STATE MIND

StakeWise Core V3

04-03-2026 - 20-04-2026

Table of contents

1. Project brief 3

2. Finding severity breakdown 5

3. Summary of findings 6

4. Conclusion 6

5. Findings report 7

Medium

- Stale meta-vault share price allows depositors to capture sub-vault rewards 7
- SubVaultsRegistry is non-upgradeable despite inheriting UUPS upgradeability 8
- MetaVault's redeem assets calculation does not account for sub-vault LTV 9

Informational

- Redundant SLOAD of metaVault in SubVaultsRegistry.enterSubVaultsExitQueue() 9
- Misleading "SLOAD to memory" comments 10
- Missing checks for sub-vault capacity 10
- Inconsistent exit position creation threshold between SubVaultsRegistry and VaultEnterExit 11
- MetaVault deposits blocks after two rounds of update rewards when no sub-vaults are registered 11
- GnoMetaVault initialization inverts the deposit token-transfer order 12
- EthMetaVaultFactory.createVault() does not validate security deposit amount 12
- KeeperRewards.harvest() does not verify vault collateralization 12
- Potential underflow in SubVaultsRegistry.redeemSubVaultsAssets() when subtracting redeemed assets 13
- Nonce synchronization is blocked when a sub vault is a non-upgraded meta vault 14
- Gas optimisations 15

	Optimisation of OsTokenRedeemer.getExitQueueMissingAssets() returning early when queued shares are zero	15
	OsTokenRedeemer.getExitQueueMissingAssets() incorrectly accounts for contract balance	16
	Additional sanity check for validation of the meta vault in OsTokenRedeemer.redeemSubVaultOsToken()	16
	NodesManager.claimExitedAssets() does not enforce vault state freshness for the balance ratio check	17
	Missing zero address checks in NodesManager setters	17
	Dust loss in NodesManager.updateOperatorState() penalty calculation due to rounding	18
Informational	User assets are transferred unnecessarily when the swap converts to zero osToken shares	18
	Surplus ETH from validator withdrawal fees is stuck in NodesManager	19
	Early return in OsTokenRedeemer.processExitQueue() may cause loss of processed shares due to state clearing	20
	Penalty donation calls in NodesManager waste gas outside pendingPenalty > 0 blocks	21
	Missing ERC-20 Transfer event in transferOsTokenPositionToEscrow for ERC-20 vault variants	21
	ERC-20 Transfer events inconsistent with balanceOf during exit queue lifecycle	22
	Immediate redemption in NodesManager.enterExitQueue() may revert due to post-exit exchange rate mismatch	22
	Operator penalty share amount depends on the NodesManager.updateOperatorState() call ordering	23
	Operator state nonce not verified in deposit flow	23

1. Project brief

Title	Description
Client	StakeWise
Project name	StakeWise Core V3
Timeline	04-03-2026 – 20-04-2026

Project Log

Date	Commit Hash	Note
04-03-2026	2eed9b3b2a3e30961febe6189f5e549de0ea56f4	Initial Commit
06-04-2026	1ad5e9c53a0e933c62724ea60b91641791671250	Reaudit commit
13-04-2026	6d7465d16e1fc348a98615c0b8c2047b26e755b0	Reaudit commit 2
20-04-2026	6a5e2244d73ea8798d91b76d389de634829299cd	Reaudit commit with donations changes and fixes

Short Overview

StakeWise V3 is a decentralized liquid staking protocol for Ethereum and other EVM-compatible chains, including Gnosis Chain. It lets users stake assets such as ETH and GNO while retaining liquidity through liquid staking tokens and earning staking rewards.

StakeWise uses a network of staking pools operated by different providers. This gives users flexibility in choosing their staking setup. Pools also support on-demand liquidity and DeFi integrations through osToken.

During this engagement, the following features were audited:

- Meta vault refactor. Part of the logic was moved into a new SubVaultsRegistry contract. Redemption functionality was also added to meta vaults to support liquidity creation.
- OsTokenRedeemer changes. These included a refactor of the vault positions redemption functions, the addition of a utility view function, and extra functionality to enable the redemption process in meta vaults.
- NodesManager components. This component introduces permissionless validation into the system. An operator can join by posting a bond and then run validators. A separate EthCommunityVault was also introduced, where the NodesManager address is hardcoded as both the fee recipient and the validator manager.

Project Scope

The audit covered the following files:

- EthMetaVault.sol
- GnoMetaVault.sol
- EthMetaVaultFactory.sol
- GnoMetaVaultFactory.sol
- EthErc20MetaVault.sol
- EthPrivMetaVault.sol
- EthPrivErc20MetaVault.sol
- VaultSubVaults.sol
- SubVaultsRegistry.sol
- SubVaultsRegistryFactory.sol
- OsTokenRedeemer.sol
- EthOsTokenRedeemer.sol
- GnoOsTokenRedeemer.sol
- NodesManager.sol
- EthNodesManager.sol
- EthCommunityVault.sol
- ExitPositions.sol

2. Finding severity breakdown

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

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

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

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

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	0 (0 fixed, 0 acknowledged)
Medium	3 (1 fixed, 2 acknowledged)
Informational	26 (18 fixed, 8 acknowledged)
Total	29 (19 fixed, 10 acknowledged)

4. Conclusion

During the audit of the codebase, 29 issues were found in total:

- 3 medium severity issues (1 fixed, 2 acknowledged)
- 26 informational severity issues (18 fixed, 8 acknowledged)

The final reviewed commit: 6a5e2244d73ea8798d91b76d389de634829299cd

5. Findings report

MEDIUM-01	Stale meta-vault share price allows depositors to capture sub-vault rewards	Acknowledged

Description

Lines:

- SubVaultsRegistry.sol#L310-L317

SubVaultsRegistry.isStateUpdateRequired() uses a **+1** tolerance when comparing the meta-vault's nonce against the global keeper nonce:

```
return subVaultsRewardsNonce + 1 < currentNonce;
```

When the keeper publishes nonce **N** and **subVaultsRewardsNonce = N-1**, this returns **false** - the meta-vault appears up-to-date. However, sub-vaults can already be harvested to nonce **N** (their **_totalAssets** updated with new rewards), while the meta-vault's **_totalAssets** remains stale because **VaultSubVaults.updateState()** has not yet been called.

Since **VaultEnterExit._deposit()** relies on **_checkHarvested()**, which delegates to **isStateUpdateRequired()**, deposits into the meta-vault are accepted at the stale share price - before sub-vault rewards are reflected in the meta-vault's **_totalAssets**.

Impact: An attacker can exploit the window between sub-vault harvests and the meta-vault's own state update to buy cheap meta-vault shares and capture a portion of the sub-vault rewards:

1. Keeper publishes new rewards (global nonce **10 → 11**).
2. Attacker calls **updateState()** on all (or only profitable) sub-vaults — their **_totalAssets** increases, reflecting new rewards. Each sub-vault's nonce becomes **11**.
3. **SubVaultsRegistry.isStateUpdateRequired()**: **subVaultsRewardsNonce (10) + 1 < 11** → **false**. The meta-vault reports no update is required.
4. **VaultEnterExit._deposit()** calls **_checkHarvested()** → passes. The attacker deposits into the meta-vault at the stale share price (**_totalAssets** does not yet include the sub-vault reward deltas).
5. Attacker calls **metaVault.updateState()** → **SubVaultsRegistry.harvestSubVaultsAssets()** → **_syncRewardsNonce()** sees all sub-vaults at nonce **11 > subVaultsRewardsNonce (10)** → returns **true** → recalculates **subVaultsTotalAssets** from actual sub-vault balances → **_totalAssets** increases.
6. The attacker's shares are now worth more. They captured a portion of the rewards that should have accrued exclusively to existing meta-vault shareholders.

Example: meta-vault has one sub-vault (vault0) with **sharesToAssets = 1.0**. Meta-vault: **totalAssets = 50**, **totalShares = 50**, **vault0shares = 50**. After **updateState()** of vault0: **sharesToAssets = 1.2**, metaVault pending **totalAssetsDelta = 10**.
Attacker **depositAssets = 60**.
If deposit after **updateState()**: meta-vault **sharesToAssets = 1.2** at deposit time -> attacker gets **60 / 1.2 = 50** shares. Meta-vault: **totalAssets = 120**, **totalShares = 100**, **sharesToAssets = 1.2** - existing shareholders are not diluted.
Without **updateState()**: attacker gets **60 / 1.0 = 60** shares at the stale price. After **updateState**, meta-vault **sharesToAssets = 120 / 110 ≈ 1.09** instead of **1.2**. The attacker extracted ~5.4 assets worth of rewards from existing shareholders.

Recommendation

We recommend that **SubVaultsRegistry.isStateUpdateRequired()** checks whether any sub-vault's nonce is ahead of **subVaultsRewardsNonce**, rather than relying solely on the **+1** tolerance against the global keeper nonce. This ensures **_checkHarvested()** blocks deposits until the meta-vault has harvested the sub-vault reward deltas via **updateState()**.

Client's comments

> Acknowledged. The Keeper updates rewards every 12 hours, so in this case, the attacker will earn rewards proportional to their stake over the past 12 hours. As with normal vaults, we aim to minimize cases where user transactions fail due

to an unharvested state. Therefore, after a Keeper update, the vault operator has a 12-hour window to update the state before user transactions begin to fail.
If an attacker deposits into the vault and calls updateState, they would receive a proportional amount of rewards accrued over the past 12 hours. However, this effect also occurs without nonce gaps: when a staker deposits into the vault, there is a delay before the validator becomes active, and during that time the rewards of existing users are diluted anyway. Additionally, the depositor cannot withdraw immediately after calling updateState. They must first enter the exit queue, which takes at least 24 hours (often longer). During this period, the attacker is exposed to the vault's slashing risk, making the strategy economically unattractive.

MEDIUM-02	**SubVaultsRegistry is non-upgradeable despite inheriting UUPS upgradeability**	Acknowledged

Description

Line: SubVaultsRegistry.sol#L36

SubVaultsRegistry inherits **UUPSUpgradeable** and overrides **_authorizeUpgrade()** to restrict the caller to the **metaVault** address:

```
function _authorizeUpgrade(address) internal view override {
    if (msg.sender != metaVault) revert Errors.AccessDenied();
}
```

The **UUPSUpgradeable.upgradeToAndCall()** function calls **_authorizeUpgrade()** and then performs the implementation upgrade. For **SubVaultsRegistry** to be upgraded, the MetaVault must call **upgradeToAndCall()** on the **SubVaultsRegistry** proxy. However, neither **VaultSubVaults** nor any of the MetaVault contracts implementations expose a function that calls **upgradeToAndCall()** on the **SubVaultsRegistry**.
As a result, **SubVaultsRegistry** cannot be upgraded through its intended UUPS mechanism, which may hinder the protocol's ability to patch bugs or introduce improvements to the sub-vaults management logic without a multi-step, coordinated upgrade of the MetaVault itself.

Recommendation

We recommend adding a function to the **VaultSubVaults** module that allows the MetaVault admin to trigger an upgrade of the **SubVaultsRegistry**.

Client's comments

The next version of MetaVault will upgrade the SubVaultsRegistry if necessary. The upgrade flow is as follows: the DAO adds the new MetaVault implementation to the VaultsRegistry; the vault admin calls the upgrade function on the MetaVault; during the MetaVault upgrade, the SubVaultsRegistry contract is also upgraded.

MEDIUM-03	MetaVault's redeem assets calculation does not account for sub-vault LTV	Fixed at: 179385c

Description

Lines:

- SubVaultsRegistry.sol#L933
- SubVaultsRegistry.sol#L951

In **SubVaultsRegistry._processRedeemRequests()**, the **redeemAssets** value is computed as the minimum of the curator's requested amount and the sub-vault's **withdrawableAssets()**. This value is then directly converted to **osTokenShares**, assuming that we can mint **osTokens** using all our collateral.
These **osTokenShares** are then passed to **mintSubVaultOsToken()**, which internally calls **VaultOsToken._mintOsToken()**.
Inside **_mintOsToken()**, the maximum mintable osToken shares are constrained by the sub-vault's LTV factor:

```
function _calcMaxOsTokenShares(uint256 assets) internal view returns (uint256) {
    uint256 maxOsTokenAssets = Math.mulDiv(assets, _osTokenConfig.getConfig(address(this)).ltvPercent, _maxPercent);
    return _osTokenVaultController.convertToShares(maxOsTokenAssets);
}
```

Since **ltvPercent** could be less than **_maxPercent** (1e18), the actual amount of osToken tokens that can be minted against a given asset position may be less than the full 1:1 conversion. As a result, when **_processRedeemRequests()** attempts to mint **osTokenShares** derived from the full **redeemAssets** value without applying the LTV discount, the call to **VaultOsToken._mintOsToken()** may revert with **Errors.LowLtv()** at VaultOsToken.sol#L195, because **position.shares** would exceed **userMaxOsTokenShares**.
This means the redeem flow can fail for amounts that the curator considers valid, as the calculation does not reflect the actual minting capacity of the sub-vault under its LTV constraint.

Recommendation

We recommend adjusting the **redeemAssets** calculation in **SubVaultsRegistry._processRedeemRequests()** to account for the sub-vault's LTV factor.

INFORMATIONAL-01	Redundant SLOAD of metaVault in SubVaultsRegistry.enterSubVaultsExitQueue()	Fixed at: b8395a7

Description

Lines: SubVaultsRegistry.sol#L470-L476
In **SubVaultsRegistry.enterSubVaultsExitQueue()**, **metaVault** is SLOADed twice: first for the access check (**msg.sender != metaVault**), then cached into a local variable (**address _metaVault = metaVault**). The second SLOAD is redundant.

Recommendation

We recommend caching **metaVault** into a local variable before the access check.

INFORMATIONAL-02	Misleading "SLOAD to memory" comments	Fixed at: 1a2ed7a

Description

Lines:

- KeeperRewards.sol#L189
- OsTokenUtils.sol#L49-L50
- SubVaultsRegistry.sol#L311

Several locations are annotated with **// SLOAD to memory** but do not perform a direct SLOAD from the contract's own storage:

1. **Reward storage lastReward = rewards[msg.sender]** - creates a storage pointer, not a memory copy. Gas benchmarking confirms the storage pointer is already cheaper than a memory copy for this packed single-slot struct, so the current code is optimal despite the misleading comment.
2. **IOsTokenConfig.Config memory config = osTokenConfig.getConfig(address(this))** - external STATICCALL, not an SLOAD.
3. **uint256 currentNonce = _getCurrentRewardsNonce()** - calls **IKeeperRewards(_keeper).rewardsNonce()**, an external call.

Recommendation

We recommend updating the comments to reflect the actual operation.

INFORMATIONAL-03	Missing checks for sub-vault capacity	Fixed at: a26680b

Description

Lines: SubVaultsRegistry.sol#L332-L333

SubVaultsRegistry.depositToSubVaults() retrieves deposit allocations from the curator via **ISubVaultsCurator(subVaultsCurator).getDeposits()**.

Each sub-vault may have a capacity limit configured through **_capacity** in **VaultState**. When a deposit is executed, **VaultEnterExit._deposit()** enforces this limit at VaultEnterExit.sol#L128:

```
if (totalAssetsAfter > capacity()) revert Errors.CapacityExceeded();
```

Since a curator may not account for each sub-vault's remaining capacity, it may allocate an amount that exceeds the capacity of one or more sub-vaults. Because **SubVaultsRegistry.depositToSubVaults()** processes all deposits sequentially in a single transaction, a **CapacityExceeded** revert on any individual sub-vault will cause the entire call to revert, preventing deposits to all other sub-vaults as well.

Recommendation

We recommend adding checks for sub-vault capacities in the **SubVaultsRegistry.depositToSubVaults()** function.

INFORMATIONAL-04	Inconsistent exit position creation threshold between SubVaultsRegistry and VaultEnterExit	Fixed at: 5c7e0d0

Description

Line: SubVaultsRegistry.sol#L384

In **SubVaultsRegistry.claimSubVaultsExitedAssets()**, a new sub-vault exit position is pushed when **leftShares > 0**. However, in the subvault's **VaultEnterExit.claimExitedAssets()**, a new exit request position is only created when **leftTickets > 1** based on the condition at VaultEnterExit.sol#L84.

Currently, **VaultEnterExit.calculateExitedAssets()** rounds **leftTickets == 1** down to 0 VaultEnterExit.sol#L62, which prevents a practical mismatch. However, if **leftShares == 1** were ever returned, **SubVaultsRegistry** would push a new sub-vault exit and increment **queuedShares**, while **VaultEnterExit.claimExitedAssets()** would not create a corresponding exit request position in the subvault. This would result in a phantom exit entry tracked by **SubVaultsRegistry**.

Recommendation

We recommend aligning the threshold in **SubVaultsRegistry** with the subvault's logic for defensive consistency.

INFORMATIONAL-05	MetaVault deposits blocks after two rounds of update rewards when no sub-vaults are registered	Fixed at: be518bc

Description

Lines: SubVaultsRegistry.sol#L310-L317

In the **EthMetaVault**, the **deposit()** function calls the internal **VaultEnterExit._deposit()**, which begins with a **_checkHarvested()** call. For MetaVaults, **VaultSubVaults._checkHarvested()** overrides the base implementation and delegates to **SubVaultsRegistry.isStateUpdateRequired()**.

In contrast, for regular vaults, the harvest check goes through **KeeperRewards.isHarvestRequired()**, which explicitly handles the non-collateralized case by returning **false** when **nonce == 0**:

```
function isHarvestRequired(address vault) external view override returns (bool) {
    // vault is considered harvested in case it does not have any validators (nonce = 0)
    // or it is up to 1 rewards update behind
    uint256 nonce = rewards[vault].nonce;
    unchecked {
        // cannot overflow as nonce is uint64
        return nonce != 0 && nonce + 1 < rewardsNonce;
    }
}
```

The **SubVaultsRegistry.isStateUpdateRequired()** has no equivalent check. During initialization, **subVaultsRewardsNonce** is set to the current Keeper **rewardsNonce**. If two or more Keeper reward updates occur before any sub-vault is added, **subVaultsRewardsNonce + 1 < currentNonce** evaluates to **true**, causing **_checkHarvested()** to revert. A MetaVault that has no sub-vaults registered may become unable to accept deposits after two Keeper reward updates, as both **deposit()** and **updateState()** revert until a sub-vault is added.

Recommendation

We recommend adding a collateralization check to **SubVaultsRegistry.isStateUpdateRequired()**. This ensures that when the MetaVault is not collateralized (no sub-vaults), the harvest check does not block deposits, matching the behavior of regular vaults.

INFORMATIONAL-06	GnoMetaVault initialization inverts the deposit token-transfer order	Fixed at: 8523498

Description

Lines: GnoMetaVault.sol#L191-L193
GnoMetaVault.__GnoMetaVault_init() calls **_deposit()** before **token.safeTransferFrom()**, inverting the order used in **GnoMetaVault.deposit()** itself (transfer first, then **_deposit()**).

Recommendation

We recommend reordering the operations in **GnoMetaVault.__GnoMetaVault_init()** to transfer tokens before calling **_deposit()**, consistent with the **GnoMetaVault.deposit()** flow itself.

INFORMATIONAL-07	EthMetaVaultFactory.createVault() does not validate security deposit amount	Acknowledged

Description

Lines: EthMetaVaultFactory.sol#L35-L46
EthMetaVaultFactory.createVault() forwards **msg.value** to **IEthMetaVault(vault).initialize{value: msg.value}(params)** without validating that **msg.value >= _securityDeposit**. The validation happens later inside **EthMetaVault.__EthMetaVault_init()**, but a failed vault initialization wastes gas on proxy deployment and registry addition that must be reverted.
The **GnoMetaVaultFactory** does not have this issue because it controls the exact **_securityDeposit** amount via **safeTransferFrom**.

Recommendation

We recommend adding an early **msg.value** check in **EthMetaVaultFactory.createVault()** before deploying the proxy, to fail fast with a clear error.

Client's comments

Acknowledged. We want to keep it consistent with the other factories, so as long as the transaction reverts, it should be fine.

INFORMATIONAL-08	KeeperRewards.harvest() does not verify vault collateralization	Acknowledged

Description

Line: KeeperRewards.sol#L166
The **KeeperRewards.harvest()** function validates that **msg.sender** is a registered vault via **_vaultsRegistry.vaults(msg.sender)**, but does not verify that the vault is collateralized before proceeding with Merkle proof verification and reward calculation. A non-collateralized vault (**rewards[vault].nonce == 0**) would pass the registry check and proceed through the proof verification logic unnecessarily. Adding a collateralization check will act as a sanity check.

Recommendation

We recommend adding an explicit collateralization check alongside the existing registry validation to provide a cleaner early revert for non-collateralized vaults.

Client's comments

Acknowledged. The Keeper contract is immutable and already deployed.

INFORMATIONAL-09	Potential underflow in SubVaultsRegistry.redeemSubVaultsAssets() when subtracting redeemed assets	Acknowledged

Description

Line: SubVaultsRegistry.sol#L598

In **SubVaultsRegistry.redeemSubVaultsAssets()**, after processing redemptions, **totalRedeemedAssets** is subtracted directly from **subVaultsTotalAssets**.

The **SubVaultsRegistry._checkHarvested()** guard invoked in **SubVaultsRegistry._calculateSubVaultsRedemptions()** only reverts when the nonce gap exceeds 1, as seen in **SubVaultsRegistry.isStateUpdateRequired()**.

This means the meta vault is permitted to operate while its **subVaultsTotalAssets** is 1 nonce behind the current rewards nonce. During that window, individual sub vaults may have already been updated to the latest nonce. For example, in the case of positive reward deltas, their actual asset balances will be higher than the stale **subVaultsTotalAssets**. When **SubVaultsRegistry._processRedeemRequests()** redeems shares from these updated sub vaults, the returned **totalRedeemedAssets** is calculated using the updated valuation of sub vaults total assets. Since **subVaultsTotalAssets** still holds the previous, lower value, the subtraction on SubVaultsRegistry.sol#L598 may underflow, causing a revert even without a full redemption.

The contract already handles an analogous situation in **SubVaultsRegistry.claimSubVaultsExitedAssets()**, where **exitedAssets** is checked against **_subVaultsTotalAssets** and clamped to zero when it exceeds the tracked total, with the excess routed to **_unaccountedExitedAssets**.

Recommendation

We recommend applying the same overflow-safe pattern used in **SubVaultsRegistry.claimSubVaultsExitedAssets()** to the subtraction in **SubVaultsRegistry.redeemSubVaultsAssets()**.

Client's comments

We will keep it as is, since the meta vault can be updated to the latest state before calling redeemSubVaultsAssets.

INFORMATIONAL-10	Nonce synchronization is blocked when a sub vault is a non-upgraded meta vault	Acknowledged

Description

Lines:

- SubVaultsRegistry.sol#L899-L906

SubVaultsRegistry._getSubVaultRewardsNonce() uses a try-catch pattern on **VaultSubVaults.subVaultsRegistry()** to distinguish meta vaults from regular vaults. If the call succeeds, the sub-vault is treated as a meta vault, and its nonce is fetched from its **SubVaultsRegistry**. If the call reverts, the sub-vault is assumed to be a regular vault, and the fallback queries **KeeperRewards** directly.

This logic does not account for the case where a sub-vault is a meta vault that has not been upgraded to the version that introduces **SubVaultsRegistry**. An old meta vault lacks the **VaultSubVaults.subVaultsRegistry()** function, so the call reverts and the fallback path is taken. However, meta vaults are not harvested through the Keeper: their **rewards[vault].nonce** in **KeeperRewards** remains zero. As a result, **SubVaultsRegistry._getSubVaultRewardsNonce()** returns zero for such a sub-vault. In **SubVaultsRegistry._syncRewardsNonce()**, this causes permanent revert of the nonce synchronization for the parent meta vault.

If an upgraded meta vault contains a non-upgraded meta vault in its sub-vault set, nonce synchronization may be permanently blocked, preventing state updates and reward distribution for the parent meta vault.

Recommendation

We recommend validating during the migration that any sub vault, which is itself a meta vault, has already been upgraded to the new version that exposes **VaultSubVaults.subVaultsRegistry()**. Alternatively, **SubVaultsRegistry._isSubVaultCollateralized()** and **SubVaultsRegistry._getSubVaultRewardsNonce()** could be extended to handle the case of a non-upgraded meta vault explicitly.

Client's comments

> All current meta-vaults will be upgraded to the new version, as the MetaVault factory is currently permissioned and the meta-vault admins are known entities.

INFORMATIONAL-11	Gas optimisations	Fixed at: 0e9a484

Description

Lines:

- OsTokenRedeemer.sol#L197-L199
- OsTokenRedeemer.sol#L340-L344
- SubVaultsRegistry.sol#L450-L452

1. Redundant full struct copy in **OsTokenRedeemer.setRedeemablePositions()**.

In the **OsTokenRedeemer.setRedeemablePositions()** function, the entire **_redeemablePositions** struct is loaded into memory only to check the **merkleRoot** field:

```
// SLOAD to memory
RedeemablePositions memory currentPositions = _redeemablePositions;
if (newPositions.merkleRoot == currentPositions.merkleRoot) {
    revert Errors.ValueNotChanged();
}
```

RedeemablePositions contains both a **bytes32 merkleRoot** and a **string ipfsHash**. Loading this struct into memory allocates a memory buffer and copies both fields, whereas only **merkleRoot** is used in the comparison.

2. Inside the first loop of **OsTokenRedeemer.redeemOsTokenPositions()**, the calculated value is stored in a local variable **sharesToRedeem** but is immediately assigned to **position.sharesToRedeem**. All subsequent operations within the loop read directly from **position.sharesToRedeem** rather than utilizing the local variable. This results in redundant memory reads, rather than using the value cached on the stack.
3. In **SubVaultsRegistry.harvestSubVaultsAssets()**, the **balances** array is declared and populated via **SubVaultsRegistry._getSubVaultsBalances()**, but it is never read after the assignment. The function only uses the second return value, **newSubVaultsTotalAssets**.

Recommendation

1. We recommend accessing **_redeemablePositions.merkleRoot** directly from storage instead of copying the entire struct into memory.
2. We recommend keeping the **sharesToRedeem** local variable to cache the calculation result for the loop execution, assigning it to **position.sharesToRedeem** only at the end of the cycle.
3. We recommend discarding the first return value of **SubVaultsRegistry._getSubVaultsBalances()** call at SubVaultsRegistry.sol#L452.

INFORMATIONAL-12	Optimisation of OsTokenRedeemer.getExitQueueMissingAssets() returning early when queued shares are zero	Fixed at: 7258b0e

Description

Line: OsTokenRedeemer.sol#L176

In **OsTokenRedeemer.getExitQueueMissingAssets()**, the function checks whether **totalTickets >= targetCumulativeTickets** to determine if all tickets have already been processed and returns zero early. When there are no queued shares, there is nothing pending to be processed, so the function could also return zero immediately.

Currently, when **_queuedShares** is zero and **totalTickets < targetCumulativeTickets**, the function proceeds to compute **Math.min(totalTicketsToCover, 0)** at OsTokenRedeemer.sol#L184, which evaluates to zero.

Recommendation

We recommend checking if **_queuedShares** is zero in the early return condition to avoid redundant computation.

INFORMATIONAL-13	OsTokenRedeemer.getExitQueueMissingAssets() incorrectly accounts for contract balance	Fixed at: 263429d

Description

Lines: OsTokenRedeemer.sol#L187-L188

The **OsTokenRedeemer.getExitQueueMissingAssets()** function computes **availableAssets** as the contract's asset balance minus **unclaimedAssets**, and subtracts it from the calculated **missingAssets**. This assumes that any excess balance held by the redeemer contract will eventually be used to satisfy exit queue positions. However, when **OsTokenRedeemer.processExitQueue()** creates a checkpoint, it only accounts for **processedAssets**, which is the sum of **redeemedAssets** and **swappedAssets**. Assets that were directly transferred or donated to the contract are not tracked by either of these accumulators and, therefore, will never be included in any checkpoint pushed to the exit queue. As a result, such assets will remain idle on the contract while **OsTokenRedeemer.getExitQueueMissingAssets()** incorrectly reports fewer missing assets than are actually needed.

Recommendation

We recommend removing the **availableAssets** adjustment and returning **missingAssets** directly.

INFORMATIONAL-14	Additional sanity check for validation of the meta vault in OsTokenRedeemer.redeemSubVaultOsToken()	Acknowledged

Description

Line: OsTokenRedeemer.sol#L289

The **OsTokenRedeemer.redeemSubVaultOsToken()** function validates that **subVault** is registered in the vaults registry and that **msg.sender** is a meta vault via the **_isMetaVault()** helper. However, the **_isMetaVault()** check only verifies that the caller is a registered vault and that it responds to the **subVaultsRegistry()** selector, without verifying that the caller's underlying implementation is an approved one via **VaultsRegistry.vaultImpls()**.

The intended flow is that **OsTokenRedeemer.redeemSubVaultOsToken()** is called by a meta vault as part of the **SubVaultsRegistry.redeemSubVaultsAssets()** execution path, initiated through **OsTokenRedeemer.redeemSubVaultsAssets()**. In a case of the potential bug in the upgrade process where a meta vault is upgraded to an unapproved implementation, that meta vault could call **OsTokenRedeemer.redeemSubVaultOsToken()** directly, bypassing the intended **OsTokenRedeemer.redeemSubVaultsAssets** flow. The current **_isMetaVault()** check would still pass since the vault remains registered and still exposes the **subVaultsRegistry()** selector.

A meta vault running an unapproved implementation may invoke **OsTokenRedeemer.redeemSubVaultOsToken()** outside the intended redemption flow, potentially redeeming osToken shares from sub vaults without going through the proper **redeemSubVaultsAssets** path.

Recommendation

We recommend adding a sanity check that validates the meta vault's implementation is approved in the **VaultsRegistry**. This can be done by querying the caller's implementation address via **IVaultVersion(msg.sender).implementation()** and verifying it against **_vaultsRegistry.vaultImpls()**.

Client's comments

> The _authorizeUpgrade already validates that the new implementation is approved in vaultImpls before allowing the upgrade. A meta vault cannot be upgraded to an unapproved implementation through the normal upgrade path.

INFORMATIONAL-15	NodesManager.claimExitedAssets() does not enforce vault state freshness for the balance ratio check	Fixed at: 5e8d7ab

Description

Lines: NodesManager.sol#L377-L383

When an operator calls **NodesManager.claimExitedAssets()**, the function checks that the operator's balance ratio meets the minimum threshold:

```
uint256 balanceAssets = IVaultState(vault).convertToAssets(operatorState.balanceShares);
if (
    operatorState.totalAssets > 0
        && Math.mulDiv(balanceAssets, _wad, operatorState.totalAssets) < uint256(minBalancePercent) * 1e14
) {
    revert Errors.LowBalance();
}
```

The **VaultState.convertToAssets()** call relies on the vault's current exchange rate, which is only updated upon harvesting. The function does not call **_keeper.isHarvestRequired(vault)** or perform any other freshness validation on the vault state. This creates an asymmetry: an operator must have a reasonably fresh vault to update their state, but can claim exited assets against an arbitrarily stale vault rate.

In a scenario where a slashing event has been recorded in the keeper but not yet harvested, **VaultState.convertToAssets()** returns an inflated value, allowing an operator below the threshold to pass the check and withdraw assets.

POC: **NodesManagerPocTest.test_claimExitedAssets_staleVaultBypassesLowBalance()**

Recommendation

We recommend adding a strict vault freshness check in **NodesManager.claimExitedAssets()**, requiring the vault to be fully synced with the latest keeper nonce. Given that **NodesManager** operates a community vault with multiple independent operators, a stricter freshness requirement is warranted for a check that gates asset withdrawal.

INFORMATIONAL-16	Missing zero address checks in NodesManager setters	Acknowledged

Description

Lines:

- NodesManager.sol#L128-L132
- NodesManager.sol#L135-L139

NodesManager.setWithdrawalsManager() and **NodesManager.setValidatorsManager()** allow setting the respective addresses to **address(0)**. This would break any functionality that depends on a valid withdrawals or validators manager address.

Recommendation

We recommend adding zero address checks in both functions.

Client's comments

> We want to retain the ability to reset the validators manager and the withdrawals manager by setting them to the zero address.

INFORMATIONAL-17	Dust loss in NodesManager.updateOperatorState() penalty calculation due to rounding	Fixed at: 6a5e224

Description

Lines: NodesManager.sol#L232-L247

When the operator has sufficient shares to cover the full penalty (**totalPenaltyShares <= availableShares**), the code zeroes out **pendingPenaltyAssets[operator]**, assuming the penalty is fully applied. However, **convertToShares** rounds down, so the donated shares may be worth less than **totalPenaltyAssets**.

Example: **totalPenaltyAssets = 1101**, share rate = 1.1 assets/share:

- **totalPenaltyShares = convertToShares(1101) = 1000** (rounded down from 1000.9)
- 1000 shares are donated, worth **1000 * 1.1 = 1100** assets
- **pendingPenaltyAssets** is set to 0, but only 1100 out of 1101 penalty assets were actually applied
- 1 wei of penalty is permanently lost

This dust accumulates over repeated penalty applications.

Recommendation

We recommend incrementing **totalPenaltyShares** by 1 to round the penalty up, ensuring the protocol is always on the winning side.

INFORMATIONAL-18	User assets are transferred unnecessarily when the swap converts to zero osToken shares	Fixed at: f9d7156

Description

Lines: OsTokenRedeemer.sol#L447-L449

In **OsTokenRedeemer._swapAssetsToOsTokenShares()**, the function returns **0** early when **_osTokenVaultController.convertToShares(assets)** evaluates to zero. However, both concrete implementations: **EthOsTokenRedeemer.swapAssetsToOsTokenShares()** and **GnoOsTokenRedeemer.swapAssetsToOsTokenShares()**, transfer the caller's assets into the contract before invoking **OsTokenRedeemer._swapAssetsToOsTokenShares()**. When the conversion outputs zero shares, the function silently returns without reverting or refunding, leaving the transferred assets stranded in the contract with no recovery mechanism for the caller.

Recommendation

We recommend reverting instead of returning zero so that the transaction rolls back entirely, preventing asset transfers that would not result in any osToken shares.

INFORMATIONAL-19	Surplus ETH from validator withdrawal fees is stuck in NodesManager	Fixed at: 03df924

Description

Lines:

- NodesManager.sol#L321-L324
- ValidatorUtils.sol#L217-L219

When the **withdrawalsManager** calls **NodesManager.withdrawValidators()**, it forwards ETH via **msg.value** to cover the withdrawal fees charged by the withdrawal contract. The call is delegated to the vault, which internally executes **ValidatorUtils.withdrawValidators()**. This library function pays the exact fee for each validator and, if there is any surplus, refunds it to **msg.sender**.

Since the library executes in the vault's context, **msg.sender** is the **NodesManager** contract. The refund is therefore sent from the vault to **NodesManager**. There is no mechanism in **NodesManager** to withdraw or forward the refunded surplus.

Recommendation

We recommend adding a mechanism to either forward the surplus back to the original caller or precisely calculate the necessary fee before the actual **NodesManager.withdrawValidators()** call.

INFORMATIONAL-20	Early return in OsTokenRedeemer.processExitQueue() may cause loss of processed shares due to state clearing	Fixed at: 27261c3

Description

Lines:

- OsTokenRedeemer.sol#L400-L423
- OsTokenRedeemer.sol#L125-L130

In **OsTokenRedeemer.processExitQueue()**, the accumulated **swappedShares**, **swappedAssets**, **redeemedShares**, and **redeemedAssets** are zeroed out before the early return check on OsTokenRedeemer.sol#L413. When **processedShares > 0** but **processedAssets == 0**, the function clears all four state variables and returns without creating an exit queue checkpoint. Because **queuedShares** was already reduced during the preceding calls to **OsTokenRedeemer.redeemOsTokenPositions()** or **OsTokenRedeemer._swapAssetsToOsTokenShares()**, the shares tracked by **processedShares** are effectively lost, so they no longer exist in **queuedShares**, **redeemedShares**, or **swappedShares**, and no checkpoint records them in the exit queue.
If a redemption or swap results in non-zero **processedShares** but zero **processedAssets**, calling **OsTokenRedeemer.processExitQueue()** permanently destroys those shares from the accounting. Users who entered the exit queue and had their **queuedShares** consumed will never receive a corresponding checkpoint, resulting in their exit requests becoming unclaimable or a break in the accounting (situation with two exit requests on the same position ticket may become possible).
Additionally, **OsTokenRedeemer.processExitQueue()** is inconsistent with the **OsTokenRedeemer.canProcessExitQueue()** logic. The view function checks whether shares are non-zero (**swappedShares > 0 || redeemedShares > 0**), while **processExitQueue()** requires both shares and assets to be non-zero.

Recommendation

We recommend returning early only when **processedShares** is zero. When **processedShares > 0** but **processedAssets == 0**, the function should still clear the state variables and create a checkpoint to maintain correct accounting. This will also make **OsTokenRedeemer.processExitQueue()** consistent with **OsTokenRedeemer.canProcessExitQueue()**.
In case of a checkpoint with zero **processedAssets**, during the claim revert is possible because in that case **exitedAssets** may be zero at OsTokenRedeemer.sol#L247. So to avoid revert, we also recommend considering this place during the fix.

Client's comments

> Don't want to create zero assets checkpoints as they will collide with the special-case logic in ExitQueue.calculateExitedAssets at lines 105-113 that assumes only the single V2 to V3 migration checkpoint can have 0 assets.

INFORMATIONAL-21	Penalty donation calls in NodesManager waste gas outside pendingPenalty > 0 blocks	Fixed at: e659b6c

Description

Lines:

- NodesManager.sol#L404-L421
- NodesManager.sol#L443-L461

1. Redundant **penaltyDeducted > 0** check in **NodesManager.claimExitedAssets()**.

penaltyDeducted is declared as **uint256** (default 0) and is only assigned a non-zero value inside **if (pendingPenalty > 0)**. Within that block, **penaltyDeducted** is set to either **exitedAssets** (which is guaranteed > 0 by the check on line 388) or **pendingPenalty** (which is > 0 by the enclosing condition). Therefore, **penaltyDeducted > 0** is always true whenever **pendingPenalty > 0**, and always false otherwise – the separate check is fully redundant.

2. Nestable **penaltyShares > 0** check in **NodesManager._deposit()**.

In **NodesManager._deposit()**, **penaltyShares** is computed via **VaultState.convertToShares(pendingPenalty)**, which uses **Math.mulDiv** with **Rounding.Floor** and can return 0 for small **pendingPenalty** values. Therefore, unlike the first case, the **penaltyShares > 0** check is not redundant — it guards against calling **VaultState.donateShares(0)**. However, since **penaltyShares** is only assigned a non-zero value inside the **if (pendingPenalty > 0)** block, the check can still be nested inside it, avoiding the unnecessary gas spending.

Recommendation

We recommend moving the donation calls inside the respective **if (pendingPenalty > 0)** blocks.
In **NodesManager.claimExitedAssets()**, the **penaltyDeducted > 0** check can be removed entirely. In **NodesManager._deposit()**, the **penaltyShares > 0** guard must be preserved (since **VaultState.convertToShares()** can return 0), but nested inside.

INFORMATIONAL-22	Missing ERC-20 Transfer event in transferOsTokenPositionToEscrow for ERC-20 vault variants	Fixed at: fdcbe55

Description

Lines:

- VaultOsToken.sol#L129
- EthErc20MetaVault.sol#L151-L160

VaultOsToken.transferOsTokenPositionToEscrow calls **super.enterExitQueue(exitShares, address(_osTokenVaultEscrow))** at line 129. **super** from **VaultOsToken** resolves to **VaultEnterExit.enterExitQueue**, skipping the **EthErc20MetaVault.enterExitQueue** override that emits **Transfer(msg.sender, address(this), shares)**.
As a result, **_balances[msg.sender]** is decremented (in **VaultEnterExit._enterExitQueue** at line 184), but no ERC-20 **Transfer** event is emitted. Off-chain indexers and wallets tracking balances via **Transfer** events will show a stale (inflated) balance for the user.
The same issue applies to all ERC-20 vault variants where **enterExitQueue()** is overridden to emit **Transfer**: **EthErc20Vault**, **EthPrivErc20MetaVault**, **EthPrivErc20Vault**, and Gnosis equivalents (**GnoErc20MetaVault**, etc.).

Recommendation

We recommend overriding **transferOsTokenPositionToEscrow()** in each ERC-20 vault variant to call **super.transferOsTokenPositionToEscrow()** and emit the missing **Transfer** event.

INFORMATIONAL-23	ERC-20 Transfer events inconsistent with balanceOf during exit queue lifecycle	Fixed at: 7528c8e

Description

Lines:

- EthErc20MetaVault.sol#L157-L160
- VaultEnterExit.sol#L184-L188
- VaultToken.sol#L43

In ERC-20 vault variants, **EthErc20MetaVault.enterExitQueue()** emits **Transfer(msg.sender, address(this), shares)**, and later **VaultToken._updateExitQueue()** emits **Transfer(address(this), address(0), burnedShares)**. Together, the Transfer events describe shares moving from the user to the vault contract, then being burned from the vault contract.
However, **_balances[address(this)]** is never updated in either step. In **VaultEnterExit._enterExitQueue()**, shares go from **_balances[user]** directly into **_queuedShares**, bypassing **_balances[address(this)]** entirely. This means:

- **balanceOf(address(this))** does not reflect the Transfer events
- **sum(balanceOf(all_addresses)) < totalSupply()** while shares are queued

The same pattern exists in **EthErc20Vault**, **EthPrivErc20MetaVault**, and **EthPrivErc20Vault**.

Recommendation

We recommend documenting this behavior for integrators and off-chain indexers, explicitly noting that **balanceOf(address(this))** does not track queued exit shares and should not be used to derive vault-held token amounts.

INFORMATIONAL-24	Immediate redemption in NodesManager.enterExitQueue() may revert due to post-exit exchange rate mismatch	Acknowledged

Description

Lines: NodesManager.sol#L339-L344
In **NodesManager.enterExitQueue()**, when the vault returns **positionTicket == type(uint256).max** (immediate redemption), two independent **IVaultState.convertToAssets()** calculations occur:

1. Vault (pre-exit): computes **assets = floor(shares * totalAssets / totalShares)**, reduces **_totalAssets** and **_totalShares**, transfers **assets** ETH to **NodesManager**.
2. NodesManager (post-exit): recomputes **assets = floor(shares * totalAssets / totalShares)** where **totalAssets** and **totalShares** are already reduced.

After removing **assets** from **totalAssets** and **shares** from **totalShares**, the remaining ratio can increase due to floor-division rounding. When it does, NodesManager's recalculated amount exceeds the ETH actually received, and **NodesManager._transferAssets()** reverts.
Immediate redemption only occurs when the vault is not collateralized (**nonce == 0**). Normally, the pre-collateralization rate is 1:1 (deposits add equal assets and shares), making rounding mismatch impossible. However, if **IVaultState.donateShares()** is called before the first harvest, it burns shares without reducing assets, creating a rate != 1:1 while the vault remains non-collateralized.

Recommendation

We recommend replacing the independent **IVaultState(vault).convertToAssets()** recalculation with tracking the actual ETH received.

Client's comments

> If that corner-case scenario occurs, you can transfer 1 wei to the NodesManager and call NodesManager.enterExitQueue(). We do not expect the CommunityVault to remain in an undercollateralized state for long.

INFORMATIONAL-25	Operator penalty share amount depends on the NodesManager.updateOperatorState() call ordering	Acknowledged

Description

Line: NodesManager.sol#L195

In **NodesManager.updateOperatorState()**, penalty processing follows this sequence:

1. Convert penalty assets to shares: **IVaultState.convertToShares(totalPenaltyAssets)** = **floor(assets * totalShares / totalAssets)**.
2. Deduct penalty shares from the operator's balance.
3. Call **IVaultState.donateShares(penaltySharesToDonate)** — burns shares, **totalShares** decreases while **totalAssets** stays constant.

After step 3, the exchange rate increases: **totalAssets / totalShares** is now higher. Any subsequent **convertToShares()** call for the same penalty amount returns fewer shares.

Since **NodesManager.updateOperatorState()** has no access control on the caller, anyone can invoke it for any operator. An operator with a pending penalty can call **NodesManager.updateOperatorState()** for all other penalized operators first, then process their own state last. Each preceding **IVaultState.donateShares()** increases the rate, reducing the share-denominated penalty for the last-processed operator.

Recommendation

We recommend considering redesigning the penalty application flow in **NodesManager.updateOperatorState()** so that all operators' penalties are evaluated fairly, regardless of call order.

Client's comments

Acknowledged. Processing order does affect the exact share amount. However, we consider this a non-issue because the savings scale as operatorPenalty * otherPenalties / vaultTVL (dust with realistic vault sizes), and the benefit from donated shares is socialized across all vault shareholders. The attacker captures only their pro-rata portion, making the net edge negligible.

INFORMATIONAL-26	Operator state nonce not verified in deposit flow	Fixed at: 173fe69

Description

Line: NodesManager.sol#L436

The **NodesManager._deposit()** function applies pending penalties from **pendingPenaltyAssets[msg.sender]** against newly deposited shares, but does not verify that the calling operator's state has been synced with the current global nonce. However, **NodesManager._deposit()** reads **pendingPenaltyAssets[msg.sender]** directly without ensuring the operator has called **NodesManager.updateOperatorState()** against the latest state root. If a new state update has occurred that increases the operator's cumulative penalty, the **pendingPenaltyAssets** value used during the deposit will be stale, and the operator may repay less penalty than actually owed.

Recommendation

We recommend adding the nonce synchronization check at the beginning of **NodesManager._deposit()** for the operators that have a non-zero **totalAssets** value. This will provide an up-to-date accounting of the penalties and consider the case for the first deposit or reentering the vault.

STATE
MIND