BAIL security

Symbiotic Rewards

FINAL REPORT

November ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, 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. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Symbiotic – Rewards - Audit Report
Website	symbiotic.fi
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/symbioticfi/rewards-v2/tree/472467dec4a935a7ac8f49fd57ea54bea359b0b2/src/contracts
Resolution 1	https://github.com/symbioticfi/rewards-v2/tree/11fedf69e0d9e707626e9a4943efc07727c3876f/src/contracts
Resolution 2	https://github.com/symbioticfi/rewards-v2/tree/15762166b8f96bef1c1ddec6ceb458208b61c668/src/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no changes made)	Failed resolution	Open
High	4	3		1		
Medium	6			6		
Low	5	1		4		
Informational	16	1		15		
Governance	1			1		
Total	32	5		27		

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Global

Below we will explain the inheritance status of all deployed contracts:

a) Rewards: The Rewards module consists of:
   i) ProtocolFees
   ii) CumulativeMerkleRewards
   iii) VaultSnapshotRewards
   iv) Rewards

b) Registry: The registry module has two standalone contracts
   i) CuratorRegistry
   ii) FeeRegistry

Issue_01	Governance Privilege
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior and result in withdrawal of all funds.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_02	Rebasing tokens are unsupported
Severity	**Informational**
Description	The contract does not currently support rebasing tokens, as the accounting is internalized via storage transitions instead of the balanceOf mechanism. Any rebasing rewards will thus remain locked.
Recommendations	Consider implementing a simple governance withdrawal function.
Comments / Resolution	Acknowledged.

Registries

CuratorRegistry

The CuratorRegistry contract is a simple registry which maps a curator address to its corresponding vault. The setting is done via the setCurator function which allows initially the vault owner to set a curator and once a curator has been set, only the current curator can determine a new curator.

The getCurator function is then consulted during VaultSnapshotRewards.claimCuratorFees.

Privileged Functions

- setCurator

Core Invariants:

INV 1: setCurator can only be called by the latest curator or by the owner of the vault

INV 2: A vault can have different curators at different times

Issue_03	Symbiotic vaults are not guaranteed to have corresponding owners
Severity	**Medium**
Description	The setCurator function expects a Vault's owner to call it during the very first setting: *else if (Ownable(vault).owner() != msg.sender) {* *revert NotAuthorized();* *}* This raises the explicit assumption that each Vault must have an owner, which is not renounced. This assumption is not always true because Vaults **do actually not have a use-case for the owner** in their current state, as they incorporate an AccessControl mechanism: *contract Vault is VaultStorage, MigratableEntity, AccessControlUpgradeable, IVault* The owner is only related to MigratableEntity and the only usage of the owner is the migrate function, which can reasonably be renounced to increase trust. It is even explicitly possible to initialize a Vault without an owner: *if (owner_ != address(0)) {* *__Ownable_init(owner_);* *}* In that scenario, it will remain impossible to set a curator towards a vault which then means no operator/curator fees can ever be set.
Recommendations	Consider implementing a governance function which allows for setting a curator towards a vault.
Comments / Resolution	Acknowledged. If the described scenario ever happens, the Symbiotic team is going to handle this with an implementation upgrade. As of now, they decided to not implement further governance privileges.

Issue_04	Change of curator will forfeit fees to new curator
Severity	**Informational**
Description	Currently, it is possible to change the curator to a new address. This will inherently forfeit all rewards for the old curator and entitles the new curator to claim these. We do not consider this as an issue but it is important to point this out to curators such that they claim their fees before setting a new curator.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_05	Lack of vault validation for setCurator
Severity	**Informational**

Description	Currently, there is no validation that the Vault address is indeed corresponding to a valid entity. While this does not expose any harm, it is still required to point out this minor lack of validation.
	
	Notably, such a validation is existing within VaultSnapshotRewards.distributeVaultSnapshotRewards function:
	
	`if (!IRegistry(VAULT_FACTORY).isEntity(vault)) {`
	`  revert InvalidVault();`
	`}`

Recommendations	Consider either implementing such a check or acknowledging this issue safely.
Comments / Resolution	Resolved.

Issue_06	Locked curator fees in case of address(0) setting
Severity	**Informational**
Description	Currently, there is no sanity check within the setCurator function which prevents a curator being set to address(0). If that ever happens while there are unclaimed fees, these fees will remain essentially locked forever.
Recommendations	Consider implementing an address(0) check.
Comments / Resolution	Acknowledged.

FeeRegistry

The FeeRegistry contract is a simple registry which keeps track of the following fee types:

- Operator Default Fee
- Operator Network Fee
- Curator Default Fee
- Curator Network Fee
-

Appendix: Serialize/Deserialize

The fee-serialization format packs the enabled flag and the numerical fee into a single uint208 by shifting the fee one bit to the left and storing the enable-bit in the least significant position.

Deserialization reverses this: the LSB indicates whether a network-specific fee overrides the default, and the remaining upper bits represent the fee value. This compact encoding is used because the fee entries are stored inside timestamped checkpoint structures, which require a single atomic value per checkpoint and benefit from minimizing storage footprint. The design allows the protocol to efficiently track historical fee changes, cleanly distinguish between “enabled” network-specific overrides and fallback defaults, and maintain consistent ordering within the checkpoint history.

Serialize:

> serialized = (fee << 1) | (enabled ? 1 : 0)

Deserialize:

> enabled = (serialized & 1) > 0
> fee = serialized >> 1

Example:

isEnabled = true
fee = 42

serialize → (42 << 1) | 1 = 84 + 1 = 85 = 0x55
deserialize → (85 & 1) = 1 , 85 >> 1 = 42

Privileged Functions

- setOperatorsFee
- setOperatorsNetworkFee
- setCuratorFee
- setCuratorNetworkFee
- setProtocolFee

Core Invariants:

INV 1: Only the owner can change the protocol fee

INV 2: Only the vault curator can change the curatorNetworkFee

INV 3: Only the vault curator can change the curatorFee

INV 4: Only the vault curator can change the operatorsFee

INV 5: Only the vault curator ca change the operatorsNetworkFee

INV 6: Whenever a network fee for operator/curator is non-existent, it must fetch the default fee

INV 7: ID from protocolFee must be derived as keccak256(REWARDS_FEE_ID, rewardsType, network) or keccak256(REWARDS_FEE_ID, rewardsType)

INV 8: Only network fees/protocol fee are serialized

Issue_07	Curator and operator fee will return zero if vault timestamp is before contract deployment timestamp
Severity	**Medium**
Description	Currently, the implementation when operator/curator fees are fetched is as follows: *function getOperatorsFeeAt(address vault, address network, uint48 timestamp, bytes memory hints)* *public* *view* *returns (uint256)* *{* *OperatorsFeeAtHints memory operatorsFeeAtHints;* *if (hints.length > 0) {* *operatorsFeeAtHints = abi.decode(hints, (OperatorsFeeAtHints));* *}* *(bool isEnabled, uint256 networkFee) =* *getOperatorsNetworkFeeAt(vault, network, timestamp, operatorsFeeAtHints.operatorsNetworkFeeHint);* *if (isEnabled) {* *return networkFee;* *}* *return getOperatorsDefaultFeeAt(vault, timestamp, operatorsFeeAtHints.operatorsDefaultFeeHint);* *}* ############## If there is no network fee at the specific timestamp, it will fallback to the default fee at this timestamp: *function getOperatorsDefaultFeeAt(address vault, uint48 timestamp, bytes memory hint)*

	public *view* *returns (uint256)* *{* *return* *_feeRegistryStorage()._operatorsFee[vault].upperLookupRecent(timestamp, hint);* *}* If there is no fee at this timestamp, it will essentially result in zero fees. This design is considered as solid until we incorporate the fact that it is very likely that historical vault snapshot timestamps for distributions will be far older than the actual deployment timestamp of this contract. This means there is no possible scenario for the curator/operator fee to be non-zero at these historical timestamps, which essentially fully prevents fee setting in that specific scenario and breaks an important core functionality of this contract.
Recommendations	A possible solution would be to allow a fee setting with a reasonable historical timestamp whenever the timestamp is before the contract deployment. However, potential side-effects from such an implementation should be carefully evaluated.
Comments / Resolution	Acknowledged. It is expected that the old rewards contract will be used for historic snapshots.

Issue_08	protocolFees are not stored historically
Severity	**Informational**
Description	As per Symbiotic team, it is desired to store historical values for all registries. This is the reason why the curator address is stored historically: *_curatorRegistryStorage()._curators[vault].push(uint48(block.timestamp), uint208(uint160(curator)));* but within the actual control-flow, it will only ever consult the latest curatorFee and does never make use of any historical values: *if (ICuratorRegistry(CURATOR_REGISTRY).getCurator(vault) != msg.sender) {* *revert NotCurator();* *}* This form an inconsistency with the way how protocolFees are stored, as these are not stored historically: *_feeRegistryStorage()._protocolFee[id] = _serializeFeeData(enable, fee);*
Recommendations	Since we do not see the necessity to adjust the code due to this small inconsistency (with the risk of introducing side-effects), we recommend acknowledging this issue.
Comments / Resolution	Acknowledged.

Rewards Module

ProtocolFees

The ProtocolFees contract is responsible for the correct accumulation of the protocol fee based on the rewardType and network.

Protocol fees are only ever accumulated during

a) distributeCumulativeRewards: via the nominal protocolFee on top of the distribution amount

> totalDistributionAmount = distributionAmount + distributionAmount * protocolFees / 100_000
> protocolFee = totalDistributionAmount - distributionAmount

b) distributeVaultSnapshotRewards: via the nominal protocolFee which is deducted from the total amount

> distributionAmount = totalDistributionAmount - totalDistributionAmount * protocolFee / maxFee
> protocolFee = totalDistributionAmount - distributionAmount

Governance can then claim fees at any time via the claimProtocolFees function.

Privileged Functions

- claimProtocolFees

Core Invariants:

INV 1: _subProtocolFeesFromTotal is only ever used for vault snapshot rewards

INV 2: _addProtocolFeesToDistribution is only ever used for cumulative merkle rewards

INV 3: Any fee fetch must first attempt to fetch network fee and only afterwards default fee

Issue_09	protocolFee ignores isEnabled flag in fallback scenario
Severity	**Medium**
Description	The protocolFee function favors the network fee and if there is no network fee or if the fee is disabled, it will choose the default fee: *function protocolFee(uint64 rewardsType, address network) public view returns (uint256) {* *(bool isEnabled, uint256 fee) =* *IFeeRegistry(FEE_REGISTRY).getProtocolFee(keccak256(abi.encode (REWARDS_FEE_ID, rewardsType, network)));* *if (isEnabled) {* *return fee;* *}* *(, fee) = IFeeRegistry(FEE_REGISTRY).getProtocolFee(keccak256(abi.encode (REWARDS_FEE_ID, rewardsType)));* *return fee;* *}* A problem occurs due to the fact that the isEnabled flag is disregarded for the default fee. This means, even if there is a default fee currently set but disabled, it will still charge the fee. It effectively ignores an important part of the setting and will result in a different outcome than anticipated (**fee is charged while it should not be charged**)
Recommendations	Consider incorporating the isEnabled flag and return zero if the fee is disabled.
Comments / Resolution	Acknowledged. The symbiotic team will simply set the fee to zero directly if it is ever desired to not apply any fee - instead of disabling the fee.

Issue_10	Redundant storage adjustment in case of zero fees
Severity	**Informational**
Description	Currently, _accountProtocolFees executes an SSTORE operation even if there is no fee to increase: *uint256 fees = totalDistributionAmount - distributionAmount;* *_protocolFeesStorage()._claimableFee[token] += fees;* This operation is redundant and will increase gas cost in that scenario.
Recommendations	This issue can be safely acknowledged under the assumption that the protocolFee will be non-zero during most periods. However, if the design choice would be to have periods with zero protocolFee, then it might be worth considering an if check to only execute the SSTORE if the fee is indeed > 0.
Comments / Resolution	Acknowledged.

Rewards

The Rewards contract forms a base to unify the CumulativeMerkleRewards and VaultSnapshotRewards claim and fee calculation related mechanisms.

This contract handles the fee deviation between the VaultSnapshotRewards and CumulativeMerkleRewards by determining stateful paths as follows:

> distributionToTotalAmount control-flow is only triggered during distribution of cumulative merkle rewards

> totalAmountToDistribution control-flow is only ever triggered during distribution of vault snapshot rewards

Furthermore, it exposes one unified claimRewards function which is externally callable.

Privileged Functions

- none

No issues found.

CumulativeMerkleRewards

The CumulativeMerkleRewards contract orchestrates the distribution and claiming of rewards which uses a cumulative amount approach and a merkle-tree based validation.

Appendix: Reward Lifecycle

The distributeCumulativeMerkleRewards function allows any caller with matching signatures (signed message from protocol & rewarder) to distribute a specific amount of tokens for a Symbiotic network. The full reward lifecycle can be illustrated as follows:

a) The depositCumulativeRewards function is triggered which deposits a specific token amount tied to a network
b) Symbiotic client (network) uses the Symbiotic backend to sign a message and craft a signature. This will simultaneously also create signature for the protocol signed message.
c) The distributeCumulativeMerkleRewards function is called which sets the merkleRoot for the network, validates all signatures and handles storage adjustment
d) The claimCumulativeMerkleRewards function is called which allows any valid address (with a leaf) to claim their corresponding cumulative amount

Appendix: Reward Claim

The claimCumulativeMerkleRewards function allows any valid address to claim their corresponding cumulative rewards. This is validated in two steps:

a) **Validation of root**: The provided merkleRoot for the network must be marked as submitted/distributed
b) **Validation of leaf**: The leaf contains the following data:
    i) keccak256(**msg.sender**, **chainId**, {**CumulativeDistributionLeaf**: token, rewardeeType, amount, rewardeeDataHash})
    ii) and it must be valid in relation to the merkleRoot

Following this validation, it is ensured that only entitled addresses can claim their rewards on the specific chain for the corresponding network.

Appendix: rewardeeType

The rewardeeType is a specific component of each leaf and essentially represents one of the three types:

- Staker
- Operator
- Curator

Including this parameter in a leaf allows a single address to potentially claim more than one reward from a single distribution. For example, one distribution can have rewards for Alice being a *Staker* and *Operator* at the same time, which is gracefully handled by the introduction of this type.

Appendix: Signed Message

When a new cumulative distribution is created, it is described by a CumulativeDistributionPayload:

- the network this distribution applies to
- a timestamp used for ordering
- a merkleRoot representing all per-rewardee leaves for this global epoch across chains
- a list of TokenAmount entries, each of which says: “for chain X and token T, the cumulative total should now be A.”

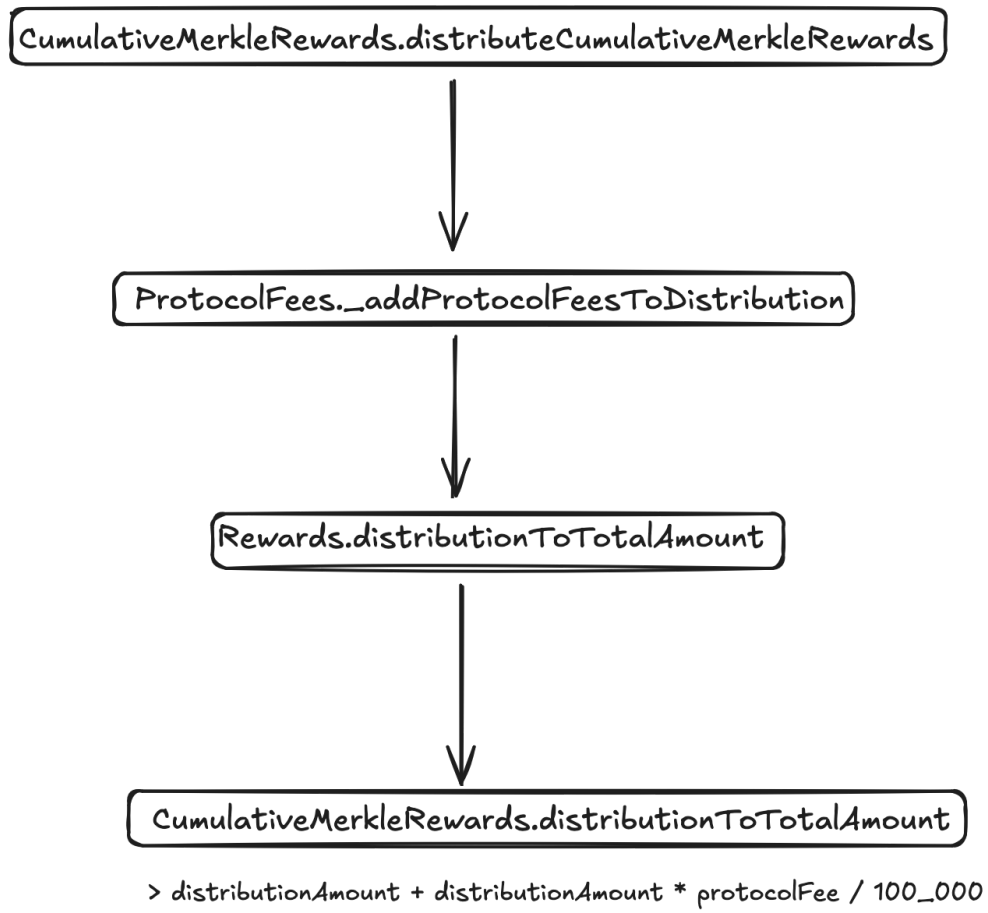
Each TokenAmount is individually hashed with the TOKEN_AMOUNT_TYPEHASH, and those hashes are packed and hashed again to produce a single array commitment.

Together with network, timestamp, and merkleRoot, this forms the EIP-712 CumulativeDistributionPayload struct.

The protocol and the per-network rewarder both sign the same typed message digest (using the cross-chain EIP-712 domain from OzEIP712), thereby jointly authorizing this new cumulative state.

Onchain, distributeCumulativeMerkleRewards recomputes the same digest, checks both signatures, verifies that the timestamp is greater than the previously accepted one for that network, and then updates per-token balances and last totals according to the delta between the new cumulative amounts and the previously stored ones.

Appendix: distributionToTotalAmount



Privileged Functions

- setProtocol

Core Invariants:

INV 1: protocol and rewarder must have signed the digest for distributeCumulativeMerkleRewards

INV 2: Only the rewarder can call withdrawCumulativeMerkleRewards

INV 3: A rewarder is always tied to a network

INV 4: depositCumulativeMerkleRewards must be called before distributeCumulativeMerkleRewards

INV 5: distributionToTotalAmount must return distributionAmount + fee on top
INV 6: balance must only be decreased by delta between

INV 7: Subsequent distributions for same network/token must have increased amount

INV 8: A rewarder can never withdraw more than the balance for his network

INV 9: For cumulative rewards, only the protocolFee is considered

INV 10: totalToDistributionAmount must not be used within this contract in any control-flow

INV 11: Any address can deposit rewards for any token/network

INV 12: Each distribution is strictly tied to a network

Issue_11	Cross-contract replay attack due to hardcoded version within OzEIP712
Severity	**High**
Description	The protocol implements a cumulative, Merkle-based reward distribution mechanism. Distributions are authorized off-chain using an EIP712-typed payload (CUMULATIVE_DISTRIBUTION_PAYLOAD_TYPEHASH) and signed by two privileged parties: - protocol() – the central protocol authority - rewarder(network) – the per-network reward manager. On-chain, distributeCumulativeMerkleRewards verifies these signatures via hashTypedDataV4CrossChain from a custom OzEIP712 base, then registers a (network, merkleRoot) pair as valid and updates the accounting deltas (_balances, _lastTotalAmounts, _lastCumulativeDistribution). The same signed message can include multiple TokenAmount entries with different chainId values, and the contract filters them at runtime using *if (totalAmount.chainId != uint64(block.chainid))* *continue;* This allows a single signature to authorize distributions on multiple chains simultaneously. Claims later rely only on: - isCumulativeDistributionRoot(network, merkleRoot) - a Merkle proof against keccak256(abi.encode(msg.sender, block.chainid, leaf) to validate a reward leaf. The EIP712 signature is not revalidated during claims, it is only used once to approve the root during

distribution.

In parallel, the contract uses a fixed EIP712 initialization:

__OzEIP712_init(OzEIP712InitParams("CumulativeMerkleRewards", "1"));

The core issue is a **potential EIP712 replay surface across contracts**, driven by a cross-chain EIP712 domain that is not clearly bound to the concrete contract instance:

bytes32 private constant CROSS_CHAIN_TYPE_HASH = keccak256("EIP712Domain(string name,string version)");

As one can see, the EIP712 domain inside hashTypedDataV4CrossChain:

- does not include verifyingContract, and
- uses a static (name = "CumulativeMerkleRewards", version = "1"),

In such a setup, the following becomes possible:

- Deploy a new contract RewardsV2 / CumulativeMerkleRewardsV2 on the same chain with identical EIP712 domain settings.

- Fund _balances[network][token] in the new contract (e.g. intending to distribute rewards under updated mechanics: different timing, safety rules, anti-abuse checks, etc).

- Any actor with access to the old signed CumulativeDistributionPayloads (which are often public or at least recoverable from logs, off-chain coordination, or the old chain) can replay them against the new contract

	- Each old cumulativeDistribution passes InvalidTimestamp checks on a fresh deployment (since lastCumulativeDistribution(network).timestamp is initially 0). - Each old merkleRoot is newly registered under _isCumulativeDistributionRoot[network][root] = true. - The deltas totalAmount.amount - lastTotalAmount are recalculated from zero and applied to _balances, consuming the new contract's funds according to old distribution schedules. This effectively allows cross-contract replay of historical distributions onto the new instance, bypassing the "new intended mechanics" of the fresh deployment.
Recommendations	Consider removing the automatic initialization via version 1: *__OzEIP712_init(OzEIP712InitParams("CumulativeMerkleRewards", "1"));* and using a version parameter. This would then avoid replay attacks because the version is unique.
Comments / Resolution	Acknowledged, the Symbiotic team is now aware of this issue and if such an unlikely scenario is ever going to occur, they will change the hardcoded version from 1 to 2 manually in the contract: *function __CumulativeMerkleRewards_init() internal onlyInitializing {* *__OzEIP712_init(OzEIP712InitParams("CumulativeMerkleRewards", "1"));* *}* Due to gas-efficiency reasons, no extra parameter has been incorporated.

Issue_12	Race condition for protocolFee increase is amplified due to permissionless nature of distributeCumulativeMerkleRewards
Severity	**Medium**
Description	Currently, the protocolFee is calculated whenever distributeCumulativeMerkleRewards is called and is calculated on top of the actual distributed amount. This means if a distribution attempts to distribute 100e18 tokens and the fee is 10%, it will decrease the _balances mapping by 110e18 tokens. If now the fee is increased just before the transaction is executed to 20%, it will decrease the _balances mapping by 120e18 tokens, while the caller expected it to only consume 110e18 tokens. Another iteration of this issue is whenever governance communicates that a fee will be decreased - a malicious actor can now simply execute the function due to the permissionless nature (if the signatures are known) - right before the fee decrease, which results in consuming more than anticipated (if the rightful caller would like to wait until fee has been decreased).
Recommendations	Consider implementing a slippage parameter for the maximum acceptable protocolFee and cross-check this during the execution.
Comments / Resolution	Acknowledged.

Issue_13	Unclaimed rewards remain permanent inaccessible
Severity	**Low**
Description	Currently, there is no withdrawal possibility for rewards which are not claimed. This is a legitimate concern because in most distribution contracts, there are scenarios where rewards are never claimed and thus remain idle.
Recommendations	Consider implementing a governance withdrawal function.
Comments / Resolution	Acknowledged.

Issue_14	Inconsistency in fee calculation behavior compared to VaultSnapshotRewards
Severity	**Low**
Description	The fee calculation of the CumulativeMerkleReward contract is different to the fee calculation from the VaultSnapshotRewards contract. The difference lies in the fact that the fee within CumulativeMerkleReward is simply added on top of the overall amount, while the fee calculation for VaultSnapshotRewards is using the known formula to calculate the fee on top of the net amount. This will create an inconsistency in the fee amount which is applied: VaultSnapshotRewards: - distributionAmount = 1_000_000e18 tokens - protocolFee = 5e4 (5%) - totalDistributionAmount = =(1_000_000e18-1 x 1e6) / (1e6 - 5e4) + 1 = = 1_052_631.578947368421052631e18 CumulativeMerkleRewards: - distributionAmount = 1_000_000e18 tokens - protocolFee = 5e4 (5%) - totalDistributionAmount = = 1000000e18 + (1000000e18 x 5e4 / 1e6) = 1_050_000e18 With the same protocol fee (5%), the MerkleRewards system requires paying ~2_631e18 tokens less than the VaultSnapshots system. ***Note**: the higher the fee, the higher the discrepancy.*
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_15	Fee rounds against the favor of the protocol
Severity	**Informational**
Description	Currently, fee calculation is rounding down which means there could be less fee than expected during a distribution. This can become specifically remarkable in the case of low decimal / high value tokens combined with the fact that the divisor is 1_000_000. - protocolFee = 1e4 (1%) - operatorFee = 3e4 (3%) - curatorFee = 2e4 (2%) - token: GUSD (2 decimals) - totalDistributionAmount = 33 wei - protocolFee = 33 x 1e4 / 1e6 = 0 - operatorFee = 33 x 3e4 / 1e6 = 0 - curatorFee = 33 x 2e4 / 1e6 = 0 - distributionAmount = 0.33e2 GUSD
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_16	Reliance on backend for signature creation is prone to downtime
Severity	**Informational**
Description	The distributeCumulativeMerkleRewards function expects the digest to be signed by the protocol and the rewarder. Symbiotic provides a backend service for this specific purpose, which means there is a reliance on this service to function - otherwise there would not be any possibility to receive a valid protocol signature. In the scenario where this service does not work, no messages can be signed and no rewards can be distributed.
Recommendations	Consider keeping this in mind and adding a fallback service mechanism.
Comments / Resolution	Acknowledged.

Issue_17	Timestamp logic is solely used as validation for subsequent distributions
Severity	**Informational**
Description	There is currently no such check which prevents users to claim rewards before the timestamp has been reached. This could be a design decision or a blunder. It furthermore needs to be pointed out that there is no “far-future” check which means if a distribution is executed with an accidentally large timestamp, it will prevent future distributions.
Recommendations	Consider elaborating whether that is an issue.
Comments / Resolution	Acknowledged.

Issue_18	Aggregate amount of leaves may not match distributed amount
Severity	**Informational**
Description	Currently, there is no on-chain validation that the amounts within the leaves indeed match the distributed amount. This means if the merkle root creation is flawed, it can happen that more rewards are claimable than rewards have been distributed. Likewise it may also happen that subsequent leaves represent a smaller amount than previous leaves and thus claims would revert because the monotonicity is not enforced on a leaf-base.
Recommendations	Consider ensuring that the backend service works properly.
Comments / Resolution	Acknowledged. It is expected that the backend works flawlessly.

Issue_19	Footgun in case signer is a smart contract
Severity	**Informational**
Description	The EIP1271 logic supports signatures from smart contract wallets. These usually simply point back to the owner(s) and checks if one of the owners has signed the message. This can become a footgun in case the protocol/rewarder address is a smart contract as now essentially this signature becomes valid not only for the owner of the contract but also for all other contracts which are owned by the same address.
Recommendations	Consider keeping that in mind.
Comments / Resolution	Acknowledged.

VaultSnapshotRewards

The VaultSnapshotRewards contract orchestrates the distribution and claiming of rewards which are tied to shares of a vault at a specific timestamp. Any registered entity (network) can distribute rewards via the distributeVaultSnapshotRewards function which essentially distributes a desired amount among the following parties:

- Users: The remaining amount after the protocol/curator/operator fee is distributed among all vault share owners proportionally to their percentage-wise ownership of shares on the overall shares at a specific timestamp.
- Operator: A operator fee is taken on the amountWithoutProtocolFee
- Curator: A curator fee is taken on the amountWithoutProtocolFee
- Protocol: A protocol fee is taken on the initial amount

Appendix: Reward Lifecycle

The reward lifecycle starts with the distributeVaultSnapshotRewards function where a linked entity can deposit rewards for a specific network and vault. A protocol fee / operator fee / curator fee is taken from this deposited amount and then users that owned vault shares at the specific timestamp can claim rewards proportionally to the owned shares.

Appendix: Reward Claim

The claimVaultSnapshotRewards function allows eligible participants to claim their rewards. Rewards are distributed proportionally to the owned amount of shares from the specified vault at the corresponding timestamp, to the overall share amount:

> activeSharesOfAt * reward.amount / totalActiveSharesAt

Appendix: Operator Fee Claim

The claimOperatorFees function allows eligible operators to claim their fee which is stored within the RewardsDistribution array for the specified token/vault/network mapping. The specific control-flow for claiming operator fees is derived from the delegator TYPE:

Type 0: Distribute rewards proportionally to the owned amount of operatorNetworkShares

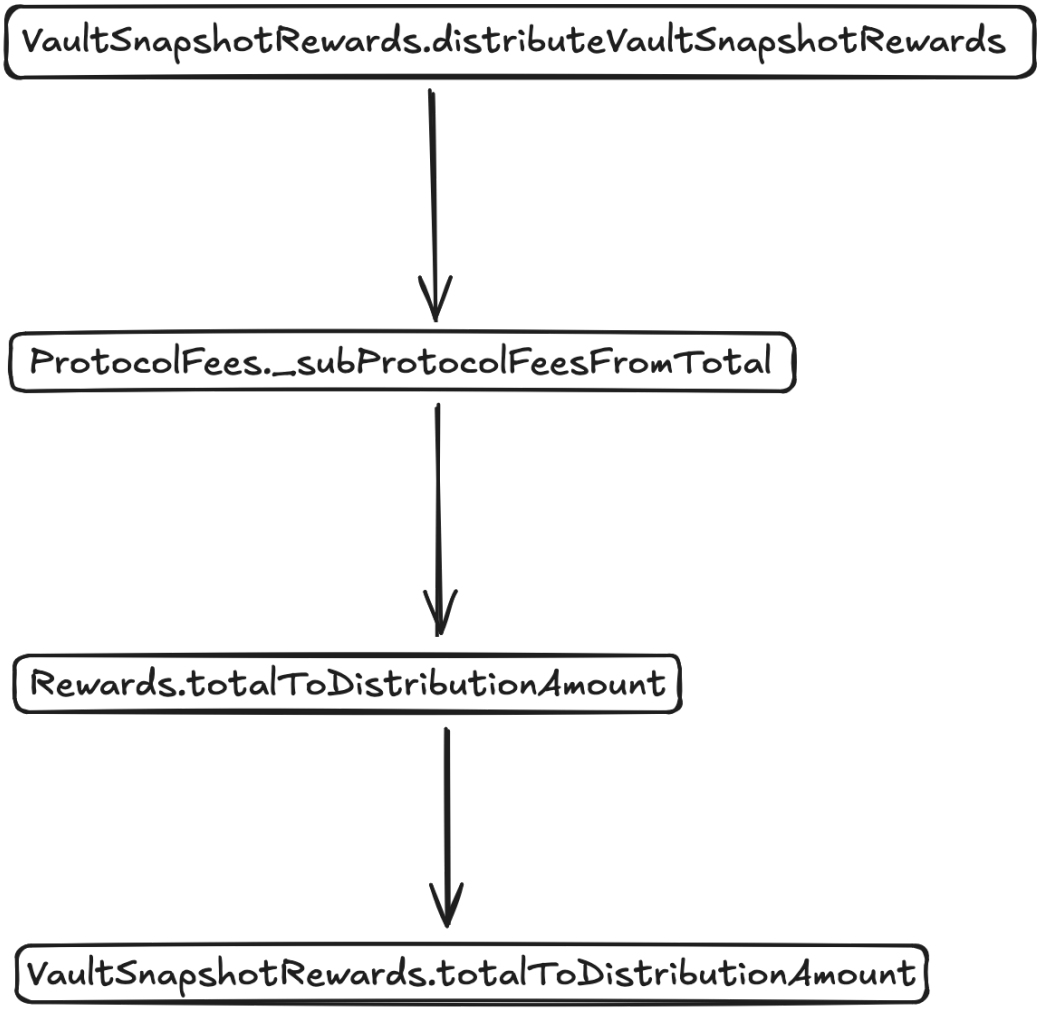
> operatorNetworkSharesAt * operatorsFees / totalOperatorNetworkSharesAt

Type 1: No execution

Type 2: Distribute all rewards if the caller is operator of OperatorSpecificDelegator

Type 3: Distribute all rewards if the caller is operator of OperatorNetworkSpecificDelegator

Appendix: totalAmountToDistribution



> totalDistributionAmount - totalDistributionAmount * protocolFee / maxFee

Issue_20	Permanently locked fees due to missing totalOperatorNetworkSharesAt > 0 check at distribution time
Severity	**High**
Description	Whenever a delegator with type = 0 wants to claim rewards, the following implementation is called: *if (reward.delegatorType == 0) {* *uint256 totalOperatorNetworkShares = INetworkRestakeDelegator(reward.delegator)* *.totalOperatorNetworkSharesAt(* *subnetwork, reward.timestamp, totalOperatorNetworkSharesHint[networkRestakeDelegatorCounter]* *);* *if (totalOperatorNetworkShares > 0) {* *amount += INetworkRestakeDelegator(reward.delegator)* *.operatorNetworkSharesAt(* *subnetwork,* *msg.sender,* *reward.timestamp,* *operatorNetworkSharesHints[networkRestakeDelegatorCounter]* *).mulDiv(reward.operatorsFees, totalOperatorNetworkShares);* *}* *unchecked {* *++networkRestakeDelegatorCounter;* *}* *}* As one can see, it is required for totalOperatorNetworkShares > 0 to distribute rewards. There is no such check within the distributeVaultSnapshotRewards function which means it will be possible that totalOperatorNetworkShares = 0 and delegator.TYPE() = 0. Thus, fees will be unclaimable.

	Notably, a check for the regular Vault shares total is exposed within distributeVaultSnapshotRewards:
	
	`if (_vaultSnapshotRewardsStorage()._activeSharesCache[vault][timestamp] == 0) {`
	`uint256 activeShares = IVault(vault).activeSharesAt(timestamp, hints.activeSharesHint);`
	`if (activeShares == 0) {`
	`revert InvalidRewardTimestamp();`
	`}`

Recommendations	Consider adding a sanity check within distributeVaultSnapshotRewards which reverts in case of delegator.type() = 0 and totalOperatorNetworkShares = 0
Comments / Resolution	Resolved. It has to be noted that technically, the claimOperatorFees function is still callable in case of type = 0 and zero operator fees. This creates technically a new issue for this specific RewardsDistributions array in case there is an entry with total shares being zero while a user may attempt to claim multiple elements in this array, it will then revert for that specific iteration, which essentially requires for claims up to the zero share element and then skipping this.

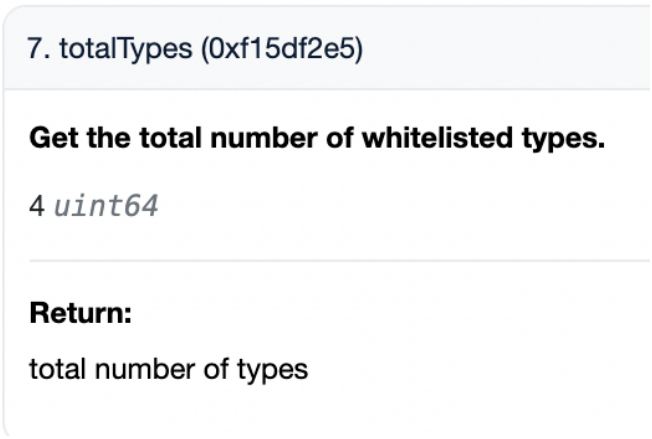
Issue_21	Permanently locked fees in case of delegator.TYPE() = 1
Severity	**High**
Description	Currently, the claimOperatorFees function has 4 different control-flows which are based on the delegator.TYPE() that is casted upon distributeVaultSnapshotRewards: *_vaultSnapshotRewardsStorage()* *._rewards[vault][network][token].push(* *RewardDistribution{{* *subnetworkId: subnetwork.identifier(),* *delegator: IVault(vault).delegator(),* ***delegatorType:*** ***IBaseDelegator(IVault(vault).delegator()).TYPE(),*** *timestamp: timestamp,* *amount: distributionAmount,* *operatorsFees: operatorsFees* *}}* In the scenario of delegator.TYPE() = 1, the control-flow will not cast any fees: *else if (reward.delegatorType == 1) {* *// pass* This means that any fees which are entitled to operators for delegator.TYPE() = 1, will remain inaccessible.
Recommendations	Consider re-routing the operator fees to the protocol or to the distributed amount in case of type = 1.
Comments / Resolution	Partially resolved. It is now fully disabled to allocate rewards if the DelegatorType is FULL_RESTAKE, instead of re-routing or zero-ing out operator fees.
Resolution 2	Resolved. Now it is allowed to distribute rewards with delegatorType = FULL_RESTAKE. These rewards are claimable in the same fashion

	as all other rewards but operator fees are zero'd out which means even if users call the claimOperatorFees function, it would simply revert.

Issue_22	FULL_RESTAKE rewards are now fully disabled
Severity	**High**
Description	During the first resolution round, a fix was implemented which prevents the stuck operator fees in case of FULL_RESTAKE (TYPE = 1). This fix however does not only prevent stuck operator fees but now fully prevents reward distribution for FULL_RESTAKE type: *else if (* *delegatorType == uint64(DelegatorType.FULL_RESTAKE) ||* *delegatorType > uint64(type(DelegatorType).max)* *) {* *revert InvalidDelegatorType();* *}* This behavior is unintended, as in that specific scenario it should still be allowed to distribute rewards but simply operator fees should be zero'd out. However, it will remain impossible to distribute any rewards at all.
Recommendations	Consider allowing for reward distribution but prevent any operator fees. As security measurement, the distributeVaultSnapshotRewards and claimOperatorFees function must be fully re-audited to ensure all edge-cases are prevented.
Comments / Resolution	Fixed in resolution 2. Only operator fees will be zeroed out, and operator claimings are reverted. Minor note: If there is any instance where a vault's delegator TYPE can change over time, this can result in the same issue as: "Potential revert for multi-claim scenarios within claimOperatorFees "

Issue_23	Race condition due to protocolFee increase
Severity	**Medium**
Description	Currently, it is possible for a distributeVaultSnapshotRewards function to be called right before protocolFees is increased. If now the distribution function remains stuck in the mempool and the fee increase is executed just beforehand, it will result in a higher protocolFee than initially anticipated.
Recommendations	Consider implementing a slippage parameter for the maximum acceptable protocolFee during distribution execution.
Comments / Resolution	Acknowledged.

Issue_24	Footgun within claimVaultSnapshotRewards can result in lost rewards
Severity	**Medium**
Description	The claimVaultSnapshotRewards function allows users to claim their rightful rewards. The implementation exposes a firstRewardToClaim parameter which determines from which index in the array to start. If a user has claimable rewards at index = 0 and provides firstRewardToClaim = 10, it will essentially skip all previous indexes and rewards from these indexes will remain permanently unclaimable. The same is true for the claimOperatorFees function. Usually, locked funds issues are rated as high severity, however, in that specific scenario the developer mentioned that this is a design choice and it would explicitly require users to provide a wrong firstRewardToClaim value. Due to both facts, we rated this issue only as medium severity. Please note that if only one of these facts would be true, it would be considered a high severity issue.
Recommendations	Consider either implementing documentation which clearly highlights this footgun or refactor the function to enforce users claiming from index = 0.
Comments / Resolution	Acknowledged, users will be informed about this behavior.

Issue_25	Lack of delegator.TYPE() validation can result in unclaimable fees
Severity	Low
Description	Currently, there are 4 delegator types. This can be found within the Delegator Factory contract: https://etherscan.io/address/0x985Ed57AF9D475f1d83c1c1c8826A0E5A34E8C7B#readContract  It is however possible that new delegator types are being added: *function whitelist(address implementation_) external onlyOwner {* *if (IEntity(implementation_).FACTORY() != address(this) ||* *IEntity(implementation_).TYPE() != totalTypes()) {* *revert InvalidImplementation();* *}* *if (!_whitelistedImplementations.add(implementation_)) {* *revert AlreadyWhitelisted();* *}* *emit Whitelist(implementation_);* *}* If that ever happens and rewards for a vault with a new delegator TYPE are being distributed, the operator fees for this distribution remain permanently locked.

	This issue has only been rated as low instead of high severity since the assumption is made that the Symbiotic architecture ensures that such delegator types are not existent.
Recommendations	Consider implementing explicit validation which re-routes operator fees to the protocol or to the distribution amount in case the delegator TYPE does not match.
Comments / Resolution	Resolved, it is now prohibited to distribute rewards in such a scenario.

Issue_26	Unclaimed rewards remain permanent inaccessible
Severity	**Low**
Description	Currently, there is no withdrawal possibility for rewards which are not claimed. This is a legitimate concern because in most distribution contracts, there are scenarios where rewards are never claimed and thus remain idle.
Recommendations	Consider implementing a governance withdrawal function.
Comments / Resolution	Acknowledged.

Issue_27	Dust accumulation over time due to truncation in pro-rata calculation
Severity	**Low**
Description	Rewards and operator fees for TYPE = 0 undergo a proportional calculation which distributes the reward amount to the corresponding owned percentage of shares. The calculation truncates the result and thus the contract will naturally accumulate dust. In extreme scenarios, it can even happen that users get no rewards at all, if their proportional share is very low.
Recommendations	Consider implementing a governance withdrawal function.
Comments / Resolution	Acknowledged.

Issue_28	Double-claim scenario in case of operator change
Severity	**Informational**
Description	The protocol distributes vault snapshot rewards and related operator fees via the VaultSnapshotRewards abstraction. For each distribution, it creates a RewardDistribution entry per (vault, network, token) that contains: - The subnetwork identifier, - The delegator contract address, - The delegatorType (0, 1, 2, or 3), - The timestamp of the reward, - The total amount for vault depositors, and - The operatorsFees bucket reserved for operators. Operator fee claims are processed through claimOperatorFees, which iterates over the relevant range of RewardDistribution entries and, depending on delegatorType, calculates how much the caller should receive. For type-2 and type-3 delegators, the entire operatorsFees amount for an entry is assigned to the operator reported by the delegator at **claim time**: *// TYPE 2* *} else if (reward.delegatorType == 2) {* *if (IOperatorSpecificDelegator(reward.delegator).operator() != msg.sender) {* *revert NotOperator();* *}* *amount += reward.operatorsFees;* *// TYPE 3* *} else if (reward.delegatorType == 3) {*

```
    if (IOperatorNetworkSpecificDelegator(reward.delegator).operator() != msg.sender) {
        revert NotOperator();
    }
    amount += reward.operatorsFees;
}
```

The delegator contracts (reward.delegator) are themselves upgradeable, and their operator role is defined by operator() in IOperatorSpecificDelegator / IOperatorNetworkSpecificDelegator. While in the current implementation this operator may be static, the upgradeable architecture means future versions can change operator() over time. Notable, this is currently only possible if Symbiotic governance added new whitelisted implementations. If at any point in the future, an implementation is used which allows for changing the operator, this will become a critical issue.

The issue arises because operator identity is not frozen at distribution time, while the claim logic is keyed per-operator address and uses the current delegator operator to gate eligibility.

At distribution time, when RewardDistribution entries are created in distributeVaultSnapshotRewards, the contract records only:

- delegator (the delegator contract address)
- delegatorType
- the timestamp
- operatorsFees

It does not snapshot the operator address that was entitled to those fees at that moment. Instead, claimOperatorFees later does the following:

- Reads the delegator address from storage.
- Calls operator() on that delegator to determine who the operator is right now.

	- Checks that operator() == msg.sender for type-2/3.
	- If equal, it adds the full reward.operatorsFees for that entry to the caller's amount.
	- The progression index _lastUnclaimedOperatorReward is scoped to (msg.sender, vault, network, token), meaning:
	  - Each distinct operator address maintains its own cursor through the reward array.
	  - Claiming as operator = A does not move the cursor for operator = B.
	
	If the delegator's operator is ever changed after distribution, this creates a straightforward double-claim path.

Recommendations	Consider storing the operator at distribution time.
Comments / Resolution	Acknowledged, the team is aware of this potential issue and will make sure all future implementations of the Vault will have an immutable operator.

Issue_29	Fee rounds against the favor of the protocol
Severity	**Informational**
Description	Currently, fee calculation is rounding down which means there could be less fee than expected during a distribution. This can become specifically remarkable in the case of low decimal / high value tokens combined with the fact that the divisor is 1_000_000.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_30	MEV concerns due to share determination based on timestamps
Severity	**Informational**
Description	Rewards are distributed proportionally to the owned amount of shares in a Vault. There is the theoretical possibility of frontrunning a new epoch with a large deposit to gain a majority of shares just to then be able to claim a majority of rewards. Likewise, users that have withdrawn shortly before the snapshot time, will not get any rewards. This exploit is however out of scope and will be handled by the Symbiotic team in the corresponding contract.
Recommendations	Consider taking appropriate steps to prevent such an attack.
Comments / Resolution	Acknowledged.

Issue_31	Griefing attack by distributing multiple 1 wei amounts between [claimTXSubmission; claimTXExecution] can result in OOG error
Severity	**Informational**
Description	The claimVaultSnapshotRewards function loops from the firstRewardToClaim index up rewardsToClaim. In specific scenarios, users will just provide a large value for rewardsToClaim, as it is anyways clamped with the maximum length: *rewardsToClaim = Math.min(rewardsToClaim, rewardsByTokenNetwork.length - firstRewardToClaim);* This technique allows users to simply claim all outstanding rewards. A theoretical griefing vector is exposed when users are using this technique, as a network can simply call the distributeVaultSnapshotRewards function multiple times with a small amount before the user claims. It will then increase the RewardsDistribution array and due to the above mentioned technique, the claimVaultSnapshotRewards function will eventually run out of gas. Please note that this is not critical as users can just call the claimVaultSnapshotRewards function again, this time using a proper rewardsToClaim value.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_32	Potential revert for multi-claim scenarios within claimOperatorFees
Severity	**Informational**
Description	Operator fees are now zero'd out in case there is a distribution with no corresponding operator shares. This behavior is correct in itself. However, the claimOperatorFees function always loops over the RewardsDistributions array and attempts to meet each element. If such an element with zero shares is pushed, it will not distribute any rewards (naturally) but it will revert due to a division by zero error, essentially reverting the whole loop. Users will essentially need to skip this exact element, which is however possible with the current claim flow.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.