pyUmbral Documentation

Release 0.3.0

Michael Egorov, Justin Myles Holmes, David Nuñez, John Pacific,

Sep 15, 2021

Table of Contents:

pyUmbral is the reference implementation of the Umbral threshold proxy re-encryption scheme. It is open-source, built with Python, and uses OpenSSL via Cryptography.io, and libsodium via PyNaCl.

Using Umbral, Alice (the data owner) can *delegate decryption rights* to Bob for any ciphertext intended to her, through a re-encryption process performed by a set of semi-trusted proxies or *Ursulas*. When a threshold of these proxies participate by performing re-encryption, Bob is able to combine these independent re-encryptions and decrypt the original message using his private key.

pyUmbral is the cryptographic engine behind nucypher, a proxy re-encryption network to empower privacy in decentralized systems.

CHAPTER 1

Installing pyUmbral

1.1 Using pip

The easiest way to install pyUmbral is using `pip`:

```
$ pip3 install umbral
```

1.2 Build from source code

pyUmbral is maintained on GitHub: https://github.com/nucypher/pyUmbral.

Clone the repository to download the source code.

```
$ git clone https://github.com/nucypher/pyUmbral.git
```

Once you have acquired the source code, you can...

... embed pyUmbral modules into your own codebase...

```
from umbral import pre, keys, config
```

... install pyUmbral with pipenv...

```
$ pipenv install .
```

... or install it with python-pip...

```
$ pip3 install .
```

1.3 Install dependencies

The NuCypher team uses pipenv for managing pyUmbral's dependencies. The recommended installation procedure is as follows:

```
$ sudo pip3 install pipenv
$ pipenv install
```

Post-installation, you can activate the pyUmbral's virtual environment in your current terminal session by running `pipenv shell`.

If your installation is successful, the following command will succeed without error.

```
$ pipenv run python
>>> import umbral
```

For more information on pipenv, The official documentation is located here: https://docs.pipenv.org/.

1.4 Development Installation

If you want to participate in developing pyUmbral, you'll probably want to run the test suite and / or build the documentation, and for that, you must install some additional development requirements.

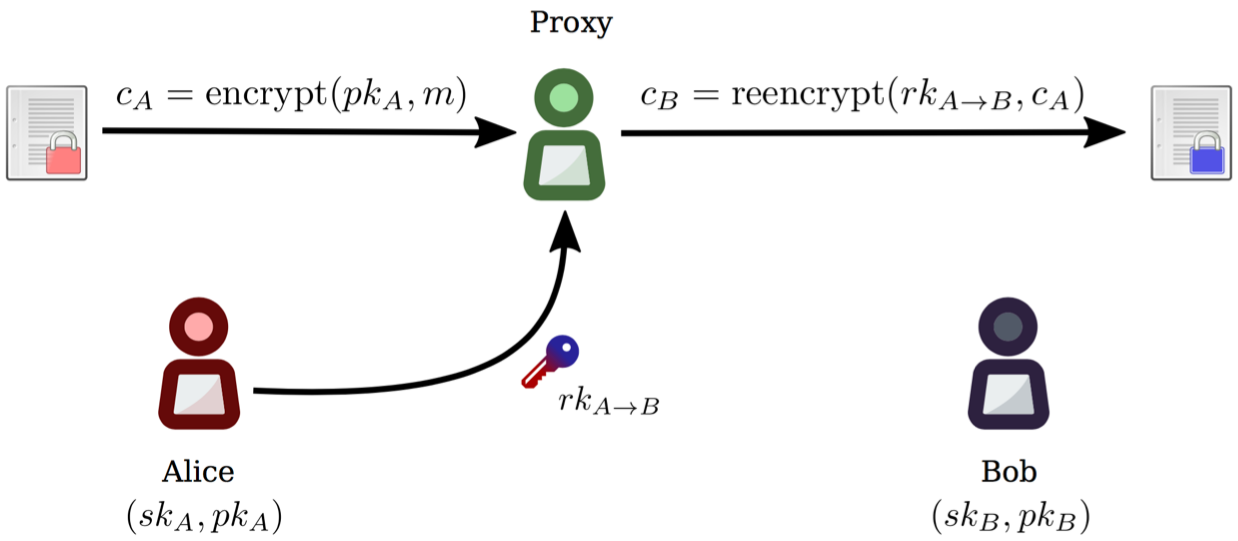
```
$ pipenv install --dev --three
```

To build the documentation locally:

```
$ pipenv run make html --directory=docs
```

CHAPTER 2

Using pyUmbral



2.1 Elliptic Curves

The matter of which curve to use is the subject of some debate. If you aren't sure, you might start here: https://safecurves.cr.yp.to/

A number of curves are available in the Cryptography.io library, on which pyUmbral depends. You can find them in the `cryptography.hazmat.primitives.asymmetric.ec` module.

Important: Be careful when choosing a curve - the security of your application depends on it.

We provide curve `SECP256K1` as a default because it is the basis for a number of crypto-blockchain projects; we don't otherwise endorse its security. We additionally support curves `SECP256R1` (also known as "NIST P-256") and `SECP384R1` ("NIST P-384"), but they cannot currently be selected via the public API.

2.2 Encryption

2.2.1 Generate an Umbral key pair

First, let's generate two asymmetric key pairs for Alice: A delegating key pair and a signing key pair.

```
>>> from umbral import SecretKey, Signer

>>> alices_secret_key = SecretKey.random()
>>> alices_public_key = alices_secret_key.public_key()

>>> alices_signing_key = SecretKey.random()
>>> alices_verifying_key = alices_signing_key.public_key()
>>> alices_signer = Signer(alices_signing_key)
```

2.2.2 Encrypt with a public key

Now let's encrypt data with Alice's public key. Invocation of *`umbral.encrypt()`* returns both a `capsule` and a `ciphertext`. Note that anyone with Alice's public key can perform this operation.

```
>>> from umbral import encrypt
>>> plaintext = b'Proxy Re-encryption is cool!'
>>> capsule, ciphertext = encrypt(alices_public_key, plaintext)
```

2.2.3 Decrypt with a private key

Since data was encrypted with Alice's public key, Alice can open the capsule and decrypt the ciphertext with her private key.

```
>>> from umbral import decrypt_original
>>> cleartext = decrypt_original(alices_secret_key, capsule, ciphertext)
```

2.3 Threshold Re-Encryption

2.3.1 Bob Exists

```
>>> bobs_secret_key = SecretKey.random()
>>> bobs_public_key = bobs_secret_key.public_key()
```

2.3.2 Alice grants access to Bob by generating kfrags

When Alice wants to grant Bob access to view her encrypted data, she creates *re-encryption key fragments*, or *"kfrags"*, which are next sent to N proxies or *Ursulas*.

Alice must specify `shares` (the total number of kfrags), and a `threshold` (the minimum number of kfrags needed to activate a capsule). In the following example, Alice creates 20 kfrags, but Bob needs to get only 10 re-encryptions to activate the capsule.

```
>>> from umbral import generate_kfrags
>>> kfrags = generate_kfrags(delegating_sk=alices_secret_key,
...                          receiving_pk=bobs_public_key,
...                          signer=alices_signer,
...                          threshold=10,
...                          shares=20)
```

2.3.3 Bob receives a capsule

Next, let's generate a key pair for Bob, and pretend to send him the capsule through a side channel like S3, IPFS, Google Cloud, Sneakernet, etc.

```
# Bob receives the capsule through a side-channel: IPFS, Sneakernet, etc.
capsule = <fetch the capsule through a side-channel>
```

2.3.4 Bob fails to open the capsule

If Bob attempts to open a capsule that was not encrypted for his public key, or re-encrypted for him by Ursula, he will not be able to open it.

```
>>> fail = decrypt_original(delegating_sk=bobs_secret_key,
...                         capsule=capsule,
...                         ciphertext=ciphertext)
Traceback (most recent call last):
    ...
ValueError
```

2.3.5 Ursulas perform re-encryption

Bob asks several Ursulas to re-encrypt the capsule so he can open it. Each Ursula performs re-encryption on the capsule using the `kfrag` provided by Alice, obtaining this way a "capsule fragment", or `cfrag`. Let's mock a network or transport layer by sampling `threshold` random kfrags, one for each required Ursula.

Bob collects the resulting cfrags from several Ursulas. Bob must gather at least `threshold` cfrags in order to open the capsule.

```
>>> import random
>>> kfrags = random.sample(kfrags,  # All kfrags from above
...                        10)      # M - Threshold

>>> from umbral import reencrypt
>>> cfrags = list()                 # Bob's cfrag collection
>>> for kfrag in kfrags:
...     cfrag = reencrypt(capsule=capsule, kfrag=kfrag)
...     cfrags.append(cfrag)        # Bob collects a cfrag
```

2.4 Decryption

2.4.1 Bob checks the capsule fragments

If Bob received the capsule fragments in serialized form, he can verify that they are valid and really originate from Alice, using Alice's public keys.

```
>>> from umbral import CapsuleFrag
>>> suspicious_cfrags = [CapsuleFrag.from_bytes(bytes(cfrag)) for cfrag in cfrags]
>>> cfrags = [cfrag.verify(capsule,
...                        verifying_pk=alices_verifying_key,
...                        delegating_pk=alices_public_key,
...                        receiving_pk=bobs_public_key,
...                        )
...           for cfrag in suspicious_cfrags]
```

2.4.2 Bob opens the capsule

Finally, Bob decrypts the re-encrypted ciphertext using his key.

```
>>> from umbral import decrypt_reencrypted
>>> cleartext = decrypt_reencrypted(receiving_sk=bobs_secret_key,
...                                 delegating_pk=alices_public_key,
...                                 capsule=capsule,
...                                 verified_cfrags=cfrags,
...                                 ciphertext=ciphertext)
```

CHAPTER 3

Public API

3.1 Keys

class umbral.**SecretKey**

Bases: *umbral.serializable.SerializableSecret*, *umbral.serializable.Deserializable*

Umbral secret (private) key.

public_key() → umbral.keys.PublicKey
Returns the associated public key.

classmethod random() → umbral.keys.SecretKey
Generates a random secret key and returns it.

classmethod serialized_size()
Returns the size in bytes of the serialized representation of this object (obtained with `bytes()` or `to_secret_bytes()`).

to_secret_bytes() → bytes
Serializes the object into bytes. This bytestring is secret, handle with care!

class umbral.**PublicKey**

Bases: *umbral.serializable.Serializable*, *umbral.serializable.Deserializable*

Umbral public key.

Created using *SecretKey.public_key()*.

__eq__(*other*)
Return self==value.

__hash__() → int
Return hash(self).

classmethod serialized_size()
Returns the size in bytes of the serialized representation of this object (obtained with `bytes()` or `to_secret_bytes()`).

class umbral.**SecretKeyFactory**
Bases: *umbral.serializable.SerializableSecret*, *umbral.serializable.Deserializable*

This class handles keyring material for Umbral, by allowing deterministic derivation of *SecretKey* objects based on labels.

Don't use this key material directly as a key.

classmethod from_secure_randomness(*seed: bytes*) → umbral.keys.SecretKeyFactory
Creates a secret key factory using the given random bytes (of size *seed_size()*).

> **Warning:** Make sure the given seed has been obtained from a cryptographically secure source of randomness!

make_factory(*label: bytes*) → umbral.keys.SecretKeyFactory
Creates a *SecretKeyFactory* deterministically from the given label.

make_key(*label: bytes*) → umbral.keys.SecretKey
Creates a *SecretKey* deterministically from the given label.

classmethod random() → umbral.keys.SecretKeyFactory
Creates a random factory.

classmethod seed_size()
Returns the seed size required by *from_secure_randomness()*.

classmethod serialized_size()
Returns the size in bytes of the serialized representation of this object (obtained with `bytes()` or `to_secret_bytes()`).

to_secret_bytes() → bytes
Serializes the object into bytes. This bytestring is secret, handle with care!

class umbral.**Signer**(*secret_key: umbral.keys.SecretKey*)
An object possessing the capability to create signatures. For safety reasons serialization is prohibited.

sign(*message: bytes*) → umbral.signing.Signature
Hashes and signs the message.

verifying_key() → umbral.keys.PublicKey
Returns the public verification key corresponding to the secret key used for signing.

class umbral.**Signature**
Bases: *umbral.serializable.Serializable*, *umbral.serializable.Deserializable*

Wrapper for ECDSA signatures.

__eq__(*other*)
Return self==value.

__hash__() → int
Return hash(self).

classmethod serialized_size()
Returns the size in bytes of the serialized representation of this object (obtained with `bytes()` or `to_secret_bytes()`).

verify(*verifying_pk: umbral.keys.PublicKey*, *message: bytes*) → bool
Returns `True` if the `message` was signed by someone possessing the secret counterpart to `verifying_pk`.

3.2 Intermediate objects

class umbral.**Capsule**
Bases: *umbral.serializable.Serializable*, *umbral.serializable.Deserializable*

Encapsulated symmetric key.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

class umbral.**KeyFrag**
Bases: *umbral.serializable.Serializable*, *umbral.serializable.Deserializable*

A signed fragment of the delegating key.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

verify(*verifying_pk: umbral.keys.PublicKey*, *delegating_pk: Optional[umbral.keys.PublicKey] = None*, *receiving_pk: Optional[umbral.keys.PublicKey] = None*) → umbral.key_frag.VerifiedKeyFrag
Verifies the validity of this fragment.

If the delegating and/or receiving key were not signed in *generate_kfrags()*, but are given to this function, they are ignored.

class umbral.**VerifiedKeyFrag**
Bases: *umbral.serializable.Serializable*

Verified kfrag, good for reencryption. Can be cast to `bytes`, but cannot be deserialized from bytes directly. It can only be obtained from *KeyFrag.verify()*.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

classmethod from_verified_bytes(*data*) → umbral.key_frag.VerifiedKeyFrag
Restores a verified keyfrag directly from serialized bytes, skipping *KeyFrag.verify()* call.

Intended for internal storage; make sure that the bytes come from a trusted source.

classmethod serialized_size()
Returns the size in bytes of the serialized representation of this object (obtained with `bytes()` or `to_secret_bytes()`).

class umbral.**CapsuleFrag**
Bases: *umbral.serializable.Serializable*, *umbral.serializable.Deserializable*

Re-encrypted fragment of *Capsule*.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

classmethod serialized_size()
Returns the size in bytes of the serialized representation of this object (obtained with `bytes()` or `to_secret_bytes()`).

verify(*capsule: umbral.capsule.Capsule*, *verifying_pk: umbral.keys.PublicKey*, *delegating_pk: umbral.keys.PublicKey*, *receiving_pk: umbral.keys.PublicKey*) → umbral.capsule_frag.VerifiedCapsuleFrag
Verifies the validity of this fragment.

class umbral.**VerifiedCapsuleFrag**
Bases: *umbral.serializable.Serializable*

Verified capsule frag, good for decryption. Can be cast to `bytes`, but cannot be deserialized from bytes directly. It can only be obtained from *CapsuleFrag.verify()*.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

classmethod from_verified_bytes(*data*) → umbral.capsule_frag.VerifiedCapsuleFrag
Restores a verified capsule frag directly from serialized bytes, skipping *CapsuleFrag.verify()* call.

Intended for internal storage; make sure that the bytes come from a trusted source.

classmethod serialized_size()
Returns the size in bytes of the serialized representation of this object (obtained with `bytes()` or `to_secret_bytes()`).

3.3 Encryption, re-encryption and decryption

umbral.**encrypt**(*delegating_pk: umbral.keys.PublicKey*, *plaintext: bytes*) → Tuple[umbral.capsule.Capsule, bytes]
Generates and encapsulates a symmetric key and uses it to encrypt the given plaintext.

Returns the KEM Capsule and the ciphertext.

umbral.**decrypt_original**(*delegating_sk: umbral.keys.SecretKey*, *capsule: umbral.capsule.Capsule*, *ciphertext: bytes*) → bytes
Opens the capsule using the delegator's key used for encryption and gets what's inside. We hope that's a symmetric key, which we use to decrypt the ciphertext and return the resulting cleartext.

umbral.**generate_kfrags**(*delegating_sk: umbral.keys.SecretKey*, *receiving_pk: umbral.keys.PublicKey*, *signer: umbral.signing.Signer*, *threshold: int*, *shares: int*, *sign_delegating_key: bool = True*, *sign_receiving_key: bool = True*) → List[umbral.key_frag.VerifiedKeyFrag]
Generates `shares` key fragments to pass to proxies for re-encryption. At least `threshold` of them will be needed for decryption. If `sign_delegating_key` or `sign_receiving_key` are `True`, the corresponding keys will have to be provided to *KeyFrag.verify()*.

umbral.**reencrypt**(*capsule: umbral.capsule.Capsule*, *kfrag: umbral.key_frag.VerifiedKeyFrag*) → umbral.capsule_frag.VerifiedCapsuleFrag
Creates a capsule fragment using the given key fragment. Capsule fragments can later be used to decrypt the ciphertext.

umbral.**decrypt_reencrypted**(*receiving_sk: umbral.keys.SecretKey, delegating_pk: umbral.keys.PublicKey, capsule: umbral.capsule.Capsule, verified_cfrags: Sequence[umbral.capsule_frag.VerifiedCapsuleFrag], ciphertext: bytes*) → bytes

Decrypts the ciphertext using the original capsule and the reencrypted capsule fragments.

3.4 Utilities

class umbral.**VerificationError**
Bases: Exception

Integrity of the data cannot be verified, see the message for details.

class umbral.serializable.**HasSerializedSize**
A base serialization mixin, denoting a type with a constant-size serialized representation.

classmethod serialized_size() → int
Returns the size in bytes of the serialized representation of this object (obtained with bytes() or to_secret_bytes()).

class umbral.serializable.**Serializable**
Bases: *umbral.serializable.HasSerializedSize*

A mixin for composable serialization.

__bytes__()
Serializes the object into bytes.

class umbral.serializable.**SerializableSecret**
Bases: *umbral.serializable.HasSerializedSize*

A mixin for composable serialization of objects containing secret data.

to_secret_bytes()
Serializes the object into bytes. This bytestring is secret, handle with care!

class umbral.serializable.**Deserializable**
Bases: *umbral.serializable.HasSerializedSize*

A mixin for composable deserialization.

classmethod from_bytes(*data: bytes*) → Self
Restores the object from serialized bytes.

CHAPTER 4

Academic Whitepaper

The Umbral scheme academic whitepaper and cryptographic specifications are available on GitHub.

> "Umbral: A Threshold Proxy Re-Encryption Scheme" *by David Nuñez*. https://github.com/nucypher/umbral-doc/blob/master/umbral-doc.pdf

CHAPTER 5

Support & Contribute

- Issue Tracker: https://github.com/nucypher/pyUmbral/issues
- Source Code: https://github.com/nucypher/pyUmbral

CHAPTER 6

Security

If you identify vulnerabilities with _any_ nucypher code, please email security@nucypher.com with relevant information to your findings. We will work with researchers to coordinate vulnerability disclosure between our partners and users to ensure successful mitigation of vulnerabilities.

Throughout the reporting process, we expect researchers to honor an embargo period that may vary depending on the severity of the disclosure. This ensures that we have the opportunity to fix any issues, identify further issues (if any), and inform our users.

Sometimes vulnerabilities are of a more sensitive nature and require extra precautions. We are happy to work together to use a more secure medium, such as Signal. Email security@nucypher.com and we will coordinate a communication channel that we're both comfortable with.

CHAPTER 7

Indices and Tables

- genindex
- modindex
- search

Python Module Index

u

Index

Symbols

C

D

E

F

G

H

K

M

P

R

S

T

U

V