Security Audit Report

Algofi AMM and Nanoswap

Delivered: March 10, 2022

Prepared for Algofi by

runtime verification

Contents

Summary

Algofi engaged Runtime Verification Inc to conduct a security audit of the smart contracts implementing two types of Automated Market Makers (AMM): a constant-product AMM (CPMM) and a stableswap AMM (Nanoswap).

The objective was to review the contracts' business logic and implementation in PyTeal and identify any issues that could potentially cause the system to malfunction or be exploited.

The audit was scheduled for a period of 7 weeks (18 Jan 2022 - 08 Mar 2022). The first 5 weeks were dedicated to reviewing the CPMM and the subsequent 2 were dedicated to reviewing the Nanoswap. We note here that the two share the same general architecture and the second phase of the audit focuses mainly on the implementation of the stableswap mathematical invariant.

The audit uncovered one medium severity isssue and several informative findings.

Scope

The audit was conducted on the following files provided by Algofi:

- `contracts/manager.py`
- `contracts/pool.py`
- `contracts/stable_pool.py`
- `contracts/stable_swap_math.py`
- `contracts/globals.py`
- `contracts/factory_logic_sig.py`
- `contracts/wrapped_var.py`

The audit focused on the following commit in the Algofi GitHub repository:

Commit: 069b8e638042f63a2fd448e47f5784e07e8066ec

Methodology

The contracts are implemented in PyTeal, a Python EDSL for writing TEAL programs. The audit is performed on the PyTeal code, *not* the compiled TEAL code.

The audit was done manually, with some tooling for testing and simulations.

Two simplified high-level models of the CPMM were written: one in plain Python and the other using an internal tool that facilitates evaluating PyTeal fragments in Python directly. The models were used to develop intuition about useful invariants and simulate test scenarios using a lightweight input format.

Similar models were developed for the Nanoswap. Additionally, alternative solutions to the stableswap invariant were implemented for comparison.

When classifying findings, *severity* refers to the impact of the given attack scenario on the application, while *difficulty* refers to the likelihood of the attack, considering the resources required to execute it. Informative findings do not pose a direct security risk.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contracts only, and make no material claims or guarantees concerning the contracts' operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of these contracts.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Issues and findings

A01. Possible inflation of LP-token value (Severity: Medium, Difficulty: Medium)

Context

It is generally desirable that the value of a single Liquidity Provider (LP) token is low enough to allow comfortable granularity for pooling and burning operations. In particular, because any user must pool assets that are worth at least `1` LP-token, a high-priced token can prohibit some users from contributing to the pool at all.

Issue

It is possible for an early liquidity provider to raise the price of a single LP-token above intended levels. Consider what happens when a user wants to swap a large amount of one asset on a pool with single-digit balances.

```
(1, 3) -> (10^9, X)
```

According to the constant product invariant, `X` should be equal to $\frac{3}{10^9}$. This value must be rounded to an integer. The safe way to round in general is in favor of the protocol. So in practice `X` will be equal to `1` and the resulting product is orders of magnitude larger than the original. Note that during this process the LP-token circulation remains constant, so the existing tokens have simply become more expensive.

Recommendation

The root problem of rounding is difficult to avoid. Instead, the protocol can avoid states with low LP-circulation. Require that any pooling operation executed on an empty pool mints at least some number of tokens. Consider locking these tokens permanently in the contract.[1]

Status

Addressed. The protocol now requires pooling and swap operations to leave at least `1000` assets in both pools. This implicitly enforces a lower bound of `1000` minted tokens on initial pools, as these are issued as the geometric mean of the pooled assets.

[1] A similar issue and recommendation can be found here, although note the technique used (donating to the pool) is different. Pool donations are not possible in Algofi, because the logic does not operate with the blockchain balances directly.

A02. Possible off-by-one calculation in price statistics (Informative)

Context

The pool application computes some cumulative sums for historical price data. When these values exceed Algorand's 64-bit unsigned integers, they are wrapped around. The application front-end then takes care to unwrap them and show the exact values.

Finding

The intuitive way to handle integer overflow wraparound is to effectively compute the values modulo `2^64`. The application code is not equivalent to this, as the value `2^64` is wrapped to `1`. This might cause inaccurate readings if the front-end handles the unwraping differently[2]. Note that this would only affect off-chain price statistics and cannot harm the correctness of the protocol itself.

Status

Addressed. The calculation is now equivalent to arithmetic modulo `2^64`.

A03. Recommended redundancy in checking OnCompletion fields (Informative)

Finding

The pool application does not check that the transaction initializing the pool is a `NoOp ApplicationCall`. In theory, this can cause problems if the field is set to `UpdateApplication`. However, initializing a pool is only meaningful in conjunction with registering the pool with the manager application and the latter *does take care* to check the `OnComplete` field of the initializing transaction in the group. While the check on the manager ensures safety in this case, we recommend that the pool application performs the same check so certain safety invariants can hold for the pool application without assuming external checks from the manager.

A related recommendation is to slightly refactor the existing checks so that admin operations are also strict with regard to the `OnCompletion` field.

Status

Addressed.

[2]Note that off-chain code is not in scope for this audit.

A04. Imperfect invariant for allowed asset ratio (Informative)

Context

The pool application requires a reasonable ratio of the two assets to function correctly. In particular, a ratio *strictly* exceeding `10^9` may cause panic behavior for some operations.

The pool application performs a check verifying that the asset ratio is *strictly* less than `10^9` after every pool and swap operation.

Finding

The final ratio check is not performed for burn operations and it is in general possible that burn operations can increase the asset ratio, due to integer division rounding. After some analysis, we believe that burn operations can cause the asset ratio to become *equal* to `10^9`, but not exceed it.

This means we currently see no attack vector based on exploiting this missing check. However, we deem the invariant to be "imperfect" because proving it in the case of the burn operation is non-trivial and therefore non-trivial to maintain should the code change in the future.

Recommendation

Pool states for which the above situation becomes relevant are improbable in practical markets. For this reason, we think it's reasonable to not complicate the logic of the application in order to attain an invariant that is easier to prove, but the point should be noted in case of updates to the code.

Status

Currently left as is, per recommendation.

A05. Unfavorable rounding in computation of stableswap invariant (Informative)

Finding

The stableswap invariant is solved using an iterative method. We found the final result to be empirically precise. However, the small errors that do occur can often benefit the user. It is generally advisable that rounding favors the protocol, if this is easy to achieve.

Recommendation

Increment the final result of the `compute_other_asset_output_stable_swap` function by `1`. Empirically, this seems to improve the precision of the calculations involved in the invariant.

In particular, it improves the chance that the value `D` [3] will not decrease over a sequence of swaps [4].

Status

Addressed.

[3] The value `D` in the invariant represents the total amount of coins when the assets have equal price. Mathematically, this value should remain constant after swap operations. In practice, swaps will change it slightly, due to the error inherent to solving the equation in integers. It is a good practice to err on the side of it increasing, if possible.

[4] The chances of `D` decreasing in practice is largely mitigated by the fee taken by the pool, which is non-zero in practical scenarios. However, we think improving the protocol's behavior in a `0`-fee scenario will increase its general robustness.