Mooniswap Audit Report

Date: August 12, 2020

The following report is an audit from Scott Bigelow, requested by the 1inch.exchange team, for their upcoming project “Mooniswap”.

This audit consists of both automated and manual review of the Mooniswap Solidity code, available here:

https://github.com/CryptoManiacsZone/mooniswap/blob/master/contracts/Mooniswap.sol

Git commit hash: 16dd67c4c5c2be0bb4d2c89bd2ac27fa9af1367f

This report is my personal review of the Mooniswap system and is unrelated to any work I have done related to any organization or company.

Overview

Mooniswap is an implementation of an “Automated Market Maker”, allowing exchange between two assets using an automated pricing mechanism based on balance ratios. While similar in approach to simple AMMs, such as Uniswap, Mooniswap utilizes a “virtual” pricing mechanism which limits the value arbitrageurs can extract from liquidity providers. Over the course of a fixed period of time, 5 minutes in the above commit, buy and sell prices continually converge on their prices based on the “real” underlying balance ratio. For example, a large buy order creates price movement which is not immediately reflected in the corresponding sell price. This new, better sell price is linearly converged on over the course of 5 minutes. When a “sell” order is placed within this 5 minute window, the price will be between the price before and after the original buy transaction. After 5 minutes with no additional trading, the price is based exclusively on the real underlying balance ratio.

In practice, I expect virtual-balance based systems to suffer some impermanent loss, but less than a normal AMM; during times of large price swings, the spread between buy and sell becomes large. Due to this conditionally larger spread, I also expect a virtual balance-based AMM to trade less volume than a traditional AMM, although how much less will only be known once the system reaches the market.

Summary of findings

The overall code quality of the project is good and it is accompanied by unit tests. The virtual balance mechanism is novel and how it attracts traders and liquidity providers will only be understood once released.

There are several defensive programming tactics, such as `Math.min()` and `Math.max()` for swap price selection, which should help prevent unforeseen token interactions and virtual balance manipulation.

The 2 critical issues that were discovered impacted newly created markets or would have required the combination of a malicious owner and a specific, somewhat rare token type. Both issues have been addressed in PR #37 and PR #35

Findings:

Critical:

Owner can withdraw all tokens on a market if any of them are ERC-777

https://github.com/CryptoManiacsZone/mooniswap/blob/16dd67c4c5c2be0bb4d2c89bd2ac27fa9af1367f/contracts/Mooniswap.sol#L264-L276

If Mooniswap is an AMM for any ERC777 token, the Owner would be able to steal them from liquidity providers using the following sequence of events.

1.) Transfer ownership to a malicious contract with an ERC777 callback hook
2.) call rescueFunds() against the address of the ERC777 through this malicious contract when there is no ERC777 surplus token supply
3.) The balance of all the ERC777 and the other token will be saved in memory
4.) Transferring the ERC777 tokens will trigger onReceived callback, which will add new liquidity to the contract via deposit(). Ensure the amount of ERC777 tokens deposited matches or exceeds the amount being stolen
5.) The `require(tokens[i].uniBalanceOf(address(this)) >= balances[i])` step that ensures the liquidity providers have not lost tokens will pass due to the balance being stored in memory before the liquidity was added.

6.) Options:
    a.) The owner could simply walk with these ERC-777 tokens
    b.) This theft creates a token imbalance which overvalues the token that was just stolen. By providing these tokens back to the market via swap(), the owner could take the other side of the market

Note: this was resolved in PR #35 by making rescueFunds `nonReentrant`

Initial liquidity provider can be negatively impacted by front-running

The initial liquidity provider specifies the initial ratio and minReturn in liquidity tokens. However, these liquidity tokens do not yet have an established value and can be manipulated by a front-run to the deposit() function. If the attacker creates an imbalanced ratio of tokens while ensuring these tokens are of little value , the “minReturn” argument from the victim does not protect them against depositing their tokens at this ratio.

Note: resolved in PR #37 by passing in both a requested amount and a minimum amount of each token, effectively setting a floor to the ratio difference from requested to actual

Medium:

No deadline on swap(), deposit(), or withdraw()

While these 3 functions contain parameters that ensure the user withdraws enough tokens or is granted enough liquidity tokens, a transaction that sits in the pending queue for long periods of time becomes a “free option”: one where if the price moves against them, it doesn’t execute, but if it moves in their favor, they could be committing to sell at a much lower price than the market is offering. This is especially profitable for miners. As an example, a user offers to buy DAI at a rate of 400 DAI per ETH, but the transaction is not provided sufficient gas to mine for a long time, long enough for the price of ETH to rise to 500 DAI per ETH. A miner could order transactions in such a way to ensure the user is only given 400 DAI per ETH using the virtual balance system, ensuring the liquidity providers receive significant benefit.

The design of the Mooniswap system decreases, but does not fully eliminate, the attacker’s economic incentive of an AMM sandwich attack by delaying price movement. The attacker can still profit through transaction ordering if they are a liquidity provider, even if they become one via a flash loan.

Late check of proper token, handing EVM execution to untrusted code

The `swap()` function takes 5 arguments, 2 of which are

- `IERC20 src`
- `IERC20 dst`

The swap function does not check that these arguments are valid (via `isToken()`) until partially through the swap, with `src.transferFrom()` having been called prior to this check. While I don't see a potential attack (given the revert that occurs later), this is an unnecessary EVM hand-off to potentially malicious code. Furthermore, the revert message for passing in an invalid token is the same as one that doesn't meet minTokens: `Mooniswap: return is not enough`.
Checking that a token is valid and the math for calculating an exchange rate should be separated, with the check happening first, with a clear revert message.

VWAP calculation vulnerable to false trading by liquidity providers

A large liquidity holder on a market is not significantly impacted by price movement from large trades since they end up retaining the majority of the assets, regardless of the balance, while the price convergence system ensures when the arbitrage opportunity does open up, the tokens are sold near market value. This liquidity holder could leverage this minimization of impact to significantly influence the price vwap mechanism

https://github.com/CryptoManiacsZone/mooniswap/blob/16dd67c4c5c2be0bb4d2c89bd2ac27fa9af1367f/contracts/Mooniswap.sol#L260-L261

Low:

Solidity version selection not specific, uses features not available in range specified

The heading of Mooniswap.sol specifies
`pragma solidity ^0.6.0;`
Enforcing any Solidity version at 0.6.0 or greater. It is recommended to specify specific versions for consistency and to ensure the features your contract is using are available in the solidity version you have specified. Several features used in the contract are not valid in this range.
https://github.com/CryptoManiacsZone/mooniswap/blob/16dd67c4c5c2be0bb4d2c89bd2ac27fa9af1367f/contracts/libraries/UniERC20.sol#L55

`address(token).staticcall{ gas: 20000 }`
Bracketed gas and value arguments were not supported until 0.6.2

https://github.com/CryptoManiacsZone/mooniswap/blob/16dd67c4c5c2be0bb4d2c89bd2ac27fa9af1367f/contracts/Mooniswap.sol#L96-L99

`mapping(IERC20 => bool) public isToken;`
Non-elementary keys were not supported until 0.6.3

https://github.com/CryptoManiacsZone/mooniswap/blob/477fbae6caf66dc2e35c13e8161833fc585ab21e/contracts/Mooniswap.sol#L195
```
function withdraw(uint256 amount, uint256[] memory minReturns)
external nonReentrant {
```

External function arguments to memory-based variables could only be `calldata` until 0.6.9

Delayed convergence through deposit/withdraw cycle

Depositing and withdrawing liquidity does not incur fees and is not subject to the virtual balance system. A user could add and remove liquidity in a single transaction, which results in balances not being changed, but convergence timestamp being pushed out 5 minutes from the current time.

The impact to the pricing mechanism is minimal and the benefit to the attacker is minimal. A large liquidity provider could force the market to offer slightly worse prices over a long period of time, ensuring they incur more fees if the market retains the volume.

Using default storage state to indicate token isETH()

Using a token address of 0x0000000000000000000000000000000000000000 to indicate the specified token is ETH is potentially risky, as an accidental storage read of an unset location could be a contributing cause to an exploit. I do not see a specific vulnerability related to the ambiguity between unset storage and isETH(), but setting this value to a non-0 sentinel value (such as 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE) can prevent an issue like this from occurring, with minimal gas cost implications.

Removing the native Ether support and using a wrapped ETH token, such as WETH, would be even better for security, allowing for safer and simpler contracts, avoiding branching isETH logic and emulating pre-transfer balances, although this use of native ETH is mentioned in the whitepaper as an intended design decision.

Withdraw minReturns dynamic array could exceed _tokens.length

Passing in a minReturns array larger than _tokens is invalid, but not checked for. I recommend adding assertion:
`require(minReturns.length <= tokens.length)`

A user could accidentally pass in 3 items (perhaps making the first argument 0x0, with the other two being the actual returns expected). This is unlikely to be a significant concern until the system allows more than 2 tokens on a Mooniswap market (which itself will require further review)

Note

Impermanent loss exaggerated by referral liquidity token allocation

The referral system hands out [small] amounts of liquidity tokens on token swaps (if referral address specified). Not all swaps are profitable for liquidity providers, potentially leading to a situation where liquidity providers are leaking value both to the swap taker in the form of trading more valuable tokens for less valuable ones AND to the referral address in the form of a stake in both assets.
Most of the time, this is not an issue on a market with acceptable levels of swaps occurring outside standard price movement arbitrage, but this might be a surprise to liquidity providers accustomed to levels of impermanent loss from liquidity providers on other systems.

fee() can be updated at will

The fee can be added and removed by the owner at will which:

1.) Changes the arrangement the liquidity providers expected
2.) Could add a fee between the time a user broadcast a transaction and the time it landed.

Fortunately for #2, minReturn is applied to the after-fee exchange amount, which ensures the swap user retains some control.

Gas Savings:

These are possible ideas for saving gas, but they all come with drawbacks. Skipping these suggestions is perfectly reasonable and does not decrease the quality or security of the contract.

Make Mooniswap.factory immutable

https://github.com/CryptoManiacsZone/mooniswap/blob/477fbae6caf66dc2e35c13e8161833fc585ab21e/contracts/Mooniswap.sol#L94

Materialize fee on Mooniswap market

https://github.com/CryptoManiacsZone/mooniswap/blob/477fbae6caf66dc2e35c13e8161833fc585ab21e/contracts/Mooniswap.sol#L114-L116
Would require calling function on every market when the fee changes (reading from factory.fee() once), but would save about 1000 gas per swap

Remove tokens[], use immutable, discreet token0/token1 storage values

Arrays can't be immutable, even though the tokens array is never changing. By changing to token0/token1, at least 2,400 gas could be saved per transaction.

If eventually generalizing to multiple tokens, there are legitimate reasons to keep the implementation generic.