pyUmbral Proxy Re-encryption Library Audit

NuCypher

November 1, 2018 – Version 1.0

Prepared for

MacLane Wilkison
Michael Egorov
John Pacific
David Núñez
Justin Holmes

Prepared by

David Wong
Maximilian Fillinger

Document Change Log

Version	Date	Change
1.0	2018-09-07	Initial report
1.1	2018-09-07	Updated for initial technical review
1.2	2018-10-01	Updated for final technical review
1.3	2018-10-12	Ready for NuCypher

Executive Summary

At the end of May 2018, NuCypher engaged NCC Group to review the pyUmbral library. The pyUmbral library implements a n-to-m proxy re-encryption scheme based on a construction from Dr. David Nuñez.[1] Over ten person-days, one consultant from NCC Group's Cryptography Services practice completed an audit of the overall security of the library as well as the implementation's conformance to the specified Umbral cryptosystem. In addition, a minimal amount of time was spent fuzzing the library. The fuzzing code is supplied at the end of this document.

Scope

Commit 8dbaf21c1f02e402621d62d64bd826bde75a44e3 of pyUmbral was audited, the scope included the following items:

Adherence to the specification. pyUmbral is a biased implementation of the Umbral cryptosystem using secp256k1 as its default curve (although other curves can be used with it), BLAKE2 as its hash function, HKDF as its key derivation function and Chacha20-Poly1305 as its authenticated encryption cypher.

Unexpected exceptions. As the library is implemented in Python, its flow was tested for exceptions being raised unexpectedly. As users of the library might not catch these exceptions, they would lead to denial of services.

Dangerous flows. Every function implied to be public and accepting user input was assessed for any dangerous path that could lead to unexpected or wrong logic.

Safety of the library API. As pyUmbral is an open source project, NCC Group assessed the safe usage and documentation of the library.

Safe implementation of the internal abstraction libraries. pyUmbral abstracts several concepts including big numbers, elliptic curve elements and elliptic curve points. The library does that through classes implementing OpenSSL functions that are available through the cryptography.io[2] Python library.

Note that NuCypher's particular usage of the library was out of scope.

Key Findings

Multiple issues were found and reported including the following:

- pyUmbral makes use of **OpenSSL** bindings created by the cryptography.io library to implement curve arithmetic in Python. In a number of locations, **memory reserved by the OpenSSL functions is not properly released after use**. This might lead to slow down in long-lasting processes and worse, to crashes.
- Formal (as well as edge-case) checks are not properly implemented in most public functions of pyUmbral. This **lack of data validation** coupled with a **lack of guidance on which functions might raise an exception** could end up being exploited in denial-of-service attacks.
- **Different applications using pyUmbral might collide** in ways that would allow their nodes to share users' data against their expectations.
- **The byte size of curve points is not correctly computed by the library**, which will lead to some curves not being usable with pyUmbral.

Strategic Recommendations

NCC Group recommends NuCypher take the following points into consideration in order to increase the security stance of the library:

- Implement input validation for all sensitive functions of pyUmbral and raise exceptions accordingly.
- Implement domain separation for different applications in order to avoid nodes being able to re-encrypt content on one application when the consent was given on another one.
- Document what functions of the library are public and can be used by external applications. Additionally, document what kind of exceptions the different functions might raise and indicate when a try-except flow must be used in order to safely take advantage of NuCypher's library.
- Consider modeling proofs of the protocol implemented in pyUmbral via a protocol proof solver like Tamarin.[3] The Umbral protocol is quite involved and its pyUmbral implementation diverges slightly from the paper. Since its internals use known cryptographic primitives, it is a good candidate for protocol proof solvers that would give an assurance of the solidity of the scheme.

[1] Umbral: A Threshold Proxy Re-Encryption Scheme
[2] https://cryptography.io/en/latest/
[3] https://tamarin-prover.github.io/

Dashboard

Target Metadata

Name	pyUmbral
Type	Library
Platforms	Python

Engagement Data

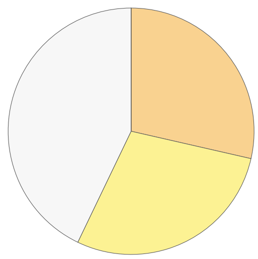
Type	Code Review
Method	Code-assisted
Dates	2018-05-21 to 2018-06-01
Consultants	1
Level of effort	10 person-days

Targets

Repository	github.com/nucypher/pyUmbral/tree/8dbaf21c1f02e402621d62d64bd826bde75a44e3

Finding Breakdown

Critical Risk issues	0
High Risk issues	0
Medium Risk issues	2
Low Risk issues	2
Informational issues	3
Total issues	**7**



Category Breakdown

Cryptography	4
Data Validation	1
Denial of Service	2

Key

Critical High Medium Low Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 15.

Title	ID	Risk
Uncaught Exceptions Might Lead to Denial Of Services	002	Medium
Lack of Domain Separation in pyUmbral's Hash Functions and PRFs	004	Medium
Progressive Memory Leak Due to Oversight in Freeing OpenSSL Structures	001	Low
Missing Checks Might Lead to Incorrect Flow or Denial of Services	005	Low
Wrong Calculation of Compressed Point Size	003	Informational
Lack of Enforcement for Supported Curves	006	Informational
Underlying Cryptographic Library Hardcodes Wrong Curve Order Size	007	Informational

Finding	**Uncaught Exceptions Might Lead to Denial Of Services**
Risk	Medium Impact: Medium, Exploitability: Medium
Identifier	NCC-nucypher-pyumbral-002
Category	Denial of Service
Location	• dem.py • point.py
Impact	An attacker could use pyUmbral public functions to submit maliciously formed inputs and crash the application resulting in denial of service conditions for legitimate users.

Description

pyUmbral attempts to differentiate private functions from public functions by adding an underscore character (_) as a prefix to the function names that are not intended to be used outside of the library. Therefore, all functions without that leading underscore character are, by default, deemed callable by applications. According to the current documentation, a number of public functions are meant to be used directly without a try-except flow. Furthermore, exceptions are not documented and in some cases not expected. This currently allows malevolent users to provide specifically crafted inputs that will raise exceptions and ultimately crash applications that have not anticipated them. Below is a list of such issues:

In dem.py, the `decrypt()` function takes a ciphertext byte string and decrypts it using the `ChaCha20Poly1305()` function from the *cryptography.io* library. This function can possibly raise an `InvalidTag`[4] exception to the caller of `decrypt()` in the `pre.py` file. Since the documentation does not recommend the use of the try-except paradigm for this call, an application may fail to catch that exception. The consequence is that malicious users could craft invalid ciphertext messages in order to crash the application using the pyUmbral library.

```
def decrypt(self, ciphertext: bytes, authenticated_data: bytes=None):
    """
    Decrypts data using ChaCha20-Poly1305 and validates the provided
    authenticated data.
    """
    nonce = ciphertext[:DEM_NONCE_SIZE]
    ciphertext = ciphertext[DEM_NONCE_SIZE:]
    cleartext = self.cipher.decrypt(nonce, ciphertext, authenticated_data)
    return cleartext
```

In point.py, the `Point` class' `from_bytes()` method transforms a bytestring into an internal representation of an elliptic curve point. If the bytestring is empty, the application will attempt to access its first byte and will raise an `IndexError` exception (list index out of range). If the application does not check for such empty bytestrings, an input containing an empty argument could be crafted in order to trigger this exception and crash the application.

```
class Point(object):
    # ...
    @classmethod
    def from_bytes(cls, data, curve: ec.EllipticCurve=None):
        """
        Returns a Point object from the given byte data on the curve provided.
        """
        # ...
```

[4] https://cryptography.io/en/latest/hazmat/primitives/aead/#cryptography.hazmat.primitives.ciphers.aead.ChaCha20Poly1305.decrypt

```
        if data[0] in [2, 3]:
            # ...
```

Recommendation

Properly name and document all the library's public functions that are meant to be used by external applications. This includes documenting what kind of exception a function might raise or if the function can be used safely without a try-except flow.

In addition, library functions that accept internal representations like Points or CurveBN as parameters should not be marked as public functions and kept only for internal use.

Finding Lack of Domain Separation in pyUmbral's Hash Functions and PRFs

Risk Medium Impact: Medium, Exploitability: Low

Identifier NCC-nucypher-pyumbral-004

Category Cryptography

Impact Different applications making use of the pyUmbral library could suffer from unwanted interactions. More clever attacks might be possible against the pyUmbral scheme if no domain separation exists in between its hash functions.

Description The Umbral cryptographic scheme uses a number of different hash functions (H_2, H_3 and H_4) and PRFs (KDF). pyUmbral implements these functions with BLAKE2b[5] and HKDF.[6] BLAKE2, as defined in its specification,[7] includes a "personalization" parameter allowing the user to use a string to create different hash functions. The same is available in HKDF as an "info" parameter to define an application-specific KDF.

The pyUmbral implementation does not use any of these domain-separation features.

These customization options allow for different applications to not collude in complex attacks that would force their separated users to interact maliciously with one another. They also prevent any in-application mixing that would happen from results of hash functions being maliciously used as results of different hash functions.

For example, consider the following attack:

- Alice and Bob use application Y and application Z.
- Alice delegates her decryption rights to Bob on application Y.
- A threshold of nodes of application Y share their re-encryption key fragments to a node of application Z.
- The node can now help Bob decrypt Alice's documents that are on application Z.

Recommendation Unfortunately, cryptography.io does not support the personalization parameter[8]:

> While the RFC specifies keying, personalization, and salting features, these are not supported at this time due to limitations in OpenSSL 1.1.0.

For this reason, NuCypher has several options:

- Use a different implementation of BLAKE2 that supports the personalization feature.
- Use a fixed-sized string as personalization, prepended to the inputs of the hash function.

For HKDF, use the `info` parameter[9] in order to set a personalization string.

In addition, NuCypher should enforce the user of the library to define an additional customization string (which could be added to BLAKE2 personalization string, HKDF's info parameter and serialized private keys) to avoid collisions between different applications making use of pyUmbral.

[5] https://blake2.net/
[6] https://tools.ietf.org/html/rfc5869
[7] https://blake2.net/blake2.pdf
[8] https://cryptography.io/en/latest/hazmat/primitives/cryptographic-hashes/#blake2
[9] https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/#cryptography.hazmat.primitives.kdf.hkdf.HKDF

Finding **Progressive Memory Leak Due to Oversight in Freeing OpenSSL Structures**

Risk Low Impact: Low, Exploitability: Low

Identifier NCC-nucypher-pyumbral-001

Category Denial of Service

Location
- openssl.py

Impact Long-lived processes using some of the pyUmbral library's functions will leak memory progressively, slowing down the program (and other processes), until it exhausts all available memory and crashes.

Description pyUmbral makes use of the cryptography.io library's bindings to the C OpenSSL library in order to perform most of its elliptic curve operations. These functions often create and return temporary structures allocated on the heap that need to be freed with the relevant OpenSSL functions like `BN_clear_free()` and `EC_POINT_free()`. pyUmbral correctly frees most of these, but still fails to release the memory allocated by some of its functions. Due to that issue, long-lived processes running pyUmbral will accumulate allocated blocks in memory, leading to a progressive program slow down and eventual crash. Below is a list of locations where the problem has been noticed.

In openssl.py, the `_bn_is_on_curve()` function directly uses cryptography.io's `backend._int_to_bn(0)` instead of pyUmbral's `_int_to_bn()` wrapper:

```
def _bn_is_on_curve(check_bn, curve_nid: int):
    # ...
    check_sign = backend._lib.BN_cmp(check_bn, backend._int_to_bn(0))
    # ...
```

As per cryptography.io's documentation,[10] this function creates a BN structure that needs to be freed explicitly:

> Converts a python integer to a BIGNUM. The returned BIGNUM will not be garbage collected (to support adding them to structs that take ownership of the object). Be sure to register it for GC if it will be discarded after use.

Another function, `_get_ec_group_by_curve_nid()`, uses the `backend._lib.EC_GROUP_new_by_curve_name()` that returns an `EC_GROUP`. This structure is subsequently never freed with `_lib.EC_GROUP_free()`[11]:

```
def _get_ec_group_by_curve_nid(curve_nid: int):
    """
    Returns the group of a given curve via its OpenSSL nid.
    """
    group = backend._lib.EC_GROUP_new_by_curve_name(curve_nid)
    # ...
```

The `_get_ec_generator_by_curve_nid()` function returns an `EC_POINT` created by the `backend._lib.EC_GROUP_get0_generator()` OpenSSL binding. The memory allocated by that structure needs to be freed later on, via the `EC_POINT_FREE()` function[12]:

[10] https://www.pydoc.io/pypi/cryptography-2.0.1/autoapi/hazmat/backends/openssl/backend/index.html#hazmat.backends.openssl.backend.Backend._int_to_bn
[11] https://github.com/pyca/cryptography/blob/50bad375f5dd3fbb7c7ea62896e2538dc5734be6/src/cryptography/hazmat/backends/openssl/backend.py#L1313
[12] https://github.com/pyca/cryptography/blob/50bad375f5dd3fbb7c7ea62896e2538dc5734be6/src/cryptography/

```
def _get_ec_generator_by_curve_nid(curve_nid: int):
    """
    Returns the generator point of a given curve via its OpenSSL nid.
    """
    # ...
    generator = backend._lib.EC_GROUP_get0_generator(ec_group)
    # ...
```

In curvebn.py, the `hash()` method uses the following function to create a BIGNUM variable containing the value one:

```
_1 = backend._lib.BN_value_one()
```

This value needs to be freed with the `backend._lib.BN_clear_free` function.

Recommendation If an OpenSSL structure (returned by an OpenSSL function) needs to be explicitly freed at some later point, create a wrapper around that function to register it with the garbage collector. This was done in pyUmbral, for example, by wrapping the OpenSSL binding `backend._int_to_bn()` into a custom `_int_to_bn()` function that takes care of registering the structure with the garbage collector via the help of cryptography.io's `backend._ffi.gc()` function.

hazmat/backends/openssl/backend.py#L1395

Finding	**Missing Checks Might Lead to Incorrect Flow or Denial of Services**
Risk	Low Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-nucypher-pyumbral-005
Category	Data Validation
Location	• pre.py • point.py • dem.py
Impact	Maliciously crafted user inputs will give out incorrect results or raise exceptions. If these exceptions are not caught it could lead to denial-of-service attacks. For example, a long-lived node that would use the pyUmbral library without a try-catch paradigm could easily be crashed by malformed public keys and ciphertexts.
Description	pyUmbral public and private functions mostly assume that given parameters are correctly formed. This is not always the case, and simple checks should be implemented to prevent accidental or malicious mis-use of the library's functions. As this issue was systemic the below list of examples might not be exhaustive: The `split_rekey()` function in `pre.py` does not correctly check for valid `threshold` and `N` values: `threshold` should always be smaller than `N`, both values should be strictly greater than `0`, etc. In `point.py`, the `from_bytes()` method does not check for the correct length of the given data before attempting to convert it into a valid Point representation. In dem.py, the `decrypt()` method does not check for a minimum valid length of the ciphertext before attempting to extract a nonce from it.
Recommendation	Implement input validation for all sensitive functions of pyUmbral and raise relevant exceptions accordingly. For example, check for correct length, valid format, logical relationship between parameters, etc.

Finding	**Wrong Calculation of Compressed Point Size**
Risk	Informational Impact: None, Exploitability: None
Identifier	NCC-nucypher-pyumbral-003
Category	Cryptography
Location	• point.py
Impact	The library would not work properly with some other curves.

Description pyUmbral uses the curve's key size (subgroup order) in several places in order to calculate the size of a curve point. A curve point's size is determined by the field size, not by the key size, hence this calculation can potentially give out a wrong size:

```
class Point(object):
    """
    Represents an OpenSSL EC_POINT except more Pythonic
    """

    @classmethod
    def get_size(cls, curve: ec.EllipticCurve=None):
        """
        Returns the size (in bytes) of a compressed Point given a curve.
        If no curve is provided, it uses the default curve.
        """
        curve = curve if curve is not None else default_curve()
        return get_curve_keysize_bytes(curve) + 1
```

This error is not directly apparent with pyUmbral's default curve secp256k1[13] for which the subgroup order has the same size (256 bits) as the underlying prime field. With other curves, this might not hold true (for example SECT233K1[14] has a different order and field size).

Recommendation Use the curve's field size instead of the order size in the computation of the `get_size()` function.

[13] https://en.bitcoin.it/wiki/Secp256k1
[14] http://www.secg.org/sec2-v2.pdf

Finding	**Lack of Enforcement for Supported Curves**
Risk	Informational Impact: None, Exploitability: None
Identifier	NCC-nucypher-pyumbral-006
Category	Cryptography
Location	config.py
Impact	Applications that use unsupported curves might encounter random decryption failure.

Description

pyUmbral's default curve is Secp256k1[15] but it can be configured to work with other elliptic curves supported by the cryptography.io library. Yet, pyUmbral's code only supports elliptic curves over prime fields with cofactor $h = 1$. This is mostly due to simplifications in the code. For example, the library's `unsafe_hash_to_point()` method skips the steps of the try-and-increment[16] algorithm that deals with $h > 1$. This can lead to a generator `U` (constructed from a hash of the string "NuCypherKMS/UmbralParameters/") being on a different subgroup that would collide with other functions of the library configured with the order of the main generator G.

It was observed that configured with the binary curve sect571r1,[17] the pyUmbral library mostly works but will sometimes produce unverifiable proofs for the `cfrags`. The fact that these curves are not expected to be supported by the library can be lost on its users since this is not mentioned by pyUmbral's documentation.

Recommendation

Create a list of supported curves (over prime fields with cofactor $h = 1$) and enforce it in `config.py`'s `set_default_curve()` method.

[15] https://en.bitcoin.it/wiki/Secp256k1
[16] https://tools.ietf.org/html/draft-sullivan-hash-to-curve-00#appendix-A
[17] http://www.secg.org/sec2-v2.pdf

Finding	**Underlying Cryptographic Library Hardcodes Wrong Curve Order Size**
Risk	Informational Impact: None, Exploitability: None
Identifier	NCC-nucypher-pyumbral-007
Category	Cryptography
Impact	No impact was found as pyUmbral does not support these misconfigured curves.

Description

pyUmbral can be configured with the different curves supported by the cryptography.io[18] library. For some of the library functionalities, the size of the order of the cyclic subgroup generated by the conventional generator point is used (for example to figure out the serialization size of a scalar value). This order is obtained directly from the cryptography.io library depending on what curve has been set by the user of the library (if no curve has been set, the default secp256k1[19] curve is used).

It was noticed during the audit of pyUmbral that these advertised values cannot always be relied upon; for example, sect571r1 has been standardized[20] with a generator spanning a subgroup of bit-size order 570, whereas the cryptography.io library defines[21] that value to be 571:

```
@utils.register_interface(EllipticCurve)
class SECT571R1(object):
    name = "sect571r1"
    key_size = 571
```

Fortunately, pyUmbral only supports curves with a cofactor $h = 1$ (as pointed out in finding NCC-nucypher-pyumbral-006 on the previous page), which are correctly configured in cryptography.io. Nonetheless, extra care should be taken when relying on that third-party cryptographic library.

Recommendation

As pointed out in finding NCC-nucypher-pyumbral-006 on the preceding page, implement a list of supported curves to avoid users of the library making use of non-valid curves.

Optionally, submit a pull request to fix these values in the cryptography.io library.

[18] https://www.cryptography.io
[19] https://en.bitcoin.it/wiki/Secp256k1
[20] http://www.secg.org/sec2-v2.pdf
[21] https://github.com/pyca/cryptography/blob/17c8f126c7c7d5ce886112a6e924277a7b203f25/src/cryptography/hazmat/primitives/asymmetric/ec.py#L138

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical Implies an immediate, easily accessible threat of total compromise.

High Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.

Medium A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.

Low Implies a relatively minor threat to the application.

Informational No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.

Medium Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.

Low Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Medium Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.

Low Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix B: Fuzzing the pyUmbral Library

A negligible amount of time was spent fuzzing the library as manual code review was deemed more useful for the duration of the audit. Nonetheless, an unexpected exception was found via the following scripts. If derived and used to fuzz other public functions of the library, these scripts might prove useful for NuCypher in the future. The source code is in this document.

The following Python script can be used to create a corpus for the fuzzing script:

```
from umbral import pre, keys, config

def main():
        # init
        config.set_default_curve()

        # Generate umbral keys for Alice.
        alices_private_key = keys.UmbralPrivateKey.gen_key()
        alices_public_key = alices_private_key.get_pubkey()
        # Generate umbral keys for Bob.
        bobs_private_key = keys.UmbralPrivateKey.gen_key()
        bobs_public_key = bobs_private_key.get_pubkey()

        with open("fuzz/setup/1_bob_private_key", "bw") as ff:
                ff.write(bobs_private_key.to_bytes())

        with open("fuzz/setup/1_alice_public_key", "bw") as ff:
                ff.write(bytes(alices_public_key))

        # Encrypt data with Alice's public key.
        ciphertext, capsule = pre.encrypt(alices_public_key, b'Proxy Re-encryption is cool!')
        print("capsule size {d}", len(capsule.to_bytes()))

        with open("fuzz/setup/1_ciphertext", "bw") as ff:
                ff.write(bytes(ciphertext))

        # Alice generates split re-encryption keys for Bob with "M of N".
        kfrags = pre.split_rekey(alices_private_key, bobs_public_key, 2, 5)
        print("kfrag size {d}", len(kfrags[0].to_bytes()))

        # Create input
        input_frags = bytes(capsule) + bytes().join(map(bytes, kfrags))

        with open("fuzz/corpus/1_kfrags", "bw") as ff:
                ff.write(input_frags)

if __name__ == "__main__":
        main()
```

The following Python script can be used in conjunction with python-afl[22] to fuzz specific functions of the pyUmbral library:

```
from umbral import (
        pre, keys, config, fragments
)
from cryptography import exceptions
import sys
import afl
from io import BytesIO
import os

# init pyUmbral
config.set_default_curve()

# setup
with open("fuzz/setup/1_bob_private_key", "br") as ff:
        bobs_private_key = keys.UmbralPrivateKey.from_bytes(ff.read())

with open("fuzz/setup/1_alice_public_key", "br") as ff:
        alices_public_key = keys.UmbralPublicKey.from_bytes(ff.read())

with open("fuzz/setup/1_ciphertext", "br") as ff:
        ciphertext = ff.read()

# persistent mode
while afl.loop(50000):
  # do the magic
  try:
    data = BytesIO(sys.stdin.buffer.read())
    capsule = pre.Capsule.from_bytes(data.read(98))
    for _ in range(5):
            kfrag = fragments.KFrag.from_bytes(data.read(227))
            cfrag = pre.reencrypt(kfrag, capsule)
            capsule.attach_cfrag(cfrag)
    pre.decrypt(ciphertext, capsule, bobs_private_key, alices_public_key)
  except ValueError:
    pass
  except pre.UmbralCorrectnessError:
    pass
  except exceptions.InternalError:
    pass

os._exit(0)
```

This fuzzing script has room for improvement. Optimizing it to make it faster would benefit its efficiency at finding bugs.

[22] https://github.com/jwilk/python-afl