Mooniswap Audit Report

dapp.org

fv@dapp.org.uk

last updated: 10.08.2020

Table of Contents

Summary

From August 4th to August 9th 2020, a team of four engineers reviewed the smart contracts for the Mooniswap decentralized trading protocol.

This work was carried out against the following git repositories:

- CryptoManiacsZone/mooniswap at `21fb05adc4c21208a42fac5c55ff4df2502ce403`

The team discovered 5 issues, two of which were of high severity, as well as proposing several code quality improvements and gas optimizations. All but one discovered issues have been addressed by the Mooniswap team as of `01b5c2fcb452d1553d84c2364e94e17905aa9773`.

Scope

The following work was carried out as part of the engagement:

General code review and security analysis

The team carried out a general code review with the aim of identifying:

- Issues impacting the functionality or security of the contract
- Code quality improvements or gas optimizations

Fuzz testing of core smart contracts

The team identified key security properties and wrote a suite of property based tests asserting these properties using the dapptools framework.

These tests are available at: https://github.com/dapp-org/mooniswap-tests/.

Team

The review was carried out by the following members of the dapp.org collective:

- David Currin
- David Terry
- Lev Livnev
- Martin Lundfall

Changelog

A revision history for this document can be found here.

Findings

Bugs

Recommendation	Severity	Likelihood	Accepted	Commit
Early `deposit` to a pool is vulnerable to theft by front-running	high	medium	???	???
Owner can generate free LP shares using reentrancy in `rescueFunds`	high	low	yes	01b5c2f
Rounding error in calculation of amount to send in `deposit`	low	high	yes	16dd67c
Amounts pulled are dependent on token ordering in `deposit`	low	high	yes	16dd67c
Interactions delay virtual balance convergence	low	high	no	-

Early `deposit` to a pool is vulnerable to theft by front-running

When a user makes the first deposit to a new `Mooniswap` pool, an attacker can front-run their `deposit` call with a `deposit` call of their own to initialise the pool with skewed token balances, and follow the user's `deposit` with an arbitrage trade, resulting in the attacker stealing most of the user's deposit.

The usual protection of supplying a `minReturn` argument is ineffective, because before the first deposit has been made, the value of an LP share can be set essentially arbitrarily. Later in the life of the pool, the same attack is prevented by the `minReturn` argument.

As a sketch example of the attack: suppose a user creates a Mooniswap pool for the USDC/PAX pair, and sends

```
deposit([100, 100], 100)
```

expecting to get 100 pool tokens that claim 100 USDC + 100 PAX. An attacker who sees this transaction, sends with higher gas price:

```
deposit([1, 100], 100)
```

If the attacker transaction mines first, the attacker receives 100 pool tokens. When the victim's transaction, the victim receives 100 pool tokens too (n.b. how the `minReturn` check is no longer meaningful). Now the attacker can withdraw their initial deposit with:

```
withdraw(100, [1, 100])
```

and then can perform a trade against the victim in the pool with:

```
swap(usdc, pax, 9, 90, 0x0)
```

buying 90 PAX for only 9 USDC, leaving the victim with 100 pool tokens which now claim only 10 USDC + 10 PAX, suffering a $81 loss.

Note that for numerical simplicity, this worked example ignores the effect of the `fee()` and `BASE_SUPPLY`, though the basic idea holds mutatis mutandis. Moreover, by depositing an even smaller amount of USDC, the attacker can increase their spoils to the user's entire deposit.

It is important to also note that the attack is feasible not only against the first deposit to a pool, but to any other deposit made soon after the first. If subsequent deposits are made either based on unconfirmed transactions, or if the blockchain may reorg in the future (either due to attacker influence or not), an attacker could manipulate the initial ratios in the first `deposit` call, and subject a depositor to an adverse trade.

Owner can generate free LP shares using reentrancy in `rescueFunds`

The `rescueFunds` function should allow the owner of Mooniswap to return any funds sent to the contract that are not from either of the tokens of the pair.

This check is done by verifying that none of the balances of the tracked tokens have decreased after a `transfer` of the selected token.

However, this function is not equipped with a `nonReentrant` modifier. For tokens which CALL the receiving address upon transfer (such as ERC777 tokens), this opens up an opportunity for the `owner` of Mooniswap to extract funds from the pool:

If the `owner` calls `rescueFunds` with any of the tokens of the pair, from a contract which then reenters `Mooniswap` with a call to `deposit`, they will have received new LP shares without providing sufficient additional liquidity (having only temporarily lent liquidity), while the balance check in `rescueFunds` following the token transfer will still succeed.

Adding a `nonReentrant` modifier to the `rescueFunds` function makes this attack impossible.

Notice that it is still possible for the `owner` to flash lend the assets of the pool, even in the presence of a `nonReentrant` modifier.

Rounding error in calculation of amount to send in `deposit`

The amounts pulled from the user as part of a call to deposit are calculated on L170.

This calculation uses a flooring division operation (`div`), which introduces a precision loss that causes the value of `fairSupply` calculated on L173 to be lower than the value precomputed in the for loop on L163 even if no fee is taken by the token.

This precision loss is then propagated through to the scaled virtual balances on L178, which can in some cases result in an improper mismatch between the real and virtual balances.

The Mooniswap team proposed to fix this issue by rounding up on the division on L173, and we agree that this is an appropriate solution.

Amounts pulled are dependent on token ordering in `deposit`

The amounts pulled from the user are calculated inside a for loop on L170.

If `totalSupply > 0` then the value of `fairSupply` used in the second iteration of the loop has already been modified as part of the call to `min` on L173, meaning that the amount pulled for the second token is always less than it would be if that token was instead in position one. Note that due to the precision loss introduced by the issue above, this is the case even if the tokens do not take a fee.

This ordering dependent behaviour could be avoided by caching the value of `fairSupply` to be used as an input to the transfer amount calculation before the loop begins.

Interactions delay virtual balance convergence

User interactions which include calls to `update` or `scale` on virtual balances have the effect of "resetting" the linear interpolation of virtual balances, resulting in a longer convergence time.

This includes not only calls to `swap` but also `deposit` and `withdraw`, see the related recommendation Do not update `time` in `scale`. Moreover, those calls can be economic no-ops, by calling with zero arguments, allowing an interested party to delay the convergence of virtual balances to some extent.

In a theoretical, continuous time, zero transaction cost setting, virtual balance convergence can be delayed to be arbitrarily slow by repeated interactions. In a practical setting, due to there being a time interval between blocks, and due to transaction costs, it is not possible to delay convergence indefinitely. For example, by sending a `deposit` every 15 seconds, it is possible to make the virtual balances move only 64% of the way to real balances after 5 minutes, and 87% of the way after 10 minutes. At the very least, participants should recognise that balances will not necessarily converge in `DELAY_PERIOD`.

Improvements

Recommendation	Accepted	Commit
Do not update `time` in `scale`	no	-
Remove dynamic arrays	no	-
Generalize benefactor address of `deposit`, `withdraw`, `swap`	no	-
Mooniswap, UniERC20: use WETH to avoid special casing	no	-
Rename `amounts` Array in `deposit`	no	-
Simplify branches in `deposit`	no	-
Prefer `calldata` to `memory` as location for external methods	no	-
Use `immutable` for the `factory` storage variable	no	-
Use stack variables instead of Balances struct	no	-
Use block number instead of timestamp in virtual balance decay	no	-

Do not update `time` in `scale`

Upon `deposit` and `withdraw`, the `scale` function updates the virtual balances for both trading directions by interpolating the line between the actual balances of the pool (`realBalances` after `deposit` / `withdrawal`) and the current virtual balances.

Since `deposit` and `withdrawals` do not change the ratio of the two tokens in the pool, the interpolation between real and virtual balances need not be performed. It is sufficient to scale the virtual balances proportionally to the growth in real balances:

```
function scale(VirtualBalance.Data storage self, uint256 num, uint256 denom) internal {
   self.balance = self.balance.mul(num).div(denom);
}
```

This ensures that `deposit` and `withdrawal` updates virtual balances to maintain a constant ratio to real balances, but leaves the subject of convergence between real and virtual balances to the `update` function called in `swap`.

Remove dynamic arrays

The `Mooniswap.sol` contract uses dynamic arrays to store the token addresses, as well as for parameters to various functions. These arrays are however used to store two values only.

It is the opinion of the audit team that these arrays should be removed and replaced with a type that more clearly reflects and enforces the invariant that a Mooniswap pool holds exactly two tokens.

In addition to their negative impact on readability the use of dynamic arrays incurs a significant gas penalty compared to static approaches.

If token pairs were referred to directly as separate storage variables they could even be stored as `immutable` state variables, eliminating at least 3 SLOAD (2 token addresses + 1 array length) costs per method call.

Generalize benefactor address of `deposit`, `withdraw`, `swap`

When calling `deposit`, `withdraw`, or `swap`, the address receiving the benefit of the function call is always `msg.sender`. For greater generality, the benefactor address could instead be set by the caller, in effect admitting a transfer of funds in combination with these methods. In particular when used by other smart contracts, this would provide a significant gas optimization.

For reference, consider tokenToEthTransferInput of Uniswap v1.

Mooniswap, UniERC20: use WETH to avoid special casing

The UniERC20 contract provides an uniform interface to perform ERC20 methods on tokens or native ETH. We find this abstraction leaky as it fails to account for the fundamental difference between them: ERC20 tokens can be pulled (via `transferFrom`), while native ETH must always be pushed (via a direct ETH transfer).

As a result, the `Mooniswap` contract ends up with plenty of special case logic to account for this: L210, L213, L143, L147.

Using ERC20 wrapped ether (WETH) instead would eliminate the need for special casing ETH, and also reduce the number of calls to unknown code, decreasing system attack surface.

Rename `amounts` Array in `deposit`

The naming of the `amounts` array in `deposit` is somewhat misleading, as the amounts within are used as upper bounds on the amount of tokens that will be pulled from the caller.

We suggest renaming to `maxAmounts` (or similar) to more accurately affect the semantics of the parameter.

This should hopefully make it clearer to consumers of the contract that they will possibly end up transferring less than the `amounts` they specify in their call to `deposit` (and perhaps alert them to the need to account for that in their integration code).

Simplify branches in `deposit`

In the `totalSupply == 0` case, we end up iterating through the `token` array one more time than is necessary.

Prefer `calldata` to `memory` as location for external methods

Saves a small amount of gas for `withdraw` and `deposit`.

Use `immutable` for the `factory` storage variable

Saves SSTORE cost upon deployment and SLOAD cost on calls to `fee()` (and subsequently `swap`).

Use stack variables instead of Balances struct

The `Balances` struct is only ever used as a local variable in the `swap` function to store the balances of the source and destination token before trade. The struct is never referenced as a whole – only its member elements are referenced.

The usage of this struct seems superfluous, and if its containing values were stored on the stack directly it would save approximately 200 gas per swap and increase readability.

Use block number instead of timestamp in virtual balance decay

Since the block timestamp is used when interpolating the virtual balances, and the miner of a block has the ability to manipulate the block timestamp, miners can directly influence the prices received by Mooniswap trades.

It is widely believed that it is impractical for a miner to manipulate a timestamp by much more than 15 seconds. Moreover, the possible impact on the price is limited to improving it up to the price implied by the contract's real balances. Nevertheless, it may be preferable to measure the virtual balance decay in terms of block numbers, rather than timestamps, since block numbers are more difficult to manipulate.

Notes and Miscellanea

- The `solc` optimizer has introduced many issues in the past. It's usage in the Mooniswap contracts increases the risk of exposure to a compiler bug.

Appendix A. Bug Classifications

Severity	
informational	The issue does not have direct implications for functionality, but could be relevant for understanding.
low	The issue has no security implications, but could affect some behaviour in an unexpected way.
medium	The issue affects some functionality, but does not result in economically significant loss of user funds.
high	The issue can cause loss of user funds.

Likelihood	
low	The system is unlikely to be in a state where the bug would occur or could be made to occur by any party.
medium	It is fairly likely that the issue could occur or be made to occur by some party.
high	It is very likely that the issue could occur or could be exploited by some parties.