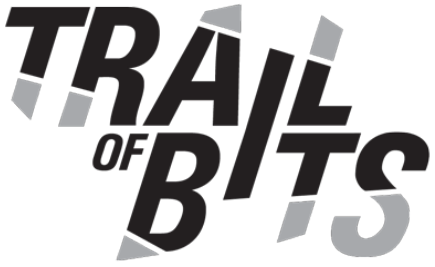

NuCypher

Security Assessment

February 26th, 2019

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

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

Will Song | *Trail of Bits*
will.song@trailofbits.com

Executive Summary

From February 11 to February 26, NuCypher engaged with Trail of Bits to review the security of NuCypher's slashing protocol, upgrade scheme, and multisig contract. Trail of Bits conducted this assessment over the course of four person-weeks with two engineers.

The first week was spent examining NuCypher's slashing protocol and the underlying Numerology library, which verifies correctness proofs produced by Ursula's. We found several code quality issues related to gas optimization, documented in Appendix B. Other than this, however, we found the quality of the slashing protocol to be extremely high.

The second week focused on the multisig contract and the upgrade protocol. The multisignature wallet appears to work as intended. It correctly manipulates its list of owners and checks for the precisely `_required` number of distinct, valid signatures before executing a transaction. There was a transaction replay issue related to `execute` that was a concern, but it is mitigated by including the transaction sender in the transaction hash. Attackers would need to acquire owners' private keys to produce a list of correct signatures with the attackers' own address.

NuCypher has chosen to use the proxy pattern for upgrading their smart contracts. We found one high severity issue, two medium severity issues, and three low severity issues in their implementation of this pattern. In general, the proxy pattern is extremely error-prone and requires a deep understanding of how the EVM and Solidity compiler lays out memory. Any error in implementing the proxy pattern will lead to catastrophic system failure. We do not recommend its use in production code. Even if NuCypher executes the proxy pattern flawlessly, updates to the Solidity compiler can alter storage layout in a way that breaks the pattern's safety. A strong alternative is migration-based upgrades, which are documented in this blog post.

Engagement Goals & Coverage

NuCypher sought to assess the safety of their slashing protocol against cryptographic- and blockchain-level attacks. In particular, they wanted to ensure that users performing erroneous proxy re-encryptions cannot produce correctness proofs that verify and vice versa. Furthermore, they wanted to ensure their multisig and upgrade strategies were robust and error-free.

Over the course of the audit, we examined the following smart contracts

- Dispatcher.sol
- Upgradeable.sol
- Numerology.sol
- SignatureVerifier.sol
- UmbralDeserializer.sol
- Issuer.sol
- MinersEscrow.sol
- MiningAdjucator.sol
- PolicyManager.sol
- UserEscrow.sol
- UserEscrowProxy.sol
- MultiSig.sol

Project Dashboard

Application Summary

Name	NuCypher, Numerology
Type	Proxy re-encryption system
Platform	EVM

Engagement Summary

Dates	February 11 - February 26, 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High Severity Issues	1	■
Total Medium Severity Issues	2	■■
Total Low Severity Issues	3	■■■
Total Informational Severity Issues	4	■■■■
Total	9	

Category Breakdown

Access Controls	1	■
Auditing and Logging	1	■
Code Quality	3	■■■
Data Validation	3	■■■
Patching	1	■
Undefined Behavior	1	■
Total	9	

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

❑ **Revise `verifyState` to accurately check memory consistency**. The current check performed to ensure memory consistency during an upgrade incorrectly assumes that type casting memory accurately reflects underlying data.

❑ **Ensure `finishUpgrade` performs the same checks as constructor**. When upgrading, contract state can be initialized to prohibited values via the `finishUpgrade` method.

❑ **Prohibit multiple calls to `finishUpgrade`**. The contract owner can call `finishUpgrade` as many times as they want, allowing them to reinitialize the contract to arbitrary values.

❑ **Do not allow the proxy to have methods that shadow the implementation**. When both the proxy and implementation inherit from the same contract, the proxy's version of the inherited method shadows that of the implementation, causing undefined behavior.

❑ **Always check for contract's existence when using `delegatecall`**. When the `delegatecall` opcode is used, it will return success when the called contract is non-existent. This will go undetected by the system and cause undefined behavior.

❑ **Optimize Numerology library to save gas.** Users will save money on gas once code has been optimized.

❑ **Make sure correct zero points are returned, and also check to make sure zero points are actually zero points in Numerology.** It does not take much extra gas to return a correct zero point. The checks will save users on gas before they go through an arduous computation to produce an off-curve point which will revert because the resulting point will not be on the curve.

Long Term

❑ **Be mindful of changes to storage layout.** The solidity compiler does not ensure storage layout remains the same across versions. Any such change could lead to system failures during an upgrade.

❑ **Use events for critical operations**. Emitting events for important system actions such as `finishUpgrade` allow the system to be monitored for suspicious or erroneous behavior.

❑ **Do not use the proxy pattern.** Relying on memory consistency across contracts is risky and difficult to test. Consider moving towards a migration-based approach.

Findings Summary

#	Title	Type	Severity
1	`verifyState` does not accurately check memory layout	Data Validation	High
2	Contract upgrades can catastrophically fail if storage layout changes	Patching	Low
3	`finishUpgrade` lacks same checks as contract constructor	Data Validation	Low
4	Contract owner can arbitrarily replay `finishUpgrade`	Access Controls	Low
5	Proxy has public methods that shadow implementation	Undefined Behavior	Medium
6	Lack of events for critical operations	Auditing and Logging	Informational
7	Dispatcher does not confirm contract's existence prior to returning	Data Validation	Medium

1. `verifyState` does not accurately check memory layout

Severity: High
Type: Data Validation
Target: `MinersEscrow, MiningAdjucator`

Difficulty: Medium
Finding ID: TOB-NCY-001

Description
The `verifyState` function is designed to ensure the memory layout of an upgraded contract is the same as that of the previous version. However, since it only verifies one entry in large structures and uses type casts to check equality, it can mistakenly succeed even when memory layouts have changed.

For example, in `MinersEscrow` the variable `minLockedPeriods` has type `uint16`. Since the variable before it is a `map` and the variable after is a `uint256`, it occupies its own 32-byte slot in memory. Suppose a developer changes the type of `minLockedPeriods` to `uint8` and creates a new `uint8` variable `foo` right after `minLockedPeriods`, but before `minAllowableLockedTokens`.

```
mapping (uint16 => uint256) public lockedPerPeriod;
uint16 public minLockedPeriods;
uint256 public minAllowableLockedTokens;
```

MinersEscrow lines 97-99

Due to the way EVM optimizes memory, both `minLockedPeriods` and `foo` will occupy the same slot in memory. When upgrading, the `verifyState` function will not report an error, since the data retrieved from that 32-byte slot is cast to a `uint16` and checked against the original contract's memory.

Another weakness in the `verifyState` function comes from the fact that it only checks one entry of a `mapping` for consistency. For example, in `MiningAdjucator`, the variable `evaluatedCFrags` is a mapping from `bytes32` to a `bool`. Currently, Boolean values are represented as `uint8`'s in storage. However, if some future upgrade of the Solidity compiler were to represent Booleans in a more memory-efficient way in, the memory layout would change while still having `verifyState` pass. This is due to the fact that in the first entry of `evaluatedCFrags` everything will appear to be the same, since getting the 32 bytes in storage will almost certainly yield a nonzero value. However, the memory layout will be entirely different.

```
bytes32 evaluationCFragHash = SignatureVerifier.hash(
    abi.encodePacked(RESERVED_CAPSULE_AND_CFRAG_BYTES), hashAlgorithm);
require(delegateGet(_testTarget, "evaluatedCFrags(bytes32)", evaluationCFragHash) != 0);
```

MiningAdjucator lines 480-482

This layout change would not be caught by other tests in `verifyState` since `evaluatedCFrags` is the last variable declared and therefore does not affect the layout prior to its beginning.

Exploit Scenario
An attacker sees that the new storage layout does not agree with the original one, and is able to overwrite `foo` when `minLockedPeriods` is changed.

The Solidity compiler optimizes storage of Booleans, which goes undetected by `verifyState` since it only checks the first entry of a mapping.

Recommendation
The proxy pattern is in no way safe or recommended.

In general, relying on type casts when testing for memory consistency is not robust to changes in layout. Similarly, checking one value of a mapping or array is not sufficient since the Solidity compiler might change memory layout.

References
- https://solidity.readthedocs.io/en/v0.4.24/miscellaneous.html#layout-of-state-variables-in-storage

2. Contract upgrades can catastrophically fail if the storage layout changes

Severity: Low
Type: Patching
Target: All upgradable contracts

Difficulty: Low
Finding ID: TOB-NCY-002

Description
The NuCypher contracts use the proxy pattern for upgradability. Due to the way in which the pattern implements upgrades, the storage layout of the contracts must not change between deployments. Unfortunately, the Solidity compiler can and does often change its storage layout between versions. Any change in the state variables (new variables, changes of type, *etc.*) will require a thorough assessment before upgrading. Extreme care must be placed in implementing inheritance, as it may also affect the storage layout.

This finding does not represent a current vulnerability in the code. However, a mismanaged upgrade can easily and immediately lead to a broken contract, constituting a high-severity issue. This finding is classified as having low severity because Solidity does not have a good track record of being backward compatible, and it is becoming increasingly hard to install older versions of the compiler.

Exploit Scenario
A newer version of `solc` is used to compile a contract upgrade, causing the storage layout to change. This will cause the contract to silently, catastrophically fail upon upgrade.

Recommendation
In the short term, document this vulnerability in the NuCypher upgrade procedures. Also record the version of Solidity used for the initial deployment and ensure that that same version of Solidity is used for *all* future deployments. Implement all of the bullet points in the recommendations section of our contract upgrade anti-patterns blog post.

In the long term, consider switching to a different contract upgrade pattern, such as contract migrations.

3. `finishUpgrade` lacks same checks as contract constructor

Severity: Low
Type: Data Validation
Target: All upgradable contracts

Difficulty: High
Finding ID: TOB-NCY-003

Description
None of the `finishUpgrade` methods contain the same checks as the constructors they replace. For example, `MinersEscrow` checks that `_minLockedPeriods` is greater than 1, but this is not reflected in `finishUpgrade`.

```
require(_minLockedPeriods > 1 && _maxAllowableLockedTokens != 0);
minLockedPeriods = _minLockedPeriods;
minAllowableLockedTokens = _minAllowableLockedTokens;
maxAllowableLockedTokens = _maxAllowableLockedTokens;
```

`MinersEscrow` constructor lines 135-138

```
function finishUpgrade(address _target) public onlyOwner {
    super.finishUpgrade(_target);
    MinersEscrow escrow = MinersEscrow(_target);
    minLockedPeriods = escrow.minLockedPeriods();
    minAllowableLockedTokens = escrow.minAllowableLockedTokens();
    maxAllowableLockedTokens = escrow.maxAllowableLockedTokens();

    // Create fake period
    lockedPerPeriod[RESERVED_PERIOD] = 111;
}
```

`MinersEscrow` lines 1160-1169

Exploit Scenario
An attacker with temporary access to the owner's keys calls `finishUpgrade` on `MiningAdjucator` even though no upgrade has occurred. The target contract has `_percentagePenaltyCoefficient` set to zero and since no check occurs, the current version of `MiningAdjucator` accepts this change. They are then able to produce malicious re-encryptions without paying a fine. This situation is made worse by the fact that no event is emitted when `finishUpgrade` is called, causing the malicious action to go undetected.

Recommendation
Ensure that each `finishUpgrade` method performs the same checks as the contract constructor.

4. Contract owner can arbitrarily replay finishUpgrade

Severity: Medium
Type: Access Controls
Target: All upgradable contracts

Difficulty: Low
Finding ID: TOB-NCY-004

Description
In all of the upgradeable contracts, the owner can call `finishUpgrade` at any time with any target contract address. This can lead to the upgraded contract being reset to the state of any contract of the same type. Furthermore, executing `finishUpgrade` does not emit an event, which allows subtle changes to be made without alerting the network.

Exploit Scenario
Someone with owner privileges wants to set a higher mining coefficient. They call `finishUpgrade` on the current version of `MinersEscrow` and set the target contract to a `MinersEscrow` with a very high mining coefficient. This goes undetected by the system since `finishUpgrade` does not emit an event and is made worse by the fact that none of the checks present in the contract constructor are present in the `finishUpgrade` method.

Recommendation
Add logic to `finishUpgrade` that prevents it from being called multiple times. Also emit an event signaling that the `finishUpgrade` method has been called.

In general, use events to signal changes being made to contract state. Prevent logic that dramatically alters state to arbitrary values from being replayed or used in an unconstrained manner.

5. Proxy has public methods that shadow implementation

Severity: High
Type: Undefined Behavior
Target: All upgradable contracts

Difficulty: Low
Finding ID: TOB-NCY-005

Description
Both the Dispatcher and it's implementing contracts inherit from `Ownable`. This shared inheritance causes them to both have identically named public functions. However, if the implementing contract decides to change the logic of one of these shared functions, the Dispatcher will never call that method, leading to undefined behavior.

Exploit Scenario
Both the `Dispatcher` and `MinersEscrow` have an `owner()` method which gets the `owner` variable. When calling the `Dispatcher`, though, the fallback function is not triggered since it inherits from `Ownable`. If both contracts use the same logic for `owner()` this is not an issue. However, suppose now that `MinersEscrow` creates a function called `owner()` which changes state and returns the `owner` variable. This new functionality will never be triggered by the `Dispatcher` since it will always call its public method `owner()`.

Recommendation
Ensure that the proxy does not have any public functions that shadow functions in the implementation. Do not allow the proxy and implementation to inherit the same functions.

This is another example of how risky the proxy upgrade pattern is, and how subtle changes to each contract can cause undefined behavior and system failure. Consider using a migration-based upgrade pattern.

6. Lack of events for critical operations

Severity: Informational | Difficulty: Low
Type: Auditing and Logging | Finding ID: TOB-NCY-006
Target: `Issuer.sol`, `MinersEscrow.sol`, `MiningAdjucator.sol`, and `PolicyManager.sol`

Description
Several critical operations do not trigger events. As a result, it will be difficult to review the correct behavior of the contracts once deployed. Users and blockchain monitoring systems will not be able to easily detect suspicious behaviors without events.

Exploit Scenario
Eve compromises the `MinersEscrow` contract calls `finishUpgrade` with a malicious target contract. The system does not detect this subtle change in parameters immediately since no event is emitted.

Recommendation
Add events for all critical operations.

Consider using a blockchain monitoring system to track any suspicious behavior in the contracts.

7. Dispatcher does not confirm contract's existence prior to returning

Severity: Medium
Type: Data Validation
Target: `Dispatcher.sol`

Difficulty: Medium
Finding ID: TOB-NCY-007

Description
The Dispatcher does not verify that a contract exists at the address it is calling `delegatecall` on. Since `delegatecall` will return success if the called account is non-existent, many methods in the NuCypher codebase will make state altering changes when calls to upgradeable contracts fail.

```
function () external payable {
    assert(target != address(0));
    (bool callSuccess,) = target.delegatecall(msg.data);
    if (callSuccess) {
        assembly {
            returndatacopy(0x0, 0x0, returndatasize)
            return(0x0, returndatasize)
        }
    } else {
        revert();
    }
}
```

The fallback function in `Dispatcher`

Exploit Scenario
During an upgrade, one of the NuCypher contracts is mistakenly replaced by a non-existent contract, either though human error or a malicious party with access to owner keys calling self destruct. This goes undetected by the system and the system behaves as though every call to the non-existent contract is successful, causing serious harm to user data and assets.

Recommendation
Always check that the contract being called through `delegatecall` has code using the `extcodesize` opcode.

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. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- The Numerology `secp256k1` library, specifically `addAffineJacobian` and `doubleJacobian`, can calculate the point at infinity more efficiently by including checks that avoid going through an entire point addition/doubling formula.
- More efficient calculations can be used for `doubleJacobian` to save a field multiplication, as per #749.
- There are minor worries about using `z=0` as a definitive class of zero points for Jacobian coordinates. When an incorrect Jacobian zero point is passed to one of the addition methods that take a Jacobian argument, it will cause the library to produce off-curve points and eventually fail an assertion when it requires points to be on curve. Currently there appear to be no issues, but when new code or a new use case is added, and it does not perform sufficiently stringent checks, it could cause the issues mentioned above. If any external exposure of Numerology is to be seen in the future, we recommend at minimum extending `is_on_curve` to Jacobian coordinates as well as checking that the inputs to all mathematical functions are correct curve points.