Security Review Report
NM-0674 OpenCover

NETHERMIND
SECURITY

(December 17, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for the OpenCover's **CoveredMetavault** contracts.

OpenCover is a specialist DeFi risk transfer provider, enabling individuals, businesses and institutions to protect themselves against technical and economic risks onchain.

The **CoveredMetavault** contract is a new primitive that implements the concept of an insured yield vault, enabling composable on-demand coverage natively onchain. It is a UUPS-upgradeable vault designed to wrap an underlying ERC-4626 compatible asset. It follows and extends the ERC-7540 standard to handle asynchronous deposits and redemptions through a request-settle-claim lifecycle managed by epochs. The protocol incorporates a configurable premium-streaming mechanism that directs a portion of settled assets to a designated collector to pay for coverage progressively from collected yield rather than upfront

The audit comprises 916 lines of the Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report nine points of attention, were two are classified as `Low`, and seven are classified as `Informational` or `Best Practices` severity. 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.

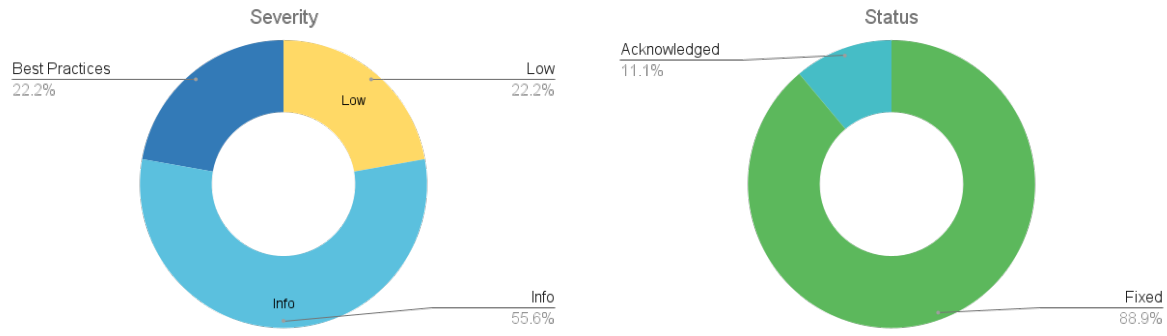

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (5), **Best Practices** (2). **Distribution of status: Fixed** (8), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	December 05, 2025
Final Report	December 17, 2025
Initial Commit	5685593c2b7cec9d3e5f60420fd653b4445ed815
Final Commit	84823a3b09f4b296d7431dd66d235c20efd5e5f2
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/CoveredMetavault.sol	767	259	33.8%	237	1263
2	src/Constants.sol	5	5	100.0%	4	14
3	src/libraries/PercentageLib.sol	16	8	50.0%	6	30
4	src/interfaces/IERC7540.sol	33	83	251.5%	16	132
5	src/interfaces/IERC7540Cancel.sol	19	37	194.7%	6	62
6	src/interfaces/ICoveredMetavault.sol	71	104	146.5%	40	215
7	src/interfaces/IERC7575.sol	5	7	140.0%	2	14
	Total	**916**	**503**	**54.9%**	**311**	**1730**

3 Summary of Issues

	Finding	Severity	Update
1	Pending assets are not reset when moving to a new era, allowing claims on the new pool	Low	Fixed
2	Users may be restricted from withdrawing their full remaining balance if it is below the minimum request amount	Low	Fixed
3	Floor rounding in premium calculation causes the claimable pool to overpay premiums	Info	Fixed
4	Premium streaming can be enabled prematurely before the first settlement	Info	Fixed
5	The `requestRedeem` function uses an external call to transfer vault's shares	Info	Fixed
6	Unfair unit distribution in `_computeUnitAssignment` favors early claimers	Info	Fixed
7	`claimableRedeemRequest` returns non-zero for already claimed request	Info	Acknowledged
8	Single-step ownership transfer can lead to permanent loss of ownership	Best Practices	Fixed
9	Use Namespaced storage for the vault	Best Practices	Fixed

4 System Overview

The `CoveredMetavault` is a UUPS upgradeable ERC-7540 vault, which implements an asynchronous process for deposits and redemptions. The vault's primary role is to stream premium for the insurance cover.

The users can make deposit and redemption requests that wait in a queue to be settled by entitled roles only. Post settlement, users can claim the processed requests.

The critical operations and management of the vault are governed by a role-based access control system with segregated roles for settlement, configuration, and pausability.

4.1 Asynchronous life cycle

The deposits and redemptions in `CoveredMetavault` are implemented as a three-phase process.

- **Request phase**: A user initiates a request for a deposit or a redemption by transferring assets or vault shares to the contract. The request is queued and is pending for processing.
- **Settlement phase**: A privileged actor, Keeper, periodically calls the `settle` function, which processes pending requests for deposit and settlement in a batch. Every settlement also advances the vault's epoch while computing the proportional assets or shares for the pending requests.

  Before settling the pending request, the vault computes the pending premiums and streams them to the premium collector account.

  As part of internal tracking, the vault clearly segregates between assets liable for premium and newly received assets that are yet to settle.
- **Claim phase**: The actual computation of receivables is performed in the vault at the time of the claiming process. Using internal accounting, the vault calculates the claim amount based on the user's proportional assets and shares.

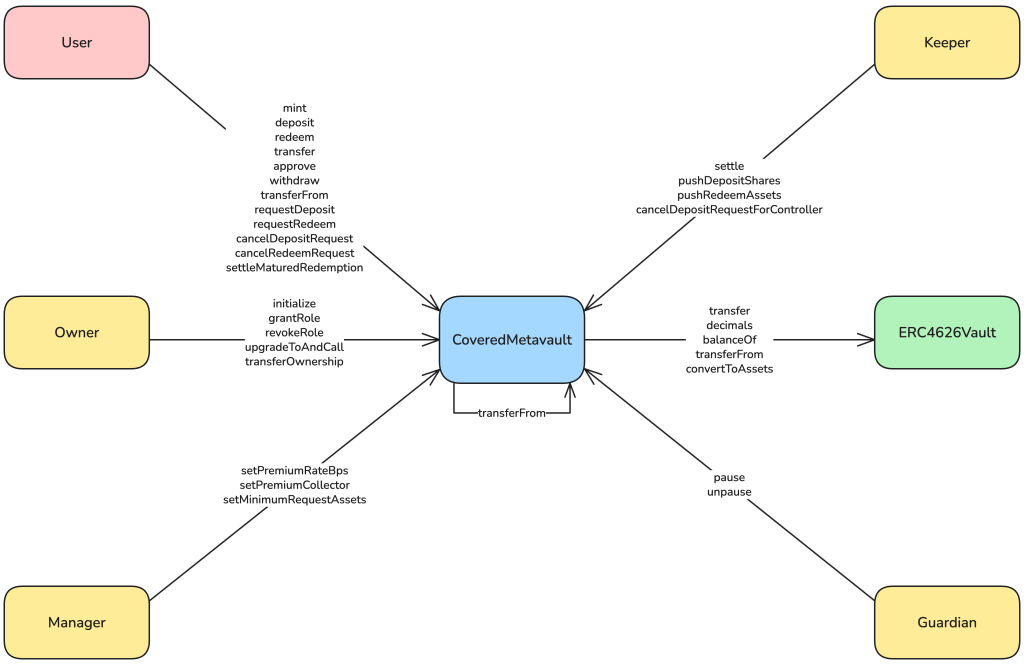

Fig. 2: CoveredMetavault Interactions

4.2 Actors and Entry points

Below are the actors and their entry points.

- **User**: Any one depositing into the `CoveredMetavault`.
  - `requestDeposit`: Submits assets to the vault to be queued for a future deposit.
  - `requestRedeem`: Submits vault shares to be queued for a future redemption.
  - `deposit/mint`: Claims vault shares after a deposit request has been settled by a Keeper.
  - `redeem/withdraw`: Claims underlying assets after a redemption request has been settled by a Keeper.
  - `cancelDepositRequest`: Cancels a pending deposit request and withdraws the submitted assets.

  - `cancelRedeemRequest`: Cancels a pending redemption request and withdraws the submitted shares.
- **Keeper**: Entitled role responsible for settling the pending requests.
  - `settle`: Processes all pending deposit and redemption requests, making them claimable by users.
  - `cancelDepositRequestForController`: Cancels a pending deposit on behalf of a specific user.
  - `pushDepositShares`: Forcefully mints and sends shares to a user from their settled deposit balance.
  - `pushRedeemAssets`: Forcefully redeems and sends assets to a user from their settled redemption balance.
- **Anyone**:
  - `settleMaturedRedemption`: Settles a specific redemption request if it has remained pending beyond a predefined maturity period, providing an exit path for users.
- **Guardian**: Entitled role with permissions to pause or unpause the `CoveredMetavault`.
  - `pause`: To pause the functionality of the vault in case of unexpected scenarios.
  - `unpause`: To release the vault from a paused state.
- **Manager**: Entitled role to configure the critical parameters of `CoveredMetavault`.
  - `setPremiumRateBps`: Configures the premium rate.
  - `setPremiumCollector`: Configures the address that receives the streamed premium.
  - `setMinimumRequestAssets`: Configures the minimum amount of assets that can be deposited into the vault.

4.3 Segregated Buckets

The segregation of assets in the `CoveredMetavault` functions as a tiered system that separates user intent from protocol accounting. The lifecycle begins with Pending Assets, which act as a temporary `loading dock`. The funds physically exist in the contract's balance but are mathematically ignored by the vault's valuation logic. This isolation ensures that unverified deposits cannot manipulate the share price, while simultaneously allowing users to cancel and receive a full refund without affecting the broader system.

Once the Keeper executes a settlement, these funds move into the Settled Assets category.This qualifies the assets to be charged the management fee, yet they remain excluded from the exchange rate calculation.

The final stage of the life cycle occurs when the user manually claims their deposit, converting Claimable Assets into Active Assets. Only at this point are shares minted to the users at the current rate.

4.4 Time clocks

The `CoveredMetavault` uses two distinct `Time Clocks` to manage the state of user deposits.While they both track time, they serve completely different purposes.

- **Epochs**: Epochs serve as a batching mechanism designed to organize incoming deposits into discrete groups. When a user deposits, their funds enter a `pending` state associated with the current epoch. Once the Keeper calls `settle()`, that epoch closes, a snapshot of the exchange rate between units and pending assets is taken, and these pending assets are effectively converted into claimable units.
- **Eras**: Eras function as a solvency safeguard that acts as a `hard reset` for the claimable deposit pool in the rare event that all assets are consumed. Because the pending pool is subject to continuous premium fees, it is mathematically possible for the asset balance to decay to zero. if this happens, the contract increments the Era counter, effectively declaring `bankruptcy` on the old pool and wiping out any unclaimed units associated with it. This ensures that when new liquidity enters the vault, it starts fresh in a new Era without being diluted by the debt of users who spent their funds in the previous Era.

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] Pending assets are not reset when moving to a new era, allowing claims on the new pool

File(s): src/CoveredMetavault.sol

Description: The `CoveredMetavault` utilizes a concept of "claimable eras" to manage the lifecycle of the deposit pool. When the `totalClaimableAssets` of the vault are fully drained (e.g., due to premium streaming), the `claimableEra` increments, effectively marking the previous pool as dead.

The `_syncEpoch(...)` function is responsible for updating a controller's state to match the current vault state. It handles two main transitions:

1. **Era Sync**: If the controller's `claimableEra` is behind the global era, it resets the controller's `claimableUnits` to zero and updates the era;
2. **Epoch Sync**: If the controller's `lastSyncedEpoch` is behind the current epoch, it processes any `pendingAssets` by calculating the corresponding unit assignment and adding them to `claimableUnits`;

The issue is that `pendingAssets` represents deposits that were requested in a previous epoch and settled into the global pool at that time. If the global `claimableEra` has incremented since the user's last sync, it implies that the pool—to which those `pendingAssets` were contributed—has been fully exhausted.

However, the logic in `_syncEpoch` clears `claimableUnits` upon an era change but does **not** clear `pendingAssets`. The execution flows immediately into the epoch sync block, where it converts these stale `pendingAssets` (using the historical `epochAllocations`) into units and adds them to the controller's `claimableUnits`.

Since the controller's era was just updated to the *new* era, these units are now valid claims against the *new* era's pool. This effectively allows a user to carry over a claim from a drained/dead pool into a fresh pool, potentially diluting new depositors or stealing assets they did not contribute to.

Recommendation(s): Consider resetting `pendingAssets` to zero and updating the `lastSyncedEpoch` when a controller is detected to be in an outdated era. This ensures that claims tied to a fully drained pool are correctly discarded.

Status: Fixed

Update from the client: Fixed in commit b34107. Resolved by pre-minting metavault shares during settlement, simplifying vault accounting and eliminating the need for eras.

6.2 [Low] Users may be restricted from withdrawing their full remaining balance if it is below the minimum request amount

File(s): src/CoveredMetavault.sol

Description: The `CoveredMetavault` enforces a minimum asset amount (`minimumRequestAssets`) for asynchronous redemption requests initiated via `requestRedeem(...)`.

This check ensures that the resulting underlying asset value from the requested shares meets the minimum threshold.

The constraint is enforced within `requestRedeem(...)` before the redemption request is recorded:

```
function requestRedeem(uint256 shares, address controller, address owner)
    external
    override
// ...
    returns (uint256 requestId)
{
    require(shares != 0, ZeroShares());
    require(controller == owner, InvalidController(controller));
    require(owner == msg.sender, InvalidOwner(owner));

    uint256 redeemAssetAmount = _convertToAssets(shares, Math.Rounding.Floor);
    _requireMinimumRequestAssets(redeemAssetAmount); // The minimum request assets check is enforced here.

// ...
}
```

If a user's total remaining share balance corresponds to an underlying asset value that is **less than** `minimumRequestAssets`, they will be blocked from initiating a withdrawal using `requestRedeem(...)` because the check in `_requireMinimumRequestAssets(...)` will revert.

While the assets are not permanently locked, this mechanism hinders a user's ability to fully exit the position if their final balance is low.

Recommendation(s): Consider modifying the minimum request asset requirement to allow requests that are either greater than or equal to the configured minimum, **OR** requests that represent the user's entire remaining balance. This maintains the protection against "dust" operations while guaranteeing that users can always fully exit their position.

Status: Fixed.

Update from the client: Fixed in commit b5a250. Removed this restriction for redemptions, allowing users to always redeem their full remaining balance. Minimums are now enforced by the offchain settlement engine that operates the keeper, while users can still settle and claim redemptions permissionlessly via `settleMaturedRedemption`.

6.3 [Info] Floor rounding in premium calculation causes the claimable pool to overpay premiums

File(s): src/CoveredMetavault.sol

Description: The `_streamPremium()` function calculates and transfers a premium fee based on the total settled assets of the vault. Since the vault's assets are logically partitioned into "backed assets" and "claimable assets", the premium payment must be proportionally deducted from both pools.

When updating the `totalClaimableAssets` to reflect the deduction, the contract calculates the new balance using the ratio of assets remaining after the premium payment:

```
// src/CoveredMetavault.sol

// ...
    if (claimableAssetsTotal != 0) {
        uint256 newClaimableAssetTotal =
            claimableAssetsTotal.mulDiv(assetsAfter, assetsBefore, Math.Rounding.Floor); // @audit-issue Rounding down
             ↪  the remaining assets
// ...
        _vaultStorage.totalClaimableAssets = newClaimableAssetTotal;
// ...
```

By using `Math.Rounding.Floor` to calculate `newClaimableAssetTotal`, the contract rounds down the remaining assets in the claimable pool. This effectively means the reduction applied to the claimable pool (the premium paid by it) is rounded *up*.

Consequently, the claimable pool bears the burden of the rounding error, paying slightly more premium than the backed assets pool. Over time, this slight overpayment creates a disadvantage for users whose assets are sitting in the claimable pool waiting to be claimed, as their claimable units will correspond to a slightly smaller total asset amount than mathematically precise.

Recommendation(s): The current implementation places the burden of precision loss on the claimable pool. While switching to Math.Rounding.Ceil would shift this burden to the backed pool, the current design systematically favors the backed assets.

Consider evaluating this trade-off. If the intent is to protect the backed pool, no change is needed, but consider documenting this behavior to incentivize users to finalize claims.

Status: Fixed

Update from the client: Fixed in commit b34107. Resolved by pre-minting shares at settlement so that premiums are applied via the share price only, eliminating the need to scale the claimable pool.

6.4 [Info] Premium streaming can be enabled prematurely before the first settlement

File(s): src/CoveredMetavault.sol

Description: The `CoveredMetavault` uses the `lastPremiumTimestamp` state variable to track when the premium was last streamed. A value of `0` is intended to serve as a sentinel indicating that the vault has not yet undergone its first settlement, ensuring that no premium is accrued or charged for the period preceding the first epoch. This logic is enforced in the `_streamPremium` function, which checks that `lastPremiumTimestamp != 0` before calculating the premium.

However, the `lastPremiumTimestamp` variable can be set to a non-zero value before the first settlement in two scenarios:

1. **During Initialization**: The `initialize(...)` function explicitly sets `lastPremiumTimestamp` to `block.timestamp` if the initial `premiumRateBps` is non-zero;
2. **Updating Premium Rate**: The `setPremiumRateBps(...)` function calls `_streamPremium()` to settle accrued premiums before updating the rate. The `_streamPremium()` function unconditionally updates `lastPremiumTimestamp` to the current timestamp at the end of execution, even if the calculation was skipped because the previous timestamp was `0`;

In both cases, `lastPremiumTimestamp` becomes non-zero before the first `settle()` call. As a result, when the first settlement eventually occurs, the vault will calculate and charge premiums for the duration elapsed since initialization or the rate update. Note that is not expected for the vault to hold funds at this time.

Recommendation(s): Consider modifying the `initialize(...)` function to ensure `lastPremiumTimestamp` remains `0`, even if a premium rate is set. Additionally, update the `setPremiumRateBps(...)` function to check if `lastPremiumTimestamp` is `0`; if so, it can skip the `_streamPremium()` call and simply update the rate, preserving the sentinel value until the first settlement.

Status: Fixed.

Update from the client: Fixed in commit 3fc8d3. Resolved by removing the lastPremiumTimestamp assignment from the initialiser and guarding premium streaming in setPremiumRateBps to preserve the sentinel value until the first settlement.

6.5 [Info] The `requestRedeem` function uses an external call to transfer vault's shares

File(s): src/CoveredMetavault.sol

Description: The `CoveredMetavault` contract functions has the `requestRedeem` function to return shares to the vault in exchange for assets. The function explicitly validates that the caller is the owner of the shares or has the appropriate permissions.

However, the contract uses an external call to transfer the shares from the user to the contract itself.

```
function requestRedeem(uint256 shares, address controller, address owner) external {
    // ...
    if (msg.sender != owner) {
        revert InvalidOwner(owner);
    }
    // ...
    // @audit-issue This requires prior approval and unnecessarily increases gas costs.
    IERC20(address(this)).safeTransferFrom(owner, address(this), shares);
    // ...
}
```

By using `safeTransferFrom` on `address(this)`, the contract triggers the standard ERC20 allowance check. This creates a circular dependency where the user must approve the vault contract to spend the vault's own tokens, even though the user is the one initiating the transaction.

This leads to a degraded User Experience as the user must sign and broadcast two transactions (approve and redeem) instead of one. Additionally, it increases the gas cost due to the external CALL opcode and the storage operations required for the allowance check.

Recommendation(s): Consider replacing the external `safeTransferFrom` call with the internal `_transfer` function inherited from the ERC20Upgradeable contract. This will bypass the redundant allowance check and allow users to redeem their shares in a single transaction.

Status: Fixed.

Update from the client: Fixed in commit 915324. Resolved by replacing the external `safeTransferFrom` with an internal `_transfer` call when locking vault shares in `requestRedeem`.

6.6 [Info] Unfair unit distribution in `_computeUnitAssignment` favors early claimers

File(s): src/CoveredMetavault.sol

Description: The `_computeUnitAssignment` function determines the share of claimable units a controller receives based on their pending assets. The algorithm calculates `maxAssignableUnits` by determining the minimum number of units that must be reserved for the remaining `unassignedAssets`.

However, the function implements an "optimistic" rounding mechanism. If the standard pro-rata calculation (`unitsToAssign`) is less than `maxAssignableUnits` and the division results in a remainder, the function explicitly increments `unitsToAssign` by `1`.

```

// ...
    // Start from floor and clamp for extra safety.
    unitsToAssign = pendingAssets.mulDiv(epochTotalUnits, epochTotalAssets, Math.Rounding.Floor);
    if (unitsToAssign > maxAssignableUnits) unitsToAssign = maxAssignableUnits;

    // @audit-issue This logic rounds up the assignment, consuming units meant for later users.
    // Grant at most +1 unit if it won't breach the remaining units limit.
    if (unitsToAssign < maxAssignableUnits) {
        uint256 remainderNumerator = mulmod(pendingAssets, epochTotalUnits, epochTotalAssets);
        if (remainderNumerator != 0) {
            unitsToAssign += 1;
            if (unitsToAssign > maxAssignableUnits) unitsToAssign = maxAssignableUnits;
        }
    }
}
```

This logic systematically favors users who sync their state (claim) earlier in the epoch. By rounding up their unit allocation, early claimers extract more value from the `epochTotalUnits` pool than their exact pro-rata share. Since the total number of units is fixed, this over-allocation reduces the `unassignedUnits` available for subsequent users. Consequently, users who claim later will inevitably receive fewer units than their fair share to balance the epoch, resulting in an unfair distribution based on claim order.

Recommendation(s): Consider removing the conditional block that increments `unitsToAssign` by `1`. Relying on standard floor rounding (`Math.Rounding.Floor`) for all unit assignments ensures that every user receives a consistent and fair share of the units relative to their assets, regardless of the order in which they sync.

Status: Fixed

Update from the client: Fixed in commit b34107. Replaced _computeUnitAssignment with a simpler, per-epoch, floor-based share allocation that removes order-dependent bias.

6.7 [Info] `claimableRedeemRequest` returns non-zero for already claimed request

File(s): src/CoveredMetavault.sol

Description: `claimableRedeemRequest` is a view function that returns the number of shares that can be claimed, if the redeem request is already settled. But, this number is valid only until the actual claim is processed. After the shares are already claimed via `redeem`, the request will not have claimable shares as they were already processed.

```
function claimableRedeemRequest(uint256 requestId, address controller)
        external
        view
        override
        returns (uint256 claimableShares)
    {
        RedeemRequestStorage storage redeemRequestStorage = _vaultStorage.redeemRequests[requestId];
        if (redeemRequestStorage.controller != controller || redeemRequestStorage.timestamp == 0) return 0;

        // Audit - this is valid only under the shares are not claimed by the user. Post redeem call, this function
         ↪  should return 0 shares considering the shares were already claimed.
 ==>    return redeemRequestStorage.settled ? redeemRequestStorage.shares : 0;
    }
```

Recommendation(s): Consider documenting this behavior for integrators and users that can use this feature.

Status: Acknowledged.

Update from the client: Documented the current behaviour in both NatSpec comments and the integrator guides.

6.8 [Best Practice] Single-step ownership transfer can lead to permanent loss of ownership

File(s): src/CoveredMetavault.sol

Description: The `transferOwnership()` function from the Vault contract is used to transfer control of the contract to a new address. The function validates that the `newOwner` is not the zero address, revokes the `owner` role from the caller, and grants the `owner` role to the `newOwner` address.

However, the function performs this transfer in a single step. It does not verify that the `newOwner` account is accessible or correct beyond the zero address check.

This means that if the current owner accidentally passes an incorrect address, the ownership of the vault will be permanently lost. Since the current owner revokes their own role immediately during the call, there is no way to recover access.

```
function transferOwnership(address newOwner) external {
    // ...
    require(newOwner != address(0), "Invalid address");

    _revokeRole(OWNER_ROLE, msg.sender);
    // @audit If newOwner is incorrect, ownership is lost forever.
    _grantRole(OWNER_ROLE, newOwner);
}
```

Recommendation(s): Consider implementing a two-step ownership transfer process. This pattern allows the current `owner` to propose a `newOwner` address, but the transfer is not finalized until the `newOwner` explicitly accepts it. This approach ensures that the `newOwner` address is valid and under control before the previous `owner` relinquishes their rights.

Status: Fixed.

Update from the client: Fixed in commit 5960af. The contract now implements a two-step ownership transfer process.

6.9 [Best Practices] Use Namespaced storage for the vault

File(s): src/CoveredMetavault.sol

Description: `CoveredMetavault` is an upgradeable contract. In order to minimise the storage layout conflicts, it is recommended to use Namespaced storage instead of using the storage slot of this contract.

Recommendation(s): Use namespace storage for storage variables.

Status: Fixed.

Update from the client: Fixed in commit 08e074. Resolved by using ERC-7201 compliant namespaced storage for vault state.

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 OpenCover's documentation

The development team provided a detailed overview for the CoveredMetaVault contract and its purpose. The code also contains inline comments to describe the functionality.The development team was collaborative during the audit, providing answers to all questions raised by the Nethermind Security team during calls and through asynchronous communications.

8 Test Suite Evaluation

8.1 Tests Output

```
> forge test
[] Compiling...
No files changed, compilation skipped

Ran 2 tests for test/unit/governance/GuardianPausable.t.sol:GuardianPausableTest
[PASS] test_Pause_Unpause_OnlyGuardian() (gas: 42800)
[PASS] test_Paused_BlocksStateChangingOperations() (gas: 158360)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.33s (182.17µs CPU time)

Ran 53 tests for test/unit/vault/Redemptions.t.sol:RedemptionsUnitTest
[PASS] test_CancelRedeem_CancelOneOfMultipleRequests() (gas: 685556)
[PASS] test_CancelRedeem_RevertWhen_ReceiverDiffersFromController() (gas: 423879)
[PASS] test_CancelRedeem_RevertsAfterSettlement() (gas: 506496)
[PASS] test_CancelRedeem_RevertsForInvalidController() (gas: 425076)
[PASS] test_CancelRedeem_RevertsForInvalidRequest() (gas: 20913)
[PASS] test_CancelRedeem_RevertsForZeroReceiver() (gas: 424716)
[PASS] test_CancelRedeem_RevertsWhenAlreadyCancelled() (gas: 450728)
[PASS] test_CancelRedeem_RevertsWhenCallerNotControllerEvenIfParamMatches() (gas: 424741)
[PASS] test_CancelRedeem_RevertsWhen_ControllerParamDiffersFromStoredRequest() (gas: 425347)
[PASS] test_CancelRedeem_SucceedsAfterMaturity() (gas: 460991)
[PASS] test_CancelRedeem_Success() (gas: 459612)
[PASS] test_ClaimableRedeemRequest_MismatchedController_ReturnsZero() (gas: 506345)
[PASS] test_ClaimableRedeemShares_BeforeSettlement_ReturnsZero() (gas: 427271)
[PASS] test_ClaimableRedeemShares_TracksSettlementAndConsumption() (gas: 518701)
[PASS] test_MaxRedeemAndWithdraw_AreZero_BeforeSettlement() (gas: 428593)
[PASS] test_PendingAndClaimableRedeemRequest_Lifecycle() (gas: 524082)
[PASS] test_PendingRedeemRequest_MismatchedController_ReturnsZero() (gas: 424424)
[PASS] test_PreviewRedeem_Reverts() (gas: 14332)
[PASS] test_PreviewWithdraw_Reverts() (gas: 14090)
[PASS] test_PushRedeemAssets_RevertsWhen_CallerNotKeeper() (gas: 509831)
[PASS] test_PushRedeemAssets_RevertsWhen_InsufficientClaimableShares() (gas: 507402)
[PASS] test_PushRedeemAssets_RevertsWhen_ZeroAddress() (gas: 20843)
[PASS] test_PushRedeemAssets_RevertsWhen_ZeroShares() (gas: 20232)
[PASS] test_PushRedeemAssets_Settled_Succeeds() (gas: 524384)
[PASS] test_Redeem_InvalidController() (gas: 507130)
[PASS] test_Redeem_RevertWhen_ReceiverDiffersFromController() (gas: 507449)
[PASS] test_Redeem_RevertsWhen_ZeroShares() (gas: 17884)
[PASS] test_Redeem_Reverts_WhenNotClaimable() (gas: 517863)
[PASS] test_Redeem_Success_EmitsWithdraw_AndTransfersAssets() (gas: 524433)
[PASS] test_Redeem_ZeroReceiver() (gas: 507208)
[PASS] test_RequestRedeem_InsufficientAllowance() (gas: 292650)
[PASS] test_RequestRedeem_InvalidOwner() (gas: 306011)
[PASS] test_RequestRedeem_MultipleRequests_SameController() (gas: 773821)
[PASS] test_RequestRedeem_PartialSettlement() (gas: 1010935)
[PASS] test_RequestRedeem_RevertWhen_BelowMinimumRequestAssets() (gas: 325284)
[PASS] test_RequestRedeem_RevertWhen_ControllerDiffersFromOwner() (gas: 306838)
[PASS] test_RequestRedeem_RevertWhen_UnderlyingValueDropsBelowMinimum() (gas: 328728)
[PASS] test_RequestRedeem_SucceedsAtMinimumRequestAssets() (gas: 437236)
[PASS] test_RequestRedeem_Success() (gas: 435680)
[PASS] test_RequestRedeem_ZeroController() (gas: 306704)
[PASS] test_RequestRedeem_ZeroShares() (gas: 17598)
[PASS] test_RevertWhen_PushRedeemAssetsRoundsDownToZeroAssets() (gas: 507208)
[PASS] test_RevertWhen_RedeemRoundsDownToZeroAssets() (gas: 507619)
[PASS] test_SettleRedeem_AlreadySettled_Reverts() (gas: 511064)
[PASS] test_SettleRedeem_InvalidRequest_Reverts() (gas: 33991)
[PASS] test_SettleRedeem_RevertWhen_AssetsFloorToZero() (gas: 458608)
[PASS] test_TotalAssets_ExcludesClaimableRedeemPool() (gas: 692112)
[PASS] test_Withdraw_InvalidController() (gas: 509176)
[PASS] test_Withdraw_RevertWhen_ReceiverDiffersFromController() (gas: 509531)
[PASS] test_Withdraw_Reverts_WhenNotClaimable() (gas: 20401)
[PASS] test_Withdraw_Success_ExactAssets() (gas: 519376)
[PASS] test_Withdraw_ZeroAssets() (gas: 17526)
[PASS] test_Withdraw_ZeroReceiver() (gas: 509008)
Suite result: ok. 53 passed; 0 failed; 0 skipped; finished in 5.34s (19.12ms CPU time)

Ran 12 tests for test/unit/vault/MaturedRedemptions.t.sol:MaturedRedemptionsUnitTest
[PASS] test_AccountingCorrect_AfterMaturedSettlement() (gas: 904029)
[PASS] test_ClaimAfterMaturation_WithoutExplicitSettlement() (gas: 515506)
```

```
[PASS] test_ClaimableRedeemRequest_RequiresSettlementEvenAfterMaturation() (gas: 508238)
[PASS] test_MixedSettlement_KeeperAndMatured() (gas: 762946)
[PASS] test_PartialMaturation_OnlyMaturedAreSettled() (gas: 774350)
[PASS] test_PendingRedeemRequest_MaturedStillPending_UntilSettled() (gas: 515780)
[PASS] test_SettleMaturedRedemption_RevertWhen_AssetsFloorToZero() (gas: 456255)
[PASS] test_SettleMaturedRedemption_RevertsAlreadySettled() (gas: 506499)
[PASS] test_SettleMaturedRedemption_RevertsInvalidRequestId() (gas: 18932)
[PASS] test_SettleMaturedRedemption_RevertsNotMatured() (gas: 424039)
[PASS] test_SettleMaturedRedemption_Success_MultipleRequests() (gas: 759889)
[PASS] test_SettleMaturedRedemption_Success_SingleRequest() (gas: 507687)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 5.37s (3.65ms CPU time)

Ran 1 test for test/integration/DepositsIntegration.t.sol:DepositsIntegrationTest
[PASS] test_MultipleUsersMultipleEpochs() (gas: 718108)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.38s (337.38µs CPU time)

Ran 1 test for test/scenario/MultiEpochScenario.t.sol:MultiEpochScenarioTest
[PASS] test_Scenario_TwoUsersMultipleEpochsWithRedeem() (gas: 1143719)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.40s (626.71µs CPU time)

Ran 3 tests for test/fuzz/PremiumFuzz.t.sol:PremiumFuzzTest
[PASS] testFuzz_PremiumStreaming_VariousDepositAmounts(uint256) (runs: 2048, : 337052, ~: 338151)
[PASS] testFuzz_PremiumStreaming_VariousDurations(uint256) (runs: 2048, : 295986, ~: 296494)
[PASS] testFuzz_PremiumStreaming_VariousRates(uint256) (runs: 2048, : 298415, ~: 298904)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 6.42s (1.10s CPU time)

Ran 3 tests for test/fuzz/DepositsFuzz.t.sol:DepositsFuzzTest
[PASS] testFuzz_DepositAfterSettlement(uint256) (runs: 2048, : 521726, ~: 521602)
[PASS] testFuzz_PartialClaim(uint256,uint256) (runs: 2048, : 521615, ~: 521617)
[PASS] testFuzz_RequestDeposit(uint256) (runs: 2048, : 122078, ~: 121974)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 6.75s (1.37s CPU time)

Ran 2 tests for test/fuzz/MaturedRedemptionsFuzz.t.sol:MaturedRedemptionsFuzzTest
[PASS] testFuzz_MaturedRedemption_VariableDelays(uint256) (runs: 2048, : 520335, ~: 520364)
[PASS] testFuzz_MultipleMaturedRedemptions(uint8) (runs: 2048, : 1390024, ~: 1066610)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 7.59s (2.22s CPU time)

Ran 1 test for test/integration/RedemptionsIntegration.t.sol:RedemptionsIntegrationTest
[PASS] test_DonationAttack_NonProfitable() (gas: 1000745)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.73s (1.11ms CPU time)

Ran 10 tests for test/unit/vault/Config.t.sol:ConfigUnitTest
[PASS] test_Initialize_RevertWhen_AssetNotERC4626() (gas: 504925)
[PASS] test_Initialize_RevertWhen_PremiumCapTooHigh() (gas: 88335)
[PASS] test_Initialize_RevertWhen_PremiumCollectorZero() (gas: 88094)
[PASS] test_Initialize_RevertWhen_PremiumRateAboveCap() (gas: 88397)
[PASS] test_Initialize_RevertWhen_PremiumRateTooHigh() (gas: 88585)
[PASS] test_Initialize_SetsMinimumRequestAssets() (gas: 69113693)
[PASS] test_Initialize_SetsPremiumConfig() (gas: 69012011)
[PASS] test_SetMinimumRequestAssets_NoOpWhenSameValue() (gas: 23568)
[PASS] test_SetMinimumRequestAssets_RevertWhen_CallerNotManager() (gas: 20350)
[PASS] test_SetMinimumRequestAssets_UpdatesValue() (gas: 29771)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 9.22s (7.65s CPU time)

Ran 8 tests for test/unit/governance/AccessRoles.t.sol:AccessRolesTest
[PASS] test_GrantRevokeRoles_AdminOnly() (gas: 61009)
[PASS] test_SetPremiumCollector_OnlyManager() (gas: 38924)
[PASS] test_SetPremiumRateBps_OnlyManager() (gas: 46464)
[PASS] test_SettleDeposits_OnlyKeeper() (gas: 40281)
[PASS] test_TransferOwnership_NewOwnerCanUseRole() (gas: 86092)
[PASS] test_TransferOwnership_OnlyOwner() (gas: 84636)
[PASS] test_TransferOwnership_RevertWhen_ZeroAddress() (gas: 17948)
[PASS] test_Upgrade_OnlyOwner() (gas: 25341511)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 9.22s (3.76s CPU time)

Ran 20 tests for test/unit/vault/Premium.t.sol:PremiumUnitTest
[PASS] test_MaxPremiumRateBps_ReturnsConfiguredValue() (gas: 14181)
[PASS] test_PremiumStreaming_FirstEpochNoStreaming() (gas: 257606)
[PASS] test_PremiumStreaming_MultipleUsers_FairReduction() (gas: 436496)
[PASS] test_PremiumStreaming_PartialYear() (gas: 292800)
[PASS] test_PremiumStreaming_SameTimestamp() (gas: 265209)
[PASS] test_PremiumStreaming_UnitPriceReduction() (gas: 293733)
[PASS] test_PremiumStreaming_WithTimeElapsed() (gas: 297961)
[PASS] test_PremiumStreaming_ZeroRate() (gas: 271694)
[PASS] test_PremiumStreaming_ZeroSettledAssets() (gas: 74110)
```

```
[PASS] test_SetPremiumCollector_Success() (gas: 28578)
[PASS] test_SetPremiumCollector_UpdateExisting() (gas: 34729)
[PASS] test_SetPremiumCollector_ZeroAddress() (gas: 17264)
[PASS] test_SetPremiumRateBps_MaxRate() (gas: 32734)
[PASS] test_SetPremiumRateBps_RevertWhen_AboveVaultCap() (gas: 69040713)
[PASS] test_SetPremiumRateBps_RevertWhen_Paused() (gas: 42909)
[PASS] test_SetPremiumRateBps_StreamsAccruedAtOldRate() (gas: 302521)
[PASS] test_SetPremiumRateBps_Success() (gas: 32112)
[PASS] test_SetPremiumRateBps_TooHigh() (gas: 21931)
[PASS] test_SetPremiumRateBps_UpdateExisting() (gas: 40289)
[PASS] test_SetPremiumRateBps_Zero() (gas: 39755)
Suite result: ok. 20 passed; 0 failed; 0 skipped; finished in 9.24s (3.87s CPU time)

Ran 90 tests for test/unit/vault/Deposits.t.sol:DepositsUnitTest
[PASS] test_CancelDeposit_OnlyAffectsSelectedController() (gas: 243443)
[PASS] test_CancelDeposit_RevertWhen_ReceiverDiffersFromController() (gas: 118346)
[PASS] test_CancelDeposit_RevertsAfterSettlement() (gas: 243216)
[PASS] test_CancelDeposit_RevertsForInvalidController() (gas: 118613)
[PASS] test_CancelDeposit_RevertsForInvalidRequestId() (gas: 116676)
[PASS] test_CancelDeposit_RevertsForZeroReceiver() (gas: 118111)
[PASS] test_CancelDeposit_RevertsWhenAlreadyCancelled() (gas: 99881)
[PASS] test_CancelDeposit_RevertsWhenNoPending() (gas: 23899)
[PASS] test_CancelDeposit_Success() (gas: 107512)
[PASS] test_ClaimableDepositRequest_AfterSettlement() (gas: 245055)
[PASS] test_ClaimableDepositRequest_MultipleEpochs() (gas: 362511)
[PASS] test_ClaimableDepositRequest_NoRequest() (gas: 15761)
[PASS] test_ClaimableDepositRequest_PendingOnly() (gas: 120944)
[PASS] test_ClaimableDepositRequest_ReturnsZero_WhenNoUnitsButPoolExists() (gas: 251862)
[PASS] test_DepositThreeParameter_InsufficientClaimable() (gas: 292793)
[PASS] test_DepositThreeParameter_InvalidController() (gas: 243407)
[PASS] test_DepositThreeParameter_PartialClaim() (gas: 519708)
[PASS] test_DepositThreeParameter_RevertWhen_ReceiverDiffersFromController() (gas: 243565)
[PASS] test_DepositThreeParameter_Success() (gas: 288097)
[PASS] test_DepositThreeParameter_Success_NonInitial() (gas: 519680)
[PASS] test_DepositThreeParameter_ZeroAssets() (gas: 18376)
[PASS] test_DepositThreeParameter_ZeroReceiver() (gas: 243347)
[PASS] test_Deposit_RevertsWhen_ZeroSharesDueToPrice() (gas: 359742)
[PASS] test_Deposit_TwoParameter_InsufficientClaimable() (gas: 32513)
[PASS] test_Deposit_TwoParameter_RevertWhen_ReceiverDiffersFromSender() (gas: 242617)
[PASS] test_Deposit_TwoParameter_Success() (gas: 284332)
[PASS] test_Deposit_TwoParameter_Success_NonInitial() (gas: 517678)
[PASS] test_Deposit_TwoParameter_ZeroReceiver() (gas: 242554)
[PASS] test_KeeperCancelDeposit_RevertsForInvalidRequestId() (gas: 19800)
[PASS] test_KeeperCancelDeposit_RevertsForUnauthorized() (gas: 122043)
[PASS] test_KeeperCancelDeposit_RevertsForZeroController() (gas: 19699)
[PASS] test_KeeperCancelDeposit_RevertsWhenNoPending() (gas: 26615)
[PASS] test_KeeperCancelDeposit_Success() (gas: 109928)
[PASS] test_MaxDeposit_NoClaimable() (gas: 16727)
[PASS] test_MaxDeposit_WithClaimable() (gas: 244946)
[PASS] test_MaxMint_WithClaimable() (gas: 475482)
[PASS] test_MintThreeParameter_ConversionPrecision() (gas: 519392)
[PASS] test_MintThreeParameter_ExactClaimableAmount() (gas: 519899)
[PASS] test_MintThreeParameter_InsufficientClaimable() (gas: 293343)
[PASS] test_MintThreeParameter_InvalidController() (gas: 245170)
[PASS] test_MintThreeParameter_PartialClaim() (gas: 519404)
[PASS] test_MintThreeParameter_RevertWhen_ReceiverDiffersFromController() (gas: 244181)
[PASS] test_MintThreeParameter_Success() (gas: 288238)
[PASS] test_MintThreeParameter_Success_NonInitial() (gas: 523472)
[PASS] test_MintThreeParameter_ZeroReceiver() (gas: 244733)
[PASS] test_MintThreeParameter_ZeroShares() (gas: 18505)
[PASS] test_Mint_TwoParameter_InsufficientClaimable() (gas: 31435)
[PASS] test_Mint_TwoParameter_RevertWhen_ReceiverDiffersFromSender() (gas: 243453)
[PASS] test_Mint_TwoParameter_Success() (gas: 285194)
[PASS] test_Mint_TwoParameter_Success_NonInitial() (gas: 520289)
[PASS] test_Mint_TwoParameter_ZeroReceiver() (gas: 244644)
[PASS] test_PendingDepositRequest_AfterSettlement() (gas: 242833)
[PASS] test_PendingDepositRequest_NoRequest() (gas: 19119)
[PASS] test_PendingDepositRequest_WithRequest() (gas: 116393)
[PASS] test_PreviewDeposit_Reverts() (gas: 15113)
[PASS] test_PreviewMint_Reverts() (gas: 15795)
[PASS] test_PushDepositShares_RevertsWhen_CallerNotKeeper() (gas: 247543)
[PASS] test_PushDepositShares_RevertsWhen_InsufficientClaimableAssets() (gas: 291634)
[PASS] test_PushDepositShares_RevertsWhen_ZeroAddress() (gas: 20543)
[PASS] test_PushDepositShares_RevertsWhen_ZeroAssets() (gas: 18634)
[PASS] test_PushDepositShares_RevertsWhen_ZeroSharesDueToPrice() (gas: 360472)
```

```
[PASS] test_PushDepositShares_Settled_Succeeds() (gas: 289792)
[PASS] test_RequestDeposit_DifferentControllers() (gas: 165465)
[PASS] test_RequestDeposit_FeeOnTransfer_UnsupportedAsset() (gas: 70507085)
[PASS] test_RequestDeposit_InsufficientAllowance() (gas: 108331)
[PASS] test_RequestDeposit_InsufficientBalance() (gas: 42423)
[PASS] test_RequestDeposit_InvalidOwner() (gas: 18458)
[PASS] test_RequestDeposit_MultipleRequests() (gas: 132981)
[PASS] test_RequestDeposit_RevertWhen_BelowMinimumRequestAssets() (gas: 41842)
[PASS] test_RequestDeposit_RevertWhen_ControllerDiffersFromOwner() (gas: 19758)
[PASS] test_RequestDeposit_RevertWhen_ExchangeRateDropsBelowMinimum() (gas: 45924)
[PASS] test_RequestDeposit_SucceedsAtMinimumRequestAssets() (gas: 131510)
[PASS] test_RequestDeposit_SucceedsWhen_ExchangeRateIncreases() (gas: 135926)
[PASS] test_RequestDeposit_Success() (gas: 128715)
[PASS] test_RequestDeposit_ZeroAssets() (gas: 18530)
[PASS] test_RequestDeposit_ZeroController() (gas: 19454)
[PASS] test_SettleDeposits_NoRequests() (gas: 33207)
[PASS] test_SettleDeposits_RevertsWhenExpectedMismatch() (gas: 264264)
[PASS] test_SettleDeposits_SettlesAllWhenZeroPassed() (gas: 257250)
[PASS] test_SettleDeposits_Success() (gas: 371273)
[PASS] test_ShareFunction() (gas: 11266)
[PASS] test_SupportsInterface() (gas: 17558)
[PASS] test_TotalAssets_AfterClaim() (gas: 286849)
[PASS] test_TotalAssets_AfterSettlement() (gas: 244276)
[PASS] test_TotalAssets_Initial() (gas: 23440)
[PASS] test_TotalAssets_WithPendingRequests() (gas: 117927)
[PASS] test_TotalPendingAssets_AccumulatesMultipleControllers() (gas: 162789)
[PASS] test_TotalPendingAssets_AccumulatesSingleController() (gas: 132822)
[PASS] test_TotalPendingAssets_InitialZero() (gas: 13872)
[PASS] test_TotalPendingAssets_ResetOnSettle() (gas: 292533)
Suite result: ok. 90 passed; 0 failed; 0 skipped; finished in 9.26s (3.90s CPU time)

Ran 10 tests for test/scenario/VaultAccountingScenario.t.sol:VaultAccountingScenarioTest
[PASS] test_Claim_ConservationAndCeilBurn() (gas: 290376)
[PASS] test_Claim_LastUnitGuard_NoGhostAssets() (gas: 404329)
[PASS] test_Fairness_SyncOnly_UsingPublicCalls() (gas: 1368206)
[PASS] test_PoolDrainsToZero_DepositAmountGap() (gas: 611320)
[PASS] test_PoolDrainsToZero_ThenNewDeposits_NoLegacyDilution() (gas: 654225)
[PASS] test_Redeem_ReserveExcludesFromTotalAssets() (gas: 511637)
[PASS] test_Stream_UnitsStableWhenPoolNonzero() (gas: 288940)
[PASS] test_Stream_ZerosAssetsAndUnits() (gas: 324540)
[PASS] test_SyncEpoch_ClearsStaleUnits_OnEraAdvance() (gas: 7002092)
[PASS] test_View_ConservativeSum() (gas: 433499)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 3.89s (19.22ms CPU time)

Ran 2 tests for test/fuzz/RedemptionsFuzz.t.sol:RedemptionsFuzzTest
[PASS] testFuzz_Redeem_PartialAndFull(uint256,uint256) (runs: 2048, : 534256, ~: 534893)
[PASS] testFuzz_Withdraw_PartialAndFull(uint256,uint256) (runs: 2048, : 538573, ~: 539165)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 4.17s (977.20ms CPU time)

Ran 5 tests for test/integration/PremiumIntegration.t.sol:PremiumIntegrationTest
[PASS] test_PremiumStreaming_ContinuousStreaming() (gas: 312505)
[PASS] test_PremiumStreaming_MultiYearCompounding() (gas: 304119)
[PASS] test_PremiumStreaming_MultiYearPlusPartial() (gas: 303926)
[PASS] test_PremiumStreaming_NewDepositsAfterPremium() (gas: 415881)
[PASS] test_PremiumStreaming_WithClaims() (gas: 327459)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 3.23s (3.89ms CPU time)

Ran 1 test for test/fuzz/VaultAccountingFuzz.t.sol:VaultAccountingFuzzTest
[PASS] testFuzz_MultiController_RandomizedSequence_NoDrift(uint96,uint8,uint256) (runs: 2048, : 614166, ~: 614962)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.52s (916.85ms CPU time)

Ran 17 test suites in 9.89s (104.05s CPU time): 224 tests passed, 0 failed, 0 skipped (224 total tests)
```

8.2 Automated Tools

8.2.1 AuditAgent

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.