Blockchain Protocol Security Analysis Report

Customer: Elys

Date: 07/03/2025

We express our gratitude to the Elys team for the collaborative engagement that enabled the execution of this Blockchain Protocol Security Assessment.

Elys Network is a versatile DeFi platform built on the Cosmos SDK, offering a comprehensive suite of financial tools including AMMs, perpetual futures trading, leveraged liquidity pools, and stablecoin staking. It prioritizes a secure and transparent environment where users control their assets and participate in governance. With its modular design, Elys seamlessly integrates new features like trade shield, token deflation, and accounted pools, ensuring adaptability within the evolving DeFi landscape.

Document

Name	Blockchain Protocol Review and Security Analysis Report for Elys
Audited By	Tanuj Soni, Reza Mir
Approved By	Nino Lipartiia
Website	https://elys.network/
Changelog	13/12/2024 - First Preliminary Report
	24/01/2025 - Second Preliminary Report
	29/01/2025 - Third Preliminary Report
	07/03/2025 - Final Report
Platform	Elys Network
Language	Golang
Tags	Cosmos, IBC, Perpetual Trading, DeFi, Staking, Stablestake, Leverage LP
Methodology	https://hackenio.cc/blockchain_methodology

Review Scope

Repository	https://github.com/elys-network/elys/
Commit	99bd50ba942ef856291aec1705e0d3acded58444

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report.

24	10	10	4
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	1
High	4
Medium	1
Low	8

Vulnerability	Severity
F-2025-8500 - Critical MEV Vulnerability in TradeShield's Order Execution	Critical
F-2024-7467 - Vulnerable Dependencies	High
F-2024-7600 - Price Exploitation from Missing On-Chain TWAP Enforcement	High
F-2025-8217 - Missing Liquidator Reward Implementation in Perpetual Trading System	High
F-2025-8461 - Inadequate Protections Against Funding Rate Manipulation	High
F-2024-7593 - Price Manipulation and Feeder Centralization Risks	Medium
F-2024-7689 - Lack of Insurance Fund in Elys Perpetual Trading Module	Low
F-2024-7736 - Counterproductive Lockup in Oracle Pools Kills Market Efficiency	Low
F-2024-8019 - External Liquidity Manipulation Vulnerability in AMM Oracle Pools with Leveraged LP Positions	Low
F-2025-8082 - Ignored Non-Existent Pools in ‘ConvertGasFeesToUsdc’ Function	Low
F-2025-8268 - Insufficient Protections in Elys Protocol AMM Logic	Low
F-2025-8341 - Division by Zero Leading to Potential Panics	Low
F-2025-8434 - Insufficient Market-Specific Leverage Limits and Missing Authority Controls	Low
F-2025-8435 - Insufficient Slippage Protection in IBC-Triggered Swaps	Low
F-2024-7484 - Code Quality Improvement	Info
F-2024-7523 - Telemetry Configs	Info
F-2024-7616 - Non-Deterministic Map Iteration	Info

Vulnerability	Severity
F-2024-7935 - Mismatch Between Comment and Return Value in Voting Power Calculations	Info
F-2024-7959 - Proofreading	Info
F-2024-7989 - Suggested Improvements to the Cosmos SDK Math Library	Info
F-2025-8077 - Clarifying Pool Existence Handling in the 'Calculate Pool APRs' Query	Info
F-2025-8154 - Identical Implementation in 'RemovePoolInfo' and 'RemoveLegacyPoolInfo' Functions	Info
F-2025-8162 - Code Quality Concerns in WithdrawElysStakingRewards	Info
F-2025-8188 - Redundant Price Removal Logic in EndBlock Function	Info

Documentation quality

- User-facing documentation for features and use cases is well-organized and readily accessible.
- Each custom module in the *x* directory is supported by detailed documentation, including READMEs, conceptual overviews, usage guides, keeper descriptions, and protobuf definitions.
- However, several modules have outdated documentation that requires revisions to reflect the current implementation accurately.
- The documentation lacks clarity on module interactions, particularly the integration of the AMM, Oracle pool, and perpetual module, as well as the execution and settlement processes for perpetual and spot orders.
- Guides for setting up a local network using Docker or Ignite are available but require updates to address compatibility and build issues.
- Documentation on off-chain components, such as the `PriceFeeder`, could be enhanced to include plans for decentralization and security measures as the Elys team transitions to the OjO network. Additionally, dependencies on third-party protocols, such as distributed oracles, should be thoroughly documented.
- IBC Documentation: Cross-chain (IBC) integration processes require expanded documentation.

Code quality

- The codebase adheres to Go programming best practices, supporting maintainability and optimal performance.
- Custom modules include test cases; however, expanding coverage and diversifying scenarios would strengthen security and stability.
- Certain functions exhibit high cyclomatic complexity, excessive length, and deeply nested structures, highlighting opportunities for refactoring.

Architecture quality

- Elys Network utilizes the Cosmos SDK, a mature and widely-adopted framework for building blockchains, contributing to a robust technical foundation.
- The Elys Network project exhibits a pragmatic architectural approach by inheriting and adapting complex modules from established projects like Osmosis and Evmos. This strategy accelerates development and leverages proven components for enhanced reliability.
- The overall architecture quality is modular and pragmatic, prioritizing efficiency and leveraging existing solutions.
- The architecture and design of external price feed to the system must be improved to ensure feeds are tamper-proof.

Table of Contents

System Overview

Elys Network is a DeFi platform built using the Cosmos SDK and featuring a range of custom modules extending its core functionalities. This overview prioritizes the description of these custom modules and their interactions.

1. **estaking:** Adds liquid staking, and burns EdenB tokens based on Elys staking changes.
2. **leveragelp:** Enables leveraged liquidity pool positions (higher risk/reward).
3. **amm:** Handles swaps via multiple pool types (standard, oracle), integrates with leveragelp/accountedpool, charges creation fees.
4. **accountedpool:** Tracks pool balances for shielded AMM pools and impermanent loss protection.
5. **perpetual:** Enables leveraged perpetual contracts with funding rates and liquidations.
6. **tradeshield:** Security layer against trading exploits.
7. **transferhook:** Enables IBC transfers with built-in token swaps.
8. **assetprofile:** Manages asset-specific settings.
9. **parameter:** Governance-controlled system parameter updates.
10. **oracle:** Provides asset price feeds for valuations/liquidations.
11. **stablestake:** Manages stablecoin staking/loans with dynamic rates.
12. **tier:** User tiers with activity-based benefits.
13. **tokenomics:** Controls token distribution/incentives.
14. **commitment:** Manages user commitments and token locks.
15. **burner:** Burns tokens regularly via epoch triggers.
16. **epochs:** Schedules timed events/triggers.
17. **masterchef:** Distributes liquidity provider rewards.

Risks

- Due to the active development of the protocol, new features and code changes are continuously introduced. These ongoing modifications may inadvertently introduce vulnerabilities or inconsistencies that fall outside the scope of this audit.

Findings

Vulnerability Details

F-2025-8500 - Critical MEV Vulnerability in TradeShield's Order Execution - Critical

Description:

The TradeShield module, designed to ensure secure and fair trading, suffers from a critical vulnerability to Miner Extractable Value (MEV) attacks, specifically within its order execution logic. The `ExecuteOrders` function, despite dealing with orders already on-chain, allows for MEV exploitation due to the way it allows arbitrary selection and execution of existing orders. This is worsened by existing vulnerabilities (*F-2024-7600*, *F-2025-8461*, *F-2024-7593*) that allow price and funding rate manipulation.

Vulnerabilities in `ExecuteOrders`

The core of the MEV vulnerability lies within the `ExecuteOrders` function in `x/tradeshield/keeper/msg_server_execute_orders.go`:

```
// From x/tradeshield/keeper/msg_server_execute_orders.go
func (k msgServer) ExecuteOrders(goCtx context.Context, msg *types.
MsgExecuteOrders) (*types.MsgExecuteOrdersResponse, error) {
ctx := sdk.UnwrapSDKContext(goCtx)

// Sequential processing of spot orders with attacker controlled Sp
otOrderIds
for _, spotOrderId := range msg.SpotOrderIds {
spotOrder, found := k.GetPendingSpotOrder(ctx, spotOrderId)
if !found {
return nil, types.ErrSpotOrderNotFound
}
// ... order execution logic
}

// Sequential processing of perpetual orders with attacker controll
ed PerpetualOrderIds
for _, perpetualOrderId := range msg.PerpetualOrderIds {
// ... similar sequential processing
}
}
```

This function exhibits a critical vulnerability that makes it susceptible to MEV exploitation.
The `ExecuteOrders` function allows *anyone* to submit a list of `SpotOrderIds` and `PerpetualOrderIds` to be executed. This means a malicious validator (or any user) can selectively choose which orders to include in the `MsgExecuteOrders` transaction, effectively controlling the order of execution and potentially manipulating the market.

Amplifying Issues

The following existing issues exacerbate the MEV vulnerability in `ExecuteOrders`:

- **F-2024-7600 | Price Manipulation Due to Lack of On-Chain TWAP Enforcement:** The lack of on-chain TWAP (Time-Weighted Average Price) enforcement allows for price manipulation, making it easier for attackers to exploit `ExecuteOrders` to their advantage. By manipulating prices, they can create more profitable scenarios for executing selected orders.
- **F-2025-8461 | Funding Rate Manipulation:** The potential for funding rate manipulation further increases the risk associated with `ExecuteOrders`. Attackers can manipulate funding rates to create arbitrage opportunities or influence the profitability of certain positions, which they can then exploit by selectively executing orders.
- **F-2024-7593 | Price Manipulation and Feeder Centralization Risks:** Centralization risks in price feeders can be exploited to manipulate prices, creating an environment where MEV extraction through `ExecuteOrders` becomes even more profitable and easier to execute.

These issues, combined with the arbitrary order selection vulnerability in `ExecuteOrders`, create a critical situation where malicious actors have significant leverage to manipulate the market and extract value at the expense of regular users.

How this Enables MEV

- **Front-running and Back-running:** A validator can observe pending orders in the mempool (which are different from the `SpotOrderIds` in `ExecuteOrders`) and then use `ExecuteOrders` to prioritize the execution of specific existing orders that would benefit them. For example, they could execute a buy order before a large market buy order in the mempool, driving up the price and profiting from the subsequent price impact.
- **Creating Favorable Matches:** By selectively executing orders, validators can create favorable matches for themselves or others. They could, for instance, execute a sell order at a higher price than it would otherwise be filled if other orders were prioritized.
- **Denial of Service (DoS):** Malicious actors could flood the network with `ExecuteOrders` transactions, each containing a carefully selected set of orders designed to disrupt the market or prevent specific users from having their orders filled.

Impact Assessment

The MEV vulnerability in `ExecuteOrders`, amplified by the existing issues, has severe consequences:

- **Financial Impact:** Users can suffer significant financial losses due to manipulated order execution. This can result in users paying higher prices, receiving lower returns, and missing potentially profitable trading opportunities.
- **Market Distortion:** MEV distorts market prices and impedes efficient price discovery. This leads to artificial price fluctuations, increased volatility, and suboptimal market outcomes, eroding the integrity and efficiency of the DeFi ecosystem.
- **Unfair Advantage:** This vulnerability gives an unfair advantage to validators and technically savvy users who can understand and exploit the order execution logic.
- **Loss of Trust:** The presence of this MEV vulnerability undermines trust in the fairness and security of the TradeShield platform.

Assets:

- tradeshield [https://github.com/elys-network/elys]

Status: Mitigated

Classification

Impact:	5/5
Likelihood:	4/5
Severity:	Critical

Recommendations

Remediation:

Order Selection and Execution:

- Remove external control over order execution in `ExecuteOrders`.
- Implement a strict FIFO queue for order processing.
- Develop automated order selection logic based on predefined rules.
- Introduce a minimum block delay between order submission and execution.
- Implement time-based batching of orders.
- Define execution window constraints.

Batch Processing:

- Implement volume-based batching.
- Randomize execution order within batches.
- Utilize a uniform clearing price mechanism.

DoS Protection:

- Enforce transaction rate limiting on `ExecuteOrders`.
- Optimize gas costs associated with `ExecuteOrders`.
- Set maximum batch size limits.

Emergency Controls:

- Implement a circuit breaker to halt trading under specific conditions.
- Develop emergency pause functionality for order execution.

Future Considerations:

- **Advanced Privacy Features:** Explore commit-reveal schemes, order obfuscation, and secure aggregation.
- **Decentralized Execution:** Explore MEV auctions and multi-party computation (MPC).

Resolution:

The implemented solution addresses MEV vulnerability by incorporating sender address bytes into swap request keys, creating a deterministic but pseudo-random storage ordering that affects retrieval sequence during execution. Order settlement logic in the AMM module now fetches requests based on this randomized ordering via functions like `GetFirstSwapExactAmountInRequest` and `GetFirstSwapExactAmountOutRequest`, making it more difficult for validators to predict which orders will be processed first. While this approach provides some protection against MEV attacks by distributing orders based on sender addresses, it maintains certain limitations. The `ExecuteOrders` function interface still allows explicit order selection, the solution depends on AMM module implementation rather than providing TradeShield-specific safeguards, and transaction ordering within blocks remains susceptible to validator manipulation. This represents a partial but meaningful improvement to the system's resistance against MEV exploitation.

Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc

---

Evidences

Attacks Combining ExecuteOrders with Other Issues

Reproduce:

F-2024-7600 (Lack of On-Chain TWAP Enforcement):

- Manipulate the oracle price at the last moment before an update.
- Use `ExecuteOrders` to front-run or sandwich trades based on the manipulated price.

F-2025-8461 (Potential Funding Rate Manipulation):

- Manipulate the funding rate by exploiting vulnerabilities in its calculation.
- Use `ExecuteOrders` to execute orders that benefit from the manipulated funding rate (e.g., open positions, trigger liquidations).

F-2024-7593 (Price Manipulation and Feeder Centralization Risks):

- Exploit centralized price feeders to manipulate oracle prices.
- Use `ExecuteOrders` to amplify the price manipulation and execute trades at favorable prices.
- This could involve:
  - Last-minute price manipulation combined with front-running.
  - Oracle manipulation combined with sandwich attacks.
  - Combined oracle and order book manipulation to create arbitrage opportunities.

General MEV Attacks via ExecuteOrders

Reproduce:

Front-Running:

- Monitor the mempool for profitable trades (e.g., large buy orders).
- Select existing orders that will create favorable price impact.
- Execute those orders before the target transaction using `ExecuteOrders`.
- Profit from the resulting price movement.

Sandwich Attack:

- Identify large pending orders in the mempool.
- Execute favorable orders before the target order using `ExecuteOrders` (e.g., buy before a large buy).
- Allow the target transaction to execute at the manipulated price.
- Execute opposing orders after the target (e.g., sell after a large buy).
- Profit from the price difference.

Denial-of-Service (DoS):

- Flood the network with `ExecuteOrders` transactions containing strategically selected orders.
- Aim to block or delay the execution of other users' orders, disrupting the market.

Order Book Manipulation:

- Monitor the order book for opportunities to manipulate prices.
- Use `ExecuteOrders` to selectively execute orders that create artificial price movements or imbalances.
- Profit from the manipulated prices by executing trades elsewhere or waiting for market correction.

Funding Rate Manipulation (Perpetuals):

- Identify situations where the funding rate is susceptible to manipulation.
- Use `ExecuteOrders` to execute specific orders that influence the funding rate calculation.
- Profit from the manipulated funding rate by holding positions that benefit from the artificial rate.

F-2024-7467 - Vulnerable Dependencies - High

Description:

A comprehensive security analysis conducted using the tools `govulncheck` and `Snyk` identified several potential vulnerabilities in the libraries utilized by the project. These findings emphasize the urgent need to address these risks promptly to maintain the system's security and stability.

Go Binary Security Review:

- **A. Description**: `go/build/constraint`: stack exhaustion in `Parse`.
  - **Impact**: Calling `Parse` on a "`// +build`" build tag line with deeply nested expressions can cause a panic due to stack exhaustion.
  - **GitHub Advisory**: GHSA-j7vj-rw65-4v26
  - **Affected Versions**: < go 1.23.0-0
  - **Fixed in**: go 1.23.1
  - **CVSS Score**: 7.5 (High)
- **B. Description**: Stack exhaustion in `Decoder.Decode` in `encoding/gob`.
  - **Impact**: Calling `Decoder.Decode` on a message which contains deeply nested structures can cause a panic due to stack exhaustion.
  - **GitHub Advisory**: GHSA-crqm-pwhx-j97f
  - **Affected Versions**: < go 1.23.0-0
  - **Fixed in**: go 1.23.1
  - **CVSS Score**: 7.5 (High)
- **C. Description**: Stack exhaustion in all `Parse` functions in `go/parser`.
  - **Impact**: Calling any of the Parse functions on Go source code which contains deeply nested literals can cause a panic due to stack exhaustion.
  - **GitHub Advisory**: GHSA-8xfx-rj4p-23jm
  - **Affected Versions**: < go 1.23.0-0
  - **Fixed in**: go 1.23.1
  - **CVSS Score**: N/A

Security Analysis of External Go Dependencies:

github.com/cometbft/cometbft : v0.38.12

- **A. Description**: CometBFT Vote Extensions: Panic when receiving a Pre-commit with an invalid data
  - **Impact**: A CometBFT node running in a network with vote extensions enabled could produce an invalid Vote message and send it to its peers. The invalid field of the Vote message is the `ValidatorIndex`, which identifies the sender in the `ValidatorSet` running that height of consensus. This field is

ordinarily verified in the processing of Vote messages, but it turns out that in the case of a Vote message of type `Precommit` and for a non-nil BlockID, a logic was introduced before this ordinary verification to handle the attached vote extension. This introduced logic (not present in releases prior to `0.38.x`) does not double-check the validity of the `ValidatorIndex` field. The result is a panic in the execution of the node receiving and processing such message.
    - **GitHub Advisory**: GHSA-p7mv-53f2-4cwj
    - **Affected Versions**: >= 0.38.0, < 0.38.15
    - **Fixed in**: 0.38.15
    - **CVSS Score**: 8.3 (High)
- **B. Description**: CometBFT's default for `BlockParams.MaxBytes` consensus parameter may increase block times and affect consensus participation.
    - **Impact**: This issue does not represent an actively exploitable vulnerability that would result in a direct loss of funds, however it may have a slight impact on block latency depending on a network's topography.
    - **GitHub Advisory**: GHSA-hq58-p9mv-338c
    - **Affected Versions**: All
    - **Fixed in**: None
    - **CVSS Score**: N/A (Low)

cosmos/cosmos-sdk : v0.50.9

- **A. Description**: The `x/crisis` package does not cause chain halt.
    - **Impact**: If an invariant check fails on a Cosmos SDK network and a transaction is sent to the `x/crisis` module to halt the chain, the chain does not halt.
    - **GitHub Advisory**: GHSA-qfc5-6r3j-jj22
    - **Affected Versions**: All versions
    - **Fixed in**: None
    - **CVSS Score**: N/A (Low)
- **B. Description**: The `x/crisis` package does not charge `ConstantFee`.
    - **Impact**: If a transaction is sent to the `x/crisis` module to check an invariant, the `ConstantFee` parameter of the chain is NOT charged.
    - **GitHub Advisory**: GHSA-w5w5-2882-47pc
    - **Affected Versions**: All versions
    - **Fixed in**: None
    - **CVSS Score**: N/A (Low)

github.com/golang/protobuf v1.5.4

- **A. Description**: Deprecated: Use the "google.golang.org/protobuf" module instead.

github.com/docker/distribution : v2.8.1+incompatible

- **A. Description**: distribution catalog API endpoint can lead to OOM via malicious user input.
  - **Impact**: Systems that run `distribution` built after a specific commit running on memory-restricted environments can suffer from denial of service by a crafted malicious `/v2/_catalog` API endpoint request.
  - **GitHub Advisory**: GHSA-hqxw-f8mx-cpmw
  - **Affected Versions**: < 2.8.2-beta.1
  - **Fixed in**: v2.8.2+incompatible
  - **CVSS Score**: 7.5 (High)

Security Analysis of External NPM Dependencies:

google-spreadsheet : ^4.0.2

- **A. Description**: Server-side Request Forgery (SSRF)
  - **Impact**: `Axios` is a promise-based HTTP client for making asynchronous requests in both browser and Node.js environments. `Axios` is vulnerable to a Server-Side Request Forgery attack caused by unexpected behavior where requests for path relative URLs get processed as protocol relative URLs. This could be leveraged by an attacker to perform arbitrary requests from the server, potentially accessing internal systems or exfiltrating sensitive data.
  - **GitHub Advisory**: GHSA-8hc4-vh64-cxmj
  - **Affected Versions**: < 4.1.3
  - **Fixed in**: 4.1.3
  - **CVSS Score**: **7.5** (High)

Assets:

- Dependencies [https://github.com/elys-network/elys]

Status: Fixed

Classification

Impact:	4/5
Likelihood:	3/5
Severity:	High

Recommendations

Remediation: To ensure the security and stability of the project, it is crucial to address the vulnerabilities identified in its dependencies.

While updating all dependencies to their latest versions is ideal, the following minimum versions are required to address the identified vulnerabilities:

- **Upgrade minimum Go version:**
  - Change `go 1.22.6` to `go 1.23.1`
- **Direct vulnerable dependencies:**
  - `github.com/cometbft/cometbft` `v0.38.15`
  - `github.com/docker/distribution` `2.8.2-beta.1`
  - `google-spreadsheet` `4.1.4`
- **Eliminate deprecated dependencies:**
  - `github.com/golang/protobuf`
- **Todo:**
  - There are two "`TODO: remove it`" comments in the `go.mod` file. Please review them.

Implementing these recommendations will significantly enhance the project's defense against known vulnerabilities, ensuring a more secure and stable environment moving forward.

Resolution: The Elys team will document exceptions for required dependencies including `github.com/golang/protobuf 1.5.4` and indirect dependencies, implementing compensating controls where feasible. The project has already upgraded to `Go 1.23.1` and updated `Cosmos SDK to v0.50.12` and `CometBFT to v0.38.17`, addressing several security concerns. Unused code such as `google-spreadsheet` will be removed from the codebase. Regular security monitoring will be established through `govulncheck` and `Snyk` tools to provide ongoing vulnerability assessment. This approach balances ecosystem compatibility requirements with security needs while aligning with established patterns in the Cosmos ecosystem.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2024-7600 - Price Exploitation from Missing On-Chain TWAP Enforcement - High

Description:

The `x/oracle` module is vulnerable to price manipulation due to the absence of on-chain enforcement of a time-weighted average price (TWAP). This vulnerability is exacerbated by the reliance on centralized exchanges for price feeds, particularly for assets where decentralized oracles, such as Band Protocol, may not be available. These assets often suffer from low liquidity on centralized exchanges, making them more prone to manipulation. Malicious actors could exploit these conditions to create artificial price fluctuations, which would then be reflected in the `x/oracle` module, potentially leading to losses in leveraged positions within the `x/leveragelp` and `x/perpetual` modules.

This issue is not merely hypothetical. Several DeFi protocols have faced significant financial losses due to similar exploits. Here is an article mentioning Oracle manipulation and exploits https://www.chainalysis.com/blog/oracle-manipulation-attacks-rising/

Vulnerability:

- **Lack of TWAP Validation:** The `x/oracle` module does not validate submitted prices against a time-weighted average, allowing outlier values to be accepted even if they deviate significantly from the historical price trend.
- **Exploiting Centralized Exchange Weaknesses:** Individuals can manipulate prices on centralized exchanges, especially for assets with low liquidity, and these manipulated prices can be propagated to the `x/oracle` module due to the lack of on-chain TWAP validation. This is exacerbated by Elys Network's support for assets not covered by distributed oracles like Band Protocol, which inherently smooth out the price feeds.

Impact:

- **Unfair Liquidations `x/leveragelp`, `x/perpetual`:** Users can lose their collateral due to manipulated liquidations, even if their positions were not genuinely at risk. This is particularly likely for assets with low liquidity on centralized exchanges, where price manipulation is easier to achieve.
- **Arbitrage Exploitation `x/amm`:** Individuals can drain funds from AMM pools by exploiting the price discrepancy between the manipulated price and the actual market value.
- **Incorrect Tradeshield Payouts `x/tradeshield`:** Tradeshield mechanisms may execute trades or issue payouts based on the manipulated price, leading to financial losses for users.

- **Distorted Market Signals:** The manipulated price disrupts the accurate reflection of market sentiment and can negatively influence other on-chain protocols and applications.

This issue is further compounded by the current "last submitter wins" logic (*F-2024-7593*) which allows a single malicious actor to control the price. While the implementation of a decentralized aggregation mechanism will mitigate this specific vulnerability, the lack of on-chain TWAP enforcement will persist and needs to be addressed separately.

Assets:

- oracle [https://github.com/elys-network/elys]

Status: Mitigated

Classification

Impact: 5/5

Likelihood: 3/5

Severity: High

Recommendations

Remediation:

1. **On-Chain TWAP Enforcement:**
   1. **Calculate TWAP on-chain:** The `x/oracle` module should calculate the TWAP based on historical price data, potentially incorporating data from decentralized exchanges. This would involve storing historical price data on-chain and implementing a robust TWAP calculation algorithm within the module itself.
   2. **Define acceptable deviation thresholds:** Establish clear deviation thresholds from the TWAP. Any submitted price that falls outside these thresholds should be automatically rejected. This prevents extreme outliers from influencing the oracle price.
   3. **Implement circuit breakers:** Incorporate a circuit breaker mechanism that temporarily halts price updates if a submitted price deviates significantly from the TWAP. This allows for investigation and manual intervention if necessary, preventing potentially manipulated prices from affecting the system.
2. **Reduce Reliance on Centralized Exchanges:**
   1. **Diversify price sources:** Incorporate price data from decentralized exchanges, which are less susceptible to

manipulation due to their transparent and open nature.
   2. **Implement stricter validation rules:** Introduce more rigorous validation rules for prices fetched from centralized exchanges. This could include outlier detection mechanisms, volume analysis, and cross-referencing with other data sources.
3. **Enhancements to Existing Mechanisms:**
   1. **Optimize time-weighted average with decay:** Ensure the time-weighted average uses an appropriate decay function that gives more weight to recent prices while preventing excessive influence from last-minute submissions. This balances the need for responsiveness to market changes with protection against manipulation.
   2. **Implement rate limiting:** Limit the frequency of price updates from individual oracles. This makes it more difficult for attackers to manipulate the TWAP by spamming the system with a series of manipulated prices.

Resolution:

Oracle Module has been replaced by Ojo Network Oracle Module, which uses Vote Extension to bring prices of assets on the chain. Mainnet migration of the fixes is still in progress.

Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc

---

Evidences

A Step-by-Step Attack (with Draining Funds)

Reproduce:

This attack exploits the vulnerability of relying on centralized exchanges (CEXs) for price feeds, especially when those CEXs have low liquidity for certain assets.

1. Identify a Vulnerable Asset

- The attacker researches and identifies an asset with low liquidity on a CEX that is used as a price source for the target protocol's oracle.
- This asset should ideally not have a decentralized price feed alternative (like Band Protocol) available, as those are generally more resistant to manipulation.

2. Take a Leveraged Position

- The attacker takes a leveraged long (betting on price increase) or short (betting on price decrease) position on the target asset within the DeFi protocol.

- This leveraged position allows them to magnify their gains (or losses) based on price movements.

3. Manipulate the Price on the CEX

- **Wash Trading:** The attacker trades the asset with themselves, creating artificial buying or selling pressure to drive the price in the desired direction. This creates the illusion of increased trading volume and legitimate price movement.
- **Exploiting Low Liquidity:** Because the asset has low liquidity, the attacker can move the price significantly with relatively small orders. They place large buy or sell orders to push the price up or down.

4. Trigger Liquidations or Favorable Price Movement & Drain Funds

The manipulated price from the CEX is propagated to the DeFi protocol's oracle.

Draining AMMs:

- **Mechanism:** The attacker exploits the artificially inflated/deflated price to make trades with the AMM, receiving more tokens than they should at the true market value. This is often done by swapping a large amount of the manipulated asset for other tokens in the pool.
- **Draining Amounts:** This depends on the pool size, the magnitude of price manipulation, and the AMM's specific algorithm. Larger pools and greater price differences allow for larger drains.
- **Likelihood and Time:** This is time-sensitive. The attacker needs to act before arbitrageurs and the market correct the price, potentially within minutes. The likelihood of success depends on the attacker's ability to manipulate the price significantly and execute the swap quickly. In some cases, they might be able to drain a significant portion of the pool's funds within a few minutes.

Draining Perpetuals:

- **Mechanism:** The attacker's leveraged position becomes highly profitable due to the manipulated price. They can then close their position, taking massive profits from the protocol. This essentially drains funds from the perpetual protocol's insurance fund or from other traders who took the opposite position.
- **Draining Amounts:** This depends on the size of the attacker's position, the leverage used, and the extent of price manipulation. Higher leverage and larger price swings lead to larger drains.

- **Likelihood and Time:** This also depends on the speed of market correction. However, perpetual protocols often have mechanisms like funding rates that can gradually reduce the attacker's profits if the manipulation persists. The attacker might need anywhere from a few minutes to a few hours to execute this attack, depending on the market's reaction to the price manipulation and the specific perpetual protocol's parameters. The likelihood of success depends on the attacker's ability to maintain the price manipulation for a sufficient period and avoid countermeasures from the protocol.

F-2025-8217 - Missing Liquidator Reward Implementation in Perpetual Trading System - High

Description:

The Elys Protocol perpetual trading module lacks a critical component: a functional liquidator reward mechanism. Despite the design specification in `x/perpetual/spec/incentivized_liquidation_system.md` explicitly outlining the need for such a mechanism, the codebase contains no logic for calculating or distributing rewards to liquidators.

This absence creates a significant security risk and undermines the economic stability of the system. Liquidators, who play a vital role in maintaining system health by closing unhealthy positions, have no financial incentive to participate. This can lead to:

- **Delayed Liquidations:** Increased risk of bad debt accumulation and potential system instability due to unaddressed unhealthy positions.
- **Higher Gas Costs for Liquidators:** Discourages participation, as liquidators would have to pay gas fees to perform liquidations without receiving any compensation. This further delays liquidations and exacerbates the accumulation of unhealthy positions.
- **Market Inefficiency:** Delayed liquidations can distort market prices and hinder efficient price discovery.
- **Reduced User Confidence:** Lack of a clear reward mechanism can erode trust in the protocol's long-term viability and discourage participation in the perpetual market.

This issue is compounded by the lack of an insurance fund (as described in *F-2024-7689*). Without an insurance fund to cover losses from unhealthy positions, the burden falls on the protocol itself, potentially leading to insolvency. The absence of liquidator rewards further exacerbates this risk, as there is no incentive for external parties to help mitigate losses by closing underwater positions.

Interplay with other Issues:

- ***F-2025-846* (ExecuteOrders Vulnerability):** The ability to selectively execute orders via `ExecuteOrders` could be used by attackers to further exploit the lack of liquidation incentives. They could manipulate the market to create a large number of unhealthy positions, knowing that there is no immediate consequence due to the absence of incentivized *liquidators.*
- ***F-2024-7600* (Price Manipulation):** Attackers could manipulate oracle prices to artificially create unhealthy

positions, and then exploit the lack of liquidator rewards to maintain these positions without being liquidated.
- ***F-2025-8461* (Funding Rate Manipulation):** Attackers could manipulate funding rates to make certain positions more likely to become unhealthy, and then exploit the lack of liquidation incentives to profit from these positions.

Assets:

- perpetual [https://github.com/elys-network/elys]
- tradeshield [https://github.com/elys-network/elys]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 3/5

Severity: High

Recommendations

Remediation: **Implement the following:**

- **Implement the incentivized liquidation system as outlined in the design specification.** This includes defining the reward structure, calculating rewards based on market conditions, and distributing those rewards to liquidators.
- **Introduce gas compensation mechanisms** to reimburse liquidators for the gas fees incurred during the liquidation process.
- **Set minimum reward thresholds** to ensure that liquidators are incentivized even for smaller positions.

Additional Considerations:

- **Dynamic Reward Calculation:** Consider implementing a dynamic reward calculation mechanism that adjusts rewards based on market conditions, such as the severity of the liquidated position or the gas price.
- **Multi-tier Rewards:** Explore the possibility of introducing a multi-tier reward system that provides higher rewards to liquidators based on their performance and contribution to the protocol's stability.
- **Advanced Incentive Structures:** Investigate more sophisticated incentive structures that incentivize liquidators to

actively monitor the market and identify potential liquidation opportunities.

Resolution: A comprehensive liquidator reward mechanism has been implemented to incentivize liquidators to close unhealthy positions. Currently, on the testnet, the Elys team will move the upgrades to the mainnet after testing.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2025-8461 - Inadequate Protections Against Funding Rate Manipulation - High

Description: The current funding rate implementation in the `x/perpetual` module is vulnerable to manipulation due to several design flaws. Attackers can exploit the system by strategically opening and closing positions to influence funding rates and extract value from other traders. This issue is exacerbated by the vulnerability identified in *F-2025-8500*, which allows arbitrary order selection and execution through the `ExecuteOrders` function. This combination creates a critical situation where attackers can manipulate both funding rates and order execution to maximize their profits at the expense of other users.

Funding Rate Calculation Flaws:

```
func (k Keeper) ComputeFundingRate(ctx sdk.Context, pool types.Pool
) (math.LegacyDec, math.LegacyDec) {
totalLongOpenInterest := pool.GetTotalLongOpenInterest()
totalShortOpenInterest := pool.GetTotalShortOpenInterest()

// Vulnerability: Direct ratio calculation without smoothing
if totalLongOpenInterest.GT(totalShortOpenInterest) {
netLongRatio := (totalLongOpenInterest.Sub(totalShortOpenInterest))
.ToLegacyDec().Quo((totalLongOpenInterest.Add(totalShortOpenInteres
t)).ToLegacyDec())
return netLongRatio.Mul(fixedRate), math.LegacyZeroDec()
}
}
```

The funding rate is calculated directly based on the current ratio of long and short open interest without considering historical data or smoothing. This makes it highly sensitive to position size changes and allows attackers to easily influence the rate with large positions.

Missing Protections:

- *No Rate Smoothing:* The lack of rate smoothing mechanisms (e.g., moving averages, time-weighted averages) allows for abrupt and potentially manipulated changes in the funding rate.
- *No Position Size Limits:* There are no caps on the influence of individual positions on the funding rate calculation. This allows large positions to dominate the rate and makes it easier for attackers with significant capital to manipulate it.

Block-Level Updates Weaknesses:

```
func (k Keeper) BeginBlocker(ctx sdk.Context) {
// Updates funding rate every block
fundingRateLong, fundingRateShort := k.ComputeFundingRate(ctx, pool
)
// ... sets new rates immediately
}
```

The funding rate is updated every block, based solely on the current state of the order book and prices within that block. This creates opportunities for frequent manipulation, as attackers can exploit short-term fluctuations in prices and open interest to influence the rate.

Additionally, the `BeginBlocker` only considers the current state and does not take into account the funding rate or market conditions of previous blocks. This can lead to severe fluctuations in the funding rate, especially if there are significant price movements or liquidations within a single block.

How *F-2025-8500* Amplifies the Issue:

- *Precise Timing of Attacks:* Attackers can use `ExecuteOrders` to execute large positions immediately before a funding rate update, maximizing their impact on the rate.
- *Creating Artificial Imbalances:* By selectively executing orders, attackers can create artificial imbalances in open interest, further distorting the funding rate.
- *Triggering Liquidations:* Attackers can manipulate the funding rate and then use `ExecuteOrders` to execute orders that trigger the liquidation of vulnerable positions.

Impact Assessment:

1. *Financial Impact:*
    1. Direct manipulation of funding payments, leading to unfair costs for traders.
    2. Forced liquidations due to manipulated funding rates.
    3. Unfair cost distribution, with attackers profiting at the expense of honest traders.
2. *Market Impact:*
    1. Artificial position imbalances created by manipulation.
    2. Distorted funding rates that do not accurately reflect market sentiment.
    3. Reduced market efficiency due to unpredictable and manipulated funding costs.
3. *System Impact:*
    1. Reduced market stability due to increased volatility and risk.
    2. Loss of trader confidence in the fairness and reliability of the perpetuals market.
    3. Potential for system gaming and exploitation, discouraging participation.

Assets:

- perpetual [https://github.com/elys-network/elys]

Status: Mitigated

Classification

Impact:	4/5
Likelihood:	3/5
Severity:	High

Recommendations

Remediation:

- **Reduce Sensitivity to Large Positions:** Adjust the funding rate calculation to be less sensitive to large individual positions. Consider using weighted averages or other techniques to dampen the impact of large trades.
- **Implement Rate Smoothing or Dampening:** Introduce mechanisms to smooth out or dampen fluctuations in the funding rate. Use moving averages, time-weighted averages, or other filtering techniques.
- **Strengthen Oracle Security:** Address the vulnerabilities in the oracle module (*F-2024-7600* and *F-2024-7593*) to prevent indirect funding rate manipulation through price manipulation.
- **Limit Order Execution Control:** Address the vulnerability in `ExecuteOrders` (*F-2025-8500*) to prevent attackers from selectively executing orders to manipulate funding rates.
- **Consider Historical Data:** Modify the `BeginBlocker` to consider the funding rate and market conditions of previous blocks, not just the current block.
- **Monitor Funding Rate Activity:** Implement monitoring tools to detect suspicious patterns in funding rate movements and identify potential manipulation attempts.

Resolution:

The vulnerability in the funding rate calculation has not been fully resolved through code changes. Instead, Elys has implemented:

1. A more secure oracle system (Ojo Network) that provides more reliable price data
2. A design that relies on economic disincentives (slippage costs) to discourage manipulation

While these are positive developments that may reduce the risk of exploitation in practice, the core design issues in the funding rate calculation remain. The recommendations from the audit report - implementing rate smoothing, adding position size limits, using time-weighted averaging, and adding rate change limits between blocks - have not been directly implemented in the code.The security of the system now relies on:

1. The robustness of the Ojo oracle for accurate pricing

2. Economic mechanisms (slippage costs) that make manipulation unprofitable
3. The assumption that the cost of attack will outweigh potential gains

Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc

Here is the response from Elys team on the issue:

> Discussed this with the team. though it is possible to manipulate the funding rate, the user who tries to do that will have to pay huge slippage to open the position as funds are borrowed from the AMM pool, thus discouraging the user from doing so, making funding rate manipulation non-beneficial

---

Evidences

Combined Attacks with ExecuteOrders (F-2025-8500)

Reproduce:

Precise Timing of Attacks:

- An attacker monitors the mempool for large pending orders that could influence the funding rate.
- They use `ExecuteOrders` to execute their own large position immediately before a funding rate update, maximizing their impact on the rate and potentially triggering liquidations.

Creating Artificial Imbalances:

- An attacker uses `ExecuteOrders` to selectively execute existing orders, creating artificial imbalances in open interest to manipulate the funding rate.

Triggering Liquidations:

- An attacker manipulates the funding rate to push traders' positions closer to liquidation.
- They then use `ExecuteOrders` to execute orders that trigger those liquidations, profiting from the liquidation penalties.

Combined Attacks with Oracle Manipulation (F-2024-7600 and F-2024-7593)

Reproduce:

Manipulating Prices to Influence Funding Rates:

- An attacker exploits vulnerabilities in the oracle module to manipulate prices.
- The manipulated prices influence the funding rate calculation, allowing the attacker to indirectly control the funding rate.

Direct Funding Rate Manipulation

Reproduce:

Exploiting Sensitivity to Large Positions:

- An attacker with significant capital opens a large long (or short) position in a perpetual market, dramatically skewing the open interest ratio.
- This causes the funding rate to become artificially high (or low), favoring the attacker's position.
- The attacker collects inflated funding payments from traders on the opposite side.
- The attacker closes their position before the next funding rate update to avoid paying the manipulated rate.

Exploiting Zero Open Interest:

- An attacker monitors the open interest for a perpetual market.
- When the open interest on one side (long or short) approaches zero, the attacker quickly closes their position on that side, temporarily forcing the open interest to zero.
- This triggers the "zero amount exploitation" vulnerability, setting the funding rate to zero.
- The attacker then re-opens their position, benefiting from the zero funding rate while the market rebalances.

Manipulating the Open Interest Ratio:

- An attacker monitors the open interest ratio and identifies a dominant side (long or short).
- They strategically open positions on the dominant side, further skewing the ratio and increasing the funding rate paid by traders on the opposite side.
- The attacker profits from the inflated funding payments.

F-2024-7593 - Price Manipulation and Feeder Centralization Risks - Medium

Description:

The `x/oracle` module suffers from a centralization vulnerability due to its pricing logic, where the final accepted price is determined solely by the most recent submission, disregarding previous ones. This setup allows a malicious actor to manipulate the price by submitting a value at the last moment, which will override all prior submissions. Furthermore, a malicious actor who controls any single whitelisted price feeder key can exploit this mechanism to manipulate prices, potentially draining all funds from the Elys Network. This vulnerability undermines the core principle of tamper-proof price feeds in decentralized finance.

Vulnerabilities:

1. **Last-Minute Price Manipulation:** A malicious actor can wait for others to submit accurate prices and then submit a manipulated price at the last moment, overriding all previous values. This grants them complete control, regardless of how many price feeders exist or how their keys are managed.
2. **Single Point of Failure:** Even with multiple feeders, relying on the last submitted price creates a single point of failure. Compromising one key allows an attacker to manipulate the price, negating the purpose of having multiple feeders.

Impact:

- **Price Manipulation:** Attackers can manipulate prices to trigger unfair liquidations, create arbitrage opportunities, and cause incorrect payouts.
- **Loss of User Funds:** Users may suffer significant financial losses due to manipulated prices affecting various modules like `x/leveragelp`, `x/perpetual`, `x/amm`, and `x/tradeshield`.
- **Total Loss of Funds:** A compromised key could be used to manipulate prices to drain all funds from the Elys Network via IBC transfers.

The specifics of how price feeder keys are handled, stored, or protected are outside the scope of this audit. However, it is crucial to acknowledge that the security of these keys is paramount, as any compromise can have catastrophic consequences for the Elys Network.

Assets:

- oracle [https://github.com/elys-network/elys]

Status:	Fixed

Classification

Impact:	5/5
Likelihood:	2/5
Severity:	Medium

Recommendations

Remediation:

Implement a Decentralized Aggregation Mechanism:

- **Median Price Feed:** Collect price feeds from multiple price feeders and use the median value. This is resistant to outliers and manipulation by a single actor.
- **Weighted Median Price Feed:** Similar to the median, but weights each price feeder's submission based on factors like historical accuracy, stake, or reputation. This further disincentivizes malicious behavior.

Enhance Price Feeder Selection and Validation:

- **Staking and Slashing:** Implement a staking mechanism where price feeders must stake tokens to participate. Slashing conditions should be defined for submitting inaccurate or manipulated prices. This economically incentivizes honest reporting.
- **Reputation System:** Introduce a reputation system that tracks the historical accuracy and reliability of price feeders. This can be used to weight their submissions in the aggregation mechanism or to exclude consistently inaccurate price feeders.

Resolution:

Switching to the Ojo network oracle module based on Vote Extensions addresses the issue. Migration to the mainnet is still in progress.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2024-7689 - Lack of Insurance Fund in Elys Perpetual Trading Module - Low

Description:

The Elys Protocol's `x/perpetual` module, responsible for facilitating perpetual trading, lacks a dedicated insurance fund to safeguard user funds against potential losses. This omission exposes users to significant financial risks, especially when engaging in leveraged trading, and threatens the overall stability of the protocol. While existing risk mitigation measures like liquidations and pool health monitoring are in place, they are insufficient to fully address the inherent risks of a system lacking a dedicated safety net.

The absence of an insurance fund within the `x/perpetual` module creates vulnerabilities, particularly when considering its reliance on the x/amm module for price information, which depends on the oracle module. This creates specific risks in the following scenarios:

1. **Module Vulnerabilities:** Undiscovered bugs or exploits within the `x/perpetual` module itself, or within its dependencies (`x/amm`, `x/leveragelp`), could lead to the depletion of pool funds. In such an event, users would have no recourse for recovering their losses.
2. **Black Swan Events:** Unpredictable and extreme market volatility ("black swan" events) can trigger cascading liquidations within the `x/perpetual` module. These events can rapidly exhaust available liquidity, resulting in substantial losses for leveraged traders. An insurance fund would act as a crucial buffer to absorb these losses and prevent a complete system collapse.
3. **Insufficient Liquidity for Margin Calls:** The `x/perpetual` module, like other margin trading systems, relies on margin calls to manage risk. When losses exceed a user's collateral, a margin call is triggered. However, if there is insufficient liquidity within the system to cover these margin calls (e.g., during a rapid price drop and a high volume of liquidations), the protocol itself can incur significant losses. An insurance fund would serve as a backstop to prevent cascading liquidations and preserve the solvency of the protocol.
4. **AMM Manipulation:** If the `x/amm` module, which provides price information to the `x/perpetual` module, is manipulated or compromised, it could lead to incorrect calculations for liquidations, margin calls, and potentially allow for exploitative trading strategies. This indirect reliance on the `x/amm` module exposes the `x/perpetual` module to risks associated with price manipulation.
5. **Oracle Manipulation:** While the `x/perpetual` module doesn't directly use the oracle module, it indirectly depends on it

through the `x/amm` module. If the oracle is manipulated (as highlighted in issues *F-2024-7593* and *F-2024-7600*), it could cascade into inaccurate price information in the `x/perpetual` module, leading to potential losses.

Existing Community Fund is Insufficient:

- The protocol currently allocates a portion of liquidated position value to a community fund (as described in `x/leveragelp/spec/incentivized_liquidation_system_spec.md`).
- However, this fund's purpose is not explicitly defined as insurance, and there's no documented mechanism for using it to compensate users specifically for losses within the `x/perpetual` module.

Assets:

- perpetual [https://github.com/elys-network/elys]

Status: Accepted

Classification

Impact: 1/5

Likelihood: 1/5

Severity: Low

Recommendations

Remediation: A dedicated insurance fund is crucial for the `x/perpetual` module. This fund should be designed to cover losses arising from:

- Exploits within the `x/perpetual` module or its dependencies.
- Unforeseen market events leading to cascading liquidations.
- Insufficient liquidity to cover margin calls.
- AMM manipulation that affects the `x/perpetual` module's functionality and leads to user losses.
- Oracle manipulation that indirectly affects the `x/perpetual` module through inaccurate price feeds.

Funding mechanisms for the insurance fund can include:

- Allocation of a portion of trading fees.
- Accumulation of a percentage of liquidated position value.
- Direct contributions from the protocol's treasury.

Resolution: **What's Still Missing:**

1. **Mechanism to Use the Fund**: While the system collects funds into the insurance fund address, there's no implemented mechanism to:
    - Use those funds to cover losses during exceptional circumstances.
    - Distribute those funds to users who experienced losses due to system failures.
    - Cover the specific risks mentioned in the audit (black swan events, insufficient liquidity, AMM manipulation)
2. **Defined Governance Process**: There's no clear governance or automation for determining:
    - When the insurance funds should be used.
    - How much should be used in different scenarios.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

Evidences

Industry Precedents

Reproduce:

Several established perpetual trading protocols utilize insurance funds to enhance user protection and protocol stability:

- **dYdX:** Uses an insurance fund to cover losses from protocol errors or other vulnerabilities, funded by a portion of trading fees.
- **Perpetual Protocol:** Employs an insurance fund primarily funded through liquidations to cover losses exceeding liquidated traders' collateral, maintaining solvency during volatile periods.
- **GMX:** Has an insurance fund funded by platform fees to cover trader losses that cannot be covered by liquidations, specifically during volatile market movements.

F-2024-7736 - Counterproductive Lockup in Oracle Pools Kills Market Efficiency - Low

Description:

The `x/amm` module's oracle pool 1-hour lockup for new liquidity providers (LPs) creates a predictable window for external liquidity manipulation, placing leveraged LP positions at increased risk of forced liquidation. While the lockup also severely disadvantages market makers, hindering their ability to manage risk and maintain market efficiency, it is the leveraged LPs who face the most immediate and potentially catastrophic consequences from this vulnerability. The lockup exacerbates existing oracle security concerns, making the platform more vulnerable to attacks, driving away market makers due to an unfavorable and risky environment, and exposing leveraged LPs to unacceptable liquidation risks.

Counterproductive Lockup - Market Maker Disincentive and Leveraged LP Endangerment:

- The lockup prevents LPs, including market makers, from quickly responding to price deviations and weight imbalances. For market makers, this increases their risk exposure and reduces their profitability. For leveraged LPs, it creates a window of extreme vulnerability where their positions can be liquidated without recourse.
- The inability to manage risk effectively will deter market makers from participating in `x/amm` oracle pools. The heightened risk to leveraged LPs will also discourage their use, reducing overall pool liquidity and usage.

Exploitation of Locked Positions - Extreme Risk for Leveraged LP Positions:

- The fixed 1-hour lockup allows attackers to manipulate external liquidity after a user joins a pool, predictably affecting `ExternalLiquidityRatio`.
- Leveraged LP positions are disproportionately impacted due to their sensitivity to price fluctuations and reliance on accurate `ExternalLiquidityRatio` for liquidation thresholds. Even small manipulations during the lockup can trigger liquidations, leading to significant financial losses.
- While this also affects market makers, the risk to leveraged LPs is far greater and more immediate due to the leveraged nature of their positions. They are more likely to be liquidated and suffer complete loss of funds.
- This is particularly concerning given the existing vulnerabilities related to oracle security (*F-2024-7593*, *F-2024-7600*) and external liquidity data (*F-2024-8019*). The lockup provides a

window where the impact of these vulnerabilities can be amplified, creating an environment where leveraged LPs are unfairly exposed to extreme risk.

Potential consequences include:

- Increased Risk to Leveraged LP Positions: The lockup creates a high probability of forced liquidations for leveraged LPs due to external liquidity manipulation, potentially leading to substantial financial losses and discouraging the use of leverage within the `x/amm` module.
- Reduced Market Maker Participation: The lockup's detrimental effect on market maker profitability and risk management will lead to a significant decline in their participation, resulting in lower liquidity, wider spreads, and increased slippage for all users.
- Inefficient Arbitrage: Preventing timely correction of price discrepancies between the AMM and external markets.
- Weakened Weight-Breaking Mechanism: Reduced effectiveness in maintaining target weights and price stability, as market makers are a key component of this mechanism.
- Suboptimal Pool Performance: Reduced efficiency and attractiveness of oracle pools due to lack of market maker involvement and increased risk to leveraged LPs.
- Unfair Treatment of Leveraged LPs: Increased risk of forced liquidations due to the predictable manipulation window.
- Increased vulnerability to external liquidity manipulation and oracle attacks. The lockup, in conjunction with issues like those identified in *F-2024-7593*, *F-2024-7600*, and *F-2024-8019*, creates a significantly higher risk environment, particularly for market makers and leveraged LP positions.
- Potential financial losses for LPs, especially for leveraged LPs and market makers, due to manipulation and inability to manage positions effectively.

Assets:

- leveragelp [https://github.com/elys-network/elys]
- amm [https://github.com/elys-network/elys]

Status: Mitigated

Classification

Impact: 1/5

Likelihood: 1/5

Severity: Low

Recommendations

Remediation:

1. **Parameterize Lockup Duration:** Make the lockup duration a configurable parameter (globally or per-pool) instead of hardcoding it. This allows for flexibility to cater to different asset types, market conditions, and the specific needs of leveraged positions.
2. **Explore Dynamic Lockup Mechanisms:** Implement a dynamic lockup mechanism that adjusts the duration based on factors like volatility, pool liquidity, `ExternalLiquidityRatio`, or weight deviation. This would allow market makers and leveraged LPs to operate more freely during normal market conditions while still providing protection during volatile periods.
3. **Introduce Tiered Lockups:** Offer different lockup durations with varying incentives (e.g., higher rewards for longer lockups). This allows LPs, including those with leveraged positions, to choose a lockup period that aligns with their risk tolerance.
4. **Enable Early Exit with Penalty:** Allow LPs, including those with leveraged positions, to exit their positions before the lockup expires, but impose a calibrated penalty. This provides an escape hatch for leveraged LPs facing unexpected market movements, mitigating the risk of complete liquidation.
5. **Allow Intra-Lockup Rebalancing:** Permit LPs to adjust the asset allocation of their locked LP position without withdrawing funds. This enables market makers to actively manage their inventory and all LPs, especially those with leveraged positions, to respond to market changes, improving their risk management and pool stability. Implement by allowing specific message types to be executed by LPs with locked funds, if those messages only affect the pool the user is locked in.
6. **Improve Oracle Security and Data Validation:** Address underlying oracle security concerns and implement robust validation of external liquidity data. This reduces the overall risk environment, making the platform more attractive to market makers and safer for leveraged LPs.
7. **Economic Analysis and Simulation:** Conduct thorough economic modeling and simulations, with a focus on market maker behavior and the impact on leveraged LP positions, to evaluate the impact of different lockup durations, mechanisms (including intra-lockup rebalancing), and the weight-breaking mechanism on pool performance, market maker profitability, leveraged LP risk, and overall protocol security.

Resolution: Moved the hardcoded 1-hour lockup to params, which allows for addressing the most immediate concerns while the community decides on the optimal duration.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2024-8019 - External Liquidity Manipulation Vulnerability in AMM Oracle Pools with Leveraged LP Positions - Low

Description: The `x/amm` module's oracle pool implementation relies on externally sourced liquidity data, specifically the `ExternalLiquidityRatio`, to adjust pool asset valuations and influence swap calculations. This reliance creates a vulnerability where attackers can manipulate the reported external liquidity, primarily through wash trading and order book manipulation on centralized exchanges (CEXs), even if the on-chain price oracle itself is decentralized and reliable. This manipulation can lead to incorrect pricing, unfair swap execution, and exploitation of the weight-breaking fee/reward mechanism within oracle pools. The issue stems from the lack of robust validation and independent verification of the externally provided liquidity data, specifically the `AssetAmountDepth` information.

Anticipated Functionality:

- **External Liquidity Data:** The system gets information about how much of an asset is available to trade on *outside* exchanges (like Binance, Coinbase, etc.). This data is called `AssetAmountDepth` and includes two key pieces:
  - `O_Tvl`: The total amount of the asset available on these external exchanges.
  - `depth`: A measure of how much the price would change on the external exchange if a large order were placed. This is fed into a formula to adjust the external liquidity based on how stable the prices are.
- **Liquidity Ratio:** The system calculates a `liquidityRatio` using a formula based on `depth`. Think of this as an adjustment factor:
  - Formula: *liquidityRatio = 1 - sqrt(1 - depth)*
- **External Liquidity Ratio:** The system then calculates the `ExternalLiquidityRatio`
  ExternalLiquidityRatio = (O_Tvl / P_Tvl) / liquidityRatio, where:
  - `O_Tvl`: Total external liquidity (from the data feed).
  - `P_Tvl`: The total amount of that asset held *inside* the AMM pool.
  - `liquidityRatio`: The adjustment factor calculated above.
  - **Intuition:** This ratio compares the external liquidity to the internal liquidity, adjusted by the `liquidityRatio`, which reflects how stable the external price is assumed to be. A higher ratio means more external liquidity is considered to be available relative to the internal pool's size.

Impact on Stakeholders:

- **Liquidity Providers (LPs):**
  - **Financial Loss:** LPs can lose a portion of their funds if attackers drain pools using manipulated exchange rates.
  - **Distorted Rewards:** The weight-breaking reward mechanism can be manipulated, leading to unfair distribution of rewards.
- **Traders:**
  - **Unfavorable Swap Rates:** Traders might get significantly worse exchange rates than they should due to the manipulated `ExternalLiquidityRatio`.
  - **Increased Slippage:** If liquidity is artificially deflated, traders might experience higher slippage than expected.

Cost and Complexity of Attack:

The cost and complexity of executing these attacks vary depending on the specific scenario but generally involve the following factors:

- **CEX Manipulation Costs:**
  - **Wash Trading:** The primary cost is the trading fees paid on the CEX. This can range from negligible (on CEXs with very low fees or maker/taker models that incentivize liquidity provision) to substantial, depending on the required volume and the CEX's fee structure. The attacker also needs capital to perform the wash trades, although this capital is not lost (except for fees).
  - **Order Book Manipulation:** This might require significant capital to place large, genuine orders to influence the order book depth, especially if trying to manipulate the `depth` parameter significantly. The attacker may be able to recover most of this capital after the attack, but there's a risk of losses if other market participants interact with these orders.
- **On-Chain Transaction Costs:** Attackers need to pay gas fees for submitting transactions to the blockchain, including:
  - Swap transactions to exploit the manipulated liquidity data.

Even though this issue assumes trusted price feeders for this specific analysis, it is absolutely crucial to address the underlying Oracle vulnerabilities (*F-2024-7593* and *F-2024-7600*). Implementing a decentralized price aggregation mechanism and on-chain TWAP enforcement will significantly enhance the overall security of the AMM module, including oracle pools that rely on external liquidity data.

Assets:

- amm [https://github.com/elys-network/elys]
- oracle [https://github.com/elys-network/elys]

Status: Accepted

Classification

Impact:	2/5
Likelihood:	1/5
Severity:	Low

Recommendations

Remediation:

I. Data Source & Aggregation - Multiple Independent Data Sources:

- Aggregate external liquidity data from multiple independent CEXs and decentralized sources.
- Employ robust aggregation methods (median, weighted average with outlier filtering) to minimize the impact of manipulation on a single source.

II. Data Validation & Integrity - Enhanced AssetAmountDepth Validation:

- Implement rigorous validation of `AssetAmountDepth` data within `FeedMultipleExternalLiquidity` and `GetExternalLiquidityRatio`.
- **Range Checks:** Ensure values are within predefined, adjustable ranges. These ranges can be adjusted at pool creation or regular updates.
- **Outlier Detection:** Detect and handle (reject or adjust) outlier values.

III. Mitigation of Manipulation Impact - Delayed Effect of ExternalLiquidityRatio Changes:

- Introduce a delay or use a time-weighted average before `ExternalLiquidityRatio` changes fully affect pool calculations, especially the weight-breaking mechanism.
- Provide a buffer to react to potential manipulations.

IV. System Safeguards - Circuit Breakers:

- Halt trading or limit the impact of external liquidity updates upon detecting extreme or suspicious values (e.g., sudden, significant `ExternalLiquidityRatio` changes).

V. Reduced CEX Dependency - Reduce Dependence on CEX Data:

- Explore on-chain data sources (e.g., trading volumes on other DEXs).
- Develop incentives for liquidity provision on decentralized exchanges.

VI. Risk Assessment - Stress Tests:

- Conduct stress tests for leveraged LP positions under manipulated external liquidity scenarios.

Resolution:

The Elys Team is working on extending the Ojo network oracle that is integrated for prices and using Vote Extensions for External Liquidity as well. This will ensure that data is from external sources but would be validated by multiple parties instead of centralized oracles.

To tackle wash trading and fake order books for tokens with very low liquidity and shallow order books. Those types of tokens will not be enabled for smart shielded pools due to the manipulation risk involved from users attempting to do so on those exchanges. Since smart shielded pools are permissioned with governance, the Elys team will evaluate only very deep liquidity tokens for consideration.

Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc

---

Evidences

The Manipulation

Reproduce:

Attackers can manipulate the `depth` and `O_Tvl` values reported by the external data feed. This can be done through techniques like *wash trading* (fake trades to inflate volume) and *spoofing* (placing fake orders to make the order book look different).

Example:

Let's say a pool has 100,000 USDC (`P_Tvl`) of internal liquidity.

Honest Scenario:

- `O_Tvl` (True External Liquidity) = 1,000,000 USDC
- `depth` (Honestly Reported) = 0.5 (meaning a large order might move the price by 50%)
- `liquidityRatio` = 1 - sqrt(1 - 0.5) ≈ 0.2929
- `ExternalLiquidityRatio` = (1,000,000 / 100,000) / 0.2929 ≈ 34.14
- **Impact:** The AMM considers the external liquidity, leading to more accurate slippage calculations for swaps.

Manipulated Scenario (Inflated Liquidity):

- `O_Tvl` = 10,000,000 (Inflated through wash trading)
- `depth` = 0.9 (Manipulated through spoofing or other means)
- `liquidityRatio` = 1 - sqrt(1 - 0.9) ≈ 0.6838
- `ExternalLiquidityRatio` = (10,000,000 / 100,000) / 0.6838 ≈ 14.62

- **Impact:** The AMM now believes there's much more external liquidity than there actually is. The attacker can swap assets in the pool and will experience **much less slippage** than they should, getting a better price and potentially draining the pool.

Manipulated Scenario (Deflated Liquidity):

- `O_Tvl` = 10,000 (Deflated, attacker might even hide their own liquidity)
- `depth` = 0.1 (Manipulated)
- `liquidityRatio` = 1 - sqrt(1 - 0.1) ≈ 0.0513
- `ExternalLiquidityRatio` = (10,000 / 100,000) / 0.0513 ≈ 1.95
- **Impact:** The AMM believes there's much less external liquidity. The attacker can manipulate the pool's weights through carefully timed swaps, potentially triggering the **weight-breaking fee mechanism** on other users, and profiting from the fees.

F-2025-8082 - Ignored Non-Existent Pools in ‘ConvertGasFeesToUsdc’ Function - Low

Description:

The `ConvertGasFeesToUsdc` function attempts to find a suitable pool for each token and convert it to the base currency. However, if a compatible pool is not found or if the swap operation fails, it simply continues to the next token, resulting in unconverted coins without providing any explicit error messages or logs.

Below is a list of specific instances where errors are silently ignored.

1. Disregarding the absence of a pool could result in unswapped tokens remaining in the fee collection address.

```
// Find a pool that can convert tokenIn to usdc
pool, found := k.amm.GetBestPoolWithDenoms(ctx, []string{tokenIn.De
nom, baseCurrency}, false)
if !found {
continue
}
```

2. Neglecting errors during the swapping process could result in undetected failures, potentially leading to inaccurate accounting or financial losses.

```
// Executes the swap in the pool and stores the output. Updates poo
l assets but
// does not actually transfer any tokens to or from the pool.
snapshot := k.amm.GetAccountedPoolSnapshotOrSet(ctx, pool)
tokenOutCoin, _, _, _, err := k.amm.SwapOutAmtGivenIn(ctx, pool.Poo
lId, k.oracleKeeper, &snapshot, sdk.Coins{tokenIn}, baseCurrency, m
ath.LegacyZeroDec(), math.LegacyOneDec())
if err != nil {
continue
}
```

3. Failing to handle errors during balance settlement may lead to inconsistencies in accounting, incomplete updates, and potential integrity issues within the system.

```
// Settles balances between the tx sender and the pool to match the
swap that was executed earlier.
// Also emits a swap event and updates related liquidity metrics.
cacheCtx, write := ctx.CacheContext()
_, err = k.amm.UpdatePoolForSwap(cacheCtx, pool, address, address,
tokenIn, tokenOutCoin, math.LegacyZeroDec(), math.LegacyZeroDec(),
math.LegacyZeroDec())
if err != nil {
continue
}
```

Assets:

- masterchef [https://github.com/elys-network/elys]

Status:	Fixed

Classification

Impact:	1/5
Likelihood:	1/5
Severity:	Low

Recommendations

Remediation: It is recommended to log warnings or track the number of skipped tokens, allowing for the identification of instances of conversion failures. If your application relies on all tokens being converted, consider generating an error message or emitting an event to indicate any missing pools or failed swaps. This approach enhances transparency and helps prevent unnoticed silent failures.

Resolution: The Elys team resolved the vulnerability by implementing comprehensive error handling and transparent logging mechanisms in the `ConvertGasFeesToUsdc` function.

Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2025-8268 - Insufficient Protections in Elys Protocol AMM Logic - Low

Description:

The Elys Protocol's Automated Market Maker (AMM) implementation, specifically within the module, highlights several areas where enhanced validation and protection mechanisms could be beneficial. It's important to emphasize that these are potential vulnerabilities that have not been definitively proven to pose actual security risks in the current codebase. This is because there might be indirect checks or safeguards in place that mitigate these issues, although they haven't been explicitly validated.

Nevertheless, to ensure robust security and prevent potential exploits, it's recommended to implement direct and explicit checks in the following areas: route validation, slippage control, input sanitization, and oracle price utilization. Additionally, the lack of a Time-Weighted Average Price (TWAP) mechanism, as reported in issue F-2024-7600, is another area where adding a direct mechanism could enhance security.

1. **Incomplete Route Validation:** The `ValidateRoutes` function fails to perform essential checks for duplicate pools, token compatibility, and circular routes. This oversight allows malicious actors to craft invalid or exploitative swap paths that can drain funds and destabilize pool compositions.
2. **Inadequate Slippage Protection:** The `SwapExactAmountIn` and `SwapExactAmountOut` functions do not adequately protect users from excessive slippage, especially in multi-hop swaps. The current implementation does not consider the cumulative slippage across all hops, potentially leading to unexpected losses for users.
3. **Insufficient Input Validation:** The `SwapExactAmountIn` and `SwapExactAmountOut` functions lack robust input validation. There are insufficient correlation checks between the provided input amounts and the expected outputs based on oracle prices and slippage tolerances. This can be exploited for pool draining attacks or result in failed transactions due to unexpected slippage.
4. **Vulnerable Oracle Price Utilization:** While the implementation uses an oracle for price verification, it lacks critical safeguards. There are no checks for price staleness, no bounds on price values, and no TWAP mechanism to mitigate price manipulation. This exposes the AMM to potential oracle price manipulation and arbitrage attacks.

Impact:

- **Fund loss:** Attackers can exploit vulnerabilities to drain funds from pools or execute profitable arbitrage attacks.
- **Degradation of pool health:** Invalid routes and price manipulation can lead to pool instability and price distortions.
- **Unexpected financial losses:** Users may experience significant losses due to inadequate slippage protection.
- **Pool draining attacks:** Insufficient input validation enables attackers to drain pools.
- **Failed transactions:** Users may encounter failed transactions due to unexpected slippage or input validation errors.
- **Oracle price manipulation:** Attackers can manipulate oracle prices to execute profitable trades at favorable prices.

The absence of a TWAP mechanism, as highlighted in issue *F-2024-7600*, significantly amplifies the severity of these vulnerabilities. Without TWAP, the AMM is highly susceptible to price manipulation attacks, especially in scenarios with volatile or illiquid assets. Attackers can exploit this to their advantage, causing substantial losses for regular users.

Assets:

- amm [https://github.com/elys-network/elys]

Status: Accepted

Classification

Impact: 1/5

Likelihood: 1/5

Severity: Low

Recommendations

Remediation:

1. **Enhance route validation:** Implement robust checks for token pair compatibility, pool uniqueness within a route, and detection of circular paths in the `ValidateRoutes` function.
2. **Implement comprehensive slippage protection:** Calculate and enforce the cumulative slippage across all hops in multi-hop swaps within the `SwapExactAmountIn` and `SwapExactAmountOut` functions.
3. **Improve input validation:** Strengthen input validation in `SwapExactAmountIn` and `SwapExactAmountOut` by correlating input amounts with expected outputs based on oracle prices and slippage tolerances.
4. **Secure oracle price utilization:** Implement checks for price staleness, bounds on price values, and integrate a TWAP

mechanism in the `GetPrice` function to mitigate oracle price manipulation.

Resolution: Issues `#1` (route validation) and `#3` (input validation) are planned for future implementation. Issue `#2` (slippage protection) is partially addressed via `TokenOutMinAmount`, ensuring minimum output amounts, with additional slippage tolerance mechanisms planned. Issue `#4` (oracle price utilization) is covered by the Ojo module integration, though currently disabled pending future activation.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2025-8341 - Division by Zero Leading to Potential Panics - Low

Description:

The code in several locations presents a risk of division by zero. Examples include functions such as `CalcMTPTakeProfitLiability`, `SwapEstimationByDenom`, and others listed below, where price-related variables in denominators (e.g., `tradingAssetPrice` or `spotPrice`) could potentially have a value of zero.

CalcMTPTakeProfitLiability :
The `CalcMTPTakeProfitLiability` function is vulnerable to a division-by-zero error if `tradingAssetPrice` is zero. This scenario may arise when the oracle provides a valid price of zero for the trading asset.

```
tradingAssetPrice, err := k.GetAssetPrice(ctx, mtp.TradingAsset) //
Could return 0 if oracle data is flawed
// ...
takeProfitLiabilities = mtp.TakeProfitCustody.ToLegacyDec().Quo(tra
dingAssetPrice) // Division by zero if tradingAssetPrice == 0
```

SwapEstimationByDenom:
The `SwapEstimationByDenom` function is susceptible to a division-by-zero error if `spotPrice` equals zero. This situation arises when `CalculateTokenARate` returns zero, which can occur if either `tokenBalanceA` or `tokenWeightB` in the pool is zero. The absence of proper error handling allows this condition to propagate through the logic chain, ultimately causing a panic during the calculation of `priceImpact`.

```
// Risk in priceImpact calculation:
priceImpact = spotPrice.Sub(impactedPrice).Quo(spotPrice) // Divisi
on if spotPrice = 0

//func (k Keeper) CalcOutRouteSpotPrice
rate, err := pool.GetTokenARate(ctx, k.oracleKeeper, &snapshot,...
)

// Underlying cause in CalculateTokenARate:
if tokenBalanceA.IsZero() || tokenWeightB.IsZero() {
return sdkmath.LegacyZeroDec() // Returns zero without error
}
```

Open:

```
tradingAssetPrice, err := k.GetAssetPrice(ctx, msg.TradingAsset)
if err != nil {
return nil, err
}
ratio := msg.TakeProfitPrice.Quo(tradingAssetPrice)
```

SettleMTPBorrowInterestUnpaidLiability:

```
tradingAssetPrice, err := k.GetAssetPrice(ctx, mtp.TradingAsset)

// ...
} else {
// custody is in usdc, liabilities needs to be in trading asset,
borrowInterestPaymentInCustody = unpaidInterestCustody.ToLegacyDec(
).Quo(tradingAssetPrice).TruncateInt()
}
```

Although rare, this issue must be addressed to prevent system crashes or inaccurate calculations. Division by zero can compromise system stability and produce invalid outcomes, undermining reliability. Implementing safeguards, such as zero checks or robust error handling, will enhance resilience in edge cases, such as faulty price feeds or empty pools. Proactively resolving this issue will ensure long-term stability, accuracy, and trust in the system's operations.

Status: Fixed

Classification

Impact: 2/5

Likelihood: 1/5

Severity: Low

Recommendations

Remediation:

- **Implement Zero Checks:** Incorporate validations to ensure denominators are non-zero before executing division operations. This will preemptively prevent division by zero scenarios.
- **Improve Error Handling:** Update functions to identify and handle conditions that may result in zero values explicitly. Rather than returning zero, these functions should return detailed error messages to prevent invalid data propagation to downstream logic.

Resolution: The Elys codebase has been updated to address division by zero vulnerabilities through the implementation of validation checks before division operations. In functions including `CalcMTPTakeProfitLiability`, `SwapEstimationByDenom`, `Open`, and `SettleMTPBorrowInterestUnpaidLiability`, zero-value validations were added for price-related denominators such as `tradingAssetPrice` and `spotPrice`.

They have been fixed in the main branch, Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc

https://github.com/elys-network/elys/blob/main/x/perpetual/keeper/calc_mtp_take_profit_liabilities.go#L28
https://github.com/elys-network/elys/blob/main/x/amm/keeper/calc_swap_estimation_by_denom.go#L75
https://github.com/elys-network/elys/blob/main/x/perpetual/keeper/open.go#L42
https://github.com/elys-network/elys/blob/main/x/perpetual/keeper/mtp_borrow_interest.go#L63

F-2025-8434 - Insufficient Market-Specific Leverage Limits and Missing Authority Controls - Low

Description:

The Elys Protocol lacks proper market-specific leverage limits and authority controls for setting maximum leverage values. This creates significant systemic risks as different markets have varying liquidity depths and volatility profiles, yet are not properly constrained by appropriate leverage limits. Currently, the protocol applies a blanket leverage limit across all markets, exposing it to potential manipulation and cascading liquidations, especially in markets with low liquidity or high volatility. While some indirect limits may exist, such as margin requirements or position size restrictions, these are not sufficient to address the complex interplay of leverage and market conditions.

Missing Market-Specific Leverage Controls:

```
// Current implementation in the perpetual module only has basic va
lidation
if !(msg.Leverage.GT(sdkmath.LegacyOneDec()) || msg.Leverage.IsZero
()) {
return errorsmod.Wrapf(ErrInvalidLeverage, "leverage (%s) can only
be 0 (to add collateral) or > 1 to open positions", msg.Leverage.St
ring())
}
```

The current implementation in the perpetual module only has basic leverage validation, without considering market-specific conditions. This means that a user can open a position with the same high leverage in both a liquid market (e.g., ETH/USD) and an illiquid market (e.g., a low-cap altcoin).

Inconsistent Leverage Limits:

```
// LeverageLP module has a fixed limit
if p.LeverageMax.GT(sdkmath.LegacyNewDec(10)) {
return fmt.Errorf("leverage max too large: %s", p.LeverageMax.Strin
g())
}
```

The `LeverageLP` module has a fixed leverage limit (10x), which may not be appropriate for all markets. This inconsistency highlights the need for a more nuanced approach to leverage limits.

No Authority Control Structure:
Currently, there is no defined authority or governance mechanism to set or adjust market-specific leverage limits. This creates a risk of inflexible and potentially inappropriate leverage limits across different markets, hindering the protocol's ability to adapt to changing market conditions.

Explicit checks for market-specific leverage limits are crucial for several reasons:

1. **Mitigating Risk in Illiquid Markets:** In illiquid markets, even relatively small trades can cause significant price swings. Allowing high leverage in such markets amplifies the risk of cascading liquidations, where the liquidation of one position triggers a chain reaction of further liquidations, potentially leading to a market crash.
2. **Preventing Manipulation:** Markets with low liquidity are more susceptible to manipulation by large traders. Explicit leverage limits tailored to market conditions can help prevent whales from exerting undue influence on prices.
3. **Protecting User Funds:** By setting appropriate leverage limits, the protocol can protect users from taking on excessive risk, particularly in volatile or illiquid markets. This helps prevent large-scale losses and promotes responsible trading practices.
4. **Ensuring System Stability:** Market-specific leverage limits contribute to overall system stability by preventing excessive leverage from destabilizing individual markets and potentially triggering contagion across the platform.
5. **Enabling Flexibility:** An authority control structure allows the protocol to adjust leverage limits as market conditions change, ensuring that the limits remain appropriate and effective.

While indirect limits like margin requirements or position size restrictions can offer some level of protection, they are not a substitute for explicit market-specific leverage limits. Here's why:

1. **Market Dynamics:** Indirect limits may not fully capture the complex interplay of leverage, liquidity, volatility, and market depth. They might be too lenient for some markets and too restrictive for others.
2. **Static Nature:** Indirect limits are often static and cannot adapt to changing market conditions. This can lead to situations where the limits are either insufficient or overly restrictive, hindering market efficiency.
3. **Circumvention:** Sophisticated traders may find ways to circumvent indirect limits, especially if they are not designed with market-specific considerations in mind.

Impact:

1. **Market Stability Risks:**
   1. High leverage in illiquid markets can lead to extreme price movements and cascading liquidations.
   2. Markets with insufficient depth are vulnerable to manipulation by large, leveraged traders.
   3. Arbitrageurs can exploit markets with inappropriately high leverage limits.

    4. Increased risk of market manipulation through over-leveraged positions.
2. **System-Wide Vulnerabilities:**
    1. The lack of correlation between market characteristics and leverage limits can lead to system-wide instability through cross-market contagion.
    2. Increased risk of bad debt accumulation due to inappropriate leverage levels.

Assets:

- perpetual [https://github.com/elys-network/elys]
- leveragelp [https://github.com/elys-network/elys]
- tradeshield [https://github.com/elys-network/elys]

Status: Fixed

Classification

Impact: 1/5

Likelihood: 1/5

Severity: Low

Recommendations

Remediation:

1. Implement market-specific leverage limits with parameters for maximum leverage, liquidity thresholds, and volatility.
2. Allow authorized entities to update these limits with validation and market condition checks.
3. Validate leverage against market-specific limits when opening positions.
4. Dynamically adjust leverage limits based on real-time market conditions.
5. Enforce access control, multi-sig requirements, and time-delays for limit updates.
6. Monitor market conditions, implement alerts, and utilize circuit breakers for extreme events.
7. Implement cooldown periods and gradual adjustments for limit updates.
8. Ensure emergency override capabilities.

Resolution: The Elys team addressed the identified leverage limit issue by implementing pool-specific maximum leverage settings through a governance-controlled mechanism. This modification replaced the previous system of a single global leverage limit with individualized limits that can be set for each trading pool. The implementation utilizes the `UpdateMaxLeverageForPool` function which ensures pool-specific leverage limits remain within the bounds of a global maximum parameter. This approach enables differentiated risk management across markets with varying characteristics, though it relies on manual governance adjustments rather than automated responses to market conditions. The system provides a basic framework for more tailored leverage controls while maintaining the protocol's existing authorization structure.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2025-8435 - Insufficient Slippage Protection in IBC-Triggered Swaps - Low

Description:

The `SwapExactAmountIn` function within the Transfer Hook Swap Handler utilizes a hardcoded `TokenOutMinAmount` of 1 when executing automated swaps for IBC-transferred tokens. This effectively disables slippage protection, leaving users vulnerable to exploitation and potential financial losses.

The vulnerability arises from the lack of slippage protection, which exposes users to multiple attack vectors, especially in scenarios involving:

- **Low-Liquidity Pools:** Small pools are susceptible to high slippage, especially with incoming IBC transfers of significant size.
- **Weight Imbalance Attacks:** Malicious actors could manipulate pool weights before IBC transfers to create unfavorable exchange rates.
- **Oracle Manipulation:** Discrepancies between pool prices and external market prices can be exploited.

Exploit Scenario:

1. **Preparation:** The attacker targets a pool with low liquidity for the token involved in an upcoming IBC transfer.
2. **Front-Running and Manipulation:** The attacker observes incoming IBC transfers and, before the user's transfer occurs:
   - Drains liquidity from the target pool, increasing slippage.
   - Uses a flash loan to manipulate the pool's weights and artificially skew the exchange rate.
3. **Exploitation:** The user's IBC transfer executes, triggering the `SwapExactAmountIn` function. Due to the manipulated pool state and the hardcoded `TokenOutMinAmount`, the user receives a minimal amount of the output token.
4. **Profit:** The attacker reverses their manipulations, restoring the pool's state while profiting from the discrepancy between the manipulated and final prices.

As a result, IBC transfers become highly susceptible to sandwich attacks. In this scenario, an attacker can front-run the transfer, manipulating the pool price by leveraging a flash loan to temporarily drain liquidity and inflate the price. The transfer proceeds despite significant slippage, causing the user to receive far less of the desired output token than anticipated. The attacker can then back-run the transaction, purchasing the asset at a lower price and profiting from the price discrepancy.

Assets:

- transferhook [https://github.com/elys-network/elys]

Status: Accepted

Classification

Impact: 1/5

Likelihood: 1/5

Severity: Low

Recommendations

Remediation:

- **Dynamic Slippage Tolerance:** Replace the hardcoded `TokenOutMinAmount` with a dynamic calculation based on the expected output amount and a user-defined or protocol-defined slippage tolerance (e.g., 1%, 3%). This would allow the transaction to revert if the price impact exceeds the acceptable threshold.
- **Oracle Price Validation:** Implement a mechanism to query and validate external oracle prices before executing the swap. This can help mitigate discrepancies between on-chain pool prices and off-chain market prices, reducing the risk of manipulation.
- **Liquidity Checks:** Implement checks to ensure sufficient liquidity exists in the target pool before executing the swap. This can help prevent scenarios where attackers intentionally drain liquidity to increase slippage.

Resolution: The Elys team plans to integrate Skip Go's end-to-end interoperability platform to create seamless cross-chain experiences, potentially removing the current implementation.

Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc

Evidences

x/transferhook/keeper/swap.go

Reproduce:

```
// x/transferhook/keeper/swap.go
func (k Keeper) SwapExactAmountIn(ctx sdk.Context, addr sdk.AccAddr
ess, tokenIn sdk.Coin, routesammtypes.SwapAmountInRoute) error {
msg:= &ammtypes.MsgSwapExactAmountIn{
TokenOutMinAmount: math.OneInt(), // Hardcoded minimum (L59)
}
//... executes IBC-triggered swap
}
```

F-2024-7484 - Code Quality Improvement - Info

Description:

As part of our ongoing audit of the Golang codebase, several areas have been identified where improvements can enhance code style, readability, and maintainability. While the code is functional, these recommendations aim to streamline development practices, making the codebase easier to manage and extend over time.
Here are the key observations, categorized by specific areas of concern:

Reserved word used as name:

We identified instances where variable names conflict with Go's built-in functions. Specifically, the codebase uses 'clear' and 'cap' as variable names, which are reserved built-in functions in Go. This practice can lead to several issues:

- Creates ambiguity between custom variables and built-in functions
- May confuse developers about the actual scope and purpose of these identifiers

Examples:

1. clear
2. cap

Missing 'case' statements for 'iota' consts in 'switch':

`switch` statements are missing `case` clauses for constants defined using the `iota` keyword. The `iota` identifier in Go is used within a `const` block to create a sequence of incrementing integer constants. While the Go compiler does not enforce handling all possible `iota` values in a `switch` statement, neglecting to do so can lead to unhandled cases, which might indicate a bug or result in unintended behavior.

When a `switch` statement evaluates a variable that can hold values from an iota-generated set of constants, it's important to include `case` clauses for each possible value or provide a `default` clause. Missing `case` statements means that certain constant values are not explicitly handled, increasing the risk of logical errors and making the code harder to read and maintain.

1. switch v.Kind()
2. switch f.Kind()

Deprecated Element Inspection:

You're using code elements (like functions, methods, types, or packages) that have been marked as deprecated in Go. When something is deprecated, it means it's still functional but no longer recommended for use - usually because there's a better alternative available or it might be removed in future versions.

Imported package name as a name identifier:

This issue occurs when a variable, argument, or function is declared with the same name as an imported package. While this is technically allowed in Go, it can lead to problems such as making the package's exported identifiers inaccessible after the conflicting declaration. Additionally, it can create confusion for developers reading the code, as it becomes unclear whether the name refers to the package or the local declaration. Such overlaps reduce code clarity and can introduce potential bugs or maintenance challenges.

Other areas to Improve:

- Redundant Parentheses
- Unnecessary Type Conversions
- Redundant Aliases
- Unhandled Errors

For detailed examples and further information, please refer to the **Evidence** section.

Assets:

- Code Quality [https://github.com/elys-network/elys]

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: It is recommended to avoid naming variables, arguments, or functions with the same name as imported packages. Instead, use descriptive and unique identifiers that clearly convey the purpose of the variable or function. This will ensure that the imported package's identifiers remain accessible and the code remains easy to read and maintain.

Avoid using reserved words or names that conflict with Go's built-in functions when naming variables, arguments, or functions. Instead, use descriptive and unique names that clearly indicate the purpose of the variable or function. For example, instead of naming a variable `cap`, consider a name like `capacity`. This practice improves code clarity, reduces ambiguity, and ensures that built-in functions remain accessible when needed.

Review all `switch` statements that operate on constants defined with `iota` to ensure that every possible value is appropriately handled. This can be achieved by:

- Adding `case` Clauses: Include `case` statements for each constant generated by `iota` within the relevant `const` blocks. This ensures that all defined values are explicitly addressed in the logic.
- Including a `default` Clause: Add a `default` clause to handle any unforeseen or undefined values. This provides a safety net by catching any values that do not match the specified `case` statements.

Address the use of deprecated code elements in your project to ensure long-term maintainability and compatibility with future versions of Go. Deprecated functions, methods, types, or packages are still functional but are no longer recommended for use, as they may have been replaced by better alternatives or could be removed in future releases.

To resolve this issue:

- Identify Deprecated Elements: Review the linter warnings to identify the specific code elements marked as deprecated. These are typically indicated by a `// Deprecated:` comment in their documentation.
- Documentation: Refer to the official Go documentation or the relevant package's documentation to understand why the element is deprecated and to find the recommended replacement.
- Update Code: Replace the deprecated elements with the suggested alternatives. This will help improve code quality, align with best practices, and reduce the risk of compatibility issues in the future.

To elevate code quality and enforce rigorous standards within your team, we recommend the following actions:

- Address Existing Warnings: Resolve the current warnings related to code style, correctness, and quality identified in the analysis. This fundamental step will eliminate inconsistencies and improve the overall health of the codebase.
- Integrate Additional Static Analysis Tools: While `golangci-lint` effectively identifies many common issues, incorporating additional static analysis tools can further enhance code quality and detect potential bugs and vulnerabilities. Consider integrating the following tools:

        - Staticcheck
        - Revive
    - Establish Style Guidelines: Develop or adopt a comprehensive Go style guide by utilizing official recommendations or community standards. Ensuring consistency and readability across the codebase will enhance collaboration and ease maintenance. Encourage all developers to adhere diligently to these guidelines.
    - Enhance Continuous Integration Pipeline: Integrate static analysis tools into your CI pipeline to facilitate early detection of linting issues during the development process. This proactive approach ensures code quality before deployment.

Always handle errors returned by functions and methods. This can involve logging the error, returning it to the caller, or taking corrective actions depending on the context. Ignoring errors should be avoided unless it is intentional and explicitly documented.

- Use Static Analysis Tools: Employ tools like errcheck to automatically detect unhandled errors in the codebase.
- Manual Code Review: Manually review the code to identify any unhandled errors that static analysis tools might miss.

Resolution: The codebase structure changed during the assessment period (e.g., the airdrop module is no longer present), which suggests that some refactoring has occurred. However, not all issues identified in the audit have been fully addressed, particularly the variable name collision with imported package names.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

Evidences

Imported package name as a name identifier

Issue details: File **abci_test.go**

- Variable 'app' collides with imported package name (2 places)

File **accounted_pool_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **airdrop_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **borrow_rate_test.go**

- Variable 'keeper' collides with imported package name

File **denom_liquidity_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **entry_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **epoch_infos_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **genesis_inflation_test.go**

- Variable 'keeper' collides with imported package name (2 places)

File **genesis_test.go**

- Variable 'app' collides with imported package name (2 places)

File **history_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **interest_rate_test.go**

- Variable 'keeper' collides with imported package name

File **keeper.go**

- Variable 'store' collides with imported package name (2 places)

File **keeper.go**

- Variable 'store' collides with imported package name (5 places)

File **msg_server_cancel_vest_test.go**

- Variable 'app' collides with imported package name (3 places)

File **msg_server_claim_rewards_test.go**

- Variable 'address' collides with imported package name

File **msg_server_claim_vesting_test.go**

- Variable 'app' collides with imported package name

File **msg_server_commit_claimed_rewards_test.go**

- Variable 'app' collides with imported package name (4 places)

File **msg_server_commit_liquid_tokens_test.go**

- Variable 'app' collides with imported package name

File **msg_server_update_vesting_info_test.go**

- Variable 'app' collides with imported package name (4 places)

File **msg_server_vest_now_test.go**

- Variable 'app' collides with imported package name (4 places)

File **msg_server_vest_test.go**

- Variable 'app' collides with imported package name (2 places)

File **pool_test.go**

- Variable 'keeper' collides with imported package name (4 places)

File **pool_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **pool_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **portfolio_test.go**

- Variable 'keeper' collides with imported package name (4 places)
- Variable 'app' collides with imported package name (3 places)

File **price_feeder_test.go**

- Variable 'keeper' collides with imported package name (3 places)

File **query_commitment_vesting_info_test.go**

- Variable 'app' collides with imported package name (2 places)

File **query_test.go**

- Variable 'app' collides with imported package name (6 places)

File **root.go**

- Variable 'app' collides with imported package name

File **time_based_inflation_test.go**

- Variable 'keeper' collides with imported package name (3 places)

Deprecated Element Inspection

Issue details: `WrapSDKContext` Deprecated: there is no need to wrap anymore as the Cosmos SDK context implements `context.Context`.

1. sdk.WrapSDKContext

2. sdk.WrapSDKContext

`NewClientProposalHandler` Deprecated: This function is deprecated and will be removed in a future release. Please use `MsgRecoverClient` and `MsgIBCSoftwareUpgrade` in favour of this legacy Handler.

1. ibcclient.NewClientProposalHandler

`ParamKeyTable` Deprecated

1. govv1.ParamKeyTable()

`WeightedProposalContent` Deprecated: Use `WeightedProposalMsg` instead.

1. []simtypes.WeightedProposalContent
2. []simtypes.WeightedProposalContent
3. []simtypes.WeightedProposalContent
4. []simtypes.WeightedProposalContent
5. []simtypes.WeightedProposalContent
6. []simtypes.WeightedProposalContent
7. []simtypes.WeightedProposalContent
8. []simtypes.WeightedProposalContent
9. []simtypes.WeightedProposalContent
10. []simtypes.WeightedProposalContent
11. []simtypes.WeightedProposalContent
12. []simtypes.WeightedProposalContent
13. []simtypes.WeightedProposalContent
14. []simtypes.WeightedProposalContent
15. []simtypes.WeightedProposalContent
16. []simtypes.WeightedProposalContent
17. []simtypes.WeightedProposalContent

`AppModule` Deprecated: use `appmodule.AppModule` with a combination of extension interfaes interfaces instead.

1. module.AppModule
2. module.AppModule

`PrintObjectLegacy` Deprecated: It will be removed in the near future!

1. clientCtx.PrintObjectLegacy(res)

`SetOutput` Deprecated: Use `SetOut` and/or `SetErr` instead.

1. cmd.SetOutput
2. cmd.SetOutput
3. cmd.SetOutput
4. cmd.SetOutput
5. cmd.SetOutput
6. cmd.SetOutput

`GenesisCoreCommand` Deprecated: use `Commands` instead.

1. genutilcli.GenesisCoreCommand

`FromTmPubKeyInterface` Deprecated: use `FromCmtPubKeyInterface` instead.

1. cryptocodec.FromTmPubKeyInterface

`ReadFile` Deprecated: As of Go 1.16, this function simply calls `os.ReadFile`.

1. ioutil.ReadFile
2. ioutil.ReadFile
3. ioutil.ReadFile
4. ioutil.ReadFile

`TempFile` Deprecated: As of Go 1.17, this function simply calls `os.CreateTemp`

1. ioutil.TempFile
2. ioutil.TempFile

`Time` Deprecated: Time based upgrades have been deprecated. Time based upgrade logic has been removed from the SDK.

1. Time

`UpgradedClientState` Deprecated: `UpgradedClientState` field has been deprecated. IBC upgrade logic has been moved to the IBC module in the sub module 02-client.

1. UpgradedClientState

Redundant Aliases

Issue details: We found several Go package imports that use unnecessary aliases. These aliases match exactly with their package names, which means they serve no purpose:

1. query
2. context
3. fmt
4. fmt
5. keeper
6. keeper
7. keeper
8. keeper
9. fmt
10. fmt
11. fmt

Redundant Parentheses

Issue details:

The “Redundant parentheses” warning indicates that your code contains parentheses that are unnecessary for the correct evaluation of an expression. These extra parentheses don’t affect how the code works but make it less clean and harder to read:

1. (swapFee.Quo(sumOfSwapFees))
2. (swapFee.Quo(sumOfSwapFees))
3. (swapFee.Quo(sumOfSwapFees))

Unhandled Errors

Issue details:

There are multiple instances where functions that return errors are not properly checked or handled. Unhandled errors occur when a function returns an error value, but the code does not verify or respond to this error. This can lead to unexpected application behavior, crashes, resource leaks, or security vulnerabilities.

As examples:

1. iter.Close()
2. f.Value.Set
3. app.SlashingKeeper.SetValidatorSigningInfo
4. app.StakingKeeper.RemoveUnbondingDelegation
5. app.StakingKeeper.SetValidator

Unnecessary Type Conversions

Issue details:

The “Redundant type conversion” warning in Go indicates that your code is performing a type conversion that is unnecessary because the variable or value is already of the target type. This redundancy can make the code less clean and harder to read without adding any functional benefit. For example, converting an `int` to an `int` or a `string` to a `string` is redundant because the value is already in the desired type:

1. sdk.AccAddress
2. sdk.AccAddress
3. sdk.AccAddress
4. sdk.AccAddress
5. math.Int
6. []byte
7. []byte
8. sdk.AccAddress
9. []byte

10. sdk.AccAddress
11. sdk.AccAddress
12. uint64
13. sdkmath.Int
14. math.LegacyDec
15. math.LegacyDec
16. math.LegacyDec
17. sdk.AccAddress
18. sdk.AccAddress
19. sdk.AccAddress

Unused Code and Legacy Cleanup

Issue details:

Unused parameter:

Unused parameters in functions can lead to confusion, bloated interfaces, and potential bugs if callers rely on them unintentionally.

Examples:

1. `snapshot` and `accPoolKeeper` in GetTokenARate
2. `tokenOutDenom` in CalcExitValueWithoutSlippage

Unused variable:

Unused variables clutter the codebase, waste memory, and suggest incomplete refactoring or outdated logic.

Examples:

1. `ErrInvalidDiscount` and `ErrInitialSpotPriceIsZero` in errors.go

Unused constant:

Constants that are defined but never referenced may indicate deprecated features or incomplete implementations.

Examples:

1. `RouterKey` in keys.go

Unused functions:

Functions that are never called increase code complexity and maintenance overhead.

Examples:

1. `createNPoolResponse` in pool_test.go

Used only in unit tests:

Functions or variables used exclusively in tests should be isolated or documented to avoid confusion in production code.

Examples:

1. SetLegacyElysStakeChange
2. DeleteLegacyElysStakeChange

Remove after migration:

Code marked for removal after migration poses technical debt and risks if retained beyond its intended lifecycle.

Examples:

```
// TODO: remove all legacy prefixes and functions after migration
func (k Keeper) DeleteLegacyElysStaked(ctx sdk.Context, address str
ing) {
store := prefix.NewStore(runtime.KVStoreAdapter(k.storeService.Open
KVStore(ctx)), types.LegacyKeyPrefix(types.LegacyElysStakedKeyPrefi
x))
store.Delete(types.LegacyElysStakedKey(address))
}

// remove after migration
func (k Keeper) GetLegacyExternalIncentiveIndex(ctx sdk.Context) (i
ndex uint64) {
store := prefix.NewStore(runtime.KVStoreAdapter(k.storeService.Open
KVStore(ctx)), types.KeyPrefix(types.LegacyExternalIncentiveIndexKe
yPrefix))
index = sdk.BigEndianToUint64(store.Get(types.LegacyExternalIncenti
veIndex()))
return index
}

// remove after migration
func (k Keeper) RemoveLegacyExternalIncentiveIndex(ctx sdk.Context)
{
store := prefix.NewStore(runtime.KVStoreAdapter(k.storeService.Open
KVStore(ctx)), types.KeyPrefix(types.LegacyExternalIncentiveIndexKe
yPrefix))
store.Delete(types.LegacyExternalIncentiveIndex())
}
```

F-2024-7523 - Telemetry Configs - Info

Description:

The Cosmos SDK enables operators and developers to gain insight into the performance and behavior of their application through the use of the `telemetry` package.

In the EndBlocker function of the `Keeper` module, there is an inconsistency in the telemetry metric key being used. Currently, the following line of code:

```
// EndBlocker of amm module
func (k Keeper) EndBlocker(ctx sdk.Context) {
defer telemetry.ModuleMeasureSince(types.ModuleName, time.Now(), te
lemetry.MetricKeyBeginBlocker)
```

uses `telemetry.MetricKeyBeginBlocker` as the telemetry key. However, this interpretation is not fully aligned, as the function is executed during the `EndBlocker` lifecycle phase. The correct telemetry key should be `telemetry.MetricKeyEndBlocker`, which more accurately reflects the stage of the block lifecycle this function is associated with.

Assets:

- Code Quality [https://github.com/elys-network/elys]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

To resolve the issue, it is recommended to take the following steps:

- **EndBlocker Function Update:** The telemetry key should be updated from `telemetry.MetricKeyBeginBlocker` to `telemetry.MetricKeyEndBlocker`. This change will ensure the telemetry system accurately tracks events during the `EndBlocker` phase, improving the reliability of lifecycle monitoring.
- **Comprehensive Telemetry Review:** A detailed review of all telemetry parameters across the relevant modules is advised to confirm their correct configuration and effectiveness in capturing the intended metrics. This proactive approach will help ensure robust performance tracking and timely detection of any discrepancies.

Resolution: The issue has been resolved. The code is now using `telemetry.MetricKeyEndBlocker` instead of the previously incorrect `telemetry.MetricKeyBeginBlocker`.

Verified on commit hash:
b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2024-7616 - Non-Deterministic Map Iteration - Info

Description:

In Golang, iterating over a map is non-deterministic, meaning the order in which key-value pairs are traversed is not guaranteed to be consistent across iterations.

In the context of a blockchain network, deterministic behavior is critical because all nodes must process transactions and state transitions in exactly the same order to ensure consensus on the ledger's state. Non-deterministic processing, such as iterating over a map in an undefined order, can result in different nodes reaching divergent states, even when processing the same blocks and transactions. This discrepancy can lead to consensus failures, potentially causing the network to fork or halt.

The function `BurnTokensForAllDenoms` in the `x/burner` module introduces a non-deterministic element through the iteration over a map:

x/burner/keeper/burn.go#L16:

```
func (k Keeper) BurnTokensForAllDenoms(ctx sdk.Context) error {
balances := k.getPositiveBalances(ctx)
for denom, balance := range balances {
if err := k.burnTokensForDenom(ctx, balance, denom); err != nil {
return err
}
}
return nil
}
```

At present, the operations executed within the loop (`k.burnTokensForDenom`) produce the same result regardless of the order in which they are invoked. Consequently, this issue is unlikely to cause chain instability or consensus failures under current conditions.

In the future, however, even minor or unrelated changes to the `Keeper` logic, such as altering how token balances are processed or introducing new dependencies, could inadvertently make the result order-dependent. This could introduce a deterministic inconsistency, leading to consensus-related vulnerabilities.

Assets:

- burner [https://github.com/elys-network/elys]
- Code Quality [https://github.com/elys-network/elys]

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: To preserve the integrity and consensus of Cosmos SDK-based blockchain networks, it is critical to eliminate non-deterministic behaviors arising from unsorted map iterations in Go.

- **Review the Codebase**: Conduct a detailed review to identify all instances of map iteration without enforced order, such as in the `BurnTokensForDenom` function.
- **Refactor Map Iterations**: For each identified instance, refactor the code by extracting map keys into a slice, sorting them (e.g., using `sort.Strings` for string keys), and iterating over the sorted keys. This ensures a predictable and consistent order of execution.

By enforcing deterministic behavior in functions like `BurnTokensForDenom`, all nodes will process map entries in the same order, safeguarding consensus and preventing potential forks or network disruptions.

Resolution: Elys Team will be fixing it in the next release, here is a comment from the them:-

> Currently not being used anywhere, will be fixing it in next release

F-2024-7935 - Mismatch Between Comment and Return Value in Voting Power Calculations - Info

Description: During the audit of the provided code, a discrepancy was identified between the comments and the actual return values of three functions:

- CalculateValidatorProjectedVotingPower
- CalculateDelegateProjectedVotingPower
- CalculateRedelegateProjectedVotingPower

The comments for these functions explicitly state that the return value represents the "calculated percentage." However, the actual implementation performs a fractional calculation, as demonstrated in `CalculateValidatorProjectedVotingPower`:

```
// Returns the projected voting power as a percentage (not a fraction)
func (min MinCommissionDecorator) CalculateValidatorProjectedVotingPower(ctx sdk.Context, delegateAmount sdkmath.LegacyDec) sdkmath.LegacyDec {
//
// Calculation of the projectedValidatorTokens and projectedTotalDelegatedTokens
//
return projectedValidatorTokens.Quo(projectedTotalDelegatedTokens)
}
```

Although this issue does not pose a direct security risk, such discrepancies between function documentation and behavior introduce avoidable confusion. This increases the likelihood of errors during development and integration, as developers may make incorrect assumptions about the return values, potentially resulting in invalid computations.

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: To address this issue, it is recommended to ensure consistency between the comments and the actual return values. There are two potential approaches to resolve the mismatch:

- **Update the Implementation to Return a Percentage:** Modify the return statements of all three functions to multiply the calculated fraction by 100 before returning it. This would align the implementation with the comments and ensure that the functions return the projected voting power as a percentage.

- **Update the Comments to Reflect the Current Behavior:** If the current implementation (returning a fraction) is the intended behavior, update the comments to accurately describe the return value.

Resolution: This will be fixed in the next release, here is the comment:-

Will be fixing it in next release.

F-2024-7959 - Proofreading - Info

Description: Several grammatical errors and typos were identified in the project documentation and code comments. These errors can lead to misunderstandings and reduce the overall clarity and professionalism of the project. These issues, while not affecting functionality, impact the professional presentation and readability of our codebase.

Examples of identified issues:

1. eixsting
2. liablities is
3. Perpertual
4. Recieving
5. avaialble
6. intemediate
7. sufficcient
8. CommentmentHook
9. declearation
10. addresss
11. shud
12. liquidty
13. Adress
14. positon
15. insufficent
16. Sucess
17. Curreny
18. Initialite
19. existant
20. entrys
21. position position
22. it close
23. a airdrop
24. whats (what's)
25. a entry
26. an normal
27. the the
28. a order
29. has spend
30. be prioritize

Assets:

- Code Quality [https://github.com/elys-network/elys]

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: Conduct a comprehensive proofreading pass across all documentation files, including comments, README files, and API documentation. Consider using automated tools like misspell to catch common errors. Additionally, establish a documentation style guide to maintain consistency in future contributions.

Resolution: During the assessment period, the Elys team did not provide any comments or proposed solutions for this finding.

F-2024-7989 - Suggested Improvements to the Cosmos SDK Math Library - Info

Description:

We identified several areas related to the math library where performance and code clarity could be improved. Specifically, we observed the use of *non-mutable functions* and *verbose zero-comparison methods*, both of which can negatively impact the efficiency and maintainability of the codebase.

- **Use of Non-Mutable Math Functions**: The code frequently employs non-mutable functions from the Cosmos SDK math library, such as `Mul`, `Add`, and `Neg`. These functions return new objects rather than modifying existing ones, leading to unnecessary object creation and increased memory usage.
- **Inefficient Zero and Sign Comparisons**: Zero and sign checks are sometimes performed using less direct methods, such as comparing values to `sdkmath.LegacyZeroDec()`. This approach can reduce code readability and make the intent less clear to future maintainers.

Benefits realized through refining the implementation include:

- **Cleaner Code**: More straightforward expressions and in-place modifications lead to cleaner and more maintainable code.
- **Performance Gains**: Optimizations can result in faster computations and a more responsive system overall.
- **Easier Maintenance**: Readable code with consistent patterns is easier for developers to understand and modify, reducing the onboarding time for new team members.

For detailed examples, please refer to the *Evidence* section.

Assets:

- Code Quality [https://github.com/elys-network/elys]

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation:

1. **Adopt Mutable Math Functions**: Replace non-mutable functions (`Mul`, `Add`, `Sub`, `Quo`, `Power`, `Abs`, `Neg`, etc.) with their

mutable counterparts (`MulMut`, `AddMut`, `SubMut`, `QuoMut`, `PowerMut`, `AbsMut`, `NegMut`) where appropriate.

2. **Use Direct Zero and Sign Checks**: Utilize methods like `IsZero()`, `IsPositive()`, and `IsNegative()` for comparisons.
3. **Code Consistency**: Ensure that these practices are applied consistently throughout the codebase. Consistency aids in maintenance and sets a standard for future development.

Resolution: This will be fixed in upcoming releases here is the comment from the Elys Team:-

> We will be upgrading to cosmossdk.io/math.Dec v1.5.x or above The operations will be different

---

Evidences

Examples of Mutable Functions

Issue details:

1. spotPrice = spotPrice.Mul(rate) → spotPrice.MulMut(rate)
2. spotPrice = spotPrice.Mul(rate) → spotPrice.MulMut(rate)
3. oracleAssetsTVL = oracleAssetsTVL.Add(v) → oracleAssetsTVL.AddMut(v)
4. joinValue = joinValue.Add(v) → joinValue.AddMut(v)
5. poolAmountOut = poolAmountOut.Neg() → poolAmountOut.NegMut()

Examples of Zero Comparisons

Issue details:

1. exp.LT(sdkmath.LegacyZeroDec()) → exp.IsNegative()
2. x.LTE(sdkmath.LegacyZeroDec()) → !x.IsPositive()
3. x.Equal(sdkmath.LegacyZeroDec()) → x.IsZero()
4. tokenPrice.Equal(sdkmath.LegacyZeroDec()) → tokenPrice.IsZero()
5. tokenPrice.Equal(sdkmath.LegacyZeroDec()) → tokenPrice.IsZero()
6. tokenPrice.Equal(sdkmath.LegacyZeroDec()) → tokenPrice.IsZero()
7. tokenPrice.Equal(sdkmath.LegacyZeroDec()) → tokenPrice.IsZero()
8. tokenPrice.Equal(sdkmath.LegacyZeroDec()) → tokenPrice.IsZero()
9. tokenPrice.Equal(sdkmath.LegacyZeroDec()) → tokenPrice.IsZero()

10. msg.Leverage.Equal(math.LegacyZeroDec()) →
    msg.Leverage.IsZero()
11. msg.MaxVotingPower.LTE(math.LegacyZeroDec()) →
    !msg.MaxVotingPower.IsPositive()
12. msg.MinCommission.LTE(math.LegacyZeroDec()) →
    !msg.MinCommission.IsPositive()
13. edenCommittedAndElysStakedDec.GT(math.LegacyZeroDec()) →
    edenCommittedAndElysStakedDec.IsPositive()
14. edenCommittedAndElysStakedDec.GT(math.LegacyZeroDec()) →
    edenCommittedAndElysStakedDec.IsPositive()
15. shareRatio.LTE(sdkmath.LegacyZeroDec()) →
    !shareRatio.IsPositive()
16. totalDelegatedTokens.LTE(sdkmath.LegacyZeroDec()) →
    !totalDelegatedTokens.IsPositive()

F-2025-8077 - Clarifying Pool Existence Handling in the 'Calculate Pool APRs' Query - Info

Description: In the `CalculatePoolAprs` function, when querying APR data for specific pool IDs, the function exhibits potentially misleading behavior when handling non-existent pools. Instead of ignoring or excluding invalid pool IDs, it silently includes entries for non-existent pools with zero APR values in the returned response.

```
poolInfo, found := k.GetPoolInfo(ctx, poolId)
if !found {
data = append(data, types.PoolApr{
PoolId: poolId,
UsdcApr: sdkmath.LegacyZeroDec(),
EdenApr: sdkmath.LegacyZeroDec(),
TotalApr: sdkmath.LegacyZeroDec(),
})
continue
}
```

Depending on your application's logic, this behavior might be acceptable. However, it is crucial to ensure this behavior is clearly communicated to the client to avoid confusion or unintended outcomes.

Assets:

- masterchef [https://github.com/elys-network/elys]

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation:

1. **Define Clear Behavior:** Explicitly document this behavior as part of your API or system documentation. Ensure it is clear to clients that non-existent pools will result in zero-initialized entries.
2. **Introduce a Configuration Flag:** Add a configurable flag to control whether non-existent pools should be included in the returned list. This provides flexibility for different use cases.
3. **Additional field:** Add an additional field to indicate that the pool is non-existent, enabling clients to distinguish this state unambiguously.

Resolution: Its intended design, here is a comment from Elys Team:-

> This is the intended design for the client. When rewards gets enabled through gov proposal, it automatically solves, till then at client end we need to show APRs

F-2025-8154 - Identical Implementation in 'RemovePoolInfo' and 'RemoveLegacyPoolInfo' Functions - Info

Description: It was observed that two functions, `RemovePoolInfo` and `RemoveLegacyPoolInfo`, have identical implementations. Both functions perform the same operation: they delete a pool's information.

RemovePoolInfo:

```
func (k Keeper) RemovePoolInfo(ctx sdk.Context, poolId uint64) {
store := runtime.KVStoreAdapter(k.storeService.OpenKVStore(ctx))
key := types.GetPoolInfoKey(poolId)
store.Delete(key)
}
```

RemoveLegacyPoolInfo:

```
func (k Keeper) RemoveLegacyPoolInfo(ctx sdk.Context, poolId uint64
) {
store := runtime.KVStoreAdapter(k.storeService.OpenKVStore(ctx))
key := types.GetPoolInfoKey(poolId)
store.Delete(key)
}
```

The presence of two functions with the same body suggests that either:

1. The `RemoveLegacyPoolInfo` function was introduced as a temporary measure during a migration or refactoring process and was never removed.
2. There was an intention to differentiate the behavior of these functions in the future, but the differentiation was never implemented.

Assets:

- masterchef [https://github.com/elys-network/elys]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

- If `RemoveLegacyPoolInfo` is no longer needed, it should be removed entirely, and all references to it should be updated to use `RemovePoolInfo`.
- If `RemoveLegacyPoolInfo` is intended to serve a different purpose in the future, the function should be updated to reflect its unique behavior. Until then, it should either be removed or marked as deprecated with a clear comment explaining its future purpose.

Resolution:

issue has been resolved by removing the redundant `RemoveLegacyPoolInfo` function, as recommended in the finding. The codebase now only contains the `RemovePoolInfo` function.

Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc

F-2025-8162 - Code Quality Concerns in WithdrawElysStakingRewards - Info

Description: In the `WithdrawElysStakingRewards` unction, the following issues have been identified:

a. The function contains a redundant error check, as illustrated below:

```
if err != nil {
return nil, err
}

if err != nil {
return nil, err
}
```

b. The call to `k.Keeper.Keeper.IterateDelegations` can be streamlined to `k.IterateDelegations`, simplifying the codebase and improving readability.

c. The sibling function, `WithdrawAllRewards`, implements the same logic, indicating redundancy. Consolidating these functions could eliminate duplication and improve maintainability.

Assets:

- estaking [https://github.com/elys-network/elys]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

- **Remove Duplicate Error Checks**: Eliminate the redundant error check to streamline the code and improve readability. Ensure that error handling is performed only once to maintain clarity.
- **Simplify Function Calls**: Update the call to `k.Keeper.Keeper.IterateDelegations` to `k.IterateDelegations` to enhance code simplicity and maintainability.
- **Evaluate Redundancy of WithdrawAllRewards**: Conduct a thorough review of the `WithdrawAllRewards` function to determine if it can be safely removed without affecting functionality. If it is

indeed redundant, consider refactoring the code to eliminate unnecessary duplication.

Resolution: Verified on commit hash: b739d7cf2c50f39024d4872456c1ac57eecc61dc:-
https://github.com/elys-network/elys/blob/main/x/estaking/keeper/msg_server.go#L89
https://github.com/elys-network/elys/blob/main/x/estaking/keeper/msg_server.go#L126

F-2025-8188 - Redundant Price Removal Logic in EndBlock Function - Info

Description: The `EndBlock` function contains two separate conditions for removing outdated prices: one based on `PriceExpiryTime` and another based on `LifeTimeInBlocks`. This design introduces redundancy, as the same price may be checked and removed twice within the same block.

abci.go:

```
for _, price := range k.GetAllPrice(ctx) {
if price.Timestamp+params.PriceExpiryTime < uint64(ctx.BlockTime().
Unix()) {
k.RemovePrice(ctx, price.Asset, price.Source, price.Timestamp)
}

if price.BlockHeight+params.LifeTimeInBlocks < uint64(ctx.BlockHeig
ht()) {
k.RemovePrice(ctx, price.Asset, price.Source, price.Timestamp)
}
}
```

Assets:

- oracle [https://github.com/elys-network/elys]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To address the redundancy and improve the efficiency of the `EndBlock` function, consider combining the two conditions into a single check using a logical OR (`||`) operator. This ensures that prices are removed if either condition is met, without redundant checks or state writes.

Example:

```
func (k Keeper) EndBlock(ctx sdk.Context) {
params := k.GetParams(ctx)
for _, price := range k.GetAllPrice(ctx) {
if price.Timestamp+params.PriceExpiryTime < uint64(ctx.BlockTime().
Unix()) ||
price.BlockHeight+params.LifeTimeInBlocks < uint64(ctx.BlockHeight(
)) {
k.RemovePrice(ctx, price.Asset, price.Source, price.Timestamp)
}
}
}
```

Resolution: Resolved in PR https://github.com/elys-network/elys/pull/1210

Disclaimers

Hacken Disclaimer

The blockchain protocol given for audit has been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in the protocol source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other protocol statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the blockchain protocol.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Blockchain protocols are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the protocol can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited blockchain protocol.

Appendix 1. Severity Definitions

Severity	Description
Critical	Vulnerabilities that can lead to a complete breakdown of the blockchain network's security, privacy, integrity, or availability fall under this category. They can disrupt the consensus mechanism, enabling a malicious entity to take control of the majority of nodes or facilitate 51% attacks. In addition, issues that could lead to widespread crashing of nodes, leading to a complete breakdown or significant halt of the network, are also considered critical along with issues that can lead to a massive theft of assets. Immediate attention and mitigation are required.
High	High severity vulnerabilities are those that do not immediately risk the complete security or integrity of the network but can cause substantial harm. These are issues that could cause the crashing of several nodes, leading to temporary disruption of the network, or could manipulate the consensus mechanism to a certain extent, but not enough to execute a 51% attack. Partial breaches of privacy, unauthorized but limited access to sensitive information, and affecting the reliable execution of smart contracts also fall under this category.
Medium	Medium severity vulnerabilities could negatively affect the blockchain protocol but are usually not capable of causing catastrophic damage. These could include vulnerabilities that allow minor breaches of user privacy, can slow down transaction processing, or can lead to relatively small financial losses. It may be possible to exploit these vulnerabilities under specific circumstances, or they may require a high level of access to exploit effectively.
Low	Low severity vulnerabilities are minor flaws in the blockchain protocol that might not have a direct impact on security but could cause minor inefficiencies in transaction processing or slight delays in block propagation. They might include vulnerabilities that allow attackers to cause nuisance-level disruptions or are only exploitable under extremely rare and specific conditions. These vulnerabilities should be corrected but do not represent an immediate threat to the system.

Appendix 2. Scope

The scope of the project includes the following components from the provided repository:

Scope Details	
Repository	https://github.com/elys-network/elys/
Commit	99bd50ba942ef856291aec1705e0d3acded58444

Components in Scope

1. Cosmos SDK Core App
2. Cosmos fork review
3. RPC
4. Cryptography and keys
5. Custom Modules:
    1. x/accountedpool
    2. x/commitment
    3. x/masterchef
    4. x/stablestake
    5. x/transferhook
    6. x/amm
    7. x/epochs
    8. x/oracle
    9. x/tier
    10. x/assetprofile
    11. x/estaking
    12. x/parameter
    13. x/tokenomics
    14. x/burner
    15. x/leveragelp
    16. x/perpetual
    17. x/tradeshield