Security Review Report
NM-0580 Hanji Derivatives Engine



NETHERMIND
SECURITY

(November 3, 2025)

Contents

1 Executive Summary **2**

2 Audited Files **3**

3 Summary of Issues **3**

4 System Overview **4**
4.1 Contract Interaction Diagram 4

5 Risk Rating Methodology **5**

6 Issues **6**
6.1 [Low] The `minUsdValue` check in `removeLiquidity` is misleading as it does not account for fees 6
6.2 [Info] Deactivating a LOB prevents claiming of outstanding orders 7
6.3 [Info] Missing input validation for `priceConfig` parameters in `setConfig` function 8
6.4 [Info] Missing validation for liquidity thresholds in `setConfig` 8
6.5 [Info] `cooldownDuration` can be cached to save an SLOAD 9
6.6 [Best Practices] Functions `mint` and `burn` can be declared `external` 9
6.7 [Best Practices] Inconsistent early return pattern in state-setting functions 10
6.8 [Best Practices] Inconsistent type casting for `LPManager` contract 11
6.9 [Best Practices] Misleading NatSpec for `withdrawFromYieldVault` function 11
6.10 [Best Practices] Missing NatSpec documentation for `updateAndGetEpochPrice` function 11

7 Documentation Evaluation **12**

8 Test Suite Evaluation **13**
8.1 Compilation Output 13
8.2 Tests Output 13

9 About Nethermind **14**

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for Hanji's **derivatives engines** contracts. The protocol provides a liquidity management system where LPs deposit assets into a central vault to earn yield, while market makers facilitate trading through integrated on-chain limit order books. The system supports integration with ERC-4626 compliant yield vaults, implements protective mechanisms including market maker slashing and LP token price validation to safeguard against unprofitable or malicious market making, and features a dynamic fee structure that adjusts based on token weight imbalances to maintain portfolio balance.

The audit comprises of 2662 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.

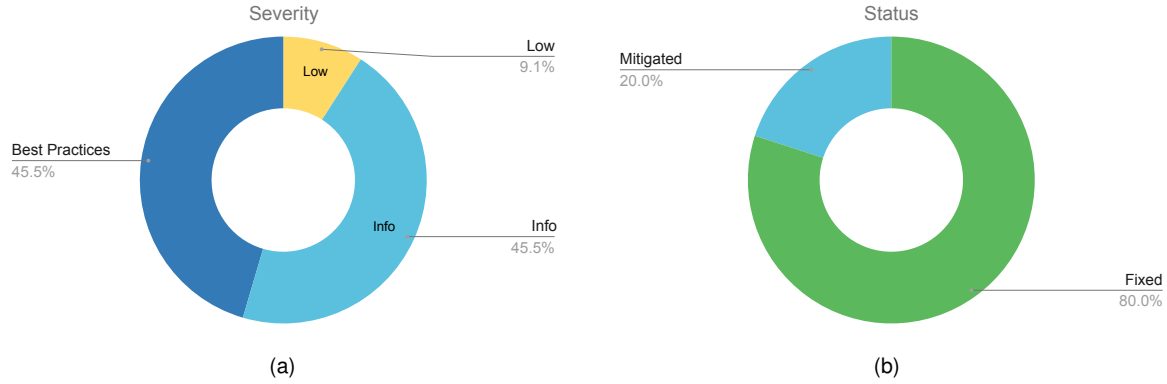


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (4), **Best Practices** (5). **Distribution of status: Fixed** (8), **Acknowledged** (0), **Mitigated** (2), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	August 28, 2025
Final Report	November 3, 2025
Initial commit	276b77ccd47770226c420bce2605813ddc08f2be
Final commit	e803ad3bc7c285bf7d406ca5189f602cefa5f84a
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/LPManagerFactory.sol	61	2	3.3%	19	82
2	src/ProxyPyth.sol	181	41	22.7%	45	267
3	src/ProxyLOBBatch.sol	57	2	3.5%	16	75
4	src/ErrorReporter.sol	57	56	98.2%	55	168
5	src/LPManager.sol	1248	352	28.2%	274	1874
6	src/Proxy.sol	5	2	40.0%	4	11
7	src/LPToken.sol	48	2	4.2%	17	67
8	src/LPManagerHelper.sol	118	21	17.8%	29	168
9	src/AggreVaultOracle.sol	54	2	3.7%	24	80
10	src/ProxyLOB.sol	214	16	7.5%	49	279
11	src/RfqProxyLOB.sol	194	10	5.2%	35	239
12	src/utils/Ecdsa.sol	53	27	50.9%	7	87
13	src/utils/PythPriceHelper.sol	26	2	7.7%	8	36
14	src/utils/TokenValueCalculator.sol	39	16	41.0%	8	63
15	src/utils/RfqOrderLib.sol	46	4	8.7%	10	60
16	src/interfaces/ILPManager.sol	150	31	20.7%	20	201
17	src/interfaces/IProxyLOB.sol	16	2	12.5%	2	20
18	src/interfaces/IOnchainTrie.sol	4	2	50.0%	2	8
19	src/interfaces/IWGAS.sol	5	2	40.0%	2	9
20	src/interfaces/IOnchainCLOB.sol	86	2	2.3%	11	99
	Total	**2662**	**594**	**22.3%**	**637**	**3893**

3 Summary of Issues

	Finding	Severity	Update
1	The `minUsdValue` check in `removeLiquidity` is misleading as it does not account for fees	Low	Fixed
2	Deactivating a LOB prevents claiming of outstanding orders	Info	Fixed
3	Missing input validation for `priceConfig` parameters in `setConfig` function	Info	Acknowledged
4	Missing validation for liquidity thresholds in `setConfig`	Info	Fixed
5	`cooldownDuration` can be cached to save an SLOAD	Info	Fixed
6	Functions `mint` and `burn` can be declared `external`	Best Practices	Acknowledged
7	Inconsistent early return pattern in state-setting functions	Best Practices	Fixed
8	Inconsistent type casting for `LPManager` contract	Best Practices	Fixed
9	Misleading NatSpec for `withdrawFromYieldVault` function	Best Practices	Fixed
10	Missing NatSpec documentation for `updateAndGetEpochPrice` function	Best Practices	Fixed

4 System Overview

The derivatives engine protocol is comprised of several core contracts that facilitate a managed liquidity and trading environment: the `LPManager` as the core of the protocol, various `Proxy` contracts for external interactions, and various helper contracts.

LPManager: The central hub contract managing liquidity provision and asset allocation across multiple tokens. Users interact with this contract to add or remove liquidity, receiving `LPToken` shares proportional to their contribution. The contract implements sophisticated fee structures including dynamic fees based on token weight imbalances, performance fees tied to high-water marks, and cooldown periods for liquidity operations. Additionally, it supports flash loan functionality and integrations with ERC-4626 compliant vaults. The contract enforces strict weight bounds for each token to maintain portfolio balance and includes emergency mechanisms such as market maker share slashing and partial pausing capabilities.

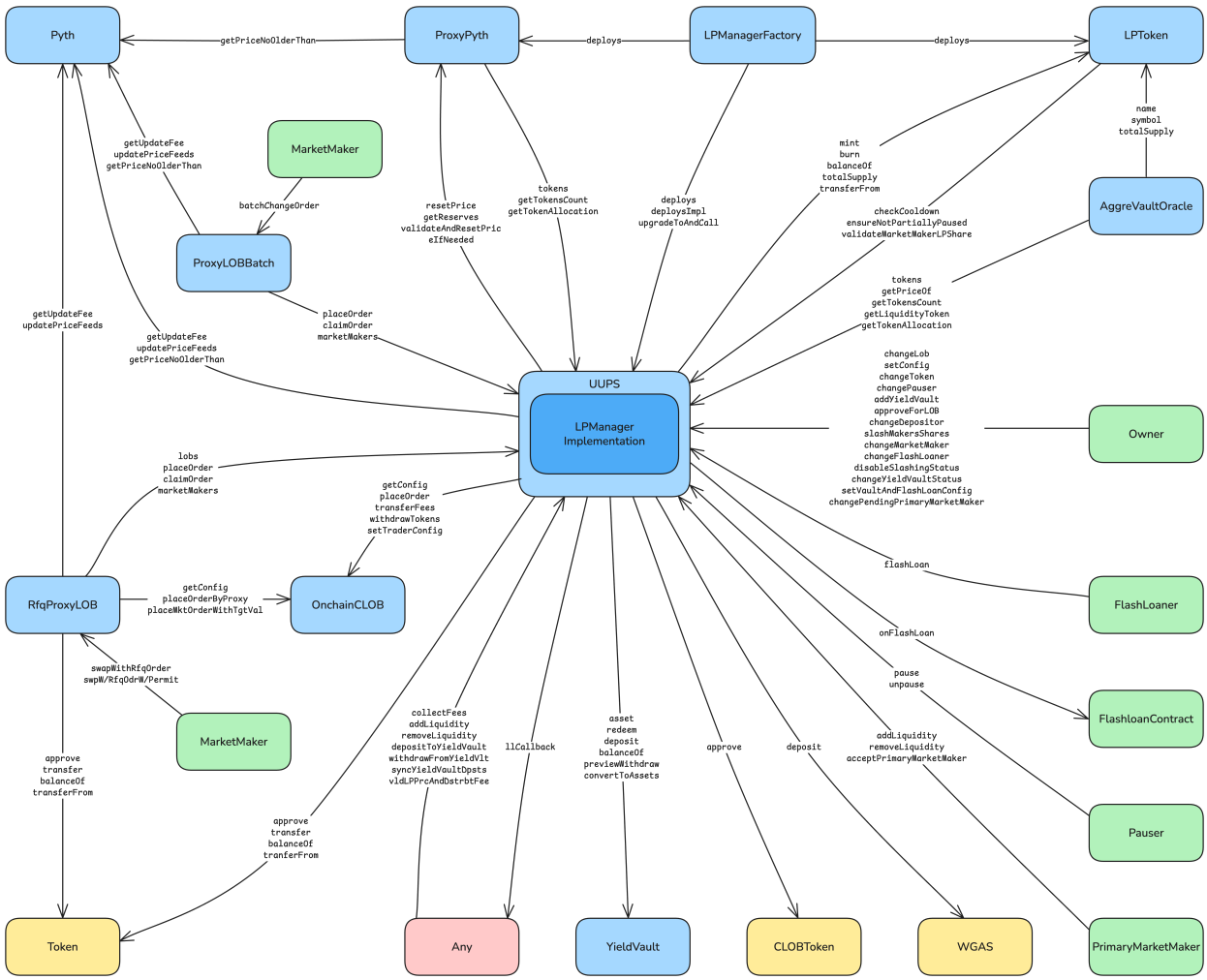
ProxyLOB: An abstract contract that bridges the `LPManager` with external CLOB (Central Limit Order Book) contracts. This proxy enables the liquidity pool to act as a market maker by placing limit orders across multiple trading pairs. It maintains internal accounting of token reserves locked in limit orders, validates order prices against oracle feeds within configurable tolerance bands, and automatically synchronizes token balances after order operations.

ProxyPyth: A specialized oracle management contract that interfaces with Pyth Network price feeds. It implements epoch-based price tracking with minimum and maximum price storage within time windows, and validates LP token price movements to detect excessive deviations.

Supporting Contracts: The protocol includes several auxiliary contracts: `LPToken` implements ERC-20 functionality with integrated cooldown validation and market maker share enforcement. `AggreVaultOracle` provides external price feeds for LP tokens for integration with external protocols. `RfqProxyLOB` enables request-for-quote functionality with EIP-712 signature validation for authenticated off-chain order matching and `LPManagerFactory` deploys new liquidity pool instances with proper initialization.

Centralization Risks: The protocol's design includes significant points of centralization. The `LPManager` is controlled by a single owner with powerful administrative privileges, including the ability to pause the system, change critical parameters, and manage the allowlist for key roles like market makers and depositors. A compromised or malicious owner could therefore exert considerable control over the protocol's operations and assets.

4.1 Contract Interaction Diagram



5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Low] The `minUsdValue` check in `removeLiquidity` is misleading as it does not account for fees

File(s): src/LPManager.sol

Description: The `removeLiquidity(...)` function is intended to protect users from slippage by using the `minUsdValue` parameter. The NatSpec comment explicitly states this parameter is "The minimum USD value the user expects to receive". However, the implementation checks this condition *before* protocol and admin fees are deducted. This means a user can receive a net value in tokens that is less than the `minUsdValue` they specified.

The check is performed on the gross `lpUSDValue`, but the final `tokenAmount` sent to the user is reduced by both `protocolFee` and `adminTokenAmount`.

```
// ...
uint256 lpUSDValue = (totalUSDValue * burnLP) / lpTotalSupply;
// @audit This check is performed against the pre-fee value.
require(
    lpUSDValue >= minUsdValue,
    ErrorReporter.InsufficientUSDValue()
);

// ...
// Multiple fees are calculated based on the initial tokenAmount.
uint256 protocolFee = tokenAmount.mulDivUp(
    protocolFeeBps,
    BASIS_POINTS_DIVISOR
);
// ...
uint256 adminTokenAmount = tokenAmount.mulDivUp(
    adminBurnLPFeeBps,
    BASIS_POINTS_DIVISOR
);
// ...

// @audit The user receives an amount reduced by fees, potentially worth less than minUsdValue.
tokenAmount -= (adminTokenAmount + protocolFee);
token.safeTransfer(msg.sender, tokenAmount);
// ...
```

This behavior is counterintuitive and makes the slippage protection less effective than implied. A user might set a minimum acceptable value but still have their transaction processed for a lower net value due to fees, which contradicts the purpose of a minimum output parameter.

Recommendation(s): Consider performing the `minUsdValue` check on the final net amount of tokens the user will receive after all fees are accounted for. Alternatively, update the NatSpec to reflect the current code.

Status: Mitigated

Update from the client: We have updated the documentation for the `removeLiquidity` function to provide clearer guidance to end users regarding the purpose and behavior of the `minUsdValue` parameter.

The primary focus of `minUsdValue` is to protect users against significant price movements during liquidity removal operations. As such, this parameter validates the USD value of the tokens before any fees are applied, ensuring the gross value meets the user's expectations relative to market conditions.

For comprehensive protection that includes fee considerations, we direct users to the `minTokenGet` parameter, which specifically enforces the minimum net token amount the user will receive after all protocol and admin fees have been deducted.

This dual-parameter approach provides users with flexibility:

- `minUsdValue` for price movement protection
- `minTokenGet` for fee-inclusive amount protection

The updated documentation now accurately reflects this intended behavior and helps users make informed decisions when configuring their liquidity removal parameters.

6.2 [Info] Deactivating a LOB prevents claiming of outstanding orders

File(s): src/LPManager.sol

Description: The `changeLob(...)` function allows the owner to deactivate a Limit Order Book (LOB) by setting its `isActive` status to `false`. However, the `claimOrder(...)` function, which is used to claim or cancel orders, can only be called on active LOBs due to the check in `_checkLOB(...)`.

If an owner deactivates a LOB that still has open orders placed by the `LPManager`, the assets backing those orders will become temporarily inaccessible. The `claimOrder(...)` function will revert for that `lobId`, preventing the recovery of the funds.

```
function changeLob(
    address lobAddress,
    bool isActive,
    // ...
) external onlyOwner {
    // ...
    } else {
        // @audit When a LOB is deactivated (!isActive), there is no process
        // @audit to claim or cancel any existing open orders.
        if (!isActive) {
            // When deactivating the pair, reset LOB contract allowances to zero
            _callApprove(
                tokens[lobs[i].tokenIdX].tokenAddress,
                lobAddress,
                0
            );
            _callApprove(
                tokens[lobs[i].tokenIdY].tokenAddress,
                lobAddress,
                0
            );
        }
    }
    // ...
}

function _checkLOB(uint8 lobId) internal view override {
    require(lobId < lobs.length, ErrorReporter.UnknownLob());
    // @audit This check prevents `claimOrder` from being called on a deactivated LOB.
    require(lobs[lobId].isActive, ErrorReporter.LobDisabled());
}
```

While the owner can resolve this by reactivating the LOB, claiming the orders, and then deactivating it again, this process is cumbersome and introduces operational risk. An owner might forget to clear outstanding orders before deactivation, leading to temporarily locked funds.

Recommendation(s): Consider adding logic to the `changeLob(...)` function to automatically cancel all outstanding `LPManager` orders on a LOB when it is being deactivated. Alternatively, add a strong warning in the NatSpec documentation for the function, advising the owner to manually claim all orders before setting a LOB's status to inactive.

Status: Mitigated

Update from the client: The `@param isActive` description now clearly warns about the consequences of deactivation and the recovery procedure.

6.3 [Info] Missing input validation for `priceConfig` parameters in `setConfig` function

File(s): `src/LPManager.sol`

Description: The `setConfig(...)` function in the `LPManager` contract allows the owner to set critical system parameters, including `baseMultiplier` and `maxAllowedPriceDeviation` from the `PriceConfig` struct. These parameters are used in the function `ProxyPyth.validateAndResetPriceIfNeeded(...)` to validate LP token price movements.

Currently, the `setConfig(...)` function does not perform any validation on these two values.

```
// ...
// slot 12
address _adminFeeRecipient = feeConfig.adminFeeRecipient;
_checkForZeroAddress(_adminFeeRecipient);

uint24 _priceValidityPeriod = priceConfig.priceValidityPeriod;
// @audit The baseMultiplier parameter is not validated.
uint64 _baseMultiplier = priceConfig.baseMultiplier;

// ...
adminFeeRecipient = _adminFeeRecipient;
priceValidityPeriod = _priceValidityPeriod;
baseMultiplier = _baseMultiplier;

// slot 13
// @audit The maxAllowedPriceDeviation parameter is not validated.
uint64 _maxAllowedPriceDeviation = priceConfig.maxAllowedPriceDeviation;
// ...
```

While this function is owner-restricted, a misconfiguration—such as setting `baseMultiplier` to a value greater than the WAD value of `1e18` or setting either parameter to zero—could compromise the price validation logic.

Recommendation(s): Consider adding input validation for the `baseMultiplier` and `maxAllowedPriceDeviation` parameters within the `setConfig(...)` function.

Status: Fixed

Update from the client: Fixed in commit `651366ac`. Added validation ensuring:

- `baseMultiplier` is restricted between 50% and 100%
- `maxAllowedPriceDeviation` cannot exceed maximum daily volatility (100% / 86400)

6.4 [Info] Missing validation for liquidity thresholds in `setConfig`

File(s): `src/LPManager.sol`

Description: The `setConfig(...)` function in the `LPManager` contract is responsible for setting various configuration parameters, including `minLiquidityValueUsd` and `maxLiquidityValueUsd`. These variables define the lower and upper bounds for the total USD value of liquidity in the pool.

The function currently lacks a validation check to ensure that `minLiquidityValueUsd` is less than or equal to `maxLiquidityValueUsd`.

```
// ...
// @audit The minimum liquidity threshold is set here.
minLiquidityValueUsd = _minLiquidityValueUsd;

// slot 11
// @audit The maximum liquidity threshold is set here.
uint128 _maxLiquidityValueUsd = liquidityConfig.maxLiquidityValueUsd;
// @audit A check is missing to ensure _minLiquidityValueUsd <= _maxLiquidityValueUsd.

maxLiquidityValueUsd = _maxLiquidityValueUsd;
// ...
```

Recommendation(s): Consider adding a `require` statement in the `setConfig(...)` function to validate that `minLiquidityValueUsd` is less than or equal to `maxLiquidityValueUsd`.

Status: Fixed

Update from the client: Fixed in commit `90f6cc55`.

6.5 [Info] `cooldownDuration` can be cached to save an SLOAD

File(s): `src/LPManager.sol`

Description: The `checkCooldown(...)` function in the `LPManager` contract reads the `cooldownDuration` state variable from storage twice if its value is not zero. Each read from storage consumes gas for an `SLOAD` operation.

```
    function checkCooldown(address account) public view {
        // @audit First SLOAD of `cooldownDuration`.
        if (cooldownDuration == 0) {
            return;
        }
        // @audit Second SLOAD of `cooldownDuration` occurs here.
        require(
            block.timestamp - lastAddedAt[account] >= cooldownDuration,
            ErrorReporter.CooldownDurationNotYetPassed()
        );
    }
```

This double read is inefficient. The value of `cooldownDuration` can be read from storage once and stored in a local memory variable.

Recommendation(s): Consider caching the `cooldownDuration` state variable in a local memory variable at the beginning of the `checkCooldown(...)` function.

Status: Fixed

Update from the client: Fixed in commit `094a80ef`.

6.6 [Best Practices] Functions `mint` and `burn` can be declared `external`

File(s): `src/LPToken.sol`

Description: The `mint(...)` and `burn(...)` functions in the `LPToken` contract are declared with `public` visibility. However, they are only designed to be called externally by the `owner` of the contract (the `LPManager`). These functions are never called internally from within the `LPToken` contract itself.

```
// ...

    function mint(address account, uint256 value) onlyOwner public {
        _mint(account, value);
        emit TokensMinted(account, value);
    }

    function burn(address account, uint256 value) onlyOwner public {
        _burn(account, value);
        emit TokensBurned(account, value);
    }
// ...
```

For functions that are only ever called from outside the contract, using `external` visibility is a best practice.

Recommendation(s): Consider changing the visibility of the `mint(...)` and `burn(...)` functions from `public` to `external`.

Status: Fixed

Update from the client: Fixed in commit `183dc5af`.

6.7 [Best Practices] Inconsistent early return pattern in state-setting functions

File(s): src/LPManager.sol

Description: There is an inconsistent pattern in how permissioned, state-setting functions handle redundant calls. Some functions, like `changePendingPrimaryMarketMaker(...)` and `acceptPrimaryMarketMaker(...)`, check if the new state is the same as the current state and return early to avoid unnecessary operations. However, other functions, namely `changeMarketMaker(...)` and `changeDepositor(...)`, do not have this check.

These functions will perform a state write and emit an event even if the new value is identical to the one already in storage.

```
    // @audit This function writes to storage even if marketMakers[marketMaker] already equals isActive.
    function changeMarketMaker(address marketMaker, bool isActive) external onlyOwner {
        marketMakers[marketMaker] = isActive;

        emit MarketMakerChanged(marketMaker, isActive);
    }

    // @audit This function writes to storage even if depositors[depositor] already equals isActive.
    function changeDepositor(address depositor, bool isActive) external onlyOwner {
        depositors[depositor] = isActive;

        emit DepositorChanged(depositor, isActive);
    }
```

In contrast, other functions in the same contract do implement this check:

```
    function changePendingPrimaryMarketMaker(
        address _primaryMarketMaker
    ) external onlyOwner {
        address currentPendingPrimaryMarketMaker = pendingPrimaryMarketMaker;
        // @audit This function correctly returns early if there is no state change.
        if (currentPendingPrimaryMarketMaker == _primaryMarketMaker) {
            return;
        }
    // ...
    }
```

This inconsistency makes the codebase harder to maintain and can lead to unnecessary gas costs from redundant `SSTORE` operations and event emissions.

Recommendation(s): Consider applying a consistent pattern across all state-setting functions.

Status: Fixed

Update from the client: Fixed in commit `7e736322`.

6.8 [Best Practices] Inconsistent type casting for `LPManager` contract

File(s): `src/LPManager.sol`

Description: In various parts of the codebase, instances of the `LPManager` contract are inconsistently cast. Sometimes they are treated as the `LPManager` type, while other times they are cast to the parent `ProxyLOB` type. While this pattern does not introduce a direct security vulnerability, it leads to an inconsistent and confusing coding style.

Recommendation(s): Consider consistently using the `LPManager` type for all variables and interactions involving instances of the `LPManager` contract. Since `LPManager` inherits from `ProxyLOB`, it already provides access to all of its parent's public and external functions.

Status: Fixed

Update from the client: Fixed in commit `045ea2f3`. We've resolved the inheritance by making `ILPManager` inherit from `IProxyLOB`. This eliminates the redundant interface and removes all `IProxyLOB` casts across integrations, ensuring consistent typing with `ILPManager` throughout the codebase.

6.9 [Best Practices] Misleading NatSpec for `withdrawFromYieldVault` function

File(s): `src/LPManager.sol`

Description: The NatSpec documentation for the `withdrawFromYieldVault(...)` function in the `LPManager` contract is inaccurate. Specifically, the comment for the `assets` parameter states that it is an "Amount of assets to withdraw directly," which does not reflect the function's actual implementation.

The function's logic takes the `assets` amount, converts it into a corresponding number of vault shares using `yieldVault.previewWithdraw(assets)`, and then adds these shares to the `shares` parameter. The final withdrawal from the vault is based on this combined total of shares.

```
    /// @notice Withdraws assets from a yield vault.
    /// @param yieldVaultId Identifier of the target vault.
    /// @param assets Amount of assets to withdraw directly.
    /// @param shares Amount of shares to redeem.
    /// @param expires Timestamp when operation expires.
    function withdrawFromYieldVault(
        uint8 yieldVaultId,
        uint256 assets,
        uint256 shares,
        uint256 expires
    ) external /* ... */ {
        // ...
        IERC4626 yieldVault = IERC4626(yieldVaultAddress);

        uint256 totalShares = shares;
        if (assets > 0) {
            // @audit The 'assets' amount is converted to shares before withdrawal.
            totalShares += yieldVault.previewWithdraw(assets);
        }

        uint256 redeemedAssets = _callRedeem(yieldVault, totalShares);
        // ...
    }
```

Recommendation(s): Consider updating the NatSpec comment for the `assets` parameter in the `withdrawFromYieldVault(...)` function.

Status: Fixed

Update from the client: Fixed in commit `7f3511c1`.

6.10 [Best Practices] Missing NatSpec documentation for `updateAndGetEpochPrice` function

File(s): `src/ProxyPyth.sol`

Description: The `updateAndGetEpochPrice(...)` function has `@notice`, `@dev`, and `@return` comments but it is missing descriptions for its input parameters.

Recommendation(s): Consider adding comprehensive NatSpec `@param` documentation.

Status: Fixed

Update from the client: Fixed in commit `ff06e0ee`. Added documentation for `maxOracleAge` and `oracleConfRel` input parameters.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Hanji's documentation

Although the `Hanji` team did not provide any written documentations, the team provided an overview of the core system components during our meetings, presenting a clear explanation of the intended functionalities and addressing questions raised by the Nethermind Security team. These sessions offered valuable insights and ensured a comprehensive understanding of the project's technical foundations.

8 Test Suite Evaluation

8.1 Compilation Output

```
user@machine derivatives-engine % forge build
[.] Compiling...
[.] Compiling 138 files with Solc 0.8.30
[.] Solc 0.8.30 finished in 40.20s
```

8.2 Tests Output

```
Ran 1 test for test/ErrorReporterTest.t.sol:ErrorReporterTest
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 322.38µs (161.38µs CPU time)
Ran 3 tests for test/PerformanceFee.t.sol:PerformanceFee
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 1.71ms (228.04µs CPU time)
Ran 4 tests for test/ProxyPyth.t.sol:ProxyPythTest
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.69ms (2.53ms CPU time)
Ran 7 tests for test/ERC3156FlashLender.t.sol:ERC3156FlashLender
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 1.81ms (1.70ms CPU time)
Ran 8 tests for test/YieldManager.t.sol:YieldManagerTest
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 1.76ms (1.03ms CPU time)
Ran 2 tests for test/PriceArbitrage.t.sol:PriceArbitrage
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.67ms (1.01ms CPU time)
Ran 4 tests for test/ProxyLOBBatch.t.sol:ProxyLOBBatchTest
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 2.50ms (1.15ms CPU time)
Ran 2 tests for test/CompromisedMarketMaker.t.sol:CompromisedMarketMaker
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 4.64ms (2.91ms CPU time)
Ran 6 tests for test/Reentrancy.t.sol:ReentrancyTest
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 4.95ms (3.33ms CPU time)
Ran 5 tests for test/Pausable.t.sol:PausableTest
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 5.28ms (577.46µs CPU time)
Ran 32 tests for test/RfqProxyLOB.t.sol:RfqProxyLOBTest
Suite result: ok. 32 passed; 0 failed; 0 skipped; finished in 5.02ms (12.95ms CPU time)
Ran 14 tests for test/PythOracle.t.sol:PythOracleTest
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 5.16ms (8.78ms CPU time)
Ran 6 tests for test/Upgradeable.t.sol:UpgradeableTest
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 5.23ms (1.81ms CPU time)
Ran 34 tests for test/LPManager.t.sol:LPManagerTest
Suite result: ok. 34 passed; 0 failed; 0 skipped; finished in 5.26ms (15.29ms CPU time)

Ran 14 test suites in 83.75ms (48.00ms CPU time): 128 tests passed, 0 failed, 0 skipped (128 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our cryptography Research team conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.