Least Authority
PRIVACY MATTERS

Cosmos SDK Liquidity Module

Security Audit Report

All in Bits

Final Audit Report: 24 June 2021

Table of Contents

Overview

- Background
- Project Dates
- Review Team

Coverage

- Target Code and Revision
- Supporting Documentation
- Areas of Concern

Findings

- General Comments
  - System Design
  - Code Quality
  - Documentation
  - Scope
- Specific Issues & Suggestions
  - Issue A: Security of Truncated SHA256 in Address Generation
  - Issue B: Linter Reports Errors
- Suggestions
  - Suggestion 1: Remove Redundant Assertions
  - Suggestion 2: Move Minting and Transfers to Separate Function
  - Suggestion 3: Perform and Document Theoretical Analysis of Liquidity Provider Wealth
  - Suggestion 4: Remove Redundant CLI Queries
  - Suggestion 5: Remove Unused Code
  - Suggestion 6: Increase and Improve Test Cases
  - Suggestion 7: Improve Error Handling
  - Suggestion 8: Separate Balance Checks
  - Suggestion 9: Improve Documentation
  - Suggestion 10: Caution on Using Strict Equality Check
  - Suggestion 11: Add Check on Demand Coin's Balance

About Least Authority

Our Methodology

Overview

Background

All in Bits has requested that Least Authority perform a security audit of the Cosmos SDK Liquidity Module, a Decentralized Finance (DeFi) application that allows the implementation of a token Decentralized Exchange (DEX) on any Cosmos SDK based network.

Project Dates

- **April 26 - May 24**: Code review *(Completed)*
- **May 27**: Delivery of Initial Audit Report *(Completed)*
- **June 21 - 23:** Verification *(Completed)*
- **June 24:** Delivery of Final Audit Report *(Completed)*

Review Team

- Anna Kaplan, Cryptography Researcher and Engineer
- Bryan White, Security Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer
- Suyash Bagad, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Cosmos SDK Liquidity Module followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- Core module: https://github.com/tendermint/liquidity/releases/tag/v1.2.4
    - Key priorities:
        - CLI: https://github.com/tendermint/liquidity/tree/develop/x/liquidity/client/cli
        - API code (which is auto-generated using gRPC): https://github.com/tendermint/liquidity/blob/develop/x/liquidity/types/tx.pb.go#L1205-L1214
- Proto: https://github.com/tendermint/liquidity/tree/develop/proto
- App: https://github.com/tendermint/liquidity/tree/develop/app

Specifically, we examined the Git revisions for our initial review:

```
5ec243fa055df853a2c7df6e0946af6178cf440c
```

For the verification, we examined the Git revision:

```
4c6ca225477d77beb5c633cce332499a836e0710
```

For the review, these repositories were cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/tendermint-liquidity

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- RFC: rfc.pdf provided to Least Authority via email on 11 February 2021
- README.md: https://github.com/tendermint/liquidity/blob/develop/README.md
- Liquidity Module Specification: https://github.com/tendermint/liquidity/tree/develop/x/liquidity/spec
- Swagger HTTP API documentation: https://app.swaggerhub.com/apis-docs/bharvest/cosmos-sdk_liquidity_module_rest_and_g_rpc_gateway_docs/2.2.1
- Go Documentation: https://pkg.go.dev/github.com/tendermint/liquidity
- D. Byun, H. Lee, 2021, "Proposal for Cosmos based Automated Market Maker." [BL21]
- Client.md: https://github.com/tendermint/liquidity/blob/develop/doc/client.md
- D. Byun, H. Lee, 2021, "Gravity DEX Economic Simulation." [BL21]
- Economic-simulation.py: https://github.com/b-harvest/Liquidity-Module-For-the-Hub/blob/master/economic-simulation.py
- Performance Testing for Liquidity Module: https://github.com/tendermint/liquidity/blob/develop/doc/Performance%20Testing%20for%20Liquidity%20Module.pdf

In addition, this audit report references the following documents:

- Blogpost, "Understanding Uniswap Returns" [P19]
- A. Evans, 2020, "Liquidity Provider Returns in Geometric Mean Markets." *arXiv:2006.08806v4 [q-fin.MF]* [E20]
- G. Angeris, H. Kao, R. Chiang, C. Noyes, T. Chitra, 2019, "An Analysis of Uniswap Markets." *arXiv:1911.03380v7 [q-fin.TR]* [AKC+21]
- M. Tassy, D. White, "Growth Rate of a Liquidity Provider's Wealth in XY = c Automated Market Makers." *https://math.dartmouth.edu/* [TW20]
- S. Vijayakumaran, 2017, "An Introduction to Bitcoin." *Cryptocurrency, 2018, IIT Bombay, Lecture notes* [V17]
- G. Angeris, T. Chitra, 2020, "Improved Price Oracles: Constant Function Market Makers." *arXiv:2003.10001v4 [q-fin.TR]* [AC20]
- A. Park, 2021, "The Conceptual Flaws of Constant Product Automated Market Making." *http://dx.doi.org/10.2139/ssrn.3805750* [P21]

Areas of Concern

Our investigation focused on the following areas:

- General
  - Correctness of the implementation;
  - Adversarial actions and other attacks on the Liquidity Module;
  - Potential misuse and gaming of the Liquidity Module;
  - Attacks that impacts funds, such as the draining or the manipulation of funds;
  - Mismanagement of funds via transactions;
  - Denial of Service (DoS) and other security exploits that would impact the Liquidity Module's intended use or disrupt the execution;
  - Vulnerabilities in the liquidity module code, particularly for swapping, earning, and creating pools;

    - Vulnerabilities in the interaction between Liquidity Module and other existing Cosmos-SDK modules;
    - Protection against malicious attacks and other ways to exploit the Liquidity Module;
    - Inappropriate permissions and excess authority;
    - Data privacy, data leaking, and information integrity;
    - Secure interfaces to the application (API and CLI);
- Network Interactions
    - Attacks on the Liquidity Module through the network and secure communication between the Liquidity Module and network components;
    - Spam attacks that bottleneck the functionality and the entire network by exhausting computation power or traffic bandwidth, leading to a monopoly of submitting transactions to the network (minimal competition among participants can lead to extreme inefficiency of price discovery);
- Economic
    - Review of the economic incentives, including the intended functions and potential impact;
    - Weaknesses in the economic model resulting in user attacks again the Liquidity Module;
    - Attack vectors that can game the trading environment and cause unnecessary costs to ordinary users; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

The Cosmos SDK Liquidity Module allows users to create a liquidity pool with any pair of tokens, participate in liquidity provision by depositing reserve tokens into the liquidity pool, and trade tokens using the liquidity pool. The Liquidity Module is a hybrid model that combines a Uniswap-like Automated Market Maker (AMM) with a traditional order book based exchange system.

System Design

It is clear that security has been considered in the design and implementation of the Liquidity Module. This is demonstrated by accurate and comprehensive documentation, well-organized and structured code, a considerable amount of test coverage, and adequate code comment coverage. In addition, sufficient checks have been implemented to identify and prevent unintended behavior. These attributes adhere to widely accepted design and development best practices, and contribute to the ability for security researchers to understand the intended function and behavior of the system, thus facilitating the ability to reason about potential security vulnerabilities.

Areas of Investigation

Our team conducted a thorough investigation and analysis of the design and implementations for liquidity pool creation, deposit of coins, swapping of coins, and withdrawals from the liquidity pool. We examined decimal handling and the truncation of passed values. In addition, we assessed the cryptography implemented in the Liquidity Module. In reviewing these components, we did not identify critical security vulnerabilities.

Furthermore, our team analyzed and reviewed the results provided in the Gravity DEX Economic Simulation report [BL21], reviewed the python script on the economic simulation, and reviewed the Performance Testing for Liquidity Module document included in the project documentation. These resources further supplemented our understanding and aided us in our review and analysis of the Liquidity Module's system design and implementation.

SHA-256 in Address Generation

We identified concerns around the security implications of the Tendermint Liquidity Pool's use of a truncated SHA-256 in address generation, which is used to ensure that different pools have unique reserve account addresses. In particular, the SHA-256 hash output is truncated to 20 bytes (or 40 hex characters) before the account address is generated, which effectively reduces the security provided by the SHA-256 hashing algorithm. As a result, we recommend conducting thorough analysis of the security of the truncated SHA-256 hash function and exploring alternative hash functions, which allow variable output digest size. For example, we suggest exploring a combination of SHA-256 with RIPEMD-160 hash functions, which could provide better preimage and collision resistance (Issue A).

Equivalent Swap Price Model

The Liquidity Module is a Cosmos SDK implementation of an AMM system with a novel economic model called the Equivalent Swap Price Model (ESPM). The key distinguishing feature of the ESPM model from the Constant Product Market Maker (CPMM) model (e.g. Uniswap) is the implementation of a hybrid system. This system combines an orderbook model exchange and a simple liquidity pool model by which the order book is governed by a set of order rules and execution is performed in batches. In the ESPM, the pool price is always equal to the last swap price, thus reducing opportunities for arbitrage.

The ESPM model is intended to provide protection against price volatility, transaction ordering vulnerabilities, and losses due to arbitrage, which are widely known vulnerabilities in AMMs such as Uniswap. Although the ESPM theoretically addresses transaction ordering issues and reduces arbitrage opportunities, the price volatility reduction does not account for risk of impermanent loss. As an alternative approach to stabilize prices in liquidity pools with high volatility tokens, the Tendermint team is considering weighted liquidity pools where an investor can deposit tokens in a pool according to a weighting factor, which may be implemented as an alternative incentive mechanism [BL21]. We acknowledge that such changes to the design are not insignifiant and require careful research and analysis prior to implementation.

Our team was not able to identify any specific security vulnerabilities in the ESPM model, however, it is difficult to predict the security implications of this AMM model implementation. Some of the existing AMM models (e.g. Uniswap and Balancer) have had the opportunity to mature in production, have undergone extensive security audits by multiple independent security research teams, and are the subject of extensive ongoing research [AKC+19, AC20, P21]. As a result, while they continue to face challenges with security, those AMM models are more predictable from a security perspective. We recommend the Tendermint team continue to carefully consider the security implications of the ESPM model and closely monitor new research.

Fuzz Testing

As a supplement to our manual review of the code, our team identified prospective fuzz testing targets and prioritized the prospective targets list based on potential severity. We implemented fuzz tests on the following targets:

- `MsgSwapWithinBatch`
- `MsgWithdrawWithinBatch`
- `MsgDepositWithinBatch`
- `GenesisState`

We did not identify any security vulnerabilities from the results of the fuzz tests. However, we recommend that the Tendermint team perform additional fuzz testing, in addition to ensuring that linters are running in the Continuous Integration (CI) pipeline and follow up on reported errors (Issue B).

Code Quality

The code is generally well-written, organized, and adheres to the conventional Cosmos SDK project structure. However, we have identified several suggested areas of improvement that would contribute to a more robust code base (Suggestion 1; Suggestion 2; Suggestion 4; Suggestion 5; Suggestion 7; Suggestion 8; Suggestion 10; Suggestion 11). We recommend implementing these suggestions, as the application of best practices reduces the opportunity for errors and bugs in the code base, which may lead to unintended consequences in the execution of the code.

Tests

We observed a considerable amount of test coverage in the code base, however, we recommend expanding test coverage to include all components (e.g. `CheckSwapPrice`, in addition to uncovered branches of `FindOrderMatch` and `OrderBook#CalculateSwap` in the `x/liquidity/types` package). Furthermore, we recommend reorganizing the test suite and properly defining function tests, failure cases, and edge cases (Suggestion 6).

Documentation

The documentation provided by the Tendermint team significantly supplemented our understanding of the core algorithm of the Liquidity Module. The code base is well commented, with sufficient comments explaining the intended functionality of each of the components.

However, we identified opportunities where the documentation could be expanded and further improved. The Gravity DEX Economic Simulation report [BL21] facilitated our understanding and reason about the economic concepts introduced by the ESPM. In particular, the simulation analyzed pool returns, impermanent losses, and arbitrage opportunities in the context of the Liquidity Module. However, we recommend building upon the economic simulation report by adding necessary formulae in each of the above topics and defining used terms for helping the reader to better understand the simulation results, in addition to improving the term definitions and references (Suggestion 9). Finally, we suggest performing and expanding the documentation to include a rigorous theoretical analysis to demonstrate how the wealth of a liquidity provider changes over time, compared with traditional AMM models (Suggestion 3).

The Proposal for Cosmos based Automated Marker Maker Lite Paper [BL21] explains the differences between general CPMM models to the proposed ESPM model. We found that some terminology in the existing documentation has yet to be defined and that variables are not declared but used and only explained through context. As a result, we recommend increasing definitions throughout the Lite Paper (Suggestion 9).

Comprehensive documentation coverage facilitates a clear understanding of the system for both security reviewers of and contributors to the project, thus minimizing the possibility of missed vulnerabilities and errors in the implementation.

Scope

We found the scope of the security audit to be sufficient in that encompassed all security critical components of the Liquidity Module. In addition, we did not identify any third-party dependency concerns.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Security of Truncated SHA256 in Address Generation	Unresolved
Issue B: Linter Reports Errors	Resolved
Suggestion 1: Remove Redundant Assertions	Resolved
Suggestion 2: Move Minting and Transfers to Separate Function	Resolved
Suggestion 3: Perform and Document Theoretical Analysis of Liquidity Provider Wealth	Unresolved
Suggestion 4: Remove Redundant CLI Queries	Resolved
Suggestion 5: Remove Unused Code	Resolved
Suggestion 6: Increase and Improve Test Cases	Resolved
Suggestion 7: Improve Error Handling	Resolved
Suggestion 8: Separate Balance Checks	Resolved
Suggestion 9: Improve Documentation	Unresolved
Suggestion 10: Caution on Using Strict Equality Check	Partially Resolved
Suggestion 11: Add Check on Demand Coin's Balance	Resolved

Issue A: Security of Truncated SHA256 in Address Generation

Location

`liquidity/keeper/liquidity_pool.go#L883`

Synopsis

Each Tendermint Liquidity Pool requires a unique reserve account address to keep track of the funds deposited or withdrawn from the pool. At present, a SHA-256 hash function is used to ensure that different pools have unique reserve account addresses.

For example, for a pool with tokens `tokenX` and `tokenY`, and pool identifier `poolId`, the reserve account address is computed as:

```
    sdk.AccAddressFromHex(SHA256("denomX/denomY/poolId")[:40])
```

The SHA-256 hash output is truncated to 20 bytes (or 40 hex characters) before the account address is generated. This effectively reduces the security provided by the SHA-256 hashing algorithm. Although the address generation requires 20 bytes as the input, we recommend analyzing the security implications of using truncated SHA-256 outputs.

Impact
It is possible for an attacker to identify input strings to the SHA-256 hash function, such that the first 20 bytes of the output matches. The result is that two different liquidity pools, both possibly created by the attacker, could have the same reserve account address. This would directly impact the deposit, withdraw, and swap functionalities of these two pools in the Liquidity Module. The truncated SHA-256 is widely used in the Cosmos ecosystem for deterministic address generation, however, in the particular case of the Liquidity Module, the negative security implications of using truncated SHA-256 exceed the benefit of simple address generation.

Remediation
While the likelihood of an attack is theoretically very small (2^{80} computations), we recommend conducting a thorough analysis of the security of the truncated SHA-256 hash function. The NIST provides guidelines on secure use of truncation in approved hash functions, such as SHA-256. We also recommend exploring alternative hash functions, such as Keccak, which allow variable output digest size. For example, we suggest exploring a combination of SHA-256 with RIPEMD-160 hash functions, which could provide better preimage and collision resistance. The SHA-256 and RIPEMD-160 combination is used in the generation of P2PKH addresses in Bitcoin [V17].

Status
The Tendermint team has responded that the SHA-256 truncation problem is not specific to the Liquidity Module and, as a result, they will not be making any changes. Given that our primary aim was to caution them about this, we consider the decision to leave the issue unresolved at the time of this verification to be acceptable.

Verification
Unresolved.

Issue B: Linter Reports Errors

Location
`tendermint-liquidity/issues/2`

Synopsis
There is a `lint` Make target which uses `golangci-lint`, an industry standard linting tool for Go. We found that the linter currently reports multiple errors. The Tendermint team has indicated that the linter results are not currently considered in their CI pipeline, however, they stated that they intend to remediate that.

Impact
`golangci-lint` runs a myriad of checks for common issues, which could have serious performance or critical security implications, depending on the specific error. We suggest referring to the linter documentation for additional information.

Feasibility
There is an increased risk that unlinted code is deployed, resulting from the absence of an important code step during the development process.

Technical Details

The `lint` Make target calls `golangci-lint` without a configuration file, which runs the default linters with their respective default configurations. However, it is not included in the GitHub Actions workflow, which the Tendermint team is currently using for CI/CD.

Mitigation

We recommend the Tendermint team consider adding review requirements to the development process (e.g. two unique approvals to merge each PR). Until the linter is included in the CI pipeline, review should include running the linter and ensuring no errors are reported.

Remediation

We recommend the Tendermint team Include the linter in the CI pipeline and review all errors for potential security vulnerabilities.

Status

The Tendermint team has included the linter in the CI pipeline and fixed all outstanding linter errors.

Verification

Resolved.

Suggestions

Suggestion 1: Remove Redundant Assertions

Location

`liquidity/keeper/liquidity_pool.go#L158`

`liquidity/keeper/liquidity_pool.go#L355`

`liquidity/keeper/liquidity_pool.go#L215`

Synopsis

There are redundant checks verifying the existence of the pool, which are unnecessary as the check is performed in the `ValidateMsgDepositLiquidityPool` and `ValidateMsgWithdrawLiquidityPool` functions.

Furthermore, since most of the checks in the context of deposit, withdrawal, or swap are aggregated in the above individual functions, the checks on the equality of the number of deposit coins (`DepositCoins`) and the number of pool reserve coins (`reserveCoins`) could also be moved in those functions.

Mitigation

We recommend removing the redundant checks to avoid confusion and to simplify the code.

Status

The Tendermint team has removed redundant assertions and have simplified storage of the liquidity pool state using its reserve account index.

Verification

Resolved.

Suggestion 2: Move Minting and Transfers to Separate Function

Location

`liquidity/keeper/liquidity_pool.go#L182-L196`

Synopsis

While depositing tokens in a new pool (by creating a pool) or in an existing pool, new pool coins are minted and transferred to the pool creator or the depositor, respectively. Additionally, the deposited amount is transferred from the depositor to the reserve account of the pool. This workflow is repeated in the functions `CreatePool` and `DepositLiquidityPool`. For better code composability, the process of minting and transfers could be moved to a separate function.

Mitigation

We recommend moving the minting and transfers into a separate function with inputs as `depositCoins` and `mintPoolCoin`. Furthermore, we recommend testing that function in isolation to check its correct usage in the liquidity pool functionalities.

Status

The Tendermint team has incorporated the suggested mitigation and has improved the code comments and nomenclature.

Verification

Resolved.

Suggestion 3: Perform and Document Theoretical Analysis of Liquidity Provider Wealth

Location

Gravity DEX Economic Simulation report [BL21]

Synopsis

The Liquidity Module is a newly proposed AMM model (i.e. ESPM), which allows for batch execution of swap orders. In contrast to the CPMM model, the pool price after a swap equals the swap price in the ESPM model. The report compares the performance of the CPMM and ESPM models in terms of liquidity providers' gains, impermanent losses, and arbitrage opportunities under various market conditions. The report claims that in most market situations, the Liquidity Module's pool investors enjoy significantly larger returns compared to the CPMM model. Although the report presents empirical evidence that the ESPM model performs better than CPMM in different market conditions, supplementing the simulation with rigorous theoretical analysis of the gains of the liquidity module's pool investors would be beneficial.

Remediation

The quantification and analysis of the wealth of liquidity providers in the CPMM model (eg. Uniswap) is a very active area of research in the DeFi space [P19, E20, AKC+21]. The recent work on "Growth Rate of a Liquidity Provider's Wealth in XY = c Automated Market Makers" [TW20] theoretically proves that, under certain assumptions on the price model of an asset, the growth rate of the Liquidity Providers' (LP) wealth would always be better than that of an unbalanced portfolio of the two assets (i.e. HODLers). This was an important breakthrough in answering the difficult question of whether LPs or HODLers fare better in terms of wealth growth. We recommend that a similar analysis be performed for the ESPM model to quantitatively prove if the liquidity module investors would fare better than HODLers in terms of wealth growth.

It is important to note that the result from [TW20] is based on the idea of modelling the pool price (`p(t) = X(t)/Y(t)`) as a Markov process. This is possible because the pre-swap and post-swap price of the pool is *always* different in CPMM (so as to maintain the constant product `X(t)*Y(t) = k` of the reserves). In the Liquidity Module, the pre and post swap prices are always equal, which enables batch execution of orders. Thus, the results from [TW20] cannot be trivially applied to the ESPM model. Nevertheless, we recommend that efforts be made to theoretically derive a closed form expression for the growth of Liquidity Module's pool investors and realize market conditions in which the LPs are to do better than HODLers. From an economic perspective, this is crucial to gain the confidence of LP investors for providing liquidity in the module.

Status

The Tendermint team has stated that their simulation report shows that pool investors benefit more in the ESPM model as compared to the CPMM model, and from the results of [TW20], pool investors perform better than HODLers under certain market conditions. As a result, they conclude that pool investors in the ESPM model would benefit more than HODLers. While such conclusions can be drawn only under specific market conditions, we nevertheless recommend that the Tendermint team continue to conduct research and analyses on theoretical comparisons of wealth growth of pool investors and HODLers specific to the ESPM model.

Verification

Unresolved.

Suggestion 4: Remove Redundant CLI Queries

Location

`client/cli/query.go#L118`

`client/cli/query.go#L203`

Synopsis

`GetCmdQueryLiquidityPool` and `GetCmdQueryLiquidityPools` command implementations call `queryClient.LiquidityPool` and `queryClient.LiquidityPools` respectively, twice.

Mitigation

We recommend removing the redundant calls to avoid confusion and to simplify the code.

Status

The Tendermint team has removed the redundant calls.

Verification

Resolved.

Suggestion 5: Remove Unused Code

Location

`liquidity/module.go#L85`

Synopsis

The Cosmos SDK AppModuleBasic interface defines a `RegisterGRPCGatewayRoutes` member, which is implemented by the liquidity `AppModuleBasic`. The liquidity `AppModuleBasic` also implements a `RegisterGRPCRoutes` member, which is not part of the interface. In addition, to the best of our knowledge, this method is not called anywhere.

Mitigation

We recommend removing the unused code to avoid confusion and to simplify the code.

Status

The Tendermint team has removed the unused code.

Verification

Resolved.

Suggestion 6: Increase and Improve Test Cases

Location

Examples, not exhaustive:

`liquidity/keeper/batch_test.go#L648-L649`

`liquidity/keeper/invariants_test.go#L78-L80`

Synopsis

While existing test coverage is thorough, we recommend identifying test cases and separating tests according to correct case, failure case, and edge case scenarios. These three categories should cover intended functionality for correct case tests, unintended functionality for failure case tests, and functionality of the implementation in edge cases tests.

For example, our team identified inconsistent evaluations in the following examples:

- `liquidity/keeper/batch_test.go#L648-L649`
- `liquidity/keeper/invariants_test.go#L78-L80`

These inconsistencies result from unclear expectations from tests and would improve from an analysis and implementation of test case scenarios.

Mitigation

We recommend separating the test suite into correct case, failure case, and edge case tests, in addition to verifying that all cases are sufficiently covered.

Status

The Tendermint team has issued pull requests 348 and 389, thus improving tests by including sufficient test cases.

Verification

Resolved.

Suggestion 7: Improve Error Handling

Location

liquidity/keeper/liquidity_pool_test.go#L60

Synopsis

The assertion library handling could be improved throughout the code base. In the example above, `ErrorIs` or `ErrorEqual`, rather than `Error`, should be used to check whether a thrown error is equal to the input assertion check. A correct assertion library implementation reduces the risk of unintended behavior.

Mitigation

We recommend reviewing the codebase and optimizing assertion library handling.

Status

The Tendermint team has improved error handling, thus optimizing assertion library handling.

Verification

Resolved.

Suggestion 8: Separate Balance Checks

Location

liquidity/keeper/liquidity_pool.go#L85

liquidity/keeper/liquidity_pool.go#L94

liquidity/keeper/liquidity_pool.go#L731

Synopsis

While creating a new liquidity pool, a check is performed to verify the pool creator has sufficient balance amounts of three assets (i.e. the two tokens forming the pool and the pool creation fee token). The balances of the pool tokens are checked at once while the pool creation fee token's balance is checked separately. In addition, the check on the pool's tokens happens twice at lines 85 and 94.

Similarly, in the `ValidateMsgWithdrawLiquidityPool` function at line 731, two checks are performed at once (i.e. the pool coin's total supply is positive and the amount of reserve coins in the pool is positive). Failure of this check returns the error `ErrDepletedPool`. The error reporting may be performed separately, specifically for the two different cases.

Mitigation

We recommend implementing separate balance checks on each token to improve error reporting. Separate balance checks result in separate error messages for each of the tokens (e.g. `ErrInsufficientBalanceTokenX`, `ErrInsufficientBalanceTokenY`, `ErrInsufficientPoolCreationFee`). Individual balance checks would imply that no redundant balance checks are performed.

Additionally, we suggest that error descriptions are made as explicit as possible for the benefit of the end user.

Status

The Tendermint team has implemented necessary separation of balance checks with explicit error messages.

Verification

Resolved.

Suggestion 9: Improve Documentation

Location

Gravity DEX Economic Simulation report [BL21]

Proposal for Cosmos based Automated Market Maker Lite Paper [BL21]

`tendermint-liquidity`

Synopsis

Cosmos based Automated Marker Maker Lite Paper

The Proposal for Cosmos based Automated Marker Maker Lite Paper explains the differences between general CPMM models to the proposed ESPM model. While these discussions are very helpful for the development team, it is important that future development can refer to common terms and concepts. Some of the terminology has yet to be defined In the existing documentation.

In addition, variables are not declared but used and only explained through context. While this can be acceptable for an experienced reader, clear definitions and precise implications improve readability and promote understanding.

Gravity DEX Economic Simulation

The report is missing clear terminology, numerical values in figures, additional documentation on value selection in Chapter 7, and further explanation on theoretical expectation in Chapter 8.5 of the report.

Code Base

Throughout calculations in the code base, decimal handling is used through truncations in divisions and multiplications. Due to the complicated nature of these calculations, written out documentation of these calculations is advised as a best practice of secure development.

Furthermore, the fraction of ownership of a depositor in the Liquidity Module is computed as `minimum{ depositAmountX/reserveAmountX, depositAmountY/reserveAmountY}`. Although the difference between the two fractions here is fairly small, it is essential to document the reasoning behind the choice of the `minimum` function for the perusal of potential pool investors.

Mitigation

We recommend the Tendermint team further expand documentation and definitions by implementing the following improvements:

Cosmos based Automated Marker Maker Lite Paper

We recommend increasing definitions throughout the Lite Paper (e.g. defining the terms of prices: `global price`, `swap price`, `pool price`, `Swap price`, `Pool price`, `post swap pool price`, `execution price`, `order price`).

We also suggest expanding the explanations of figures (e.g. Figure 4.3 of the Lite Paper).

Gravity DEX Economic Simulation
We recommend increasing definitions throughout the report (e.g. on Impermanent Loss/Impermanent Return, Pool Return, and Pool Return EX IR).

Additionally, we recommend revising figures to include numerical values instead of bar charts and further explaining intuition behind value selection in Chapter 7, next to describing the theoretical expectation in Chapter 8.5 of the report.

Code Base
We recommend writing out decimal handling used through truncations in divisions and multiplications throughout the calculations in the code base (e.g. this example), and document the reasoning behind the usage of the `minimum` function in this example.

Status
The Tendermint team has provided a plan for updating the documentation, specifically noting the following:

- Terminology;
- Variable declarations;
- Improvement in figures;
- Increased details of value selection in Chapter 7;
- Further explanation of theoretical expectation in Chapter 8.5; and
- Calculation documentation in the codebase.

We commend the Tendermint's team to diligently plan forward as it relates to improving the documentation. However, the plan details were not implemented at the time of the verification and, as such, the suggestion remains unresolved.

Verification
Unresolved.

Suggestion 10: Caution on Using Strict Equality Check

Location
`liquidity/keeper/liquidity_pool.go#L776`

Synopsis
In the `ValidateMsgSwapWithinBatch` function, there is a check on the amount of the offer coin's fee which ensures that the user has submitted the correct maximum fee.

Specifically, we check:

```
OfferCoinFee == OfferCoinAmount * (swapFee / 2)
```

where `swapFee` is set to 0.3%. The computation on the right hand side is done using the function `GetOfferCoinFee`. Since it involves a division and a multiplication operation, there might be a negligible precision loss. In the event of a precision loss, the input `OfferCoinFee` from the user might not precisely match the right hand side, leading to an unexpected error. In general, equality checks with division operations on one of the sides must be avoided. Instead, the differences between the two sides must be ensured to be negligibly small.

It is important to note that if the swap requests are generated using the liquidity module's CLI, this would not result in unexpected errors because internally, the CLI uses the same `GetOfferCoinFee` function to compute `OfferCoinFee` for a user. In case the user uses some other way to generate a swap message, this could lead to an unexpected error flag.

Mitigation

Avoid using strict equality checks when comparing two quantities, which might be the result of floating-point divisions and multiplications. Instead we suggest checking if the difference between the quantities is negligible in such cases.

Status

The Tendermint team has reviewed all instances of strict equality checks and concluded that, currently, this issue is not a priority concern. For the specific case of `OfferCoinFee`, a strict equality check will not be an issue if users use the Liquidity Module's CLI. However, in future development, the Tendermint team has stated their intended plans to accept `OfferCoinFee` amount higher than the required amount, and will therefore remove the strict equality check.

Verification

Partially Resolved.

Suggestion 11: Add Check on Demand Coin's Balance

Location

`liquidity/keeper/liquidity_pool.go#L759-L764`

Synopsis

While executing a swap order in the `ValidateMsgSwapWithinBatch` function, the amount of `OfferCoin` in the pool is confirmed as non-zero. This is necessary to ensure that swap requests to a pool which has no `OfferCoin` left in the reserves, are not executed. A similar check should be performed on the `DemandCoin` as well. Consider an edge case where a user orders to swap token X for token Y. If the amount of Y in the pool is 0, this would mean that the swap price is infinite for the user. Since there are no checks on the `DemandCoin`'s amount in the pool, if such a swap request is processed, the user would effectively receive 0 amount of token Y in return (according to the ESPM model)and potentially lose the X tokens. Although the possibility of such a scenario in practice might be insignificant, the algorithm should throw an error to stop the execution of such swaps.

Mitigation

We recommend adding a simple check to ensure the `DemandCoin` amount in the pool is non-zero. We recommend changing the aforementioned code to the following:

```
// check if the balances of offer & demand coins in the pool are
non-zero

reserveOfferCoinAmt := k.GetReserveCoins(ctx,
pool).AmountOf(msg.OfferCoin.Denom)

reserveDemandCoinAmt := k.GetReserveCoins(ctx,
pool).AmountOf(msg.DemandCoinDenom)
```

```
        if !reserveOfferCoinAmt.IsPositive() ||
        !reserveDemandCoinAmt.IsPositive() {

                return types.ErrDepletedPool

        }
```

For better error reporting, we recommend separating the checks on the amounts of `OfferCoin` and `DemandCoin` with custom error messages.

Status

The Tendermint team has changed the logic to determine if a pool is depleted. A pool is said to be depleted only if the pool coin supply becomes zero. This enables a depleted pool to correctly reject deposit or swap requests. In addition, the Tendermint team has also added tests relevant to the new definition of a depleted pool.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.