HANJI LIQUIDITY VAULT SECURITY AUDIT REPORT

April 04, 2025

MixBytes()

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

Stage goals
- Build an independent view of the project's architecture.
- Identifying logical flaws.

2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

Stage goal

Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

Stage goal

Detect inconsistencies with the desired model.

4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

Stage goals

- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

1.3 Project Overview

This Hanji Derivatives Engine provides a liquidity management system controlled by a market maker for a decentralized exchange that utilizes on-chain limit order books. The main contract, **LPManager**, manages liquidity addition and removal, applies dynamic fees, and collects protocol revenues based on fees collected from connected DEXs and the market maker's activity. Another key component, **RfqProxyLOB**, facilitates the matching of off-chain signed RFQ orders on-chain, enhancing flexibility for market makers. Overall, the system incorporates a Pyth oracle for pricing, robust fee logic, and advanced liquidity controls.

1.4 Project Dashboard

Project Summary

Title	Description
Client	Hanji
Project name	Liquidity Vault
Timeline	23.12.2024 - 26.02.2025
Number of Auditors	3

Project Log

Date	Commit Hash	Note
23.12.2024	2a14dea125700a7fbbe60c213dfc5d3004fae5be	initial commit
26.12.2024	c76c91351d99235f7a36b8bcc7bf57791ce8c78d	intermediate commit with fixes
17.02.2025	2b64cbc74e87a4c5204bd436614071d5731cac41	commit for the reaudit
26.02.2025	366b78b02aa88d9d3debd7fee0234a0b6e4fa10d	commit for the reaudit

Project Scope

The audit covered the following files:

File name	Link
src/LPManagerFactory.sol	LPManagerFactory.sol
src/ErrorReporter.sol	ErrorReporter.sol

File name	Link
src/LPManager.sol	LPManager.sol
src/Proxy.sol	Proxy.sol
src/LPToken.sol	LPToken.sol
src/ProxyLOB.sol	ProxyLOB.sol
src/RfqProxyLOB.sol	RfqProxyLOB.sol
src/utils/Ecdsa.sol	Ecdsa.sol
src/utils/RfqOrderLib.sol	RfqOrderLib.sol
src/ProxyLOBBatch.sol	ProxyLOBBatch.sol
src/ProxyPyth.sol	ProxyPyth.sol
src/utils/TokenValueCalculator.sol	TokenValueCalculator.sol
src/utils/PythPriceHelper.sol	PythPriceHelper.sol

Deployments

File name	Contract deployed on mainnet	Comment
LPManagerFactory.sol	0x0A3AA8...a3FB1582	LPManagerFactory
Proxy.sol	0x4C911b...c9e2c1d6	LPManager Proxy (WXTZ, WETH, USDC, WBTC)
LPManagerFactory.sol	0x013069...480403C8	EmptyImplementation
LPManager.sol	0x4940d5...8833d2Dc	LPManager Implementation
LPToken.sol	0x1cd88f...9003A930	

File name	Contract deployed on mainnet	Comment
LPManagerHelper.sol	0x6b285F...b83cF64d	
ProxyLOBBatch.sol	0x78B916...804dE2b7	
RfqProxyLOB.sol	0xC4D749...6Ef51091	
ProxyPyth.sol	0x2b1802...46C55D77	

1.5 Summary of findings

Severity	# of Findings
Critical	5
High	2
Medium	4
Low	8

ID	Name	Severity	Status
C-1	Insecure Liquidity Valuation with Oracle Updates	Critical	Fixed
C-2	Incorrect Remainder Allocation in Market-Only Orders	Critical	Fixed
C-3	Improper Bid Trie Reference Causing Reserve Overstatement	Critical	Fixed
C-4	Double-Counted Fees in Partial Bid Claims	Critical	Fixed
C-5	Missing Price Factor in Claimed Shares Calculation	Critical	Fixed
H-1	Zero-Address `rfqOrderSigner` Allows Unrestricted Order Execution	High	Fixed
H-2	Centralization Risks and Market Maker Potential Exploits	High	Fixed
M-1	Imprecise Balance Check in `removeLiquidity`	Medium	Fixed
M-2	Weak Enforcement of Market Maker's LP Share Requirement	Medium	Fixed
M-3	Irrevocable RFQ Orders Allow Stale Execution	Medium	Fixed
M-4	Paused State Overly Restricts Liquidity Removal and Order Cancellation	Medium	Fixed

L-1	Insufficient Validation for Configuration Parameters	Low	Fixed
L-2	Rounding-Based Fee Bypass	Low	Fixed
L-3	Suboptimal Initial LP Minting Logic	Low	Fixed
L-4	Unnecessary State Storage of `pyth` Address	Low	Fixed
L-5	Public Function Can Be External	Low	Fixed
L-6	Separate Price Update from Validation for Improved Clarity	Low	Fixed
L-7	Risk of Incorrect Spread Calculation	Low	Fixed
L-8	Excessive Arguments Reduce Readability and Increase Complexity	Low	Fixed

1.6 Conclusion

We conducted an audit of the project, with particular focus on the attack vectors outlined below.

1. **Arbitrage Opportunities in Different Price Scenarios**: Simulated various price scenarios to identify arbitrage opportunities by generating streams of opposing RFQ (Request for Quote) orders, analyzing selective order execution and reordering strategies to exploit market conditions.
2. **Reentrancy**: Tested contract functions for vulnerabilities to reentrancy attacks by simulating recursive calls during state updates and fund transfers, ensuring the system prevents unauthorized re-entry and maintains consistency.
3. **Simultaneous Order Execution via LOB and LPManager**: Analyzed the execution of orders through both the Limit Order Book (LOB) and the LPManager to identify potential arbitrage opportunities
4. **Calculation and Logic Implementation Errors**: Examined the system for inaccuracies in calculations and flaws in logic implementation that could lead to unintended behavior, financial loss, or security vulnerabilities.
5. **Arbitrage via Liquidity Addition and Removal during Predictable Oracle Price Updates**: Tested scenarios where the predictability of upcoming oracle price updates could be exploited by strategically adding or removing liquidity to create arbitrage opportunities.
6. **Manipulation of Swap Function Parameters in Various Market Conditions**: Analyzed the impact of adjusting swap function parameters under different market scenarios, including price changes and variations in LOB liquidity volume, to identify potential vulnerabilities or exploitable behaviors.
7. **Improper Protocol Configuration**: Tested scenarios involving misconfigured protocol parameters, such as a null rfqOrderSigner, excessively high maxOracleAge, or zero cooldown, to identify potential security vulnerabilities.
8. **Signature Handling and Replay Attack Resistance**: Verified the correctness of signature validation mechanisms to ensure proper authentication and tested the system's resistance to replay attacks by attempting to reuse valid signatures in different contexts, ensuring nonce or timestamp protections are effective.
9. **Integration with LOB Contract**: Reviewed the integration with the Limit Order Book (LOB) contract to ensure proper functionality, data consistency, and the absence of vulnerabilities during order processing and execution.
10. **Deployment Phase Attacks**: Analyzed the deployment process to identify potential vulnerabilities, such as race conditions, misconfigurations, or exploitable states before the system becomes fully operational.
11. **Liquidity Drain via Adverse Orders by Malicious Market Makers**: Examined scenarios where a malicious market maker generates orders deliberately disadvantageous to liquidity providers, converting their losses into profits for an affiliated account. This represents a distinct and more severe issue than centralization, as market makers could be third parties, not directly under the protocol's control.
12. **Centralization & Admin Privileges**: Examined the upgradeable contract's architecture and administrative controls to assess potential risks of malicious code injection, unauthorized modifications and pausing the liquidity removal, which could grant a single entity complete control over user funds.

Given the complexity and size of the project, as well as the significant number of issues identified during our audit, we strongly recommend conducting an additional audit with another team. This step would mitigate the risk of human error, provide a broader and deeper analysis, and further enhance community trust by

demonstrating a commitment to transparency and security. An independent review could help identify any overlooked issues, ensuring the project's security and reinforcing its reputation in the market.

Additionally, we note that more comprehensive test coverage before initial audit stage could have helped identify many vulnerabilities during the development. The identified vulnerabilities are listed in the section below.

2. FINDINGS REPORT

2.1 Critical

C-1	Insecure Liquidity Valuation with Oracle Updates
Severity	Critical
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `addLiquidity` and `removeLiquidity` functions of the `LPManager` contract.

Because Pyth oracles can be updated atomically by users in the same transaction, it is possible to supply liquidity at a low price and subsequently remove it at a higher price, all within a single transaction. This creates an arbitrage opportunity that can rapidly extract value from the protocol.

The issue is classified as **critical** severity because it allows attackers to exploit sudden price changes for profit, potentially causing substantial losses to the protocol.

Recommendation

We recommend introducing a system of price epochs with duration equals to `maxOracleAge` and updating the minimum/maximum tracked prices at each `addLiquidity`/`removeLiquidity`. Specifically:

- For `addLiquidity`, use the highest price from the last two epochs to calculate the minted shares.
- For `removeLiquidity`, use the lowest price from the last two epochs to calculate the amount of tokens to transfer.
- Update these historical price bounds every time liquidity is added or removed.

This approach helps mitigate abrupt, short-time price manipulations.

Client's commentary

> Fixed as recommended. Additionally, the problem has been significantly mitigated by setting maxOracleAge to its minimum acceptable value.

C-2	Incorrect Remainder Allocation in Market-Only Orders
Severity	Critical
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `_placeOrder` function of the `ProxyLOB` contract.

When `marketOnly` is set to `true` the order is not included into `IOnchainTrie` within the `IOnchainLOB.placeOrder`, but the code still calculates `remainShares` as `quantity - executedShares`, instead of correctly setting it to zero. As a result, these non-existent leftover shares are added to `lobReservesByTokenId` and factored into `_getReserves`, ultimately inflating the protocol's reported total balance. Since `_getReserves` then shows a value higher than what truly exists, this discrepancy can lead to insolvency of the protocol.

The issue is classified as **critical** severity because overstated reserves severely undermine the protocol's financial stability, risking potential losses for liquidity providers.

Recommendation

We recommend setting `remainShares` to zero when `marketOnly` is `true`, ensuring no residual shares are mistakenly recorded. This accurately reflects that market orders have no remaining portion and preserves correct reserve accounting.

C-3	Improper Bid Trie Reference Causing Reserve Overstatement
Severity	Critical
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `_claimOrder` function of the `ProxyLOB` contract.

When `isAsk = false` (a bid), the final bit of the `orderId` indicates the relevant on-chain trie (bid trie expects a zero-ending bit). However, `_claimOrder` does not convert or mask the `orderId` before querying the trie, causing the bid order to appear non-existent and returning zero values. This erroneously inflates the contract's reserves by failing to correctly match and settle claimed amounts from the reserves of the claimed order.

The issue is classified as **critical** severity because it can produce a chain of incorrect calculations that ultimately overstate the protocol's asset balances.

Recommendation

We recommend adjusting `_claimOrder` to convert the `orderId` correctly when `isAsk = false`, ensuring that bid orders are properly identified in the bid trie.

C-4	Double-Counted Fees in Partial Bid Claims
Severity	Critical
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `_claimOrder` function of the `ProxyLOB` contract.

When placing a bid (`isAsk = false`), the contract transfers `quantity * price + passivePayoutCommission` into the LOB and records the same total in `lobReservesByTokenId`. However, if the order is partially executed, and `claimOrder` is invoked with `only_claim = false`, the unfilled portion of the fee is returned to the LPManager but is never subtracted from `lobReservesByTokenId`. Consequently, `_getReserves` continues to count fees that no longer remain, causing an inflated reserve calculation.

The issue is classified as **critical** severity because overstating reserves can lead to protocol insolvency if the system operates on incorrect balance assumptions.

Recommendation

We recommend updating `_claimOrder` to remove the unclaimed fee portion from `lobReservesByTokenId` once it is returned to the LPManager. This prevents double-counting in reserve calculations. For instance, change:

```
uint256 executedValue = executedShares * price;
uint256 passiveFee = executedValue.mulWadUp(passiveCommissionRate);
```

to:

```
uint256 returnedValue = totalClaimedShares * price;
uint256 passiveFee = returnedValue.mulWadUp(passiveCommissionRate);
```

C-5	Missing Price Factor in Claimed Shares Calculation
Severity	Critical
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `_claimOrder` function of the `ProxyLOB` contract.

When `isAsk = false`, the current code subtracts `(totalClaimedShares + passiveFee) * scalingFactorTokenY` from the reserves without multiplying `totalClaimedShares` by `price`. Consequently, the actual value of the claimed shares is not correctly accounted for, resulting in a higher-than-accurate reserves value.

The issue is classified as **critical** severity because it leads to a miscalculation of the claimed token's worth, which inflates the protocol's reported balances.

Recommendation

We recommend using `(totalClaimedShares * price + passiveFee) * scalingFactorTokenY` for the subtraction. This ensures the claimed amount is converted into its proper value before being removed from the reserves.

2.2 High

H-1	Zero-Address `rfqOrderSigner` Allows Unrestricted Order Execution
Severity	High
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `_swapWithRfqOrder` functions of the `RfqProxyLOB` contract.

If `rfqOrderSigner` is set to `address(0)`, the signature verification logic will treat any invalid signature as signed correctly, because `signer == rfqOrderSigner` trivially matches when both are zero. This effectively disables signature checks, permitting anyone to execute unauthorized orders and drain the contract's funds.

The issue is classified as **high** severity because it entirely bypasses RFQ signature security and can lead to significant fund loss if misconfigured.

Recommendation

We recommend enforcing a strict requirement that `rfqOrderSigner` must never be the zero address. Specifically:

1. Validate in the setter function and initializer that reverts if `rfqOrderSigner == address(0)`.
2. Add an additional check in the signature verification logic to prevent any orders from processing if the signer address is zero.

Client's commentary

> The RFQ logic has been extracted into a separate contract, which substantially reduces risks for the LPManager contract itself.

H-2	Centralization Risks and Market Maker Potential Exploits
Severity	High
Status	Fixed in 2b64cbc7

Description

This issue has been identified regarding the protocol's upgradeability and the market maker's control over user funds.

Admin Centralization: As the contract is upgradeable, the admin can modify core protocol logic at any time. If a single, potentially compromised individual or entity controls this role, they could introduce malicious code or alter system parameters without user consent, leading to a total takeover of user funds.

Market Maker Privilege: The market maker can execute trades on its own LOB positions, potentially at rates harmful to liquidity providers. For instance, a 1%-below-market swap with an address under its control could drain a large portion of the pool's value if made repeatedly. This risk grows if the market maker is less trusted than the admin but still has considerable influence.

The issue is classified as **high** severity because either the admin or the market maker can perform actions that jeopardize all participant funds, up to an almost complete loss.

Recommendation

We recommend:

1. Implementing a multisig with a timelock for critical administrative (upgrade) functions. This restricts any single party from pushing harmful updates immediately.
2. Introducing a safety deposit mechanism for the market maker. If the market maker's trades cause protocol losses, they are first covered by its deposit. If losses approach the deposit limit, further market maker actions should be blocked, and the deposit seized if necessary.

These measures reduce the risks of centralized exploitation and ensure the market maker is financially liable for damaging trades.

2.3 Medium

M-1	Imprecise Balance Check in `removeLiquidity`
Severity	Medium
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `removeLiquidity` function of the `LPManager` contract.

When removing liquidity, the contract's calculations assume that the `protocolFee` will be deducted from the balance. However, this fee remains on the contract's balance rather than being immediately subtracted. This can make `nextTotalUsd` and `nextTokenUsd` appear lower than they actually are, causing legitimate liquidity removal requests near the `lowerBoundWeight` to revert.

The issue is classified as **medium** severity because it can disrupt normal user operations and prevent valid withdrawals under specific conditions.

Recommendation

We recommend adjusting the balance check logic to ensure only truly removable tokens are counted. Any protocol fee portion still on the contract's balance should be excluded or otherwise handled to avoid erroneously restricting valid withdrawal requests.

M-2	Weak Enforcement of Market Maker's LP Share Requirement
Severity	Medium
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `_checkPlaceOrderAllowance()` function of the `LPManager` contract.

Currently, the primary market maker's LP share requirement is only enforced when they remove liquidity or other users add liquidity to the protocol. There is no corresponding check for other flows, such as transferring or removing LP tokens by other users, allowing the market maker to bypass the minimum share constraint and block deposits of other users.

The issue is classified as **medium** severity because it can undermine the intended minimum stake rule and block deposits.

Recommendation

We recommend extending the `_checkPlaceOrderAllowance()` logic to also check outgoing transfers from the primary market maker. This prevents the market maker from reducing their LP share below the required proportion through other mechanisms.

M-3	Irrevocable RFQ Orders Allow Stale Execution
Severity	Medium
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `swapWithRfqOrder` and related functions of the `RfqProxyLOB` contract.

Although there is an `expires` field to limit order lifetimes, a market maker might need to cancel an RFQ order immediately, well before `expires` is reached. Currently, there is no way to invalidate such an order once signed. Consequently, a user can hold onto old orders until a price swing makes them profitable. Additionally, contradictory RFQ orders (e.g., buying at one price and selling at a higher price) could be executed back-to-back by a single user, creating a risk-free arbitrage.

The issue is classified as **medium** severity because it grants users anadvantage ("free option"), potentially harming the market maker if order cancellation is required before expiration.

Recommendation

We recommend implementing a robust on-chain invalidation method, such as allowing the market maker to revoke specific order hashes. This ensures that orders can be canceled immediately if market conditions change.

Client's commentary

> Fixed by inverting the logic: the client signs the order and sends it to market maker, after which the market maker updates the quotes and submits the client's order.

M-4	Paused State Overly Restricts Liquidity Removal and Order Cancellation
Severity	Medium
Status	Fixed in 366b78b0

Description

This issue arises when `_validateLPPriceAndDistributeFees` detects a significant drop in the LP token price and triggers `_pause()`.

In this paused state, the `removeLiquidity()` function is disabled, preventing LP holders from withdrawing their tokens at a time when market conditions could be deteriorating. Moreover, market makers cannot cancel their outstanding orders in the LOB contract, leaving those orders to hang in a potentially volatile situation. This can compound losses if prices move further against them while the contract is paused.

The issue is classified as **medium** severity because it restricts user access to their funds and prohibits cancelling potentially risky orders during severe market shifts.

Recommendation

We recommend refining the pause mechanism so that liquidity providers can still remove liquidity and market makers can cancel open orders when severe market shifts occur. Instead of setting a global pause state for the entire protocol, a dedicated flag such as `slashingAvailableStatus` could be utilized. If `slashingAvailableStatus` is true, only the `primaryMarketMaker` should be restricted from removing liquidity, transferring tokens, and the protocol should be paused for placing new orders with enabled ability to claim open positions. This ensures that other users can exit the protocol promptly while preventing further potentially harmful actions from the primary market maker.

2.4 Low

L-1	Insufficient Validation for Configuration Parameters
Severity	Low
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `setConfig` function of the `LPManager` contract.

There are minimal checks on parameters like `cooldownDuration` and `maxOracleAge`. If set to very long values, they could inadvertently disrupt normal protocol operations or lead to unpredictable behaviors.

The issue is classified as **low** severity because it mainly involves a risk of misconfiguration rather than an immediate exploit.

Recommendation

We recommend setting reasonable upper and lower limits for both `cooldownDuration` and `maxOracleAge` to prevent harmful configurations.

Client's Commentary

> MixBytes: An upper limit for `maxOracleAge` was introduced, while leaving `cooldownDuration` without restrictions. Marked as fixed, since an excessively high cooldown parameter won't cause intermediate disruption to protocol operations and can be easily adjusted.

L-2	Rounding-Based Fee Bypass
Severity	Low
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `addLiquidity` function of the `LPManager` contract.

Integer division in fee calculations can allow an attacker to exploit rounding, shaving off small portions of fees. Repeated usage could slightly diminish the overall fees collected.

The issue is classified as **low** severity because the financial impact is minimal, though it may accumulate over time.

Recommendation

We recommend using rounding-up division (e.g. using `solmate.FixedPointMathLib.mulDivUp`) in fee calculations to ensure no fractions are lost due to division.

L-3	Suboptimal Initial LP Minting Logic
Severity	Low
Status	Fixed in 2b64cbc7

Description

This issue has been identified within the `addLiquidity` function of the `LPManager` contract.

When the pool is initialized (i.e., `lpTotalSupply == 0`), the logic uses only `tokenUSDValue` of the newly added liquidity instead of accounting for any tokens already in the contract. This might let sophisticated users transfer tokens in advance, influencing the first LP mint in ways that create an imbalance in share price.

The issue is classified as **low** severity because it doesn't pose security risk, and user-defined parameters like `minLPMinted` often mitigate large gains.

Recommendation

We recommend basing the initial mint on the total pool value (existing tokens plus newly added tokens) to ensure a more accurate and fair LP token distribution.

L-4	Unnecessary State Storage of `pyth` Address
Severity	Low
Status	Fixed in 366b78b0

Description

The `pyth` address is stored as a regular state variable. If this address is never intended to change, making it effectively immutable could save gas on repeated reads.

The issue is classified as **low** severity because it focuses on optimization rather than a security flaw.

Recommendation

We recommend defining `pyth` as an `immutable` variable within the constructor if updates are unnecessary post-deployment.

L-5	Public Function Can Be External
Severity	Low
Status	Fixed in 366b78b0

Description

This issue has been identified in the `getReserves` function of the `ProxyPyth` contract. Currently, it is marked as `public` but is only used externally, not called from within the contract itself. Changing it to `external` allows for slightly more efficient compilation and reduces gas consumption for calls.

Recommendation

We recommend updating the function signature from `public` to `external`. This optimizes how Solidity handles the function call interface, potentially saving some gas over time.

L-6	Separate Price Update from Validation for Improved Clarity
Severity	Low
Status	Fixed in 366b78b0

Description

This issue has been identified in the current design, which relies on `validatePriceMovement` for both updating and validating price movements with multiple parameters. This approach can lead to confusion about the purpose of each parameter and makes the contract logic harder to audit and maintain.

The issue is classified as **low** severity because it does not introduce a direct vulnerability but negatively affects code clarity and maintainability.

Recommendation

We recommend separating the logic into two dedicated functions:

1. `resetPrice()`: Responsible only for fetching and storing the default price.
2. `validatePriceMovement()`: Called afterward to verify that price movements remain within expected boundaries.

This separation of functionality will make the code more intuitive, minimize confusion about parameters, and improve long-term maintainability.

L-7	Risk of Incorrect Spread Calculation
Severity	Low
Status	Fixed in 366b78b0

Description

In the `updateAndGetEpochPrice` function, when a new price is fetched from the oracle, the stored exponent (`epochPriceData.expo`) is compared with the current oracle exponent (`pythPrice.expo`). If these values differ, the function rescales the stored price variables (`cMaxPrice`, `cMinPrice`, `pMaxPrice`, and `pMinPrice`) by dividing or multiplying by a power of ten. When the exponent decreases—meaning the new exponent is greater than the stored one—the rescaling involves dividing these values using integer arithmetic, which always rounds down.

This downward rounding can significantly alter the `cMaxPrice` and `pMaxPrice` values depending on the magnitude of the exponent difference. Since the protocol calculates the spread based on the difference between the minimum and maximum prices, a substantially reduced maximum price due to truncation can distort the spread calculation. Such an incorrect spread could, in rare cases, lead to unwarranted user enrichment.

Recommendation

We recommend rounding the maximum prices upward instead, ensuring that the spread calculation remains in favor of the protocol.

L-8	Excessive Arguments Reduce Readability and Increase Complexity
Severity	Low
Status	Fixed in 366b78b0

Description

This issue has been identified within functions such as `setConfig` and `getConfig`, where more than five separate parameters are passed rather than using a single structured parameter or return object. Handling numerous individual parameters can significantly reduce code readability, complicate integration, and increase the risk of errors during debugging.

Although one might assume that using structures in memory could incur extra gas costs, the impact is minimal under typical `mstore` and `mload` operations. Consequently, consolidating these parameters does not introduce a significant extra gas cost but rather improves maintainability and clarity.

Recommendation

We recommend consolidating these multiple function parameters (and corresponding return values) into memory structures where logical grouping is possible. This refactoring can simplify the function signatures and enhance overall code clarity.

3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

Contacts

https://github.com/mixbytes/audits_public

https://mixbytes.io/

hello@mixbytes.io

https://twitter.com/mixbytes