Synthetix

Solidity Security Review

Unipool Smart Contract

Version: 2.0

February, 2020

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the `Unipool` smart contract, which replaces the current Uniswap sETH liquidity provider incentivisation mechanism with an *onchain* staking contract. This review focused solely on the security aspects of the Solidity implementation of the contract, but also includes general recommendations and informational comments. The more general economic structure of the system and related economic game theoretic attacks on the platform are outside the scope of this assessment.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review regarding, the underlying business model or the individuals involved in the project. This document is based upon a time-boxed analysis of the underlying smart contracts. Statements are not guaranteed to be accurate and do not exclude the possibility of undiscovered vulnerabilities.

Document Structure

The first section provides an overview of the functionality of the (`Unipool` smart contract contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given, which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*. Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities found within the `Unipool` contract.

Overview

The Synthetix platform aims to produce collateralized stable coins. It uses two types of tokens:

1. **Synths:** Tokens which aim to be stable compared to fiat or crypto currencies (sUSD, sAUD, sEUR, sBTC, etc.);
2. **Synthetix:** Fixed-supply token which receives fees from the exchanging of Synths via synthetix.exchange.

The Synthetix platform has been assessed by Sigma Prime on multiple previous occasions.

The current liquidity provider incentivisation mechanism for the sETH/ETH pair relies on an offchain spreadsheet combined with a python script to calculate the reward of liquidity providers.

The `Unipool` smart contract replaces this process and moves it *onchain* and is meant to work as follows:

1. Users add liquidity to the sETH uniswap exchange and receive uniswap `UNI` tokens;
2. Users can stake these `UNI` tokens inside the `Unipool`
3. Everytime the `Synthetix.mint()` function is called to mint the inflationary supply, a portion of these SNX tokens are sent to the `Unipool` contract
4. Liquidity providers (i.e. stakers) are assigned a share of the SNX rewards based on the percentage of staked tokens compared with the rest of the pool of liquidity providers
5. Liquidity providers can claim their SNX rewards by calling a withdraw function on the `Unipool` contract.

The detailed specification related to this contract is described in the following Synthetix Improvement Proposal SIP #31.

Security Review Summary

This review was initially conducted on commit 4cb9b50, and targets exclusively the sole file `Unipool.sol`. This contract implements the SIP #31 specification, which was accessed at commit afa636f.

This smart contract inherits the `LPTokenWrapper` and `IRewardDistributionRecipient` contracts.

During the course of this engagement, a new commit hash (commit 4feb1cb) was provided which mitigates issue UNI-03 but introduced UNI-01.

Retesting activities targeted commit 53df522.

The manual code-review section of the reports are focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focuses on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

In prior reviews, Sigma Prime have raised vulnerabilities regarding centralization aspects (i.e. administrative control from the owner account). Synthetix have previously acknowledged and accepted these vulnerabilities, so we omit them from this review and direct the reader to our prior reviews for more information.

The testing team identified a total of seven (7) issues during this assessment, of which:

- One (1) is classified as critical risk,
- Two (2) are classified as high risk,
- One (1) is classified as medium risk,
- Three (3) are classified as informational.

All these vulnerabilities have been acknowledged and/or resolved by the development team.

To support this review, the testing team used the following automated testing tools:

- Rattle: `https://github.com/trailofbits/rattle`
- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within Synthetix's `Unipool` smart contract. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including comments not directly related to the security posture of the `Unipool` contract, are also described in this section and are labelled as *"informational"*.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team;
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk;
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
UNI-01	Wrong Order of Operations Leads to Exponentiation of `rewardPerTokenStored`	Critical	Resolved
UNI-02	Staking Before Initial `notifyRewardAmount` Can Lead to Disproportionate Rewards	High	Resolved
UNI-03	External Call Reverts if Period Has Not Elapsed	High	Resolved
UNI-04	Gap Between Periods Can Lead to Erroneous Rewards	Medium	Resolved
UNI-05	Rounding to Zero if Duration is Greater Than Reward	Informational	Resolved
UNI-06	`Withdrawn` Event Log Poisoning	Informational	Resolved
UNI-07	Miscellaneous General Coments	Informational	Resolved

UNI-01	Wrong Order of Operations Leads to Exponentiation of `rewardPerTokenStored`		
Asset	`Unipool.sol`		
Status	**Resolved:** Resolved in commit 6aab0b2		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `Unipool.sol` contract was updated in commit 4feb1cb.

The updated `rewardPerToken()` function was modified from:

$$rewardPerToken = rewardPerTokenStored + \frac{lastTimeRewardApplicable() - lastUpdateTime}{totalSupply()} * rewardRate * 1e18$$

to:

$$rewardPerToken = \frac{rewardPerTokenStored + lastTimeRewardApplicable() - lastUpdateTime}{totalSupply()} * rewardRate * 1e18$$

The result is that `rewardPerTokenStored` will grow exponentially thereby severely overstating each individuals rewards earned. Individuals will therefore either be able to withdraw more funds than should be allocated to them or they will not be able to withdraw their funds at all as the contract has insufficient `SNX` balance.

This vulnerability makes the `Unipool` contract unusable.

Recommendations

Adjust the function `rewardPerToken()` to represent the original functionality.

Resolution

The typo introduced as been removed in commit 6aab0b2.

UNI-02	Staking Before Initial `notifyRewardAmount` Can Lead to Disproportionate Rewards		
Asset	`Unipool.sol`		
Status	**Resolved:** Resolved in commit df79f70		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

If a user successfully stakes an amount of `UNI` tokens before the function `notifyRewardAmount()` is called for the first time, their initial `userRewardPerTokenPaid` will be set to zero.

The first time `notifyRewardAmount()` is called, both `rewardRate` and `periodFinish` will be set; however `lastUpdateTime` will remain zero.

The staker may then call the `earned()` function which will state their balance as:

$$earned = now * rewardRate * percent\_of\_stake$$

Since both `lastUpdateTime` and `userRewardPerTokenPaid` are zero.

This can therefore allow a staker to call `get_reward()` at a later date when the contract has sufficient funds. The staker would be paid out funds greater than their share of the `SNX` rewards.

For a proof-of-concept, see `test_get_reward.py/test_get_rewards_stake_early()`.

Recommendations

We recommend preventing `stake()` from being called before `notifyRewardAmount()` is called for the first time.

Alternatively, set `lastUpdatetime` to `block.timestamp` during the first execution of `notifyRewardAmount()`.

Resolution

The second recommendation has been introduced in commit df79f70.

UNI-03	External Call Reverts if Period Has Not Elapsed		
Asset	`Unipool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The function `notifyRewardAmount()` will revert if `block.timestamp >= periodFinish`. However this function is called indirectly via the `Synthetix.mint()` function. A revert here would cause the external call to fail and thereby halt the mint process. `Synthetix.mint()` cannot be successfully called until enough time has elapsed for the period to finish.

Recommendations

Consider handling the case where the reward period has not elapsed without reverting the call.

Resolution

The development team has resolved this issue in commit 4feb1cb.

UNI-04	Gap Between Periods Can Lead to Erroneous Rewards		
Asset	`Unipool.sol`		
Status	**Resolved:** Resolved in commit df79f70		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `SNX` rewards are earned each period based on `reward` and `duration` as specified in the `notifyRewardAmount()` function.

The *"rate per second"* at which rewards are distributed is given by:

$$rewardRate = \frac{reward}{duration}.$$

Rewards are paid out for all stakers following the formula:

$$rewards\_distributed = rewardRate * (lastTimeRewardApplicable - lastUpdateTime)$$

However, `lastUpdateTime` includes the time between periods and therefore, if there is a gap between periods:

$$(lastTimeRewardApplicable - lastUpdateTime) > duration$$

Therefore, resulting in:

$$rewards\_distributed > reward$$

Where $reward$ is the amount from `notifyRewardAmount()` and $rewards\_distributed$ is the total amount earned by all stakers.

The result is the contract will output more rewards than it receives. Therefore if all stakers call `getReward()` the contract will not have enough `SNX` balance to transfer out all the rewards and some stakers may not receive any rewards.

For a proof-of-concept, see `test_get_reward.py/test_get_rewards_slippage()`

Recommendations

We recommend enforcing each period start exactly at the end of the previous period.

Resolution

The above recommendation has been implemented in commit df79f70.

UNI-05	Rounding to Zero if Duration is Greater Than Reward
Asset	`Unipool.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `rewardRate` value is calculated as follows:

$$rewardRate = \frac{reward}{duration}$$

Due to the integer representation of these variables, if `duration` is larger than `reward` the value of `rewardRate` will round to zero. Thus, stakers will not receive any of the reward for their stakes.

Furthermore, due to the integer rounding, the total rewards distributed may be rounded down by up to one less than `duration`. As a result, the `Unipool` contract may slowly accumulate `SNX`.

For a proof-of-concept, see `test_get_reward.py/test_get_rewards_larger_duration()`

Recommendations

Beware of the rounding issues when calling the `notifyRewardAmount()` function. We also recommend some way of allowing the excess `SNX` reward from rounding to be claimed or withdrawn from the `Unipool` contract.

Resolution

The testing team removed the `duration` input parameter from the `notifyRewardAmount()` function. The smart contract is now using a default 7 days duration instead.

For any input amount in wei that is larger than `604800 wei / (7 days duration in seconds)`, `rewardRate` should give a integer that is more than 0.

UNI-06	`Withdrawn` Event Log Poisoning
Asset	`Unipool.sol`
Status	**Resolved:** Resolved in commit 53df522
Rating	Informational

Description

Calling the `withdraw()` function will emit the `Withdrawn` event. No `UNI` tokens are required as this function can be called with `amount = 0`.

As a result a user could continually call this function, creating a potentially infinite amount of events.

This can lead to an event log poisoning situation where malicious external users spam the `Unipool` contract to generate arbitrary `Withdrawn` events.

Recommendations

Consider adding a `require` or `if` statement preventing the `withdraw()` function from emitting the `Withdrawn` event when the `amount` variable is zero.

Resolution

The testing team updated the `withdraw()` and `stake()` functions to introduce `require` statements, preventing the `Staked` and `Withdrawn` events from being called with `amount = 0`.

UNI-07	Miscellaneous General Coments
Asset	`Unipool.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have a direct security implication:

- The variables `snx` and `uni` can be declared as `constant` s.
- In the updated `notifyRewardAmount()` function (provided in commit 4feb1cb), the following lines are duplicated (i.e. present in both `if` and `else` code blocks):
  - `periodFinish = block.timestamp.add(duration);`
  - `emit RewardAdded(reward, duration);`

  By moving these instructions outside of the `if/else` statements, $15,709$ gas can be saved on contract deployment.
  ✓ Resolved in commit [df79f70]

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

- Since the development team uses a *mock* contract for testing purposes that allows setting the `uni` and `pool` variables, the first recommendation cannot be implemented without significantly impacting the internal testing process.
- The second recommendation has been implemented by the development team in commit df79f70.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `pytest` framework was used to perform these tests and the output is given below.

```
tests/test_deploy.py::test_deploy                           PASSED  [8%]
tests/test_exit.py::test_exit                               PASSED  [16%]
tests/test_exit.py::test_exit_rewards                       PASSED  [25%]
tests/test_get_reward.py::test_get_rewards                  PASSED  [33%]
tests/test_get_reward.py::test_get_rewards_larger_duration  PASSED  [41%]
tests/test_get_reward.py::test_get_rewards_stake_early      PASSED  [50%]
tests/test_get_reward.py::test_get_rewards_multiple         PASSED  [58%]
tests/test_get_reward.py::test_get_rewards_slippage         PASSED  [66%]
tests/test_staking.py::test_stake                           PASSED  [75%]
tests/test_withdraw.py::test_withdraw                       PASSED  [83%]
tests/test_withdraw.py::test_over_withdraw                  PASSED  [91%]
tests/test_withdraw.py::test_withdraw_no_balance            PASSED  [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

		Likelihood		
Impact		Low	Medium	High
	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].