7c6fxL3ERfD/mpLO01gQKBgQDsxOWTf+p2C+det5p2j6iZ5N/
↪dyWL5tUm9WPnmX3r83h3VkOdEdeGPa8QNhVrHv8us9VYOUVDh/
↪0YDNzTJQog+QeV205NGQ272C2oRwkfo6ltHk/
↪0DEjyPFSw0viyPSoBN0gytRqULmA0ULXZ7LcL36zDrQlD0DUU3HJvG58NGyQKBgQCyPcFPU5Vlx3ll0YSezfj9e1N82/
↪bws7Tgk+3r6A8kdNvZN/9xZ7QKJ6a2ihHfs8HIGCGfTLCj5nXcbERfc784Cx4/
↪jvaMj2BICuqf5K4Xatab0FmAF9waOwtkO61/
↪dd8OKPf9nE04C8HZDQvIg5FqDtdHcOt9QsbudBrr7VKeLQKBgCuq2MiOY/
↪ink2GFrUhGkIrpilxGQynYxKPWYCib3Xv7nzb/
↪RZf7wcEI2BzCRo7mkbLxgJCdcLRtt0TqjqK70ZLh5mc2+EeSMknQqxxhX4/WgUU/
↪Rv+lAmRFPGTx2hgHXoh7v/
↪jJObFctrTM+bgYJYhB6UDKd1G7jNNwRE63+ez5AoGAVhbp1YzDbgNoqTsHWUSW7KeybW442Y2S4Z3Rns3Y8nzW6xXW9Ulndjgsl
↪XwtNqi8rQx5Rgc+Tf51jirtT/5fi1o+/8MO/+5zzy+sWTS/
↪zMk52+eybSXDfSdGiEueUJkdUQXL+jN2i4o8NJLW/SLGmB5/
↪WhReT7u+eEItIkCgYEAronO5VDCjZXEhJGebUKbKACHamp5DVxqhWsDxUHEldqA7V0OISYOpfOuVMeE7mIAae6yAGXLIhpSALr2
```

Initialising the cipher using the specified parameters will result in a signed string. This should be encoded in base64. RSASSA-PSS will generate a different signature every time, so running this example generate a different example each time.

Adding the authorization header

Add to the request an HTTP header named Authorization, whose value includes: - The algorithm of the CVT1 signing scheme. - The ID of the identity making the request. - The signed headers. - The signature calculated in Stage 3 (above).

The contents of the header are created after you calculate the signature as described in stage 3, so the Authorization header is not included in the list of signed headers. Although the header is named Authorization, the signing information is actually used for authentication.

The following pseudocode shows the construction of the Authorization header:

```
Authorization: algorithm Identity=identityId, SignedHeaders=signedHeaders,␣
↪Signature=signature
```

The following example shows a finished Authorization header (the algorithm is the CVT1 request signing scheme as designated by the string `CVT1-RSA4096-SHA256`):

```
Authorization: CVT1-RSA4096-SHA256 Identity=b15e50ea-ce07-4a3d-a4fc-0cd6b4d9ab13,␣
↪SignedHeaders=content-type;host;x-cvt-date,␣
↪Signature=ZwccJzSaGuNO0GRleZFpMqZ3VBs59VAxB7J6COubsCJTnVccmgyx/
↪uxtpRpi9qP20Ytk83SLY0ZyeXRphlZTyW7OqLB1I5U3as9AKqD4WpQK1iNPn7z6K1X3nODq3jWk2TqbcW2pMoFZXGvaCyN5j1ma
↪iEYGVDtGzzMdrGKKMfN6GUWVM9nwozXn82eqgjtvxw7X2eA/
↪ecGs44fy10KygdXHiaB+lkzTDfNh1k26FfHF5YEeiBCwCQahYHo89aac0/
↪LeWjjXqlqUiQntYnwUM7hYphbX8ArES75+4VtqIEGf1NCON52ctbifVLjXzhb8j20CfJgXhsl0fwoQpQ==
```

Note the following:

- There is no comma between the algorithm and Identity. However, the SignedHeaders and Signature are separated from the preceding values with a comma.
- The value is the identifier generated by Delta during identity registration.

CHAPTER 13

CVT1 Request Verification

The Delta service will process all requests to establish authenticity and set the identity for the actions requested. The following elements are extracted from the Authorization header:

- identityId
- signedHeaders
- signature

Using these elements, the following actions are performed on the request to ensure the signature is valid:

1. Create a canonical request, consisting of:
    - The HTTP request method
    - The canonical path
    - The canonical query string
    - The canonical headers, as determined by the list of signedHeaders
    - The signedHeaders list
    - The hashed payload
2. Create a string to sign, using the canonical request from step 1.
3. Calculate a SHA256 digest of the string to sign from step 2. The output must be hex-encoded and be lowercase, as defined by Section 8 of RFC 4648.
4. Retrieve the public signing (verification) key of the identity matching the identifierId.
5. Decrypt the signature with the public signing (verification) key using RSASSA-PSS with the following parameters:
    - SHA256 as the digest function
    - MGF1 with SHA256 as the mask generator function
    - 32 bytes as the salt value for MGF1

6. Check the decrypted signature (from step 5) matches the digested string to sign (from step 3)- if match, then the request is allowed to continue, otherwise a HTTP status 403 is returned

CHAPTER 14

Glossary

Delta A framework for protecting content so that it can be shared securely across networks and organisations. Delta achieves this by allowing an identity to create and distribute secrets to other identities.

Identity An entity (such as user, device, or another service) registered with Delta, that is uniquely identifiable, and has possession of an encryption key pair and a signing key pair.

IdentityId An identifier generated by Delta that is unique and associated with retrieval and designation of an identity.

Identity metadata Optional, textual, owner-provided key-value pairs used for lookup, associated with an identity.

Encryption key pair An asymmetric key pair (consisting of the public encryption key and the private decryption key), associated with an identity for the purpose of encryption and decryption of secret encryption keys.

Public encryption key The public key of an asymmetric key pair (the private counterpart being the private decryption key), functioning as a key encryption key (KEK), to encrypt a secret encryption key. The public encryption key is associated with an identity and published on Delta.

Private decryption key The private key of an asymmetric key pair (the public counterpart being the public encryption key), used to decrypt a secret encryption key. The private decryption key must be managed outside of Delta.

Signing key pair An asymmetric key pair (consisting of the private signing key and the public signing verification key), associated with an identity for the purpose of request signing and authentication.

Public signing verification key The public key of an asymmetric key pair (the private counterpart being the private signing key), used to verify request authenticity and ownership. The public signing key is associated with an identity. However, unlike public encryption keys, public signing verification keys are not published by Delta.

Private signing key The private key of an asymmetric key pair (the public counterpart being the public signing verification key), used to sign requests as required by Delta so that the service can verify request authenticity and ownership. The private signing key must be managed outside of Delta.

Secret An entry in Delta, comprising of protected secret content, encryption details, core attributes, and secret metadata.

SecretId An identifier generated by Delta that is unique and associated with retrieval and designation of a secret.

Secret content Plaintext data that an identity wants to protect (such as a password, symmetric key or small text file). Secret content is limited to 200KB in size.

Protected secret content Encrypted secret content, protected by a secret encryption key.

Secret metadata Optional, textual, owner-provided key-value pairs used for lookup, associated with a secret.

Encryption details Client-generated attributes associated with the encryption of the secret content. The encryption details contain the protected secret encryption key, initialization vector (IV), and any other keying material required for the decryption of the protected secret content.

Core attributes Service-generated attributes associated with a secret (SecretId, Owning IdentityId, CreatedDate, ModifiedDate).

Secret encryption key A symmetric key used to encrypt or decrypt the secret content. The secret encryption key in turn is protected by a public encryption key of an Identity.

Protected secret encryption key The secret encryption key, encrypted with the public encryption key of an identity and stored as part of a secret.

Base secret A secret whose secret content is encrypted using the public encryption key of the owning identity.

Owning identity An identity in Delta responsible for creation of a base secret.

Receiving identity An identity in Delta that is the recipient of a derived secret.

Derived secret A secret that should contain the same secret content as a base secret, where this secret content is protected/encrypted by a secret encryption key, which in turn is encrypted using the public encryption key of the receiving identity. Derived secrets are the mechanism through which secret content is shared between identities. A derived secret can only be created by the owning identity of the base secret.

Events Operations performed on identity and secret entries in Delta that can be retrieved in a structured manner.

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

CHAPTER 16

License

Python Module Index

C

Index

I

R

S

U