NuCypher

Security Assessment

August 27th, 2018

Prepared For:
MacLane Wilkison | *NuCypher*
maclane@nucypher.com

Prepared By:
JP Smith | *Trail of Bits*
jp@trailofbits.com

Ben Perez | *Trail of Bits*
benjamin.perez@trailofbits.com

Changelog:

June 15, 2018:	Initial report delivered
August 27, 2018:	Updated for public release

Executive Summary

From May 10 to June 29, NuCypher engaged with Trail of Bits to review the security of NuCypher's blockchain platform and the PyUmbral proxy re-encryption library. Trail of Bits conducted this assessment over the course of twelve person-weeks with two engineers.

The first two weeks were spent examining the PyUmbral library for cryptographic flaws. The third week was spent reviewing the smart contracts in NuCypher KMS. The final two weeks comprised of looking at collusion and DoS-based attacks on the NuCypher network.

The PyUmbral library is high quality, overall. The discovered issues are primarily due to the fact that the library is curve-agnostic, allowing users to choose their own curves instead of forcing the use of curves known to be secure. Defining PyUmbral exclusively over one curve, such as secp256k1, would be a substantial improvement.

NuCypher KMS contained a variety of high-severity issues. As of June 29th, malicious miners in the NuCypher network can mint free money by outputting random numbers instead of valid re-encryption keys. Furthermore, the network's lack of an anonymization mechanism could lead to collusion-based attacks that would result in the compromise of users' private keys. Finally, NuCypher KMS contained several medium-severity issues stemming from unimplemented functionality and lack of input validation.

To protect users of their network from impersonation, NuCypher must fix the variety of data-validation and signature issues outlined in this report. Furthermore, as it currently stands, users are not anonymous, which can lead to their private keys being leaked by malicious nodes. This issue must be fixed before users have a financial stake in the network. Finally, the overall robustness of the network is weakened by the fact that malicious miners cannot be challenged or detected, allowing them to flood the network with fake re-encryption keys and getting paid for their malicious behavior. Creating a scheme to prevent this is non-trivial, but is absolutely essential to a functional NuCypher product.

Engagement Goals & Coverage

NuCypher sought to assess the safety of their proxy re-encryption library, PyUmbral, against cryptanalytic attacks. They also wanted to ensure that their key management system on the Ethereum blockchain was free of Solidity bugs, network vulnerabilities, and collusion-based attacks.

During the engagement, we conducted a thorough audit of the PyUmbral library, examining its openSSL use and writing tests to ensure its serialization routines were correctly implemented. Also, we manually reviewed the cryptography in NuCypher KMS, as well as the server code, policy management system, and Solidity contracts. We began writing automated tests with Echidna for the blockchain component but were unable to complete this activity due to time constraints.

PyUmbral

- ✓ Manually review all cryptographic code
- ✓ Read and review specifications for design flaws
- ✓ Check all serialization and deserialization code for possible risks
- ✓ Automate serialization testing with random inputs
- ❏ Check all call sites of the API for dangerous usage

Blockchain

- ✓ Perform static analysis on all Solidity code using Slither
- ✓ Manually review all Solidity code
- ✓ Manually review all Python code
- ✓ Run automated tests for ERC20 vulnerabilities
- ❏ Write property tests and fuzz UserEscrow, MinerEscrow, Issuer, and PolicyManager contracts
- ✓ Explore collusion-based attacks

Crypto

- ✓ Manually review all cryptographic code
- ✓ Check all call sites of the crypto API for dangerous usage
- ✓ Check all signature usage
- ✓ Check all serialization and deserialization code for possible risks
- ✓ Check all X.509 certificate code for standards compliance and safety
- ✓ Ensure that the only randomness available via API is cryptographically secure, even in pathological operating conditions
- ✓ Analyze the consequences of collusion attacks as they specifically pertain to cryptographic code

Keystore

- ✓ Manually review all Keystore code for logic errors
- ✓ Check call sites of the crypto API for dangerous usage

Network

- ✓ Review server code for code execution or resource exhaustion bugs
- ✓ Review server code for logic bugs
- ✓ Consider denial-of-service scenarios

Policy

- ✓ Review model code for logic bugs with security implications

Misc

- ✓ Review Bytestring splitter code for possible functionality issues
- ✓ Review Constant Sorrow for vulnerabilities

Project Dashboard

Application Summary

Name	NuCypher, PyUmbral
Type	Proxy re-encryption system
Platform	Python, EVM

Engagement Summary

Dates	May 10 - June 29, 2018
Method	Whitebox
Consultants Engaged	2
Level of Effort	Full-time

Vulnerability Summary

Total High Severity Issues	6	■■■■■■
Total Medium Severity Issues	4	■■■■
Total Low Severity Issues	4	■■■■
Total Informational Severity Issues	0	
Total	14	

Category Breakdown

Access Controls	1	■
Configuration	1	■
Cryptography	2	■■
Data Exposure	1	■
Data Validation	1	■
Denial of Service	7	■■■■■■■
Timing	1	■
Total	14	

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Add a salt parameter to the KDF**. The utils.py library provides a KDF but sets salt to none. All KDF applications must be salted to ensure independence across different uses of the hash function.

❑ **Define which curve the system is using in a single location.** Many functions in PyUmbral take a curve parameter but default to a global configuration if the curve is not specified. This allows users to generate private keys over one curve but generate public keys over another, leading to an inconsistent state.

❑ **Only allow users a small whitelist of curves to select from.** PyUmbral lets users choose arbitrary–-even insecure–-curves for its cryptographic functionality.

❑ **Validate signatures in `signature.py`.** The signature class does not validate incoming signatures, allowing for the construction of signatures that don't depend on the private key used to sign.

❑ **Mitigate known ERC20 race conditions.** Strict adherence to the ERC20 standard can lead to situations where attackers can reorder transactions to take more tokens than they were approved for. This should be mitigated in either the `approve` function or any client used to interact with the system.

❑ **Require an access token for API usage and limit the rate of requests.** The server code does not limit the rate at which its clients can make requests, leaving it vulnerable to DoS attacks.

❑ **Add application logic for reverting data storage to a previous state.** Many functions write to the database with no validation of input data. Currently, there is no way to revert the database if it enters an undesirable state.

❑ **Use file permissions to prevent unprivileged users from accessing the database.** The keystore database is stored unencrypted. No permission-based access controls are present.

❑ **Refactor the policy class to require a unique nonce and signature per policy.** Lack of kfrag validation and policy revocation leaves an attacker open to replay attacks.

❑ **Validate signatures on work orders.** The work order submission process happens without a blockchain component or replay attack prevention.

❑ **Require Ursula to sign responses to work orders.** Ursula never signs responses to work orders, allowing anyone to impersonate her to a given Bob.

Long Term

❑ **Ensure that cryptographic primitives are used in accordance with their specifications.** Even if it is unlikely cryptographic functions will be used outside of a very specific context, they should always adhere to their specification.

❑ **Build a system so that NuCypher can detect and penalize malicious nodes.** The NuCypher system cannot currently detect malicious miners, leading to a potential DoS attack or miners minting free money without adding value to the network.

❑ **Design APIs resistant to front-running attacks.** Due to the public nature of all data and transactions on the blockchain, malicious miners can and will front-run valuable transactions.

❑ **Make sure any exposed cryptographic APIs are resistant to misuse.** Users should not be trusted to make sound cryptographic choices, especially in curve selection.

❑ **Devise a system of pseudonyms for all users.** Users of the NuCypher network are not anonymized. Due to the mathematical properties of threshold schemes, malicious nodes can collude to learn a specific user's private key.

❑ **Adopt layered defenses for network and compute resources.** Assume the network will fall prey to DoS attacks and accidental abuse from users.

❑ **Assume attackers can observe, modify, and replay traffic on the wire.** Cryptographic systems must be designed to prevent attackers on the wire from reusing old messages.

❑ **Sign and check signatures for communications with security implications.** All cryptographically important information should be signed and then validated to prevent impersonation and replay attacks.

Findings Summary

#	Title	Type	Severity
1	Unsalted HKDF in utils.py	Cryptography	Low
2	Multiple issues related to curve specification	Configuration	High
3	Multiple issues related to parametrization over arbitrary curves	Cryptography	High
4	Insufficient validation of signatures	Data Validation	High
5	Network cannot detect malicious nodes	Denial of Service	High
6	NuCypherKMSToken may be vulnerable to transaction reordering attacks	Timing	Medium
7	Server implements no rate-limiting functionality	Denial of Service	High
8	Database has no snapshot and rollback functionality	Denial of Service	Medium
9	Lack of anonymity allows collusion-based attacks	Data Exposure	High
10	Database has no access controls	Access Controls	Low
11	`ProxyRESTServer.set_policy` can be used to invalidate policy arrangements	Denial of Service	Medium
12	Several issues related to policy issuance	Denial of Service	Medium
13	Work orders have no protection from replay attacks	Denial of Service	Low
14	Impersonating Ursula is trivial	Denial of Service	Low

1. Unsalted HKDF in utils.py

Severity: Low
Type: Cryptography
Target: `pyUmbral/umbral/utils.py`

Difficulty: High
Finding ID: TOB-NCY-001

Description
`utils.py` in the Umbral codebase provides a function, `kdf`, which wraps an HKDF from cryptography.io but sets `salt` to `None`. RFC 5869, which specifies HKDF usage, recommends salt be used when possible (§ 3.1), as it greatly increases robustness.

```python
def kdf(ecpoint, key_length):
    data = ecpoint.to_bytes(is_compressed=True)

    return HKDF(
        algorithm=hashes.BLAKE2b(64),
        length=key_length,
        salt=None,
        info=None,
        backend=default_backend()
    ).derive(data)
```

Fig. 1: the `kdf` function

Exploit Scenario
An attacker attempts to derive some input of `kdf` from its output via brute force. As no salt is used, they can brute force all applications of `kdf` at once.

Recommendation
Add a salt parameter to kdf (even one that can take a value of `None`) and use it in accordance with RFC 5869 § 3.1.

Going forwards, ensure all cryptographic primitives are used as their specifications dictate.

References
- HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

2. Multiple issues related to curve specification

Severity: High
Difficulty: Undetermined
Type: Configuration
Finding ID: TOB-NCY-002
Target: Several

Description
A number of operations and object constructors take an optional curve parameter. If a parameter isn't provided, the operations will inherit from a global configuration. This can be confusing, allows for inconsistency, and leads to some functionality bugs. Notably:

- When deserializing a capsule, the logic to determine its activation status assumes a keysize of 32, which is not true for every curve. Specifically, it assumes 6 sections of size key_size with 5 additional bytes will serialize to exactly 197 bytes.
- In keys.py, the `UmbralPrivKey` class has a get_pubkey method, which returns the `UmbralPubKey` corresponding to the private key. Both classes carry information about the curve being used. However, the get_pubkey method does not take curve information. It always returns an `UmbralPubKey` on the default curve.

```
# CurveBNs are the keysize in bytes, Points are compressed and the
# keysize + 1 bytes long.
if len(capsule_bytes) == 197:
    e = Point.from_bytes(capsule_buff.read(key_size + 1), curve)
    v = Point.from_bytes(capsule_buff.read(key_size + 1), curve)
    sig = CurveBN.from_bytes(capsule_buff.read(key_size), curve)
    e_prime = Point.from_bytes(capsule_buff.read(key_size + 1), curve)
    v_prime = Point.from_bytes(capsule_buff.read(key_size + 1), curve)
    ni = Point.from_bytes(capsule_buff.read(key_size + 1), curve)
```

Fig. 2: `CurveBN.from_bytes` (excerpt)

```
def get_pubkey(self):
    """
    Calculates and returns the public key of the private key.
    """
    return UmbralPublicKey(self.bn_key * self.params.g)
```

Fig. 3: `UmbralPrivKey.get_pubkey`

Exploit Scenario
An implementer instantiates a private key over SECP521R1.
She doesn't update the default curve from SECP256K1.
She attempts to generate a public key.
Now she cannot decrypt any incoming messages.

Recommendation
Replace `197` in the above snippet with `6 * key_size + 5`. Force `get_pubkey` to use the same curve as the private key it's being generated from. Most importantly, define which curve the system is using in a single, canonical location.

Whenever possible, make sure illegal states are unrepresentable.

3. Multiple issues related to parametrization over arbitrary curves

Severity: High
Type: Cryptography
Target: pyUmbral

Difficulty: High
Finding ID: TOB-NCY-003

Description
Umbral: A Threshold Proxy Re-encryption Scheme doesn't restrict the curve used in its implementation. It just requires that the curve be defined over a group of prime order (the actual phrasing is somewhat ambiguous. This is our interpretation.) The implementation implicitly limits this to any cryptography.io elliptic curve object supported by OpenSSL, but that may still be overly broad.

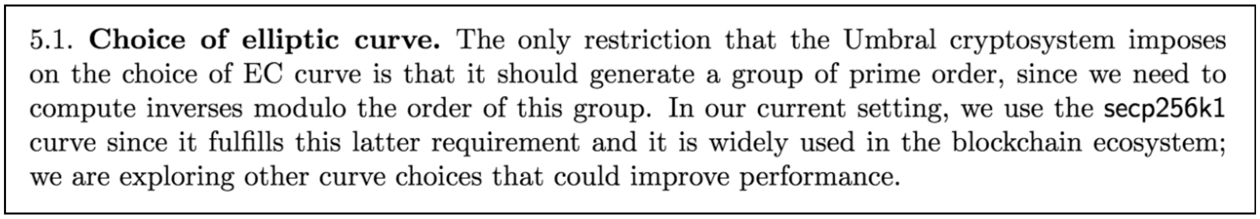
5.1. **Choice of elliptic curve.** The only restriction that the Umbral cryptosystem imposes on the choice of EC curve is that it should generate a group of prime order, since we need to compute inverses modulo the order of this group. In our current setting, we use the `secp256k1` curve since it fulfills this latter requirement and it is widely used in the blockchain ecosystem; we are exploring other curve choices that could improve performance.

Fig. 4: An excerpt from the paper

Furthermore, this parametrization causes undefined behavior in pyUmbral. In `openssl.py`, the `_get_new_EC_POINT` takes an optional `ec_group` and `curve_nid`. If it is passed a `curve_nid` it sets `ec_group` to the group associated with `curve_nid`. However, if both parameters are passed to the function and are in disagreement, `ec_group` will default to the group associated with curve_nid.

If a curve with a cofactor other than one is ever used, several parts of the application will require reengineering to, for example, prevent small subgroup attacks by validating the order of public keys.

```
def _get_new_EC_POINT(ec_group=None, curve_nid: int=None):
    """
    Returns a new and initialized OpenSSL EC_POINT given the group of a curve.
    If curve_nid is provided, it retrieves the group from the curve provided.
    """
    if curve_nid:
        ec_group = _get_ec_group_by_curve_nid(curve_nid)
    elif not ec_group:
        raise ValueError("No group provided.")
```

Fig. 5: `_get_new_EC_POINT`

Exploit Scenario
Serialization code doesn't take into account the variety of curves over which it can be called and contains hard-to-reach logic bugs. A PyUmbral implementer chooses an unusual elliptic curve and cannot correctly deserialize incoming messages.

Recommendation
Choose one curve over which Umbral is canonically defined, or at least implemented. If that's infeasible, a small, well-tested whitelist can suffice. Rewrite 5.1 from *Umbral: A Threshold Proxy Re-encryption Scheme* to use slightly more precise terminology, e.g. "Any curve over a field of prime order."

Ensure that specifications give enough detail to prevent potential misuse. While generality is appropriate in some circumstances, it cannot come at the cost of security.

References
- Weak Curves In Elliptic Curve Cryptography
- A state-of-the-art Diffie-Hellman function

4. Insufficient validation of signatures

Severity: High
Difficulty: Undetermined
Type: Data Validation
Finding ID: TOB-NCY-004
Target: `nucypher/crypto/signature.py`

Description
The `Signature` class implements no verification for the r and s parameters, allowing for the construction of signatures that don't depend on the private key used to sign. This can pose a significant risk if these signatures are ever checked without also checking the validity of the r and s parameters.

Right now the impact of this is mitigated because every curve in cryptography.io will ultimately use OpenSSL's `ossl_ecdsa_verify_sig`, which does parameter verification. However, `ossl_ecdsa_verify_sig` is not guaranteed to be used for every OpenSSL curve. Furthermore, this validation should be done as soon as a signature is initialized to prevent the unvalidated signature from being used in another context.

```
def __init__(self, r: int, s: int):
    #  TODO: Sanity check for proper r and s.
    self.r = r
    self.s = s
```

Fig. 6: `Signature.__init__`

Exploit Scenario
A client chooses to implement Curve1174 in OpenSSL themselves and use it with NuCypher-KMS. They forget to check for the case where r is equal to 0 modulo the curve order when verifying signatures. An adversary can now craft valid signatures for any message from any key.

A signature is created with r=0 and stored in NuCypherKMS. It's verified outside the system by software that overlooks parameter validation. As before, an attacker can now craft signatures that verify as if from any private key.

Recommendation
Make r and s `CurveBN`s, reusing existing validation there.

An unbounded integer is rarely the right type for elements used in cryptographic computation.

References

- The Elliptic Curve Digital Signature Algorithm (ECDSA)

5. Network cannot detect malicious nodes

Severity: High
Difficulty: Low
Type: Denial of Service
Finding ID: TOB-NCY-005
Target: `nucypher/blockchain/`

Description
NuCypherKMS does not currently have a way to establish whether a miner is performing valid re-encryptions. At best, this severely reduces the reliability of the network. At worst, it could lead to malicious nodes mining free money.

Exploit Scenario
A miner runs a node that simply outputs random numbers. Their cfrags do not directly cause any failures in the network, since re-encryption uses a threshold scheme. However, this miner will be able to mint free money until they are manually removed from the network. Even after being removed, there is nothing stopping this malicious user from creating another node.

The lack of validation also could lead to a DoS attack. There is nothing preventing a flood of malicious nodes from rendering the NuCypherKMS useless. This attack may require a large amount of upfront capital, since all miners must put up collateral in the MinersEscrow contract, but such a scenario is not unimaginable. A group of semi-affluent actors could collude to short NuCypher tokens. Even if the network is running a decent number of nodes, say 9,000, it would only take 3,000 fake nodes to ensure that re-encryption fails 25% of the time when k/n = 3/4.

Recommendation
We propose the following system for incentivizing correct and safe proxy re-encryption coordinated via the Ethereum blockchain.

Work is coordinated via a “dispatcher” contract. Users may interact with this contract in several capacities. First, let’s consider the case in which someone wants to perform re-encryption in exchange for NU tokens.

We’ll call this actor Ursula. We wish to incentivize Ursula to reply to every request for re-encryption with a correct answer (as opposed to either replying with an incorrect answer or simply failing to reply, which, from a utilitarian perspective, are equivalent). Ursula must commit to performing re-encryption for some time to be useful as a participant in the Umbral cryptosystem. We cannot simply pay per job. As a result, we propose the following:

Ursula makes a commitment to being available for some period of time, during which she’ll reply to some portion of requests with a correct response. When she makes this commitment, she stakes some amount of tokens. The dispatcher assembles multiple

Ursulas into cohorts based on the assumed rate at which other Ursulas drop out, the commitments of the cohort, and the Umbral threshold parameter.

When Alice requires re-encryption services, she pays the dispatcher for access to some cohort of Ursulas for some time. She can then make requests (at some maximum rate) for re-encryption consisting of a set of kfrags and a public key. Each request has an associated nonce. Once a request is published, the chosen Ursulas form a Plasma-like sidechain. This is to prevent exorbitant gas costs and latency during verification. On the sidechain, Ursulas provide timestamped proofs to a verifier contract that they did the claimed work. Once the sidechain has reached consensus and the contractual obligation has finished, the sidechain syncs with the main Ethereum blockchain and the Ursulas are paid.

6. NuCypherKMSToken may be vulnerable to transaction reordering attacks

Severity: Medium
Difficulty: High
Type: Timing
Finding ID: TOB-NCY-006
Target: `nucypher/blockchain/NuCypherKMSToken.sol`

Description
There's a well-known race condition in the ERC20 standard where an attacker with the ability to reorder transactions can use `transferFrom` to take more tokens than they were allotted with `approve`. While mitigations for this vulnerability exist, NuCypherKMSToken does not implement them. In addition, a very similar vulnerability exists in `decreaseApproval`.

`approve` sets the allowance of the given user to some number regardless of their current allowance. This allows front-running attacks, where an attacker can use a victim's tokens in unexpected ways by preempting their calls to `approve` with a call to `transfer`. `decreaseApproval` has a similar but less serious problem. It will behave identically if an attacker can insert a `transfer` between its origination and resolution, which could conceivably deceive a user.

Exploit Scenario
Alice approves Bob to transfer up to 100 of her tokens. Later, she decides to approve Bob for 150 tokens. Bob is watching for transactions from Alice that call `approve`. When he observes one, he introduces another transaction with a lower timestamp that transfers 100 tokens to his own address. After Alice's transaction finalizes, he transfers 150 tokens to himself for a total of 250 tokens extracted (instead of the 150 intended).

Alice approves Bob to transfer up to 100 of her tokens. Later, she wants to make those tokens available to Carol. Alice calls `decreaseApproval` to set Bob's allowance to 0. As before, Bob notices Alice's `decreaseApproval` and preempts it with a transfer of those 100 tokens. Alice's `decreaseApproval` transaction still appears to work, so she approves Carol to use 100 tokens. Carol then transfers them out. 200 tokens are transferred out of Alice's account (instead of the 100 intended).

Recommendation
Either in the client or the NuCypherKMSToken contract, ensure you cannot `approve` someone for a nonzero number of tokens if they aren't currently approved for zero tokens (N.B.: doing this in the contract is technically an ERC20 standard violation). Change `decreaseApproval` to return the actual number of tokens the allowance it's modifying decreased by

Always assume malicious entities can front-run transactions for financial gain.

References

- ERC20 API: An Attack Vector on Approve/TransferFrom Methods
- Method decreaseApproval in StandardToken.sol is unsafe
- Implementation of ‘approve’ method violates ERC20 standard

7. Server implements no rate-limiting functionality

Severity: High
Difficulty: Low
Type: Denial of Service
Finding ID: TOB-NCY-007
Target: `nucypher/network`

Description
The NuCypher server code features no functionality that limits the rate at which its clients can make requests. As many of its endpoints modify database state or perform cryptographic computation, this lack of functionality makes it exceedingly easy for a client to exhaust the server's resources.

The risk from this issue increases dramatically from the fact that this code is designed to be executed as part of a "testnet" so that developers can build applications against NuCypher's API. Developing code can easily have bugs that lead to accidental extreme API usage.

Exploit Scenario
A developer is integrating their app with the NuCypher system and wants to create one policy per file in some directory. Due to a simple bug each time a re-encryption is performed, a logfile is produced in the same directory, meaning policies are created in an infinite loop. The databases of every consumer of policies from this developer fill up with junk and can no longer contribute to the network.

Recommendation
Require an access token for API usage. Limit the requests made per access token per unit of time.

Always assume developers will grossly misuse APIs and develop accordingly.

8. Database has no snapshot and rollback functionality

Severity: Medium
Difficulty: Undetermined
Type: Denial of Service
Finding ID: TOB-NCY-008
Target: `nucypher/keystore`

Description
Right now, many functions write to the database with no validation of input data. Especially in concert with TOB-NCY-007, we believe it is likely some databases become large to the point of being unwieldy and filled with mostly useless data. Short of clearing the database and starting over, NuCypher offers no solution for reverting these changes.

Exploit Scenario
Due to poor termination logic in a loop, some developer accidentally publishes billions of keys to the network. As keys are identified with a unique integer, all key space in each DB is exhausted. State was completely fine 24 hours ago, but the only way to return each database to functionality is either to discard all previously known keys or handwrite a SQL query to delete all records matching some pattern (bugs in which could cause serious problems).

Recommendation
Add application-level logic for reverting data storage to a previous state.

Assume states will become invalid, and add both aggressive validation logic and ways to recover from inconsistency.

9. Lack of anonymity allows collusion-based attacks

Severity: High
Difficulty: Medium
Type: Data Exposure
Finding ID: TOB-NCY-009
Target: `nucypher/keystore`

Description
Currently the NuCypher system does not anonymize its users. Therefore re-encryption nodes can acquire information about both the senders and recipients of the data they are re-encrypting. This allows for attacks where Ursulas collude with Bob to learn Alice's private key.

Exploit Scenario
Alice runs a Netflix-like service which Bob subscribes to. Bob wishes to stop paying for the service while still being allowed to stream media. He creates the threshold number of Ursula nodes, which proceed to enter into a Policy with Alice for Bob's re-encrypted key. Since Bob and the Ursulas are owned by the same person, they can collude to learn Alice's private key. Bob can stop paying for Alice's service while still accessing its data.

Recommendation
Devise a system where users are all pseudonymous. This could be done by the booker in the scheme outlined in TOB-NCY-006, since it would be able to mask each job's origin. Once the policy has been created, its involved parties become de-anonymized. The fact that all parties remain anonymous during job selection prevents many collusion attacks.

Since Umbral leaks Alice's private key if the Ursulas and Bob collude, collusion attacks must be taken very seriously.

10. Database has no access controls

Severity: Low
Difficulty: High
Type: Access Controls
Finding ID: TOB-NCY-010
Target: `nucypher/keystore`

Description
The keystore sqlite database is stored unencrypted. No permission-based access controls are present. An attacker with even temporary filesystem access can add, modify, or delete entries at will.

Exploit Scenario
Due to an unpatched operating system, An attacker briefly obtains a shell on several NuCypher nodes. He uses this access to remove some user's keys from all nodes' databases. This user can no longer make use of the NuCypher network.

Recommendation
Explicitly ensure that file permissions on the database prevent unprivileged users from accessing it, and use sqlite's encryption facilities to ensure it's stored encrypted.

Whenever possible, adopt layered "defense in depth" strategies to mitigate partial compromises.

References
- How To Compile And Use the SQLite Encryption Extension (SEE)

11. ProxyRESTServer.set_policy can be used to invalidate policy arrangements

Severity: Medium
Difficulty: Low
Type: Denial of Service
Finding ID: TOB-NCY-011
Target: `nucypher/network/server.py`

Description
`set_policy` decrypts an AEAD-encrypted message containing a kfrag to attach to a given policy arrangement. However, due to several individual small bugs and questionable design decisions, an attacker with network access can make policy arrangements held by any node effectively useless. Specifically, we note the following issues:

- `verify_from` returns a decrypted message even if no valid signature is provided
- `set_policy` does not check whether `verify_from` succeeded
- `attach_kfrag_to_saved_arrangement` does not check if a kfrag is attached to some arrangement already

Exploit Scenario
An attacker observes a user's network traffic, and records a message she sends to `set_policy`. They then doctor the message such that the kfrag is different, and the message no longer contains a signature. `set_policy` commits the updated signature to the DB, preventing re-encryption.

Recommendation
Make sure `verify_from` performs verification. Don't let it fail silently anywhere it is called. Separate logic for creating and updating kfrags.

Assume attackers can observe, modify, and replay traffic on the wire for malicious ends.

12. Several issues related to policy issuance

Severity: Medium
Type: Denial of Service
Target: `nucypher`

Difficulty: Low
Finding ID: TOB-NCY-012

Description
The policy system presents the following issues:

- The m parameter is never signed
- Alice's signature is optional, and not included when creating a policy with from_alice
- The signed portion of the policy is just kfrags, which are left in the keystore after the policy expires
- Kfrags are never validated
- Ursula doesn't respond to arrangements made
- Alice doesn't have to publish arrangements on the blockchain, as Ursula doesn't validate against them
- There is no policy revocation mechanism
- The policy and arrangement code offers no protection from replay attacks

As a result, there are several ways to craft policies that appear to be from Alice and convince Ursulas to execute them, frequently with different parameters than the original policy intended. Alice cannot tell if this is taking place, and Ursula cannot distinguish legitimate policies from fake ones.

Exploit Scenario
An attacker observes some of Alice's policies over the wire. They replay Alice's policies with different ms and different kfrags to all Ursulas without publishing anything on the blockchain. Ursula realizes Alice isn't playing fair (she's issuing policies that cannot be fulfilled and aren't paid for) and refuses to accept further policies from her (possible due to TOB-NCY-009). Alice can no longer use the Nucypher system, but cannot tell why (other than Bob's work orders never being fulfilled).

Alternatively, a developer accidentally issues dozens of bad policies without publishing on the blockchain, and cannot revoke them. Ursula, as before, now refuses to do business with that Alice.

Recommendation
Refactor the policy class to require a nonce and signature per policy. Require Ursula to track nonces and ensure they aren't reused. Implement a fix for TOB-NCY-009.

Always consider the ways in which an attacker can observe and replay traffic to compromise a system.

13. Work orders have no protection from replay attacks

Severity: Low
Difficulty: Low
Type: Denial of Service
Finding ID: TOB-NCY-013
Target: `nucypher/keystore/keystore.py`

Description
The work order submission process happens with no blockchain component and no protection against replay attacks. An attacker that can observe one valid work order can replay it indefinitely and Ursulas will honor it. In addition, the signature field is never validated.

Exploit Scenario
An attacker observes Bob's network traffic, records a work order submission to some Ursula, and replays it. Data to all other Ursulas may be manipulated as well, straining the network's resources.

Recommendation
Validate Bob's signature on work orders. Add a nonce, which cannot be repeated.

Always consider the ways in which an attacker can observe and replay traffic to compromise a system.

14. Ursula's responses are unauthenticated

Severity: Low
Difficulty: Low
Type: Denial of Service
Finding ID: TOB-NCY-014
Target: `nucypher/characters.py`

Description
Ursula never signs her response to work orders. Anyone can pretend to be her and respond to a given Bob.

Exploit Scenario
An attacker Eve observes that Bob submits a request to some Ursula that Eve wants to sabotage, then spams him with grossly incorrect results. Ursula is then not paid for any valid re-encryption she may perform.

Recommendation
Require Ursula to sign responses to work orders. Require Bob to validate Ursula's signed response.

If communications have security implications, they should be signed and that signature should be validated.

Appendix A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
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 protecting the privacy or integrity of 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
Documentation	Related to documentation accuracy
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
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important

Medium	Individual user’s information is at risk, exploitation would be bad for client’s reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client’s reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

Appendix B: Serialization testing code

This code tests the serialization functionality of pyUmbral. It randomly generates cryptographic objects, then serializes them, deserializes the resulting data, and ensures no changes have occured. This code requires hypothesis, and can be run with `pipenv run pytest roundtrip.py`

```
from cryptography.hazmat.backends.openssl import backend
from cryptography.hazmat.primitives.asymmetric.ec import SECP224R1

from hypothesis import HealthCheck, given, settings, unlimited
from hypothesis.strategies import binary, booleans, integers, lists, text

from umbral.config import set_default_curve, default_curve
from umbral.curvebn import CurveBN
from umbral.fragments import CorrectnessProof, KFrag
from umbral.keys import UmbralPrivateKey, UmbralPublicKey
from umbral.params import UmbralParameters
from umbral.point import Point, unsafe_hash_to_point
from umbral.pre import Capsule
from umbral.openssl import _get_ec_order_by_curve_nid
from umbral.utils import get_curve_keysize_bytes

# crypto constants

set_default_curve(SECP224R1)
curve = default_curve()
params = UmbralParameters(curve)
ks = get_curve_keysize_bytes(curve)

# generators

bns = integers(min_value=1, max_value=backend._bn_to_int(params.order)).map(
    lambda x: CurveBN.from_int(x))

points = binary(min_size=1).map(
    lambda x: unsafe_hash_to_point(x, label=b'hypothesis', params=params))

# utility

def assert_kfrag_eq(k0, k1):
    assert(all([ k0._id                    == k1._id
               , k0._bn_key                == k1._bn_key
               , k0._point_noninteractive  == k1._point_noninteractive
               , k0._point_commitment      == k1._point_commitment
               , k0._point_xcoord          == k1._point_xcoord
               , k0._bn_sig1               == k1._bn_sig1
               , k0._bn_sig2               == k1._bn_sig2
               ]))

def assert_cp_eq(c0, c1):
    assert(all([ c0._point_e2               == c1._point_e2
               , c0._point_v2               == c1._point_v2
               , c0._point_kfrag_commitment == c1._point_kfrag_commitment
               , c0._point_kfrag_pok        == c1._point_kfrag_pok
               , c0._bn_kfrag_sig1          == c1._bn_kfrag_sig1
               , c0._bn_kfrag_sig2          == c1._bn_kfrag_sig2
               , c0._bn_sig                 == c1._bn_sig
               , c0.metadata                == c1.metadata
               ]))

# tests

@given(bns)
@settings(max_examples=10000, timeout=unlimited)
def test_bn_roundtrip(bn):
    assert(bn == CurveBN.from_bytes(bn.to_bytes()))

@given(points, booleans())
@settings(max_examples=10000, timeout=unlimited)
def test_point_roundtrip(p, c):
    assert(p == Point.from_bytes(p.to_bytes(is_compressed=c)))

@given(binary(min_size=ks, max_size=ks), bns, points, points, points, bns,
bns)
@settings(max_examples=10000, timeout=unlimited)
def test_kfrag_roundtrip(d, b0, p0, p1, p2, b1, b2):
    k = KFrag(d, b0, p0, p1, p2, b1, b2)
    assert_kfrag_eq(k, KFrag.from_bytes(k.to_bytes()))

@given(points, points, bns)
@settings(max_examples=10000, timeout=unlimited)
def test_capsule_roundtrip_0(p0, p1, b):
    c = Capsule(p0, p1, b, None, None, None)
    assert(c == Capsule.from_bytes(c.to_bytes()))

@given(points, points, bns, points, points, points)
@settings(max_examples=10000, timeout=unlimited)
def test_capsule_roundtrip_1(p0, p1, b, p2, p3, p4):
    c = Capsule(p0, p1, b, p2, p3, p4)
    assert(c == Capsule.from_bytes(c.to_bytes()))

@given(points, points, points, points, bns, bns, bns)
@settings(max_examples=10000, timeout=unlimited)
def test_cp_roundtrip(p0, p1, p2, p3, b0, b1, b2):
    c = CorrectnessProof(p0, p1, p2, p3, b0, b1, b2)
    assert_cp_eq(c, CorrectnessProof.from_bytes(c.to_bytes()))

@given(points)
@settings(max_examples=10000, timeout=unlimited)
def test_pubkey_roundtrip(p):
    k = UmbralPublicKey(p)
    assert(k == UmbralPublicKey.from_bytes(k.to_bytes()))

@given(binary(min_size=1))
@settings(max_examples=100, timeout=unlimited,
suppress_health_check=[HealthCheck.hung_test])
def test_privkey_roundtrip(p):
    k = UmbralPrivateKey.gen_key()
    rt = UmbralPrivateKey.from_bytes(k.to_bytes(password=p), password=p)
    assert(k.get_pubkey() == rt.get_pubkey())
```

Appendix C. Code Quality Recommendations

PyUmbral

- `unsafe_hash_to_point` in PyUmbral doesn't work when a label isn't provided. If the provided `label` is `None` (or no label is provided), it uses a label of `[]`, but you can't concatenate lists and bytes.

NuCypher-KMS

- The provided Pipfile doesn't build `bytestring_splitter` or `constant_sorrow`, both of which are required for NuCypher-KMS to work.
- The `CAPSULE_LENGTH` constant in constants.py is only accurate for unactivated capsules (activated capsules are 197 bytes long). This propagates to `capsule_splitter` and thus `UmbralMessageKit`.
- The blockchain portion of the system implements staking, but is not ERC900 compatible. Adopting this interface would enable usage of standard contracts and testing tools designed for this model.
- NuCypherKMSToken doesn't allow transfers from a person to themselves, either via `transfer` or `transferFrom`. This can cause unexpected behavior when these functions are called as part of a larger function and unexpectedly revert.
- The constants `KFRAG_LENGTH` and `CFRAG_LENGTH_WITHOUT_PROOF`, defined in `Constants.py`, are neither initialized nor used anywhere else in the codebase
- The constant `_EXPECTED_LENGTH` defined in `models.py` is unused

Appendix D. Slither static analysis

Trail of Bits has included our Solidity static analyzer, Slither, with this report. Slither works on the Abstract Syntax Tree (AST) generated by the Solidity compiler and detects some of the most common smart contract security issues, including:

- The absence of a constructor
- The presence of unprotected functions
- Uninitialized variables
- Unused variables
- Functions declared as constant that change the state
- Deletion of a structure containing a mapping
- ... and many more.

Slither is an unsound static analyzer and may report false positives. The lack of proper support for inheritance and some object types (such as arrays) may lead to false positives.

In order to use Slither, simply launch the analysis on the Solidity file:

```
$ python /path/to/slither.py file.sol
```

Ensure that import dependencies and libraries, such as OpenZeppelin, can be found by the solc compiler in the same directory.