



Security Audit Report

Osmosis Q2 2023

Authors: Aleksandar Ljahovic, Ivan Golubovic

Last revised 23 June, 2023

Table of Contents

Audit overview.....	1
The Project	1
Scope of this audit	1
Conducted work	1
Conclusions	1
Audit dashboard.....	2
Target Summary	2
Engagement Summary	3
Severity Summary	3
System Overview.....	4
Concentrated liquidity	4
Threat inspection	10
User categories	10
Threats	10
Findings	14
Division by zero protection in distribute.go	15
Insufficient validation on gauge creation	17
Optimization opportunity in addToPosition for last position withdrawal	19
Missing validation for negative number of shares when creating accumulator position	21
Redundant checks	23
Store loading optimization	24
Minor code improvements	26
Redundant condition check in loop for qualifyingLiquidity evaluation	27
Alignment issue between comments and code in tick comparison logic	29
Redundancies in GetAllPositionIdsForPoolId() function and ensurePositionOwner() usage	31
Inefficient string concatenation	33
Inefficient comparison methods and unnecessary object creation	35
Appendix: Vulnerability Classification	37
Impact Score	37
Exploitability Score	37

Severity Score	38
Disclaimer.....	40

Audit overview

The Project

Osmosis has created an innovative take on concentrated liquidity. Drawing inspiration from Uniswap's Automated Market Maker (AMM) design, Osmosis introduces a novel approach that enhances capital efficiency by 200-300 times compared to traditional AMMs. By allowing Liquidity Providers (LPs) to concentrate their capital within specific price ranges, Osmosis ensures assets closely align with their desired spot prices. This targeted liquidity provision offers reduced price impact, increased market stability, and opens up new possibilities for incentivizing LPs based on their proximity to the current price and position duration. With its groundbreaking implementation of concentrated liquidity, Osmosis pioneers more efficient and robust decentralized exchange mechanisms within the DeFi ecosystem.

Scope of this audit

Initially scheduled from May 15, 2023, to June 12, 2023, the audit timeline was extended to June 21, 2023, in agreement with Osmosis. The extension allowed for a thorough examination of subsequent changes made after the initial commit, as well as critical issues identified by Osmosis. The audit team consisted of the following personnel:

- Ivan Golubovic
- Aleksandar Ljahovic

During the audit, our focus was on analyzing the x/concentrated-liquidity module and related changes in x/incentives, x/superfluid, x/gamm, osmoutils/accum, and other relevant areas concerning the concentrated liquidity concept.

Conducted work

The audit project encompassed the following activities:

- Manual code inspection of the x/concentrated-liquidity module and associated changes in other components. We conducted a thorough examination of the codebase, documenting our insights in the "System Overview" section. The manual code inspection revealed the majority of the findings presented in this report.
- Where possible, we attempted to reproduce the findings using Osmosis' integration/unit/end-to-end test suite. Any relevant tests used to reproduce the findings are detailed in the respective sections.

Conclusions

Overall, we found the codebase to exhibit exceptional quality, characterized by well-structured and comprehensible code. The comprehensive test suite includes unit, integration, and valuable end-to-end testing environments. During the audit, we identified 1 high severity finding, along with several medium, low, or informational severity findings. Based on our analysis, we recommend the following areas for immediate and relatively low effort improvements:

- Optimization in data storage: Some findings highlight potential improvements in how KV stores are utilized. Small adaptations can yield significant enhancements in storage mechanisms.
- Performance optimizations: Another noteworthy area pertains to findings related to performance improvements. Although these changes may be small individually, their cumulative impact can lead to substantial overall improvements over time.

Audit dashboard

Target Summary

- **Type:** Specification and Implementation
- **Platform:** Golang
- **Artifacts:**
 - **Commit hash:** 9e1ca7beb15efaf027282946b7231a4bdfc4f32f
 - Additional list of PRs to be reviewed:
 - Key malleability - possible to claim ownership of someone else's position: <https://github.com/osmosis-labs/osmosis/pull/5467>
 - Tick rounding issue causing funds drained: <https://github.com/osmosis-labs/osmosis/pull/5493>
 - Incorrect bound check for liquidity amounts, allowing to withdraw more than eligible for: <https://github.com/osmosis-labs/osmosis/pull/5474>
 - Incorrect handling of forfeited incentives: <https://github.com/osmosis-labs/osmosis/pull/5495>
 - Tick iterator bugs: <https://github.com/osmosis-labs/osmosis/pull/5491>
 - Incentive emission share calculation not accounting for active tick liquidity: <https://github.com/osmosis-labs/osmosis/pull/5417>
 - remove repeated reallocations in swap step iterations: <https://github.com/osmosis-labs/osmosis/pull/5211>
 - separate fees into different module account: <https://github.com/osmosis-labs/osmosis/pull/5230>
 - MsgSwapExactAmountOut audit: <https://github.com/osmosis-labs/osmosis/pull/5179>
 - v16 upgrade handler updates: <https://github.com/osmosis-labs/osmosis/pull/5213>
 - change KVStore value from posID to boolean byte: <https://github.com/osmosis-labs/osmosis/pull/5237>
 - update rewards splitting logic to only use bonded classic pool balances: <https://github.com/osmosis-labs/osmosis/pull/5239>
 - MsgUnlockAndMigrateSharesToFullRangeConcentratedPosition audit (part 2/2): <https://github.com/osmosis-labs/osmosis/pull/5160>
 - reinvest dust fees back into pool accum: <https://github.com/osmosis-labs/osmosis/pull/5245>
 - iterator improvements for swap in given out and liquidity for full range query: <https://github.com/osmosis-labs/osmosis/pull/5248>
 - multiple fee tokens gov prop: <https://github.com/osmosis-labs/osmosis/pull/5261>
 - bump sdk fork with sqrt perf improvements: <https://github.com/osmosis-labs/osmosis/pull/5249>
 - remove repeated pool exists check: <https://github.com/osmosis-labs/osmosis/pull/5278>
 - avoid refetching ticks during swaps; parse from iterator value: <https://github.com/osmosis-labs/osmosis/pull/5288>
 - liquidity net in direction `sdk.Int` -> `int64`: <https://github.com/osmosis-labs/osmosis/pull/5299>
 - liquidity for full range `sdk.Int` -> `int64`: <https://github.com/osmosis-labs/osmosis/pull/5300>
 - remove repeated pool unmarshaling and uptime accum refetching overhead in swaps: <https://github.com/osmosis-labs/osmosis/pull/5295>
 - Fix tick range helper and update tests/comments: <https://github.com/osmosis-labs/osmosis/pull/5313>
 - Fix discount rate bound check: <https://github.com/osmosis-labs/osmosis/pull/5314>
 - expect single synthetic lock per native lock ID: <https://github.com/osmosis-labs/osmosis/pull/5265>
 - fungify message audit: <https://github.com/osmosis-labs/osmosis/pull/5317>
 - Remove double rounding in `CalcAmount1Delta`: <https://github.com/osmosis-labs/osmosis/pull/5326>
 - pools gov prop supercharge liquidity: <https://github.com/osmosis-labs/osmosis/pull/5345>

- Update Accumulator while collecting incentives: <https://github.com/osmosis-labs/osmosis/pull/5290>
- Make calc accrued incentives nonmutative: <https://github.com/osmosis-labs/osmosis/pull/5361>
- twap record upgrade handler: <https://github.com/osmosis-labs/osmosis/pull/5363>
- single migration entry point: <https://github.com/osmosis-labs/osmosis/pull/5360>
- Add 2 week supported uptime: <https://github.com/osmosis-labs/osmosis/pull/5349>
- Fix SwapInGivenOut to track fees with the correct denom: <https://github.com/osmosis-labs/osmosis/pull/5388>
- Delete wrong check in `ValidateBasic` of `MsgUnlockAndMigrateSharesToFullRangeConcentratedPosition`: <https://github.com/osmosis-labs/osmosis/pull/5393>
- fix uptime Accumulator init genesis: <https://github.com/osmosis-labs/osmosis/pull/5411>
- Fix incentive emission share calculation: <https://github.com/osmosis-labs/osmosis/pull/5417>
- redirect distribution record on migrate link: <https://github.com/osmosis-labs/osmosis/pull/5400>
- feat: NoLock gauge type and external gauge creation wiring to CL: <https://github.com/osmosis-labs/osmosis/pull/5459>
- refactor/fix(CL): add IDs to incentive record keys to avoid overwriting: <https://github.com/osmosis-labs/osmosis/pull/5496>

Engagement Summary

- **Dates:** 15.05.2023 to 21.06.2023
- **Method:** Manual code review, protocol analysis
- **Employees Engaged:** 2

Severity Summary

Finding Severity	#
Critical	0
High	1
Medium	1
Low	1
Informational	7
Total	10

System Overview

Osmosis is a decentralized blockchain protocol built on top of the Cosmos network. It aims to provide an infrastructure for efficient and secure token exchange. Osmosis focuses on the concept of liquidity pools, allowing users to create and manage pools of tokens for trading purposes.

By using the Osmosis platform, users can create and manage liquidity pools by depositing pairs of tokens. These pools enable the exchange of tokens and serve as the basis for price discovery. Liquidity providers earn transaction fees and receive liquidity pool tokens in return for providing liquidity. Osmosis supports multiple pool types (balancer, stableswap) and the scope of this audit is around concentrated liquidity pools.

Concentrated liquidity

The design of the Automated Market Maker, introduced by Uniswap, which is based on creating the possibility to liquid in specific price ranges using the concept of ticks is called concentrated liquidity. It allows liquidity providers (LPs) to focus their capital in a specific price range, resulting in higher efficiency and lower price impact for traders. Osmosis team used further developed Uniswap's design and created its own concentrated liquidity AMM.

The architecture of concentrated liquidity introduces the concept of a "position" that concentrates liquidity within a fixed range. Instead of tracking individual token reserves, the design tracks the liquidity amount and the square root of the price ratio. This enables efficient calculation of outcomes for swaps and pool joins.

Ticks are discrete points utilized in concentrated liquidity (CL) pools. In the conventional approach, ticks have a fixed difference of 0.01% between adjacent prices. However, Osmosis adopts a geometric tick spacing methodology with additive ranges. This approach offers enhanced granularity and control over tick prices, enabling liquidity providers (LPs) to trade at desired spot prices.

The formulas involved in tick spacing establish the relationship between ticks and prices. The precision factor is defined at the starting tick ($\text{exponentAtPriceOne}$), and the tick spacing determines the increment distance. By calculating geometric exponents and additive ticks, the corresponding price for a tick can be determined, and vice versa.

The decision to employ geometric tick spacing with additive ranges facilitates trading at desired spot prices and enhances the user experience. It eliminates the need for rounding or restricting trades to specific prices. However, multiple ticks may represent the same price, and in such cases, a larger tick representing the same price is chosen when creating a position.

Key distinctions between Osmosis' implementation and Uniswap CL are as follows:

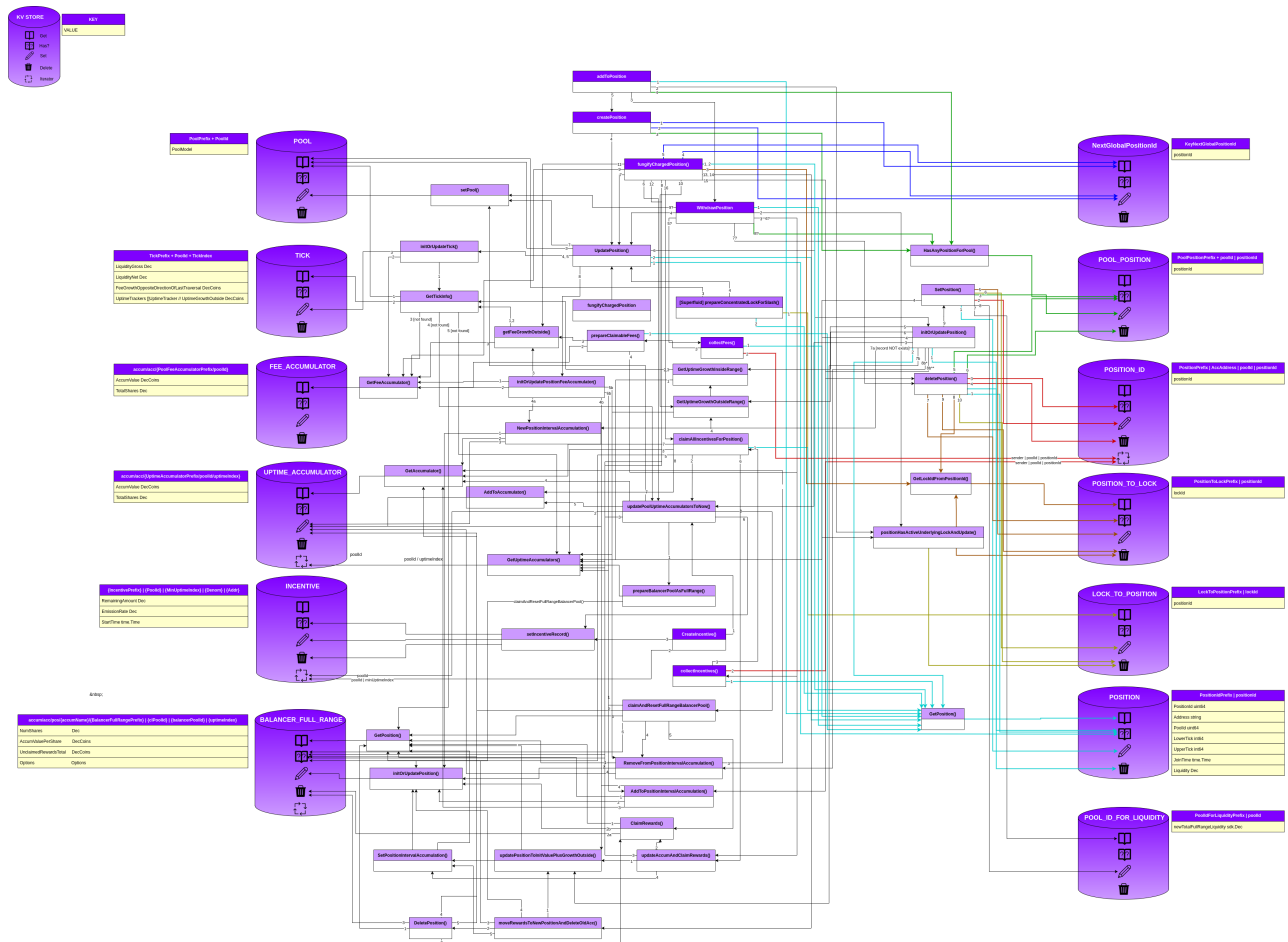
1. *Geometric tick spacing with additive ranges*
Osmosis improves upon Uniswap's logarithmic spacing by employing geometric tick spacing with additive ranges. The core idea is that by adjusting the precision for each power of 10, human-readable values align with actual tick spots.
2. *Superfluid full range positions*
Osmosis introduces the ability to superfluidly stake positions, but only as full range positions within the CL pool. Notably, the balancer position is analogous to a full range CL position, which is why it can be superfluid staked.
3. *Migration logic*
It is possible to migrate a position from another pool (e.g., Balancer) to the CL pool. To enable this, a governance-approved connection must exist between a single Balancer pool and a single CL pool when the CL pool is created. Partial migration is also allowed. The Balancer position to be migrated can be in one of four different states, and the migration path is determined based on its current state. These states are:
 - a. superfluid staked and it's bonded for 14 days,
 - b. superfluid unbonding position (still unbonding, e.g day 7 of 14),
 - c. locked but not superfluid staked,
 - d. not locked at all.
4. *Shared incentive logic*
In Osmosis' design, all rewards are calculated based on CL pool positions, but transactions can be

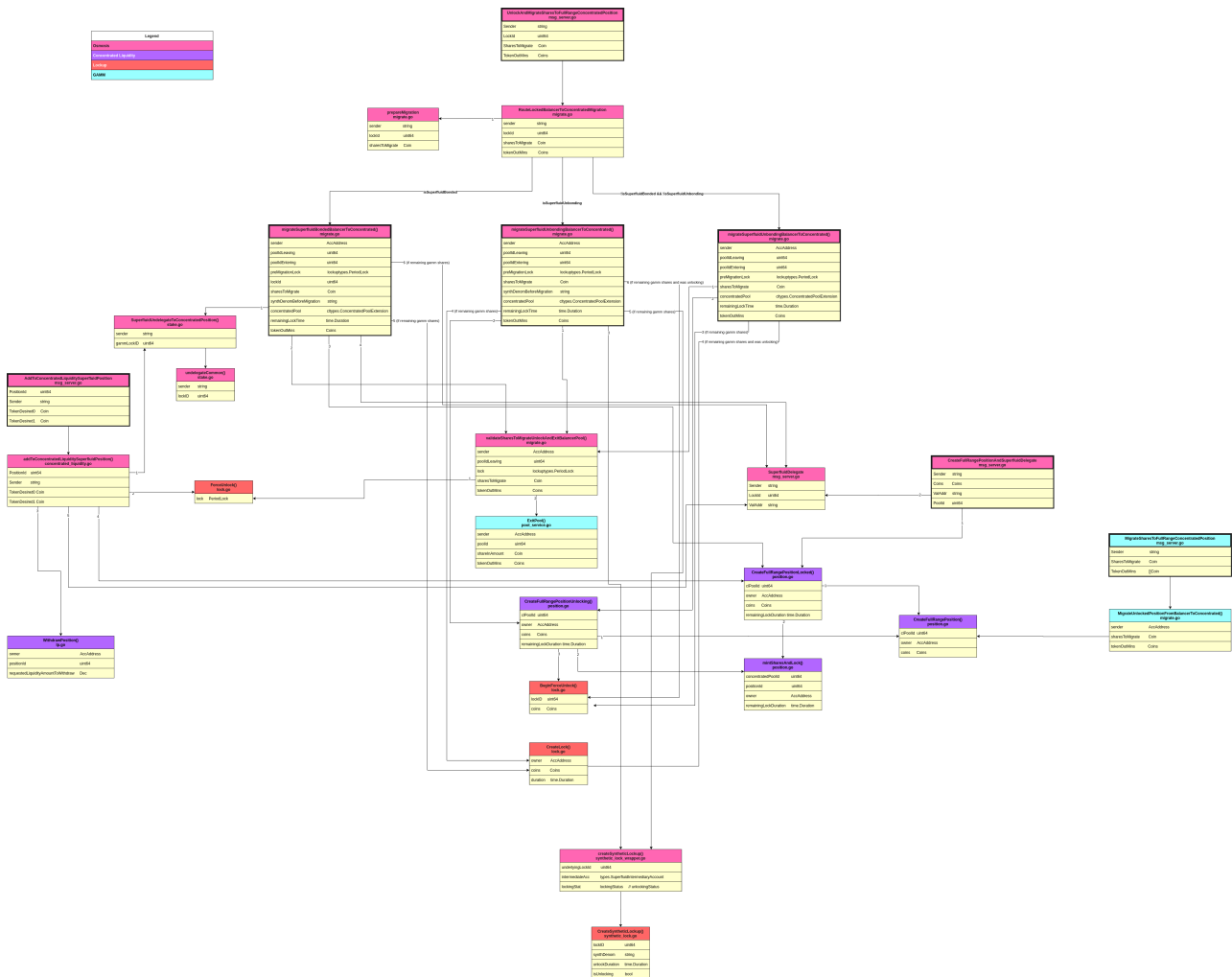
conducted through both the CL and Balancer pools, with both pools receiving incentives. To facilitate this, the Balancer pool is represented as a single full range position within the CL pool. During the calculation process, the funds to which the Balancer pool is entitled are routed based on the calculations for its position within the CL pool.

5. *Charging to be qualified for uptime incentives (1ns)*

Originally, a position was entitled to rewards upon creation but could only collect them after a certain period of uptime. Osmosis deemed this requirement too strict and modified the uptime to 1ns, allowing rewards to be collected immediately. This adjustment serves as an interim approach until their new design for uptimes is finalized.

During the audit, two detailed diagrams were created to help in the process of system understanding and finding possible vulnerabilities. The first is showing the interactions with the stores while the other one is showing the flow in migration logic.





To better explain the core concepts of Osmosis' design, messages from concentrated liquidity module are described below.

MsgCreateConcentratedPool

Pool initialization is called from the `poolmanager` module but is executed within the `concentrated-liquidity` module. During initialization, the tick spacing, swap fee, and quote denominations (extracted from the creation message) are validated to ensure authorized values. The initialization process also encompasses creating fee and uptime accumulators, setting the timestamp for the last liquidity update, saving the pool to storage, and invoking specific listeners.

System Overview

MsgCreatePosition

This message is employed to create positions in a concentrated liquidity pool. The message includes the following data: the pool's ID where the position is being created, the address of the message sender, the lower and upper tick bounds of the position, the quantity of tokens provided for the position, and the desired minimum amounts of specific tokens to be added or removed from the pool.

To create a position, all the forwarded data in the message undergoes validation and recalculation based on the pool's properties. For instance, the liquidity amounts to be provided are proportionate to the existing reserves. Additionally, the created position and the actual token amounts calculated are checked against the user's desired amounts of tokens to be received or sent.

A crucial calculation involved in this message is determining the price based on the provided ticks. Osmosis employs a customized approach to ticks representation and these calculations, which are thoroughly explained in the provided documentation.

Finally, once the correct token amounts are obtained, they are transferred from the user to the pool, followed by emitting a specific event.

MsgAddToPosition

This message is used to add a certain amount of tokens to already created positions. According to the current design, the position associated with the forwarded ID in the message parameters is deleted during the process, while a new position is created, combining the tokens already positioned under the mentioned ID and the ones sent via the message. The sender of this message must be the same as the creator of the position being updated, and the position must not be superfluidly staked.

After initial input validations, the process begins with withdrawing from the position with the current ID. This process also encompasses collecting fees and incentives from that position. Essentially, the withdrawal process represents an update to the position with a new liquidity amount. The amount can be equal to or less than the currently available liquidity, and in the former case, the position is deleted. When withdraw is invoked from `MsgAddToPosition`, it withdraws the full liquidity amount each time to delete the position to which funds are being added (based on the current design). The withdrawal procedure concludes with the actual transfer of tokens from the pool to the user, followed by emitting an event.

The withdrawal process is succeeded by creating the new position with the updated token amounts. An event is also triggered to announce the position creation.

MsgWithdrawPosition

The `MsgWithdrawPosition` message is utilized to withdraw funds from a specific position in the pool and transfer them to the user. Validations for this message include verifying that the owner and creator of the position are the same, ensuring that the withdrawal amount is not negative, and checking that the requested liquidity amount to be withdrawn is not greater than the available liquidity in the specified position. The withdrawal process also includes checks for any underlying locks on the position.

In addition to the mentioned validations, this process involves collecting incentives for the position. If the complete liquidity is being withdrawn, fees are also collected. In such cases, the pool must be uninitialized, with the square root price and tick set to zero. Lastly, the position being withdrawn from is updated in terms of the tick and liquidity amount.

MsgCollectFees

The `MsgCollectFees` message is used to collect fees from specified positions. The message contains information about the positions from which the fees should be obtained, including the sender of the message and the position IDs. The sender must also be the owner of the positions. The following process occurs for each position in the message:

1. Retrieve the position using the position ID.
2. Determine the position owner and verify that it matches the message sender.
3. Use the fee accumulator to calculate the amount of fees earned by the position.
4. Obtain the pool address and transfer the fee amount from the pool to the eligible user.
5. Emit an event indicating that the fees for a single position have been collected.

After processing all the positions, the calculated fees are summed up, and the total amount is emitted in another event at the end of the message.

MsgCollectIncentives

Similar to collecting fees, the `MsgCollectIncentives` message is used to collect incentives on a per-range basis. The message includes the owner and position IDs of the positions for which rewards are to be claimed. After validating the sender, the following process is performed for each position ID in the message parameters:

1. Retrieve the position using the position ID and verify that the owner matches the message sender.
2. Use the rewards accumulator (uptime accumulators) to determine the amount of rewards earned by the position.
3. Check if the calculated reward amount is nonzero and transfer it from the pool address to the user.
4. Emit an event indicating that rewards have been collected for the specific position.

After processing all the positions, the total sum of collected and forfeited rewards is calculated and emitted in an event.

There are two key points specific to Osmosis' design worth mentioning. Both of these are incorporated in the second step of the incentives calculation flow described earlier.

The first point involves synchronizing all accumulators to ensure they are up to date for the relevant pool. This includes time calculations and a unique approach where one position represents the entire Balancer pool within the concentrated liquidity (CL) pool, enabling rewards to be calculated for that specific pool.

The second important aspect is the calculation of uptime growth beyond the given tick range. This is another Osmosis-specific approach that takes into account all rewards claimed by the entire pool.

It is also noteworthy that some of the calculated rewards are returned to the pool for other liquidity providers to receive if the current provider is ineligible. If there are no remaining positions, these funds are sent to the community pool.

MsgCreateIncentive

The creation of incentives is facilitated through the `MsgCreateIncentive` message. The goal is to transfer incentive coins (provided in the message) from the owner (message sender) to the incentives address of the pool, while also creating a record of the incentives for the given pool.

The validation process for this message includes the following checks:

- Ensuring the validity of the incentive coins.
- Verifying that the start time is after the current block time.
- Confirming that the emission rate (provided in the message) is positive.
- Validating that the minimum uptime (provided in the message) is authorized.

If all the validations pass, the process continues by checking the balance of the sender's address before actually transferring the funds to the pool's incentives address.

Since the uptimes are stored in uptime accumulators, they are also updated to synchronize with the current block time. The message processing concludes with the creation of the incentives record and emitting an event.

MsgFungifyChargedPositions

The `MsgFungifyChargedPositions` message is employed to combine multiple positions into a single position. The input data for this message includes position IDs and the sender of the message. Each position is validated according to the following criteria:

- Ownership of the positions belongs to the same user.
- The positions are located within the same tick range.
- The positions belong to the same pool.
- The positions are fully charged.
- The positions are not locked.

The process of merging multiple positions into one results in the creation of a new position that inherits the properties of all the original positions. This means that unclaimed rewards, liquidity, and the timestamp of the new position are derived from the original positions (e.g., the liquidity is the sum of all the liquidities from the previous positions). As the old positions are processed, eligible rewards are claimed, unclaimed rewards are transferred to the new position, and the old positions are deleted. Upon creating the new position, the fee and uptime accumulators are updated. The process concludes with emitting an event.

Threat inspection

User categories

- **Osmosis stakeholders:** Osmosis owners, or whoever has an economic upside from the Osmosis itself
- **Liquidity providers (LPs):** Users who provide their tokens in concentrated liquidity pools, and thus give liquidity for trades. Their main incentive is to get upside from trades via fees. Dedicated incentives for LPs are also included.
- **Traders:** Users who perform trades on Osmosis. Their incentive is to extract value via trading tokens at different exchanges at varying valuations.

Threats

Threat: Miscalculations in processes based on geometric tick spacing with additive ranges

When liquidity providers (LPs) submit their funds into liquidity pools, the price is calculated using Osmosis' unique approach, which employs geometric tick spacing with additive ranges. It is crucial to ensure that the calculated prices are fair with respect to all parameters.

Impacted users:

- Liquidity providers

Possible impacts:

- LPs submitting funds may receive an unfair price for their amount.
- A lower price would result in a loss of value for LPs' funds.
- A higher price would make the system exploitable, potentially causing other liquidity providers to lose their funds.

Possible ways of exploitation:

- Discovering a flaw in the economic mechanism and submitting a crafted transaction to exploit it.

Audit actions:

- Examine the new approach on tick, price, squared price calculations
- Review all the data precision and any other mathematical relations being influenced by the previous calculations

Threat: Out-of-thin-air token generation due to rounding

Calculations involving prices, ticks, or any other computations are complex, involving multiplication, truncation, and multiple scenarios. Rounding errors could lead to the generation of residual tokens, which, although small in quantity, may deplete the reserves.

Impacted users:

- Osmosis stakeholders
- Liquidity providers
- Traders

Possible impacts:

- Loss of funds

Possible ways of exploitation:

- Submitting multiple transactions with specially crafted inputs, such as very large values.

Audit actions:

- Inspect rounding practices throughout the solution
- Make sure the rounding is always performed in favor of the pool
- Check if provided transaction types can lead to token generation out of nowhere

Threat: Miscalculation that can be abused to lead to unfair allocation of rewards

Miscalculations in terms of amount can harm both users and chain holders. In addition to the amount, the denomination must always be correct to avoid rewarding arbitrary tokens.

Balancer pool receives incentives based on calculations for the position created in the CL pool, where that position represents the Balancer. The representation and calculation processes must be scrutinized for any flaws that could result in unfair incentives for the Balancer pool.

Impacted users:

- Liquidity providers

Possible impacts:

- LPs submitting funds may receive an inaccurate amount of incentives.

Possible ways of exploitation:

- Discovering a flaw in the incentives mechanism and submitting a crafted transaction to exploit it.

Audit actions:

- Examine the incentives calculation mechanisms (reward amount, reward denoms)
- Inspect the unique approach of migrating positions to CL pool in order to be incentivised

Threat: Routing tokens to incorrect user

Given that the number of users involved in liquidity providing is presumably arbitrary, it is essential to ensure that rewards are correctly routed and allocated to accounts that genuinely contributed to liquidity provision and deserve the reward.

Impacted users:

- Liquidity providers

Possible impacts:

- LPs submitting funds may not receive incentives or may not receive the complete amounts.

Possible ways of exploitation:

- Discovering a flaw in the incentives mechanism and submitting a crafted transaction to exploit it.

Audit actions:

- Examine all the places where Bank module is used to route the rewards to user/module
- Examine that destination account addresses cannot be fabricated

Threat: A malicious user intentionally creates transaction that slows down the chain

It is crucial to prevent users from abusing messages, such as creating a pool, creating a position, collecting fees, collecting incentives, etc., to create transactions that would significantly slow down or potentially halt the chain (e.g., exceeding block time).

Impacted users:

- Liquidity providers

- Osmosis stakeholders
- Traders

Possible impacts:

- All users involved with the CL may experience delays or unavailability of the system.

Possible ways of exploitation:

- Submitting a crafted transaction to exploit the chain's performance.

Audit actions:

- Determine if there is possibility to utilize any of the present practices in the code (especially loops or repeated actions) to slow down the chain
- Inspect the possibility of halting the chain (e.g. panic from BeginBlocker/EndBlocker)

Threat: Abuse the connection between CL and Balancer pool

Osmosis' design incorporates a connection between the Balancer and CL pools. While this connection serves various purposes, it is crucial to examine whether it can be exploited to propagate bugs or flaws that could damage not only one pool but both of them.

Another aspect that requires examination is the implementation of Osmosis-specific superfluid staking, which is currently accessible in the CL pool. Furthermore, the Balancer position undergoing migration may exist in different states, two of which are superfluid staked positions, namely, bonded or unbonding.

Impacted users:

- Liquidity providers
- Traders

Possible impacts:

- Pools may lose a portion or the entirety of their reserves.
- Lose superfluid stakes.

Possible ways of exploitation:

- Propagating a flaw through the pool's connection using a crafted transaction.

Audit actions:

- Examine the mechanisms used to connect the two pools and determine if there is a flaw or a feature inside that could be utilized to maliciously drain or lose the funds from one pool upon receiving the data over the connection

Threat: Inaccurate migration of balancer position to CL pool

The migration logic allows users to move their positions from the Balancer pool to the CL pool. This migration process is designed to attract users by leveraging CL's ability to utilize capital more effectively. However, it is crucial that the migration is executed precisely, without any flaws in the algorithm that could result in unintended changes to the created positions.

Impacted users:

- Liquidity providers

Possible impacts:

- Created positions may be altered due to algorithmic flaws.

Possible ways of exploitation:

- Transactions used for migrating positions may result in unwanted changes.

Audit actions:

- Check all possible migration flows and states
- Examine preservation of desired properties of positions before and after the migration

Threat: Inaccurate merging of positions in CL pool

The `MsgFungifyChargedPositions` message enables users to combine multiple positions into a single position within the CL pool. The merging process involves aggregating the properties of the individual positions and creating a new position with these combined properties. It is important to verify whether the new position accurately represents the original positions from which it originated.

Impacted users:

- Liquidity providers

Possible impacts:

- Some properties of positions are lost during merging.

Possible ways of exploitation:

- Transaction used for merging positions may result in unwanted changes.

Audit actions:

- Examine the preservation of position properties before and after the merge
- Examine the properties of individual position after the merging when interacting with the user

Threat: Abuse permissionless approach for messages

Since all messages in the module are currently permissionless, except for pool creation (which will likely become permissionless as well), it is essential to validate and verify all inputs from message senders to prevent them from creating a state that could harm the pool, other users, or the chain.

Impacted users:

- Liquidity providers
- Traders

Possible impacts:

- Delays in operation, denial-of-service attacks, chain halts, loss of user funds, and various other consequences resulting from maliciously created inputs that lead to an unwanted system state.

Possible ways of exploitation:

- Creating malicious inputs through crafted transactions.

Audit actions:

- Examine all the inputs available for user to forward in pool creation process
- Inspect the edge case values of data originating from the user

Findings

There were errors rendering macro:

- An unknown error occurred.

Division by zero protection in distribute.go

Title	Division by zero protection in distribute.go
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	3 HIGH
Issue	

Involved artifacts

- [osmosis/x/incentives/keeper/distribute.go](#)

Description

During the review of the new feature introduced in the pull request [PR #5459](#), we identified a potential issue related to the lack of protection against division by zero in the code segment. The specific code areas of concern are as follows:

[distribute.go#L318](#)

```
remainAmountPerEpoch := remainCoin.Amount.Quo(sdk.NewIntFromUint64(remainEpochs))
```

[distribute.go#L290-L293](#)

```
remainEpochs := uint64(1)
if !gauge.IsPerpetual {
    remainEpochs = gauge.NumEpochsPaidOver - gauge.FilledEpochs
}
```

While reviewing the code, it was observed that there is no protection against division by zero when calculating `remainAmountPerEpoch`. Additionally, `remainEpochs` can be zero if `NumEpochsPaidOver` and `FilledEpochs` have the same values within the `gauge`. This issue can result in a panic, causing a chain halt since the code is invoked from the `BeginBlocker`.

The problem can be recreated and panic caused using [this](#) test, just setting `NumEpochsPaidOver=0`.

Problem Scenarios

The lack of protection against division by zero can lead to the following problems:

1. **Division by Zero Error:** If `remainEpochs` is zero, division by zero will occur when calculating `remainAmountPerEpoch`. This will cause a runtime panic, leading to a chain halt and disrupting normal operations.
2. **Chain Halt:** A panic resulting from the division by zero error can cause the chain to halt, potentially impacting the overall stability and availability of the system.

Recommendation

To address the division by zero issue and prevent potential chain halts, we recommend implementing input validation for the `NumEpochsPaidOver` parameter and adding appropriate protection in the code.

1. **Input Validation:** When accepting the `NumEpochsPaidOver` input, ensure that it is greater than zero and validate it before further processing. Reject any invalid inputs to prevent the division by zero scenario.
2. **Protection against Division by Zero:** Before performing the division operation `(remainCoin.Amount.Quo(sdk.NewIntFromUint64(remainEpochs)))`, check if `remainEpochs` is non-zero. If `remainEpochs` is zero, handle the situation appropriately, such as returning an error or taking alternative actions based on the specific use case requirements.

By implementing these recommendations, you can safeguard against division by zero errors and prevent chain halts caused by panics. Additionally, it is essential to perform thorough testing and review of the updated code to ensure its correctness and resilience in handling various scenarios.

Insufficient validation on gauge creation

Title	Insufficient validation on gauge creation
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Issue	

Involved artifacts

- [osmosis/x/incentives/keeper/gauge.go](#)
- [osmosis/x/incentives/keeper/msg_server.go](#)
- [osmosis/x/incentives/keeper/msg_server_test.go](#)
- [osmosis/x/incentives/types/msgs_test.go](#)

Description

The [message used for creating](#) a gauge contains the property `NumEpochsPaidOver`. This property, along with other necessary data, is forwarded from the message server to the gauge creation function. However, neither the [message server](#) nor the creation function includes any validation for this `uint64` value.

It is important to note that this value plays a significant role in the [calculation and distribution of rewards](#). Specifically, it affects the amount of rewards allocated over a particular period, with higher values resulting in decreased rewards.

To confirm that the validation is not present, an [existing test](#) can be changed so that it receives maximum value of `uint64`:

```
func TestMsgCreatePool(t *testing.T) {
    // generate a private/public key pair and get the respective address
    pk1 := ed25519.GenPrivKey().PubKey()
    addr1 := sdk.AccAddress(pk1.Address())

    // make a proper createPool message
    createMsg := func(after func(msg incentivetypes.MsgCreateGauge)
incentivetypes.MsgCreateGauge) incentivetypes.MsgCreateGauge {
        distributeTo := lockuptypes.QueryCondition{
            LockQueryType: lockuptypes.ByDuration,
            Denom:           "lptoken",
            Duration:        time.Second,
```

```

    }

    properMsg := *incentivetypes.NewMsgCreateGauge(
        false,
        addr1,
        distributeTo,
        sdk.Coins{},
        time.Now(),
        18446744073709551615,
    )
    ...

```

When executed, the test passed with no notifications or errors about this value.

It can also be tested using message server `test` with same value as the previous one:

```

msg := &types.MsgCreateGauge{
    IsPerpetual:    tc.isPerpetual,
    Owner:          testAccountAddress.String(),
    DistributeTo:   distrTo,
    Coins:          tc.gaugeAddition,
    StartTime:      time.Now(),
    NumEpochsPaidOver: 18446744073709551615,
}

```

The test also passed with no errors reported.

Problem Scenarios

An adversarial user could exploit this system by creating a gauge with an excessively large value for `NumEpochsPaidOver`. This manipulation would lead to rewards of small value being distributed over an extended duration.

Additionally, if there are limitations on the number of gauges, this behavior negatively impacts other users attempting to create gauges, as the extended epochs specified by the malicious user can impede the creation of new gauges by legitimate users.

Recommendation

Determining the precise value to set as a limitation for `NumEpochsPaidOver` is not a straightforward task. However, it is advisable to establish a limit based on testing statistics, anticipated user behaviors, or sensible expected values for the number of epochs over which awards are distributed.

By imposing a limit, it helps mitigate potential abuses or unintended consequences resulting from excessively large values. Careful consideration should be given to factors such as system performance, resource allocation, and overall user experience. The specific limit should be determined based on a thorough analysis of the system's requirements and the expected behavior of users within the context of award distribution.

Optimization opportunity in addToPosition for last position withdrawal

Title	Optimization opportunity in addToPosition for last position withdrawal
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	2 MEDIUM
Issue	

Involved artifacts

- osmosis/x/concentrated-liquidity/lp.go

Description

While examining the accuracy of position checks and the number of positions in the pool, we identified a potential opportunity for optimization in the `addToPosition` process. The current implementation involves withdrawing the position first, which results in the position being deleted, and then checking if the pool contains any remaining positions. If there are no positions left, an error is returned, and the message execution is halted. The code segment of interest can be found [here](#).

Problem Scenarios

The existing logic presents a few potential problems:

1. **Inefficient Withdrawal Process:** The current approach of withdrawing the position first and then checking for remaining positions can lead to inefficiencies, especially when the withdrawal operation is resource-intensive. In cases where the position being processed is the last one, unnecessary withdrawal operations are performed.
2. **Resource-Intensive Rollback Procedure:** If an error occurs during the withdrawal of the last position, the entire function needs to be rolled back, which can involve complex computations or database operations. Performing unnecessary withdrawals increases the frequency and complexity of the rollback procedure, leading to resource wastage.

Recommendation

To address the identified issues and optimize the code, we recommend implementing a check to determine if the position being processed is the last one in the pool before initiating the withdrawal. By performing this check upfront, the withdrawal process can be bypassed for the last position, resulting in cost savings and a simplified

rollback procedure. This optimization will contribute to improved efficiency and resource utilization within the `addToPosition` functionality.

It is important to carefully consider the implementation details, such as how the last position is identified and the potential impact on the overall system behavior. Thorough testing should be conducted to ensure the proposed optimization does not introduce any unintended side effects.

By implementing the suggested modification, the execution efficiency of the `addToPosition` process can be enhanced, leading to a more streamlined and resource-efficient codebase.

Missing validation for negative number of shares when creating accumulator position

Title	Missing validation for negative number of shares when creating accumulator position
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	1 LOW
Issue	

Involved artifacts

- [osmosis/osmoutils/accum/accum.go](#)
- [osmosis/osmoutils/accum/accum_test.go](#)

Description

There is currently no validation in [this](#) function to prevent negative `numShareUnits`. To ensure data integrity and prevent the creation of positions with a negative number of shares, it is advisable to include such a check.

It is worth noting that the `AddToPosition` function already performs this validation. Therefore, it would be appropriate to add a similar validation in the mentioned function to maintain consistency and prevent the creation of positions with negative share units.

To confirm the described scenario, an existing test can be reused. In `accum_test.go` a following change could be added:

```
var (
    testAddressOne   = sdk.AccAddress([]byte("addr1_____")).String()
    testAddressTwo   = sdk.AccAddress([]byte("addr2_____")).String()
    testAddressThree = sdk.AccAddress([]byte("addr3_____")).String()

    emptyPositionOptions = accumPackage.Options{}
    testNameOne          = "myaccumone"
    testNameTwo          = "myaccumtwo"
    testNameThree        = "myaccumthree"
    denomOne             = "denomone"
    denomTwo             = "denomtwo"
    denomThree          = "denomthree"
```



```

emptyCoins = sdk.DecCoins(nil)
emptyDec   = sdk.NewDec(0)

initialValueOne      = sdk.MustNewDecFromStr("100.1")
initialCoinDenomOne  = sdk.NewDecCoinFromDec(denomOne, initialValueOne)
initialCoinDenomTwo  = sdk.NewDecCoinFromDec(denomTwo, initialValueOne)
initialCoinDenomThree = sdk.NewDecCoinFromDec(denomThree, initialValueOne)
initialCoinsDenomOne  = sdk.NewDecCoins(initialCoinDenomOne)

positionOne = accumPackage.Record{
    NumShares:      sdk.NewDec(-100),
    AccumValuePerShare: emptyCoins,
    UnclaimedRewardsTotal: emptyCoins,
}
...

```

and then, execution of `func (suite *AccumTestSuite) TestNewPosition()` is successful even though the number of `NumShares` is negative.

Problem Scenarios

By implementing this validation, we can mitigate potential problems where a position with a negative number of shares is created. Such scenarios can negatively impact subsequent calculations involving incentives or fees in the accumulators.

Recommendation

Although the current calls to this function in the `fees.go` and `incentives.go` files from the `x/concentrated-liquidity` package ensure that negative values are not passed, it is still recommended to add the validation. This is essential in case of future changes that may overlook these checks before calling the function.

Redundant checks

Title	Redundant checks
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- [osmosis/x/concentrated-liquidity/model/pool.go](#)
- [osmosis/x/poolmanager/create_pool.go](#)
- [osmosis/x/concentrated-liquidity/math/tick.go](#)
- [osmosis/x/concentrated-liquidity/position.go](#)

Description

1. This validation check is currently implemented in the `ValidateBasic` function. It would be beneficial to reassess the necessity of this check within the existing flow. In the `current` scenario, the `NewConcentratedLiquidityPool` function is invoked from within the `CreatePool` function. Since the message is validated prior to this invocation, the mentioned check becomes superfluous.
2. The `check` for `upperTick <= lowerTick` is redundantly performed before each call to this function. It would be more appropriate to restrict this check to the function itself.
3. Is it not imperative to verify the length against 2? The minimum requirement for the number of positions is 2, so allowing validation for messages containing only 1 position seems undesirable. Furthermore, this check is repetitive, as the length of the positions collection is already validated before invoking this function, so one of the checks can be removed (probably better to leave the one inside the `validatePositionsAndGetTotalLiquidity`).
4. Within this function, the same check (`check1`, `check2`) is redundantly executed twice, resulting in unnecessary repetition. Moreover, there is no modification made to the validated data between these checks. Therefore, it is recommended to eliminate the duplication by performing the check only once. This optimization ensures code efficiency and avoids redundant evaluations.

Store loading optimization

Title	Store loading optimization
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- [osmosis/osmoutils/accum/accum.go](#)
- [osmosis/x/concentrated-liquidity/incentives.go](#)
- [osmosis/x/concentrated-liquidity/lp.go](#)
- [osmosis/x/concentrated-liquidity/tick.go](#)
- [osmosis/x/concentrated-liquidity/fees.go](#)
- [osmosis/x/concentrated-liquidity/swaps.go](#)

Description

1. It is unnecessary to extract the position from the store at [this](#) point and return the `numShares` separately. Instead, the position can be directly referenced as `position.numShares` since it has already been retrieved a few lines above. This approach eliminates the need for duplicate retrieval and reduces the number of load operations from the store. The same redundant process is repeated [here](#) as well.
2. To optimize load operations from the store, the function `ClaimRewards` can be modified to accept the `Position` as an [argument](#) instead of its name. By doing so, we avoid [retrieving the position from the store](#) every time the function is called. The only necessary adjustment in this case would be made [here](#). Implementing this change reduces the number of load operations required from the store and improves efficiency.
3. At [this](#) point, it is feasible to eliminate an additional store load operation. Since the position is already retrieved from the store at the start of the function, the liquidity required in the specified section of the code can be defined simply as `position.Liquidity`. This adjustment leverages the existing data and avoids the need for an additional store retrieval.
4. Excessive invocations of the `getPoolById()` function lead to increased loads on the data stores. Upon analysis, we have identified several instances where the `poolId` is passed into functions and subsequently retrieved from the stores. This flow could be optimized by caching the pool at the outset of the operation and reusing it later, thereby reducing the number of load operations. In the `GetTickInfo()` function, the `poolId` is passed as an argument and forwarded to three distinct

functions: `getInitialFeeGrowthOppositeDirectionOfLastTraversalForTick`, `updatePoolUptimeAccumulatorsToNow`, and `getInitialUptimeGrowthOppositeDirectionOfLastTraversalForTick`. Each of these

functions loads the pool from the store using the aforementioned `getPoolById()` function. An enhanced approach would involve storing the retrieved pool once and subsequently utilizing it without the need for additional store loading. Since the pool remains unaltered throughout the flow, data consistency can be maintained while achieving improved code efficiency.

- a. An example where the pool is obtained at the very beginning of the function and then forwarded to `GetTickInfo()` is the function `getFeeGrowthOutside()`. Later, `poolId` gets forwarded to already mentioned three functions with loading from the store in each of them.
 - b. Similarly, in the `UpdatePosition()` function, where the pool is obtained at the beginning, two calls to `initOrUpdateTick()` are made, which also depend on the `GetTickInfo()` function.
 - c. Another comparable pattern is observed within the `computeOutAmtGivenIn()` function, which commences with the retrieval of the pool from the store. Subsequently, the `crossTick()` function is invoked, wherein both `GetTickInfo()` and `updatePoolUptimeAccumulatorsToNow()` perform additional load operations. This can be made more efficient by passing the pool itself from the outset of the `computeOutAmtGivenIn()` function, thereby avoiding redundant load operations.
 - d. A nearly identical scenario arises in the `computeInAmtGivenOut()` function, featuring the same calls to `crossTick()` and `updatePoolUptimeAccumulatorsToNow()`.
5. An opportunity for optimizing storage operations arises by eliminating unnecessary invocations of the `GetPosition()` function in flows where the position has already been retrieved from the store. To illustrate this possibility, let's consider a flow that currently includes five redundant calls to `GetPosition()`. Within the `WithdrawPosition()` function, there exists an initial call to `GetPosition()`. Subsequently, using the same identifier, the position is obtained from the store an additional five times in the following functions:
- a. `GetPositionLiquidity()`
 - b. `collectIncentives()` (called twice)
 - c. `claimAllIncentivesForPosition()`
 - d. `collectFees()`
 - e. `prepareClaimableFees()`

To improve efficiency, these redundant calls to `GetPosition()` can be removed since the position has already been acquired and stored. By eliminating these redundant operations, we can reduce the load on the storage system and enhance the overall performance of the code.

Minor code improvements

Title	Minor code improvements
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- [osmosis/x/concentrated-liquidity/position.go](#)
- [osmosis/x/concentrated-liquidity/lp.go](#)
- [osmosis/x/concentrated-liquidity/fees.go](#)

Description

1. Instead of evaluating the value of `lockIsMature` at [this](#) point, a more concise function return statement can be implemented as follows: `return !lockIsMature, lockId, nil`. This simplifies the code and provides a clear output from the function.
2. While the comment explicitly specifies that the `requestedLiquidityAmountToWithdraw` should be a positive value, the [current validation](#) check using `IsNegative()` only verifies that the value is not below zero, neglecting the consideration of zero itself.
3. To ensure a logical sequence of operations, it is advisable to execute the collection of rewards after validating the amount of liquidity intended for withdrawal. Hence, a recommended modification [here](#) would involve swapping these two actions.
4. The [function](#) `GetFeeAccumulator` can be optimized for simplicity by consolidating it into a single line: `return accum.GetAccumulator(ctx.KVStore(k.storeKey), types.KeyFeePoolAccumulator(poolId))`. This approach reduces code complexity while achieving the desired outcome.

Redundant condition check in loop for qualifyingLiquidity evaluation

Title	Redundant condition check in loop for qualifyingLiquidity evaluation
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- [osmosis/x/concentrated-liquidity/incentives.go](#)

Description

During our analysis, we observed a code segment in the pull request [PR #5417](#) that introduced changes related to the calculation of `qualifyingLiquidity`. Consequently, within the for loop in the `incentives.go` file [here](#), there is a code segment that is no longer required:

```
for uptimeIndex := range uptimeAccums {  
    // ...  
    if qualifyingLiquidity.LT(sdk.OneDec()) {  
        continue  
    }  
    // ...  
}
```

Problem Scenarios

The existing implementation can lead to the following issues:

1. **Redundant Condition Check:** Since the `qualifyingLiquidity` value remains constant within the loop, evaluating the same condition repeatedly (`qualifyingLiquidity.LT(sdk.OneDec())`) introduces unnecessary overhead. This condition check can be performed once before entering the loop, eliminating the need for redundant checks.
2. **Code Complexity and Performance Impact:** The redundant condition check increases code complexity without adding any value to the loop's functionality. It can negatively impact performance.

Recommendation

To optimize the code and enhance performance, we recommend moving the condition check for `qualifyingLiquidity` outside the loop. By performing this check only once, prior to entering the loop, we can eliminate the redundant condition checks within the loop. The modified code snippet would be as follows:

```
if qualifyingLiquidity.LT(sdk.OneDec()) {  
    for uptimeIndex := range uptimeAccums {  
        // ...  
        // ...  
    }  
}
```

By making this modification, we can eliminate the unnecessary condition check within the loop and reduce the computational overhead. It improves the code's simplicity, readability, and maintainability.

Alignment issue between comments and code in tick comparison logic

Title	Alignment issue between comments and code in tick comparison logic
Project	Osmosis Q2 2023
Type	IMPLEMENTATION DOCUMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- osmosis/x/concentrated-liquidity/incentives.go

Description

During the review [PR #5474](#), it was observed that the comments and comparisons related to `currentTick`, `lowerTick`, and `upperTick` are not aligned correctly. Specifically, the comments describing the conditions for using specific formulas do not match the comparison operators used in the [code](#):

```
if currentTick < lowerTick {
    // If the current price is less than or equal to the lower tick, then we use
    the liquidity0 formula.
    liquidity = Liquidity0(amount0, sqrtPriceA, sqrtPriceB)
} else if currentTick < upperTick {
    // If the current price is between the lower and upper ticks (inclusive of
    the lower tick but not the upper tick),
    // then we use the minimum of the liquidity0 and liquidity1 formulas.
    liquidity0 := Liquidity0(amount0, sqrtPrice, sqrtPriceB)
    liquidity1 := Liquidity1(amount1, sqrtPrice, sqrtPriceA)
    liquidity = sdk.MinDec(liquidity0, liquidity1)
} else {
    // If the current price is greater than the upper tick, then we use the
    liquidity1 formula.
    liquidity = Liquidity1(amount1, sqrtPriceB, sqrtPriceA)
}

return liquidity, nil
```


Problem Scenarios

The comments explaining the logic for selecting the liquidity formula based on the relationship between `currentTick`, `lowerTick`, and `upperTick` are inconsistent with the actual comparison operators used in the code.

Recommendation

To improve clarity and alignment between the comments and the code, we suggest one of the following options:

1. **Update Comment:** If the intention is to check for "less than or equal to," revise the comment for the first condition to match the code. For example:

```
if currentTick <= lowerTick {  
    // If the current price is less than or equal to the lower tick, then  
    we use the liquidity0 formula.  
    ...  
} else if currentTick < upperTick {  
    // If the current price is between the lower and upper ticks,  
    ...  
}
```

2. **Update Comparison:** If the intention is to check for "less than," update the comparison in the code to match the comments. For example:

```
if currentTick < lowerTick {  
    // If the current price is less than the lower tick, then we use the  
    liquidity0 formula.  
    ...  
} else if currentTick < upperTick {  
    // If the current price is between the lower and upper ticks (inclusive  
    of the lower tick but not the upper tick),  
    ...  
}
```

By aligning the comments and the code, you can enhance the readability and understanding of the code logic, reducing confusion for developers who review or maintain the code in the future.

Redundancies in GetAllPositionIdsForPoolId() function and ensurePositionOwner() usage

Title	Redundancies in GetAllPositionIdsForPoolId() function and ensurePositionOwner() usage
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- [osmosis/x/concentrated-liquidity/position.go](https://osmosis.zone/concentrated-liquidity/position.go)
- [osmosis/x/concentrated-liquidity/incentives.go](https://osmosis.zone/concentrated-liquidity/incentives.go)

Description

In the [PR #5237](#), it has been identified that the `poolId` parameter in the `GetAllPositionIdsForPoolId()` function appears to be redundant. The purpose of this function is to iterate over the store using a prefix that consists of both `poolId` and an address, or just the address if the `poolId` is not provided. However, the current implementation includes a `check` `keyPoolId == poolId || poolId == 0` to filter the position IDs based on the desired `poolId`. Since the iterators are expected to function correctly, this check is unnecessary and can be removed.

Additionally, the `ensurePositionOwner()` function is identified as potentially redundant. This function is called from `collectIncentives()` and `collectSpreadRewards()`, but before both calls, the `GetPosition()` function is used to retrieve the `Position` struct. The `Position` struct already contains the `address` and `poolId` fields, which are the same as the ones stored in the `KeyAddressPoolIdPositionId` (PositionPrefix KV Store). Therefore, instead of calling `ensurePositionOwner()` and iterating over the store, the check `position.Address == sender.String()` can be directly used, simplifying the code and avoiding unnecessary iterations.

Problem Scenarios

The current implementation of `GetAllPositionIdsForPoolId()` includes a redundant check `keyPoolId == poolId || poolId == 0`. This check adds unnecessary complexity to the function and is not required for iterating over the store using the provided prefix. Similarly, the usage of `ensurePositionOwner()` function before `collectIncentives()` and `collectSpreadRewards()` can be eliminated by utilizing the `Position` struct obtained from `GetPosition()` and directly comparing the address with `sender.String()`.

Recommendation

To improve the codebase and eliminate redundancies, the following recommendations are proposed:

1. Modify `GetAllPositionIdsForPoolId()` Function:
 - Remove the `poolId` parameter from the function since it is not needed.
 - Rename the function to `GetAllPositionIdsForPrefix` or simply `GetAllPositionIds` to reflect its updated purpose.
 - Remove the unnecessary check `keyPoolId == poolId || poolId == 0` as the iterators are expected to function correctly without this condition.
2. Eliminate Redundant Usage of `ensurePositionOwner()` :
 - Replace the calls to `ensurePositionOwner()` in `collectIncentives()` and `collectSpreadRewards()` with a direct check of `position.Address == sender.String()` using the `Position` struct obtained from `GetPosition()`.
3. Update the PositionPrefix KV Store Structure:
 - Consider changing the structure of the PositionPrefix KV store to `[owner, poolId] -> listOfPositionIds`.
 - Since the store is specifically used in `GetUserPositions()` / `GetAllPositionIdsForPoolId()`, this modified structure can provide a more efficient and logical representation.

Inefficient string concatenation

Title	Inefficient string concatenation
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- [osmosis/x/txfees/types/gov.go](#)

Description

In the `UpdateFeeTokenProposal.String()` function, there is an inefficient string concatenation operation performed inside a loop. Specifically, the code concatenates the string representation of `feeToken` values to the `recordsStr` variable using the `+` operator. This operation is repeated for each element in the `p.FeeTokens` slice.

Problem Scenarios

The current implementation of string concatenation using the `+` operator inside the loop can lead to performance inefficiencies and increased memory overhead. This is because strings in Go are immutable, so each concatenation operation creates a new string instance, resulting in unnecessary memory allocations.

Recommendation

To improve the performance and reduce memory overhead, it is recommended to utilize the `strings.Builder` type for string concatenation within the loop. Here is an updated version of the code:

```
func (p UpdateFeeTokenProposal) String() string {
    var b strings.Builder

    b.WriteString(fmt.Sprintf(`Update Fee Token Proposal:
Title:      %s
Description: %s
`, p.Title, p.Description))
}
```

```
    b.WriteString("  Records:  ")
    for _, feeToken := range p.Feetokens {
        b.WriteString(fmt.Sprintf("(Denom: %s, PoolID: %d) ", feeToken.Denom,
feeToken.PoolID))
    }

    return b.String()
}
```

By using `strings.Builder`, the code appends the string representations of `feeToken` values directly to the builder without creating intermediate string instances. This improves performance and reduces memory allocations, especially for large `p.Feetokens` slices.

Inefficient comparison methods and unnecessary object creation

Title	Inefficient comparison methods and unnecessary object creation
Project	Osmosis Q2 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- [osmosis/x/concentrated-liquidity/math/math.go](#)
- [osmosis/x/concentrated-liquidity/swaps.go](#)
- [osmosis/x/concentrated-liquidity/incentives.go](#)
- [osmosis/x/concentrated-liquidity/tick.go](#)
- [osmosis/osmoutils/accum/accum.go](#)

Description

The codebase contains inefficiencies related to comparison methods and unnecessary object creation. These issues can impact performance and increase memory usage.

1. **Inefficient Comparison Methods:** The codebase uses `.Equal(sdk.ZeroDec())` in multiple instances where `.IsZero()` would be a more efficient alternative. Similarly,
 - `.LT(sdk.ZeroDec())` can be replaced with `.IsNegative()`,
 - `.GT(sdk.ZeroDec())` with `.IsPositive()`,
 - `.LTE(sdk.ZeroDec())` with `!IsPositive()`, and
 - `.GTE(sdk.ZeroDec())` with `!IsNegative()`.
2. **Unnecessary Object Creation in Arithmetic Operations:** Immutable functions for arithmetic operations are used in situations where there is no need to create new objects. This leads to unnecessary memory allocation. To address this issue, it is recommended to replace immutable operations (such as `Add`, `Sub`, `Mul`, `Quo`) with their mutable counterparts (`AddMut`, `SubMut`, `MulMut`, `QuoMut`) where applicable. By utilizing mutable operations, the code can modify existing objects instead of creating new ones, resulting in improved performance and reduced memory overhead.

Problem Scenarios

Some parts of the codebase create new `sdk.Dec` objects unnecessarily, leading to potential memory overhead:

- The variable `denominator` at [math.go#L136](#) can be eliminated by utilizing mutable operations to modify the product value directly.
- Instead of using `.Sub`, `.SubMut` can be used at [math.go#L28](#) and [math.go#L51](#) to modify the operand in place.
- At [math.go#L33](#), `.MulMut` and `.QuoMut` can be used instead of creating new `sdk.Dec` objects.
- At [math.go#L56](#), a new `sdk.Dec` object can be avoided by utilizing `.QuoMut`.
- Instead of creating a new `newLiquidity` object at [swaps.go#L550-L552](#), the calculation can be performed without creating a new object. It can be achieved by using `swapState.liquidity.AddMut(liquidityNet)`.
- The `AddMut` function can be used for `totalSpreadFactors` at [swaps.go#L483-L485](#). In the same code section, the use of assignment operators seems unnecessary since the functions like `SubMut` already modify their operand.
- The code at [tick.go#L64-L68](#) can utilize `SubMut` and `AddMut` methods and remove the assignments for improved efficiency.
- Instead of using `.LTE(sdk.ZeroDec())`, the code at [accum.go#L232](#) can utilize `!IsPositive()` for better readability. Similarly, `IsZero()` can be used at [accum.go#L284](#), and `IsNegative()` at [accum.go#L375](#).
- The line at [swaps.go#L355](#) can be modified to `totalSpreadFactors.AddMut(spreadRewardCharge)` for a more efficient operation.
- The line at [swaps.go#L593](#) can be changed to `tokenIn.Amount.SubMut(spreadFactorsRoundedUp.Amount)` to avoid creating a new `sdk.Int` object.
- The code at [incentives.go#L377](#) can utilize `IsZero()` instead of comparing to `ZeroDec()`.
- The code at [incentives.go#L381-L383](#) can use `IsNegative()` instead of the current implementation for better efficiency.
- The code at [incentives.go#L543](#) can utilize `IsPositive()` instead of comparing to `ZeroDec()` for improved readability and efficiency.

Recommendation

In addition to the specific code locations mentioned earlier, we recommend conducting a thorough investigation of the codebase to identify other areas where similar optimizations can be applied.


Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of [Common Vulnerability Scoring System \(CVSS\) v3.1](#), which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the [Impact score](#), and the [Exploitability score](#). The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ [CVSS Qualitative Severity Rating Scale](#), and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

Impact Score	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
 None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

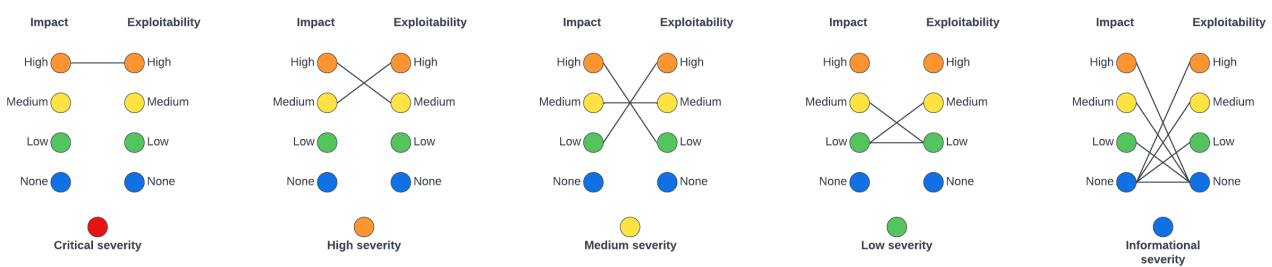
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
- *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

Exploitability Score	Examples
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
● None	illegitimate actions taken in a coordinated fashion by all actors


Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.



As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Severity Score	Examples
● Critical	Halting of chain via a submission of a specially crafted transaction

Severity Score	Examples
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
 Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal Systems has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal Systems makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal Systems to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.