Security Review Report
NM-0662 - Hyperbeat Vault

NETHERMIND
SECURITY

(October 1, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for Hyperbeat's Vault contracts. The protocol is an asset management vault designed to execute trading strategies on HyperCore. Users can deposit assets into the vault and receive VaultTokens, which represent their share of the total assets under management.

The audit comprises 1223 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools, and (c) simulation and creation of test cases.

Along this document, we report 10 points of attention, where two are classified as `Medium`, five are classified as `Low`, and three are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

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.

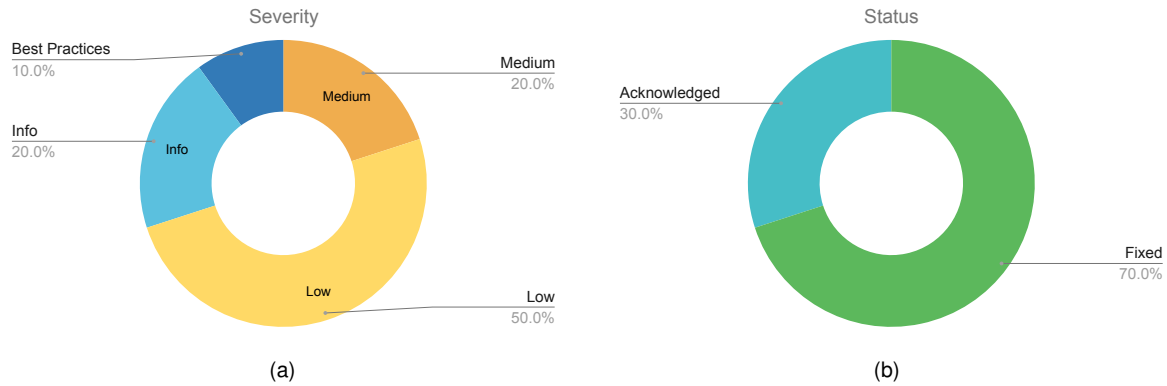

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (5), **Undetermined** (0), **Informational** (2), **Best Practices** (1). **Distribution of status: Fixed** (7), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	September 19, 2025
Final Report	October 1, 2025
Initial Commit	9c72d9772f469ed683b0ad53ba806a857838ed72
Final Commit	5643798e0267bb90ac81e3c9e4ee0828cd62981b
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/OracleAggregator.sol	40	1	2.5%	8	49
2	src/interfaces/IAggregatorV3.sol	22	1	4.5%	3	26
3	src/interfaces/IMidasPriceFeed.sol	19	1	5.3%	2	22
4	src/interfaces/ICoreWriter.sol	4	1	25.0%	1	6
5	src/vault/ExchangeRateUpdater.sol	295	79	26.8%	39	413
6	src/vault/DnCoreWriterVault.sol	211	114	54.0%	44	369
7	src/vault/DepositReceiver.sol	61	38	62.3%	14	113
8	src/vault/Pricer.sol	309	87	28.2%	35	431
9	src/vault/VaultToken.sol	41	13	31.7%	9	63
10	src/vault/WithdrawalQueue.sol	157	41	26.1%	22	220
11	src/vault/Depositor.sol	59	27	45.8%	13	99
12	src/vault/interfaces/IPriceProvider.sol	5	5	100.0%	1	11
	Total	**1223**	**408**	**33.4%**	**191**	**1822**

3 Summary of Issues

	Finding	Severity	Update
1	Tokens sent to `WithdrawalQueue` can be permanently stuck making the vault insolvent	Medium	Fixed
2	`totalSupply(...)` on a single chain is used for valuation	Medium	Fixed
3	Exchange rate can be incorrectly calculated due to HyperEVM-HyperCore communication timings	Low	Acknowledged
4	Instant withdrawal fees can be bypassed through rounding exploitation	Low	Fixed
5	Potential underflow in `getPrice()` can lead to Denial of Service	Low	Fixed
6	`recoverERC20(...)` function can be blocked	Low	Acknowledged
7	`rejectWithdrawalRequest(...)` sends tokens to the recipient instead of the original depositor	Low	Fixed
8	Lack of sanity checks when modifying `dnCoreWriters` and `assetConfigs`	Info	Acknowledged
9	Users have no control over the exchange rate for their withdrawal requests	Info	Fixed
10	`createWithdrawalRequest` does not validate for zero-amount withdrawals	Best Practices	Fixed

4 Protocol Overview

The system under review is an asset management vault designed to generate yield by actively managing user-deposited assets. Users deposit ERC20 tokens and receive `VaultToken`s, which represent their proportional share of the vault's total assets. The system deploys these assets into multiple accounts in HyperCore, via the `DnCoreWriterVault` contracts. The value of the `VaultToken`s fluctuates based on the performance of the vault's portfolio.

4.1 Core Components

The protocol is composed of several key smart contracts, each with a specialized role:

- **`VaultToken.sol`**: An ERC20-compliant omnichain fungible token (OFT) that represents a share in the vault. Its supply is controlled by designated `minter` and `burner` contracts.
- **`Depositor.sol`**: The public-facing contract for handling user deposits. It calculates the corresponding `VaultToken` amount based on the current exchange rate and mints them for the user.
- **`DepositReceiver.sol`**: A central holding contract for idle assets that have been deposited but not yet deployed for trading. It acts as a dispatcher of funds to the trading vault, to users for instant withdrawals, and to the fee recipient.
- **`DnCoreWriterVault.sol`**: The vault's operational "hot wallet". It holds the assets actively used for trading and exposes functions for privileged roles to interact with the external HyperCore system.
- **`WithdrawalQueue.sol`**: Manages the entire withdrawal process. It supports two modes: a standard, asynchronous queued withdrawal and an instant withdrawal option that incurs a fee.
- **`Pricer.sol`**: Serves as the authoritative source for the vault's Net Asset Value (NAV), represented as an exchange rate. It is responsible for calculating management and performance fees and is consulted during all deposits and withdrawals.
- **`ExchangeRateUpdater.sol`**: A dedicated contract responsible for calculating the total value of assets under management (AUM). It aggregates asset values from on-chain balances and off-chain data from HyperCore, then updates the NAV in the `Pricer` contract.
- **`OracleAggregator.sol`**: A utility contract used to derive asset prices by combining data from one or more Chainlink price feeds.

4.2 Actors & Privileged Roles

The system operates with a set of well-defined actors and privileged roles that govern its operation and administration:

- **User**: An external actor who can deposit assets, create withdrawal requests, and transfer `VaultToken`s.
- **Admin (`ADMIN_ROLE`)**: The most privileged role, capable of configuring critical parameters across the entire system, such as setting fees, pausing contracts, managing supported assets, and assigning other roles.
- **Keeper (`KEEPER_ROLE`)**: An automated off-chain entity expected to periodically call `updateExchangeRate()` to keep the vault's NAV current with market conditions.
- **Solver (`SOLVER_ROLE`)**: A privileged actor responsible for processing queued withdrawal requests, ensuring users receive their assets.
- **Allocator (`ALLOCATOR_ROLE`)**: Authorized to move funds from the idle pool in the `DepositReceiver` to the `DnCoreWriterVault` for active deployment in trading strategies.
- **Trading & Bridge Roles**: Specialized roles (`TRADING_AGENT_ROLE`, `BRIDGE_ROLE`) that can execute trading and bridging operations on HyperCore through the `DnCoreWriterVault`.

4.3 Key Workflows

4.3.1 Deposit Flow

a. A user calls `deposit(...)` on the `Depositor` contract.

b. The `Depositor` contract transfers the user's tokens to the `DepositReceiver`.

c. It queries the `Pricer` to determine the amount of `VaultToken`s to issue based on the current exchange rate.

d. The `Depositor` (acting as the designated `minter`) calls `mint(...)` on the `VaultToken` contract to issue shares to the user.

4.3.2 Net Asset Value (NAV) Update Flow

a. A Keeper with the `KEEPER_ROLE` calls `updateExchangeRate()` on the `ExchangeRateUpdater`.

b. The `ExchangeRateUpdater` calculates the total AUM by aggregating idle on-chain asset balances and reading account data from HyperCore. It uses `OracleAggregator` contracts to price all assets.

c. It then calls `updateExchangeRate(...)` on the `Pricer`. The `Pricer` validates the new rate, calculates management and performance fees, adjusts the rate accordingly, and stores the new official NAV.

4.3.3 Withdrawal Flow

The system provides two distinct paths for users to withdraw their funds.

Queued Withdrawal:

a. A user calls `createWithdrawalRequest(...)` on the `WithdrawalQueue`, transferring their `VaultToken`s to the contract.

b. The request is added to a queue, capturing the exchange rate at that moment.

c. A Solver with the `SOLVER_ROLE` later calls `processWithdrawalRequests(...)`.

d. The contract transfers the appropriate amount of `baseAsset` to the user and calls `burn(...)` on the `VaultToken` contract to burn the user's shares. It uses the least favorable exchange rate (current vs. request time) for the user.

Instant Withdrawal:

a. A user calls `instantWithdraw(...)` on the `WithdrawalQueue`.

b. The contract immediately calls `burn(...)` on the `VaultToken` contract, using the user's balance.

c. A withdrawal fee is calculated and subtracted from the withdrawal amount.

d. The contract calls `sendAssetsForInstantWithdrawal(...)` on the `DepositReceiver` to transfer the final asset amount to the user from the idle asset pool.

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 [Medium] Tokens sent to `WithdrawalQueue` can be permanently stuck making the vault insolvent

File(s): src/vault/WithdrawalQueue.sol

Description: The `WithdrawalQueue` contract only handles the `baseAsset` token. Its withdrawal and recovery mechanisms can only transfer the `baseAsset`. There are no functions to manage or transfer any other ERC20 token that might be sent to it.

If any token other than `baseAsset` is sent to the `WithdrawalQueue` contract, those funds will be locked. The protocol's total asset calculation includes the balances held by the `WithdrawalQueue` contract, this would lead to an inflated Net Asset Value (NAV), as it would count funds that are no longer accessible. This would make the protocol insolvent.

Recommendation(s): Consider modifying the `recoverERC20(...)` function to allow the withdrawal of other ERC20 tokens.

Status: Fixed.

Update from the client: Fixed in commit 64a1b2.

Update from Nethermind: Recover actions of `vaultTokens` should be restricted.

Update from the client: Acknowledged.

6.2 [Medium] `totalSupply(...)` on a single chain is used for valuation

File(s): src/vault/VaultToken.sol, src/vault/ExchangeRateUpdater.sol

Description: The `VaultToken` is an omnichain token that inherits from LayerZero's `OFT.sol`. This allows the token to be bridged across different blockchains. The protocol's valuation mechanism, located in the `ExchangeRateUpdater` contract, calculates the value of one `vaultToken` (the exchange rate) using the formula: `ExchangeRate = TotalAssets / TotalSupply`.

An issue arises because `vaultToken.totalSupply()` only returns the number of tokens on the _current_ chain. When an `OFT` is bridged from a source chain to a destination chain, it is burned on the source and minted on the destination. However, the underlying assets that back the `vaultToken` remain on the source chain.

```
// In ExchangeRateUpdater.sol
function updateExchangeRate() external requiresAuth {
    (uint256 totalAssets, uint8 baseAssetDecimals) = _calculateTotalAssets();
    // ...
    // @audit-issue `IERC20Metadata(vaultToken).totalSupply()` provides the token supply
    //              only on the current chain, not the global supply across all chains.
    uint256 newNav = _convertDecimals(
        totalAssets.mulDiv(
            10 ** IERC20Metadata(vaultToken).decimals(),
            IERC20Metadata(vaultToken).totalSupply()
        ),
        baseAssetDecimals,
        Pricer(pricer).decimals()
    );
    Pricer(pricer).updateExchangeRate(newNav.toUint96());
    // ...
}
```

This allows adversaries to manipulate the exchange rate by moving shares cross-chain before an update.

Recommendation: Consider using an `OFTAdapter` pattern instead of making the `VaultToken` `OFT`. This would change to a `lock-mint` mechanism to avoid the burning of shares in the HyperEVM chain.

Status: Fixed.

Update from the client: Fixed in commit cbf347.

6.3 [Low] Exchange rate can be incorrectly calculated due to HyperEVM-HyperCore communication timings

File(s): src/vault/ExchangeRateUpdater.sol

Description: The `ExchangeRateUpdater` contract, which resides on HyperEVM, determines the `vaultToken`'s value by reading asset balances from HyperCore via the `l1Read` contract. Due to the ordering of operations in Hyperliquid's architecture, state changes made on HyperCore may not be readable on HyperEVM until the next block.

If the keeper calls `updateExchangeRate()` in the same block that a large deposit has occurred on HyperCore, the NAV calculation will not account for the tokens sent as these will not be part of the contract's balance or spot balance. If the calculation uses this data that doesn't include a recent large deposit, the exchange rate will be set artificially low, causing subsequent depositors to be minted an excessive number of shares and diluting existing holders.

Recommendation(s): To mitigate this operational risk, consider adding a safeguard to prevent the exchange rate from being updated in the same block after HyperCore deposits.

Status: Acknowledged.

Update from the client:

6.4 [Low] Instant withdrawal fees can be bypassed through rounding exploitation

File(s): src/vault/WithdrawalQueue.sol

Description: The `WithdrawalQueue` contract offers an `instantWithdraw(...)` feature that allows users to bypass the standard withdrawal process for a fee. This fee is calculated using the formula `_amount.mulDiv(instantWithdrawalFee, 10_000)`.

The `mulDiv` implementation from `FixedPointMathLib` rounds the result of the division down.

```
function instantWithdraw(address _user, uint256 _amount) external {
    if (isPaused || isInstantWithdrawalPaused) {
        revert WithdrawalQueue__Paused();
    }
    VaultToken(vaultToken).burn(msg.sender, _amount);
    // @audit-issue `mulDiv` rounds the result down to the nearest integer.
    uint256 fee = _amount.mulDiv(instantWithdrawalFee, 10_000);
    uint256 amountToWithdraw = _amount - fee;
    // ...
}
```

An attacker can bypass the fee mechanism by performing multiple small withdrawals instead of a single large one. By choosing a withdrawal `_amount` that is small enough, the attacker can ensure that the numerator of the division (`_amount * instantWithdrawalFee`) is less than the denominator (`10_000`). In this case, the `mulDiv` function will round the resulting fee down to zero.

Recommendation(s): Consider rounding up when computing the withdrawal fee.

Status: Fixed.

Update from the client: Fixed in commit 64a1b2.

6.5 [Low] Potential underflow in `getPrice()` can lead to Denial of Service

File(s): src/OracleAggregator.sol

Description: The `OracleAggregator` contract is designed to calculate a combined price from two distinct Chainlink price feeds, a `baseFeed` and a `quoteFeed`. The public function `getPrice()` orchestrates this by first calculating a raw price and then adjusting it for the correct decimal precision.

The issue lies in the calculation of the input decimals passed to the `_convertDecimals(...)` function. The calculation is performed as `baseDecimals - quoteDecimals + 18`. If the `quoteFeed` has a higher number of decimals than the `baseFeed` (i.e., `quoteDecimals > baseDecimals`), the initial subtraction `baseDecimals - quoteDecimals` will underflow because both variables are of type `uint8`.

This underflow will cause the `getPrice()` function to revert, leading to a Denial of Service. Any contract or user attempting to fetch the price from this aggregator will be unable to do so, potentially halting critical functionalities that depend on this price feed.

```
function getPrice() public view virtual returns (uint256) {
    // @audit-issue If `quoteDecimals` > `baseDecimals`, this calculation will underflow.
    return _convertDecimals(_getPrice(), baseDecimals - quoteDecimals + 18, decimals);
}
```

Recommendation(s): Consider reordering the arithmetic operations to prevent the possibility of an intermediate underflow.

Status: Fixed.

Update from the client: Fixed in commit 64a1b2.

6.6 [Low] `recoverERC20(...)` function can be blocked

File(s): `src/vault/WithdrawalQueue.sol`

Description: The `WithdrawalQueue` contract has an administrative function, `recoverERC20(...)`, intended to allow a privileged user to recover surplus `baseAsset` from the contract. To execute this function, the caller must provide an array of all `_activeWithdrawals` currently pending in the contract.

The administrator must construct the `_activeWithdrawals` array off-chain based on the current state of the contract. However, any user can call `createWithdrawalRequest(...)` at any time, which modifies the on-chain set of active withdrawals.

```
function recoverERC20(WithdrawalRequest[] memory _activeWithdrawals) external requiresAuth {
    bytes32[] memory activeWithdrawalRequestIds = getActiveWithdrawals();
    uint256 activeWithdrawalsLength = _activeWithdrawals.length;
    // @audit This check fails if the on-chain set changes after the caller
    //        has prepared the transaction.
    if (activeWithdrawalsLength != activeWithdrawalRequestIds.length) {
        revert WithdrawalQueue__InvalidActiveWithdrawals();
    }
    uint256 totalAssets = 0;
    for (uint256 i = 0; i < activeWithdrawalsLength; i++) {
        // @audit-issue This check also fails if a new request is added,
        //              as the contents and order will no longer match.
        if (activeWithdrawalRequestIds[i] != keccak256(abi.encode(_activeWithdrawals[i]))) {
            revert WithdrawalQueue__InvalidActiveWithdrawalRequest();
        }
        WithdrawalRequest memory withdrawalRequest = _activeWithdrawals[i];
        totalAssets += withdrawalRequest.baseAssetAmount;
    }
    baseAsset.safeTransfer(depositReceiver, baseAsset.balanceOf(address(this)) - totalAssets);
}
```

A malicious user could try to front-run calls to `recoverERC20(...)` with `createWithdrawalRequest(...)`. This action will add a new request to the on-chain set, causing the administrator's `recoverERC20(...)` transaction to fail its validation checks and revert.

Recommendation(s): Consider redesigning the `recoverERC20(...)` function to not rely on a user-provided array.

Status: Acknowledged.

Update from the client:

6.7 [Low] `rejectWithdrawalRequest(...)` sends tokens to the recipient instead of the original depositor

File(s): src/vault/WithdrawalQueue.sol

Description: The protocol's withdrawal process allows a user to initiate a withdrawal by calling `createWithdrawalRequest(...)`. The caller (`msg.sender`) provides the `vaultTokens` and specifies a `_user` address, which is the designated recipient for the final `baseAsset` payout once the withdrawal is successfully processed.

A privileged role can cancel a pending request using the `rejectWithdrawalRequest(...)` function. Per the current design, when a request is rejected, the function sends the `vaultTokens` that were locked for withdrawal to the `_user` address.

```
function createWithdrawalRequest(address _user, uint256 _amount)
    external
    returns (WithdrawalRequest memory)
{
    // ...
    // @audit The `_user` is designated to receive the final `baseAsset`.
    WithdrawalRequest memory withdrawalRequest = WithdrawalRequest({
        nonce: withdrawalNonce,
        user: _user,
        amount: _amount,
        //...
    });
    // ...
}
```

The rejection logic sends the `vaultTokens` to this same `_user`.

```
function rejectWithdrawalRequest(WithdrawalRequest memory _withdrawalRequest)
    external
    requiresAuth
{
    // ...
    // @audit-issue `vaultToken`s are sent to `_withdrawalRequest.user`,
    //              which was the intended recipient of `baseAsset`.
    vaultToken.safeTransfer(_withdrawalRequest.user, _withdrawalRequest.amount);
    // ...
}
```

This design is problematic because the `_user` is expecting to receive `baseAsset`, not `vaultToken`. If the designated `_user` is a smart contract, it may not be programmed to accept or manage incoming `vaultTokens`. In such cases, the returned `vaultTokens` could be permanently locked within the recipient contract, leading to a loss of funds.

Furthermore, this behavior deviates from standard and safer design patterns, where a cancelled action reverts the state for the original initiator. The user who provided the capital for the withdrawal (`msg.sender`) should be the one to receive their tokens back if the request is cancelled.

Recommendation(s): Consider modifying the withdrawal logic to ensure that a rejected request returns the `vaultTokens` to the original depositor (`msg.sender` of the `createWithdrawalRequest(...)` call) instead of the intended asset recipient.

Status: Fixed.

Update from the client: Fixed in commit 64a1b2.

6.8 [Info] Lack of sanity checks when modifying `dnCoreWriters` and `assetConfigs`

File(s): src/vault/ExchangeRateUpdater.sol

Description: The `ExchangeRateUpdater` contract calculates the Net Asset Value (NAV) of the vault, which is an essential for determining the `vaultToken`'s exchange rate. This calculation directly depends on the `dnCoreWriters` and `assetConfigs` state arrays, which are part of the data sources for the protocol's assets.

A privileged admin role has the authority to add and remove entries from these arrays using functions like `addDnCoreWriterAddress(...)` and `addAssetConfig(...)`. Currently, these functions do not perform any validation or sanity checks on the addresses being added.

In some cases these added or removed configuration could impact the NAV considerably causing major changes in the `exchangeRate`

Recommendation(s): Consider strengthening the administrative functions by adding on-chain sanity checks.

Status: Acknowledged.

Update from the client:

6.9 [Info] Users have no control over the exchange rate for their withdrawal requests

File(s): src/vault/WithdrawalQueue.sol

Description: Users can request a withdrawal by calling the `createWithdrawalRequest(...)` function. This function transfers the user's `vaultToken`s to the contract and creates a withdrawal request, storing the exchange rate at the time of creation.

```
function createWithdrawalRequest(address _user, uint256 _amount) external returns (WithdrawalRequest memory) {
// ...
    vaultToken.safeTransferFrom(msg.sender, address(this), _amount);
    uint256 currentExchangeRate = Pricer(pricer).getRate();
    uint256 baseAssetAmount = Pricer(pricer).getAssetAmount(baseAsset, _amount);
    WithdrawalRequest memory withdrawalRequest = WithdrawalRequest({
// ...
// @audit The exchange rate at the time of request creation is stored.
        exchangeRate: uint128(currentExchangeRate),
        baseAssetAmount: baseAssetAmount
    });
// ...
}
```

These requests are processed at a later time by a privileged solver role, who calls the `processWithdrawalRequests(...)` function. This function checks if the current exchange rate is lower than the rate stored in the user's request. If the rate has dropped, the user receives a smaller amount of the base asset, calculated with the new, less favorable rate.

```
function processWithdrawalRequests(WithdrawalRequest[] memory _withdrawalRequestsToProcess) external requiresAuth {
// ...
    uint256 currentExchangeRate = Pricer(pricer).getRate();
    for (uint256 i = 0; i < requestsLength; i++) {
        WithdrawalRequest memory withdrawalRequest = _withdrawalRequestsToProcess[i];
// ...
        uint256 amountToSend = withdrawalRequest.baseAssetAmount;
// @audit-issue If the current rate is worse (lower), the user receives fewer tokens.
        if (withdrawalRequest.exchangeRate > currentExchangeRate) {
            amountToSend = Pricer(pricer).getAssetAmount(baseAsset, withdrawalRequest.amount);
            exchangeRateApplied = currentExchangeRate;
        }
        // ...

        baseAsset.safeTransfer(withdrawalRequest.user, amountToSend);
// ...
    }
}
```

The issue is that users have no control over when their withdrawal is processed and cannot set a minimum acceptable exchange rate (i.e., slippage protection). This exposes them to the risk of their withdrawal being executed at a significantly worse rate than when they initiated it, leading to a poor user experience and potential financial loss.

Recommendation(s): Consider implementing a mechanism that gives users more control over their withdrawals. This could involve allowing users to specify a minimum output amount when they create a request. If the amount of base assets to be received falls below this user-defined threshold during processing, the transaction should revert or be handled differently.

Status: Fixed.

Update from the client: Fixed in commit 64a1b2.

Update from Nethermind:

- Previously, `exchangeRateApplied` was set to `currentExchangeRate` if `withdrawalRequest.exchangeRate > currentExchangeRate` whereas now that line is removed. So the event will emit incorrect exchange rate in this case;
- `minAmountOut` is not checked if the `exchangeRate` used is the one at the time of the `withdrawalRequest` creation;

6.10 [Best Practice] `createWithdrawalRequest` does not validate for zero-amount withdrawals

File(s): src/vault/WithdrawalQueue.sol

Description: The `createWithdrawalRequest(...)` function in the `WithdrawalQueue` contract is the entry point for users to initiate a withdrawal. It takes an `_amount` of `vaultToken`s to be withdrawn.

The function lacks input validation to ensure that the `_amount` provided is greater than zero. As a result, it is possible for a user to create a withdrawal request for `0` tokens.

```
function createWithdrawalRequest(address _user, uint256 _amount)
    external
    returns (WithdrawalRequest memory)
{
    if (isPaused) {
        revert WithdrawalQueue__Paused();
    }
    // @audit-issue The `_amount` parameter is not validated to be greater than zero.
    vaultToken.safeTransferFrom(msg.sender, address(this), _amount);
    uint256 currentExchangeRate = Pricer(pricer).getRate();
    uint256 baseAssetAmount = Pricer(pricer).getAssetAmount(baseAsset, _amount);
    WithdrawalRequest memory withdrawalRequest = WithdrawalRequest({
        nonce: withdrawalNonce,
        user: _user,
        amount: _amount,
        createdAt: uint64(block.timestamp),
        exchangeRate: uint128(currentExchangeRate),
        baseAssetAmount: baseAssetAmount
    });
    // @audit This adds a functionally useless request to the `_withdrawalRequests` set.
    bytes32 withdrawalRequestId = keccak256(abi.encode(withdrawalRequest));
    bool success = _withdrawalRequests.add(withdrawalRequestId);
    // ...
}
```

Allowing zero-amount requests serves no functional purpose and has several negative consequences:

- **State Bloat**: It allows for the creation of useless entries in the `_withdrawalRequests` set, unnecessarily increasing the contract's storage footprint;
- **Increased Gas Costs**: Any function that iterates over the set of withdrawal requests will incur slightly higher gas costs due to these extra, meaningless entries;
- **Event Spam**: It emits a `WithdrawalRequestCreated` event for a non-event, creating unnecessary noise on the blockchain and making it harder to track legitimate withdrawals;

Recommendation(s): Consider adding a `require` statement at the beginning of the `createWithdrawalRequest(...)` function to ensure the `_amount` is greater than zero. For example: `require(_amount > 0, "Amount must be greater than zero");`.

Status: Fixed.

Update from the client: Fixed in commit 64a1b2.

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 Hyperbeat's Vault contract documentation

The Hyperbeat team provided an overview of the core system components during meetings, presenting a clear explanation of the intended functionalities and addressing questions raised by the Nethermind Security team. Additionally, the Hyperbeat team provided written documentation in the form of markdown files describing the protocol's different flows.

8 Test Suite Evaluation

8.1 Tests Output

```
forge test --match-path "**vault**"
[] Compiling...
No files changed, compilation skipped

Ran 3 tests for tests/vault/e2e/VaultE2ETest.sol:VaultE2ETest
[PASS] test_shouldClaimFeesInDifferentAssetThanBaseAsset() (gas: 444565)
[PASS] test_shouldReflectYieldInExchangeRateForProcessedWithdrawals() (gas: 577077)
[PASS] test_shouldUpdateExchangeRateProperly() (gas: 322864)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 24.36ms (1.58ms CPU time)

Ran 24 tests for tests/vault/Pricer.t.sol:PricerTest
[PASS] testFuzz_GetAssetAmount(uint256) (runs: 1000, : 34870, ~: 35026)
[PASS] testFuzz_GetShareAmountInShareDecimals(uint256) (runs: 1000, : 34838, ~: 34983)
[PASS] test_RoleCapabilities_AdminCanCallAllSetters() (gas: 212231)
[PASS] test_RoleCapabilities_OwnerCanCallAllSetters() (gas: 215262)
[PASS] test_RoleCapabilities_RemoveCapabilityRemovesAccess() (gas: 57523)
[PASS] test_RoleCapabilities_RemoveRoleRemovesAccess() (gas: 38338)
[PASS] test_RoleCapabilities_UserCannotCallSetters() (gas: 244646)
[PASS] test_SetAssetConfig_UpdatesProperly() (gas: 916193)
[PASS] test_UpdateDelay_UpdatesProperly() (gas: 39990)
[PASS] test_UpdateExchangeRate() (gas: 107454)
[PASS] test_UpdateExchangeRateMultipleTimes() (gas: 155023)
[PASS] test_UpdateExchangeRate_RevertsIfCalledTooSoon() (gas: 35525)
[PASS] test_UpdateExchangeRate_RevertsIfOutsideLowerBound() (gas: 36441)
[PASS] test_UpdateExchangeRate_RevertsIfOutsideUpperBound() (gas: 36234)
[PASS] test_UpdateFeeRecipient_RevertsIfZeroAddress() (gas: 25750)
[PASS] test_UpdateFeeRecipient_UpdatesProperly() (gas: 43699)
[PASS] test_UpdateLower_RevertsIfTooLarge() (gas: 25779)
[PASS] test_UpdateLower_UpdatesProperly() (gas: 39968)
[PASS] test_UpdateManagementFee_RevertsIfTooLarge() (gas: 25777)
[PASS] test_UpdateManagementFee_UpdatesProperly() (gas: 40019)
[PASS] test_UpdatePerformanceFee_RevertsIfTooLarge() (gas: 25756)
[PASS] test_UpdatePerformanceFee_UpdatesProperly() (gas: 39978)
[PASS] test_UpdateUpper_RevertsIfTooSmall() (gas: 25778)
[PASS] test_UpdateUpper_UpdatesProperly() (gas: 39999)
Suite result: ok. 24 passed; 0 failed; 0 skipped; finished in 52.70ms (81.58ms CPU time)

Ran 20 tests for tests/vault/Depositor.t.sol:DepositorTest
[PASS] testFuzz_Deposit_VariousAmounts(uint256) (runs: 1000, : 137018, ~: 137046)
[PASS] test_Deposit_AfterExchangeRateUpdate() (gas: 212223)
[PASS] test_Deposit_BaseAsset() (gas: 129686)
[PASS] test_Deposit_DepositToken1() (gas: 137119)
[PASS] test_Deposit_DepositToken2() (gas: 137208)
[PASS] test_Deposit_RevertsForNonDepositToken() (gas: 865596)
[PASS] test_Deposit_RevertsWhenPaused() (gas: 65425)
[PASS] test_Deposit_ToDifferentReceiver() (gas: 128079)
[PASS] test_MultipleDeposits_DifferentUsers() (gas: 207239)
[PASS] test_RoleCapabilities_OwnerCanCallAllSetters() (gas: 3221724)
[PASS] test_RoleCapabilities_RemoveCapabilityRemovesAccess() (gas: 851741)
[PASS] test_RoleCapabilities_RemoveRoleRemovesAccess() (gas: 38378)
[PASS] test_SetDepositReceiver_AdminCanSet() (gas: 38812)
[PASS] test_SetDepositReceiver_UserCannotSet() (gas: 29531)
[PASS] test_SetPricer_AdminCanSet() (gas: 2397206)
[PASS] test_SetPricer_UserCannotSet() (gas: 2388013)
[PASS] test_ToggleDepositToken_AdminCanToggle() (gas: 817611)
[PASS] test_ToggleDepositToken_UserCannotToggle() (gas: 804996)
[PASS] test_TogglePaused_AdminCanToggle() (gas: 38738)
[PASS] test_TogglePaused_UserCannotToggle() (gas: 27891)
Suite result: ok. 20 passed; 0 failed; 0 skipped; finished in 79.60ms (60.84ms CPU time)

Ran 17 tests for tests/vault/WithdrawalQueue.t.sol:WithdrawalQueueTest
[PASS] testFuzz_InstantWithdrawalFee(uint256) (runs: 1000, : 301451, ~: 301721)
[PASS] test_CreateWithdrawalRequest_MultipleUsers() (gas: 363237)
[PASS] test_CreateWithdrawalRequest_RevertsWhenPaused() (gas: 308107)
[PASS] test_CreateWithdrawalRequest_Success() (gas: 328285)
[PASS] test_InstantWithdraw_RevertsWhenPaused() (gas: 279326)
[PASS] test_InstantWithdraw_Success() (gas: 304139)
[PASS] test_ProcessWithdrawalRequests_Success() (gas: 414597)
```

```
[PASS] test_ProcessWithdrawalRequests_WorksWithLowerExchangeRate() (gas: 411097)
[PASS] test_ProcessWithdrawalRequests_processWithLowerExchangeRate() (gas: 405933)
[PASS] test_RejectWithdrawalRequest_Success() (gas: 347570)
[PASS] test_RoleCapabilities_OwnerCanCallAllSetters() (gas: 2438912)
[PASS] test_SetInstantWithdrawalFee_AdminOnly() (gas: 40914)
[PASS] test_SetInstantWithdrawalPaused_AdminOnly() (gas: 46694)
[PASS] test_SetPaused_AdminOnly() (gas: 46716)
[PASS] test_SetPricer_AdminOnly() (gas: 2401132)
[PASS] test_WithdrawalQueueFlow_FullIntegration() (gas: 383989)
[PASS] test_WithdrawalWithDifferentExchangeRates() (gas: 469228)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 160.89ms (144.19ms CPU time)

Ran 4 test suites in 167.72ms (317.56ms CPU time): 64 tests passed, 0 failed, 0 skipped (64 total tests)
```

8.2 Automated Tools

8.2.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

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.