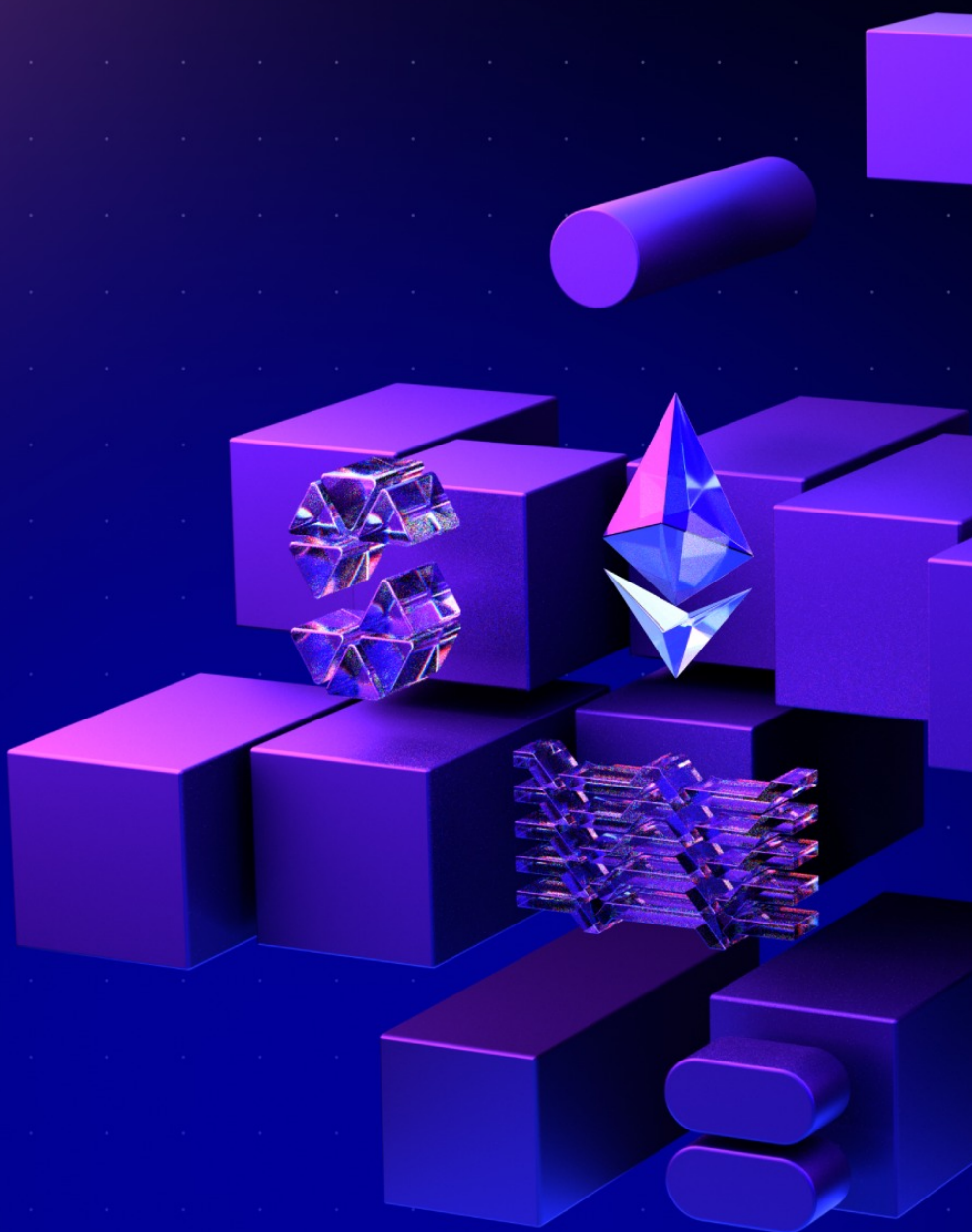
ackee
blockchain security

Symbiotic

Rewards V2

January 12, 2026



Audit Report

Contents

1. Document Revisions . . . 3
2. Overview . . . 4
    2.1. Ackee Blockchain Security . . . 4
    2.2. Audit Methodology . . . 5
    2.3. Finding Classification . . . 6
    2.4. Review Team . . . 8
    2.5. Disclaimer . . . 8
3. Executive Summary . . . 9
    Revision 1.0 . . . 9
    Revision 1.1 . . . 11
4. Findings Summary . . . 12
Report Revision 1.0 . . . 14
    Revision Team . . . 14
    System Overview . . . 14
    Trust Model . . . 14
    Findings . . . 15
Appendix A: How to cite . . . 36
Appendix B: Wake Findings . . . 37
    B.1. Detectors . . . 37
Appendix C: Wake Arena Findings . . . 38
    C.1. Discovered Findings . . . 38

1. Document Revisions

1.0-draft	Draft Report	December 20, 2025
1.0	Final Report	December 21, 2025
1.1	Fix Review	January 12, 2026

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**
   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**
   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**
   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**
   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**
   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

6. **Wake Arena assisted vulnerability discovery**
   As the last step, the scope is checked against Wake Arena, an LLM-powered audit tool, to identify potentially missed vulnerabilities. This step is executed at the end of the audit process to avoid distracting auditors from manual review.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Lukáš Rajnoha	Lead Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Symbiotic is a shared-security marketplace where networks obtain economic security from asset stakers, allowing stake to be shared and utilized across multiple networks. Its flexible architecture lets stake providers maximize returns while networks secure the guarantees they need. The Rewards V2 is a rewards distribution component that enables networks to fund and distribute ERC-20 rewards across vaults and operators.

Revision 1.0

Symbiotic engaged Ackee Blockchain Security to perform a security review of Symbiotic Rewards V2 with a total time donation of 7 engineering days in a period between December 11 and December 19, 2025, with Lukáš Rajnoha as the lead auditor.

The audit was performed on the commit `716eddc`[1] in the contracts repository and the scope was the following:

- `src/contracts/CumulativeMerkleRewards.sol`
- `src/contracts/CuratorRegistry.sol`
- `src/contracts/FeeRegistry.sol`
- `src/contracts/ProtocolFees.sol`
- `src/contracts/Rewards.sol`
- `src/contracts/VaultSnapshotRewards.sol`

The `src/contracts/libraries/Checkpoints.sol` library was not included in the scope.

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. For testing, we involved the Wake testing framework. During the review, we paid special attention to:

- ensuring the fee math and other arithmetic of the system is correct;
- verifying cumulative Merkle distributions are correctly integrated;
- ensuring deposit, distribution, and withdrawal accounting updates balances atomically;
- reviewing initialization/upgradeability patterns;
- ensuring access controls are not too relaxed or too strict; and
- looking for common issues such as data validation.

Our review resulted in 10 findings, ranging from Info to Medium severity. The most severe finding M1 involved stuck funds due to the ability to skip some claimable rewards. Another 3 Low findings involved a logic error in protocol fee accounting in L1, missing network registry checks in L2, and insufficient zero address validation in case of L3.

At the end of the review, we engaged the Wake Arena tool, which discovered the issue W4 regarding a potential division-by-zero error.

Ackee Blockchain Security recommend Symbiotic to:

- ensure proper initialization of deployed contracts;
- validate off-chain infrastructure involving Merkle roots/proofs and signatures is working correctly;
- ensure `CumulativeMerkleRewards` has updated versioning in case of future redeployments to prevent replay attacks due to the intentionally reduced EIP-712 domain (only `(name, version)`); and
- address all identified issues.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

Symbiotic engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision.

Symbiotic provided a final commit `30f5381`[2] with the fixes and provided comments to most of the acknowledged findings.

The review was performed on January 5, 2026 on the commit `30f5381`[3].

From the reported 10 findings:

- 3 issues were fixed; and
- 7 issues were acknowledged.

No new findings were reported.

[1] full commit hash: `716eddccd21f64be77d0af8f882952ca41fbf4e8`, link to commit

[2] full commit hash: `30f5381cfd6e7614bae6d7170d6b3f86c31319a7`, link to commit

[3] full commit hash: `30f5381cfd6e7614bae6d7170d6b3f86c31319a7`, link to commit

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	0	1	3	4	2	**10**

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
M1: Skippable rewards in `VaultSnapshotRewards` become unclaimable	Medium	1.0	Acknowledged
L1: `protocolFee` ignores disabled default fee	Low	1.0	Acknowledged
L2: `CumulativeMerkleRewards` does not enforce network registry checks	Low	1.0	Acknowledged
L3: Missing zero-address check in `setCurator` locks role and fees	Low	1.0	Acknowledged

Finding title	Severity	Reported	Status
W1: `FeeRegistry` contract does not disable initializers	Warning	1.0	Fixed
W2: Ambiguous reverts degrade UX	Warning	1.0	Acknowledged
W3: Inconsistent zero-address recipient validation	Warning	1.0	Fixed
W4: Division-by-zero in `VaultSnapshotRewards.claimOperatorFees`	Warning	1.0	Acknowledged
I1: Unused using-for directive	Info	1.0	Fixed
I2: Several public functions should be external	Info	1.0	Acknowledged

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Lukáš Rajnoha	Lead Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Symbiotic Rewards V2 is a unified rewards layer that combines two flows under a single orchestrator contract: off-chain constructed `CumulativeMerkleRewards`, which support multi-chain allocations, and fully on-chain `VaultSnapshotRewards`, which require no off-chain work and use historical snapshots for pro-rata fee splits. The `CuratorRegistry` tracks vault-to-curator ownership and gates curator fee withdrawals, while `FeeRegistry` stores protocol, curator, and operator fee configurations with optional network-specific overrides and historical snapshots.

Trust Model

For cumulative rewards, users rely on the off-chain data committed by the protocol signer and network rewarder, though anyone can verify roots against published data. For snapshot rewards, users rely on the network to submit correct distributions and past timestamps, with funds moved atomically on distribution. Users trust the correctness of core registries and any upgrade or ownership controls that govern signers, roles, and deployments, and accept ERC-20 token transfer behaviors reflected in accounting. Users must trust the protocol owner who can change protocol fees at any time.

Findings

The following section presents the list of findings discovered in this revision.

For the complete list of all findings, Go back to Findings Summary

M1: Skippable rewards in `VaultSnapshotRewards` become unclaimable

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	VaultSnapshotRewards.sol	Type:	Logic error

Description

The `VaultSnapshotRewards` contract records per-snapshot distributions and pays them out to stakers and operators. Claims are made via `claimVaultSnapshotRewards` and `claimOperatorFees`, which iterate from each account's `lastUnclaimedReward`/`lastUnclaimedOperatorReward` cursor with pagination via `firstRewardToClaim` and `rewardsToClaim` parameters.

Because claims may start from a `firstRewardToClaim` greater than the recorded `lastUnclaimed*`, the contract advances the cursor past earlier entries, effectively skipping them. Those skipped distributions cannot be claimed later by that user and are permanently stuck.

Listing 1. Excerpt from VaultSnapshotRewards

```
271 function claimVaultSnapshotRewards(
272     address recipient,
273     address network,
274     address token,
275     address vault,
276     uint256 lastUnclaimedRewards,
277     uint256 firstRewardToClaim,
278     uint256 rewardsToClaim,
279     bytes[] memory activeSharesHints
280 ) public nonReentrant {
281     if (recipient == address(0)) {
282         revert InvalidRecipient();
283     }
284
285     // Check lastUnclaimedRewards vs on-chain for reorgs
```

```
286     if (lastUnclaimedRewards != lastUnclaimedReward(msg.sender, vault,
    network, token)) {
287         revert InvalidLastUnclaimedReward();
288     }
289
290     RewardDistribution[] storage rewardsByTokenNetwork =
291         _vaultSnapshotRewardsStorage()._rewards[vault][network][token];
292
293     firstRewardToClaim = firstRewardToClaim > lastUnclaimedRewards ?
    firstRewardToClaim : lastUnclaimedRewards;
294     if (firstRewardToClaim > rewardsByTokenNetwork.length) {
295         revert NoRewardsToClaim();
296     }
297
298     rewardsToClaim = Math.min(rewardsToClaim, rewardsByTokenNetwork.length -
    firstRewardToClaim);
299     if (rewardsToClaim == 0) {
300         revert NoRewardsToClaim();
301     }
302
303     if (activeSharesHints.length == 0) {
304         activeSharesHints = new bytes[](rewardsToClaim);
305     }
306
307     uint256 amount;
308     for (uint256 i; i < rewardsToClaim; ++i) {
309         RewardDistribution storage reward;
310         unchecked {
311             reward = rewardsByTokenNetwork[firstRewardToClaim + i];
312         }
313
314         amount += IVault(vault).activeSharesOfAt(msg.sender,
    reward.timestamp, activeSharesHints[i])
315             .mulDiv(reward.amount,
    _vaultSnapshotRewardsStorage()._activeSharesCache[vault][reward.timestamp]);
316     }
```

Exploit scenario

The rewards array contains four rewards: `[r0, r1, r2, r3]`. Alice, a staker who has not claimed any rewards yet, has `lastUnclaimedReward = 0`.

1. Alice calls the `claimVaultSnapshotRewards` function with `firstRewardToClaim`

`= 2` and `rewardsToClaim = 2`.

2. The contract calculates `firstRewardToClaim = max(2, 0) = 2`.
3. Alice claims rewards `r2` and `r3`.
4. The contract updates `lastUnclaimedReward[alice] = 2 + 2 = 4`.
5. Alice's `lastUnclaimedReward` is now 4. She cannot call the `claimVaultSnapshotRewards` function again with `firstRewardToClaim = 0` and `rewardsToClaim = 2` to claim rewards `r0` and `r1`, making them permanently stuck.

Recommendation

Ensure that claims cannot become stuck in the protocol, while also adressing the finding W4 in parallel to prevent denial of service if additional parameter restrictions are added (such as `firstRewardToClaim == lastUnclaimedReward`).

Alternatively, properly document the behavior of skipped rewards becoming unclaimable to inform users of the risk.

Acknowledgment 1.1

The issue was acknowledged by the team, confirming that the reward-skipping functionality is intended.

Go back to Findings Summary

L1: `protocolFee` ignores disabled default fee

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	ProtocolFees.sol	Type:	Logic error

Description

The `FeeRegistry` contract stores protocol fee configuration, including both network-specific or default (network-agnostic) fees. These fees are configured via the `setProtocolFee` function, which takes in `enable` and `fee` parameters, with requirements only being that the `fee` parameter is less than `MAX_FEE`. This allows for setting the fee configuration to `(enable = false, fee > 0)` (i.e. disabling the fee while keeping the non-zero value).

Listing 2. Excerpt from FeeRegistry

```
241 function setProtocolFee(bytes32 id, bool enable, uint256 fee) public
    onlyOwner {
242     if (fee >= MAX_FEE) {
243         revert FeeTooHigh();
244     }
245
246     _feeRegistryStorage()._protocolFee[id] = _serializeFeeData(enable, fee);
247     emit SetProtocolFee(id, enable, fee);
248 }
```

In the `ProtocolFees` contract, the `protocolFee` function is used to get the current fee for a given network. When no network-specific fee is enabled, the function falls back to the default fee. However, this fallback does not check the `isEnabled` flag for the default entry. A disabled default fee with a non-zero value is still returned and applied.

Listing 3. Excerpt from ProtocolFees

```
67 function protocolFee(uint64 rewardsType, address network) public view returns
```

```
    (uint256) {
68      (bool isEnabled, uint256 fee) =
69          IFeeRegistry(FEE_REGISTRY).getProtocolFee(
    keccak256(abi.encode(REWARDS_FEE_ID, rewardsType, network)));
70      if (isEnabled) {
71          return fee;
72      }
73
74      (, fee) = IFeeRegistry(FEE_REGISTRY).getProtocolFee(
    keccak256(abi.encode(REWARDS_FEE_ID, rewardsType)));
75      return fee;
76  }
```

Exploit scenario

1. Bob, a governance actor, disables the default protocol fee by calling `setProtocolFee` with `enabled = false` and a non-zero `fee`. No network-specific override is set.
2. Alice, a network operator, initiates a reward distribution via `VaultSnapshotRewards.distributeVaultSnapshotRewards` or `CumulativeMerkleRewards.distributeCumulativeMerkleRewards`.
3. The contract calls `ProtocolFees.protocolFee(rewardsType, network)` to get the current fee for the given network. No network-specific fee is enabled, so the function falls back to the default.
4. The fallback ignores the `isEnabled` flag, which is set to `false`, and applies the non-zero fee. Recipients are charged despite governance having disabled the default fee.

Recommendation

Check the `isEnabled` flag for the default fee in the fallback branch. Return zero when the default fee is disabled.

Acknowledgment 1.1

The issue was acknowledged by the team, confirming that the

implementation is intended.

Go back to Findings Summary

L2: `CumulativeMerkleRewards` does not enforce network registry checks

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	CumulativeMerkleRewards.sol	Type:	Data validation

Description

Unlike the `VaultSnapshotRewards` contract, the `CumulativeMerkleRewards` contract does not enforce that a network exists in the registry across key entry points. This allows deposits to non-existent networks and operations on invalid networks:

- `depositCumulativeMerkleRewards` accepts deposits without verifying the network exists.

Listing 4. Excerpt from CumulativeMerkleRewards

```
218 function depositCumulativeMerkleRewards(address network, address token,
    uint256 amount) public nonReentrant {
219     uint256 balanceBefore = IERC20(token).balanceOf(address(this));
220     IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
221     amount = IERC20(token).balanceOf(address(this)) - balanceBefore;
222
223     if (amount == 0) {
224         revert InsufficientDeposit();
225     }
226
227     _cumulativeMerkleRewardsStorage()._balances[network][token] += amount;
228     emit DepositCumulativeMerkleRewards(network, token, amount);
229 }
```

- `setRewarder` writes `_rewarder[msg.sender]` with no registry check to ensure `msg.sender` is a registered network.

Listing 5. Excerpt from CumulativeMerkleRewards

```
275 function setRewarder(address rewarder_) public {
276     _cumulativeMerkleRewardsStorage()._rewarder[msg.sender] = rewarder_;
277     emit SetRewarder(msg.sender, rewarder_);
278 }
```

While the missing registry check in the `setRewarder` function does not result in material harm, the `depositCumulativeMerkleRewards` function, due to its missing network existence check, allows crediting `_balances[network][token]` for an unregistered network. Because no valid `rewarder(network)` exists in case of an unregistered network and withdrawals require `rewarder(network) == msg.sender`, these balances have no withdrawal path and remain stuck.

Exploit scenario

1. Alice, a network operator, approves tokens and calls `depositCumulativeMerkleRewards`, mistakenly specifying a non-existent network.
2. The contract credits `_balances[network][token]` for the non-existent network.
3. Alice is no longer able to withdraw her tokens and deposit them to the correct network, resulting in these balances remaining stuck.

Recommendation

Enforce a network existence check in the `depositCumulativeMerkleRewards` function.

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

L3: Missing zero-address check in `setCurator` locks role and fees

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	CuratorRegistry.sol, VaultSnapshotRewards.sol	Type:	Data validation

Description

The `setCurator` function in the `CuratorRegistry` contract allows setting the curator address to `address(0)` without any validation. Once the curator is set to the zero address, the role becomes permanently locked and cannot be reassigned. This occurs because the authorization check compares `msg.sender` against the stored curator address:

Listing 6. Excerpt from CuratorRegistry

```
69 if (address(uint160(value)) != msg.sender) {
70     revert NotAuthorized();
71 }
```

When the curator is `address(0)`, this condition evaluates to `address(0) != msg.sender`, which is always true for any valid caller, causing all subsequent `setCurator` calls to revert with `NotAuthorized()`. Additionally, any accumulated curator fees become permanently unclaimable because `claimCuratorFees` in `VaultSnapshotRewards` requires `msg.sender` to be the current curator:

Listing 7. Excerpt from VaultSnapshotRewards

```
420 if (ICuratorRegistry(CURATOR_REGISTRY).getCurator(vault) != msg.sender) {
421     revert NotCurator();
422 }
```

Since no address can satisfy `msg.sender == address(0)`, the fees are locked forever.

Exploit scenario

1. Alice, the vault owner, calls `setCurator(vault, curator)` to set Bob, the initial curator, for the vault.
2. Bob accumulates 10,000 tokens in curator fees over time.
3. Bob mistakenly calls `setCurator(vault, address(0))` intending to renounce the curator role, or a malicious Bob intentionally griefs the system.
4. The curator role is now permanently set to `address(0)`.
5. No one can call `setCurator` to fix this and all subsequent calls revert with `NotAuthorized()`.
6. The 10,000 tokens in accumulated fees and all future curator fees for this vault are permanently locked in the contract.

Recommendation

Add a zero address check in the `setCurator` function.

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

W1: `FeeRegistry` contract does not disable initializers

Impact:	Warning	Likelihood:	N/A
Target:	FeeRegistry.sol	Type:	Configuration

Description

The `FeeRegistry` contract is missing the `_disableInitializers()` internal function call in its `constructor`, making the implementation contract initializable (allowing an owner to be set).

Listing 8. Excerpt from FeeRegistry

```
64 constructor(address curatorRegistry_) {
65     CURATOR_REGISTRY = curatorRegistry_;
66 }
67
68 /* PUBLIC FUNCTIONS */
69
70 function initialize(address owner) public initializer {
71     __Ownable_init(owner);
72 }
```

Recommendation

Add the `_disableInitializers()` internal function call in `FeeRegistry.constructor` to prevent the implementation contract from being initialized.

Fix 1.1

The issue was fixed by the team in the commit `3cbad0f`[1] by adding the `_disableInitializers()` internal function call in the `FeeRegistry.constructor`.

Go back to Findings Summary

W2: Ambiguous reverts degrade UX

Impact:	Warning	Likelihood:	N/A
Target:	Rewards.sol, CumulativeMerkleRewards.sol	Type:	Data validation

Description

The protocol contains multiple ambiguous revert error paths, instead of custom descriptive errors that would provide more context to the user:

- The `CumulativeMerkleRewards.distributeCumulativeMerkleRewards` function relies on underflow revert in case of insufficient deposited funds. When distributing, the contract computes `totalDistributionAmount` and decrements `_balances[network][token]` without checking sufficiency. If the balance is too low (e.g. deposit did not cover fees), this underflows and reverts.

Listing 9. Excerpt from CumulativeMerkleRewards

```
196     TokenAmount calldata totalAmount = totalAmounts[i];
197     if (totalAmount.chainId != uint64(block.chainid)) {
198         continue;
199     }
200
201     uint256 distributionAmount = totalAmount.amount -
    lastTotalAmount(network, totalAmount.token);
202     uint256 totalDistributionAmount = _addProtocolFeesToDistribution(
203         uint64(IRewards.RewardsType.CUMULATIVE_MERKLE), network,
    totalAmount.token, distributionAmount
204     );
205
206     _cumulativeMerkleRewardsStorage()._balances[network][totalAmount.token]
    -= totalDistributionAmount;
207
    _cumulativeMerkleRewardsStorage()._lastTotalAmounts[network][totalAmount.tok
    en] = totalAmount.amount;
208 }
```

- The `CumulativeMerkleRewards.withdrawCumulativeMerkleRewards` function also relies on underflow revert without checking whether sufficient balance is available. During withdrawal, the contract decrements `_balances[network][token]` by the `amount` parameter. If the balance is too low, an underflow revert occurs.

Listing 10. Excerpt from CumulativeMerkleRewards

```
232 function withdrawCumulativeMerkleRewards(address recipient, address network,
    address token, uint256 amount)
233     public
234     nonReentrant
235 {
236     if (rewarder(network) != msg.sender) {
237         revert NotRewarder();
238     }
239
240     _cumulativeMerkleRewardsStorage()._balances[network][token] -= amount;
241     IERC20(token).safeTransfer(recipient, amount);
242
243     emit WithdrawCumulativeMerkleRewards(network, token, amount);
244 }
```

Recommendation

Add the appropriate checks with descriptive revert errors (e.g. `InsufficientDeposited()`) to the affected functions clearly communicate the issue and improve user experience.

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

W3: Inconsistent zero-address recipient validation

Impact:	Warning	Likelihood:	N/A
Target:		Type:	Data validation

Description

When claiming or withdrawing rewards or fees, the protocol usually expects a `recipient` address to send the requested funds to. Several `claim*` functions include a zero-address check (reverting with `InvalidRecipient()`) to prevent loss of funds, namely `VaultSnapshotRewards.claimVaultSnapshotRewards` and `VaultSnapshotRewards.claimOperatorFees`.

Listing 11. Excerpt from VaultSnapshotRewards

```
271 function claimVaultSnapshotRewards(
272     address recipient,
273     address network,
274     address token,
275     address vault,
276     uint256 lastUnclaimedRewards,
277     uint256 firstRewardToClaim,
278     uint256 rewardsToClaim,
279     bytes[] memory activeSharesHints
280 ) public nonReentrant {
281     if (recipient == address(0)) {
282         revert InvalidRecipient();
283     }
```

Many similar functions, however, are missing this check:

1. `VaultSnapshotRewards.claimCuratorFees` transfers to recipient without a zero-address check.
2. `CumulativeMerkleRewards.withdrawCumulativeMerkleRewards` transfers to recipient without a zero-address check.

3. `CumulativeMerkleRewards.claimCumulativeMerkleRewards` transfers to `recipient` without a zero-address check.
4. `ProtocolFees.claimProtocolFees` transfers to `recipient` without a zero-address check.

Recommendation

Standardize recipient validation across all `claim*` and `withdraw*` functions.

Fix 1.1

The issue was fixed by the team in the commit `d45dfe0`[2] by adding the zero-address check to the left-over functions.

Go back to Findings Summary

W4: Division-by-zero in `VaultSnapshotRewards.claimOperatorFees`

Impact:	Warning	Likelihood:	N/A
Target:	VaultSnapshotRewards.sol	Type:	Denial of service

Description

The `claimOperatorFees` function iterates through each operator and computes operator rewards for `NETWORK_RESTAKE` entries as `operatorNetworkSharesAt.mulDiv(operatorsFees, totalOperatorNetworkSharesAt)`. When a distribution was created at a timestamp where `totalOperatorNetworkSharesAt` was zero, the contract correctly sets `operatorsFees = 0` during distribution. However, at claim time the function unconditionally calls mulDiv with the historical `totalOperatorNetworkSharesAt` as denominator. If that denominator is 0, `Math.mulDiv` reverts with a division by zero error, causing the entire claim to revert. This causes a denial of service for straightforward batched claims that include such an index. While callers can skip indices via the `firstRewardToClaim` parameter, doing so worsens the user experience and breaks automated batching flows.

Listing 12. Excerpt from VaultSnapshotRewards

```
366 uint256 amount;
367 uint256 networkRestakeDelegatorCounter;
368 for (uint256 i; i < rewardsToClaim; ++i) {
369     RewardDistribution storage reward;
370     unchecked {
371         reward = rewardsByTokenNetwork[firstRewardToClaim + i];
372     }
373     if (reward.delegatorType == uint64(DelegatorType.NETWORK_RESTAKE)) {
374         amount += INetworkRestakeDelegator(reward.delegator)
375             .operatorNetworkSharesAt(
376                 network.subnetwork(reward.subnetworkId),
377                 msg.sender,
378                 reward.timestamp,
```

```
379                 operatorNetworkSharesHints[networkRestakeDelegatorCounter]
380             )
381             .mulDiv(
382                 reward.operatorsFees,
383                 INetworkRestakeDelegator(reward.delegator)
384                     .totalOperatorNetworkSharesAt(
385                         network.subnetwork(reward.subnetworkId),
386                         reward.timestamp,
387 
    totalOperatorNetworkSharesHint[networkRestakeDelegatorCounter]
388                     )
389             );
```

Recommendation

Avoid calling `mulDiv` with a zero denominator for `NETWORK_RESTAKE` entries by adding an explicit check to ensure the denominator is non-zero before performing the calculation.

Acknowledgment 1.1

The issue was acknowledged by the team as intended behavior.

Go back to Findings Summary

I1: Unused using-for directive

Impact:	Info	Likelihood:	N/A
Target:	ProtocolFees.sol	Type:	Code quality

Description

The `ProtocolFees` contract contains an unused using-for directive.

Listing 13. Excerpt from ProtocolFees

```
16 using Math for uint256;
```

Recommendation

Remove the unused using-for directive.

Fix 1.1

The issue was fixed by the team in the commit `30f5381`[3] by removing the unused using-for directive.

Go back to Findings Summary

I2: Several public functions should be external

Impact:	Info	Likelihood:	N/A
Target:		Type:	Code quality

Description

Several functions are declared `public` even though they are never invoked from within their contract or inheritance tree. Functions that are safe to change to `external`:

- `CumulativeMerkleRewards.balance`
- `CumulativeMerkleRewards.setProtocol`
- `CumulativeMerkleRewards.distributeCumulativeMerkleRewards`
- `CumulativeMerkleRewards.depositCumulativeMerkleRewards`
- `CumulativeMerkleRewards.withdrawCumulativeMerkleRewards`
- `CumulativeMerkleRewards.setRewarder`
- `CuratorRegistry.initialize`
- `CuratorRegistry.getCuratorAt`
- `CuratorRegistry.getCurator`
- `CuratorRegistry.setCurator`
- `FeeRegistry.initialize`
- `FeeRegistry.getOperatorsFeeAt`
- `FeeRegistry.getOperatorsFee`
- `FeeRegistry.getCuratorFeeAt`
- `FeeRegistry.getCuratorFee`
- `FeeRegistry.getProtocolFee`

- `FeeRegistry.setOperatorsFee`
- `FeeRegistry.setOperatorsNetworkFee`
- `FeeRegistry.setCuratorFee`
- `FeeRegistry.setCuratorNetworkFee`
- `FeeRegistry.setProtocolFee`
- `ProtocolFees.claimProtocolFees`
- `Rewards.initialize`
- `VaultSnapshotRewards.rewardsLength`
- `VaultSnapshotRewards.rewards`
- `VaultSnapshotRewards.distributeVaultSnapshotRewards`
- `VaultSnapshotRewards.claimOperatorFees`
- `VaultSnapshotRewards.claimCuratorFees`

Recommendation

Change the visibility of the listed functions from `public` to `external`.

Acknowledgment 1.1

The issue was acknowledged by the team, as the visibility of the functions does not affect contract code size in Solidity version 0.8.26.

Go back to Findings Summary

[1] full commit hash: `3cbad0f2cc56619be51e1c9d028be930315a0a61`, link to commit

[2] full commit hash: `d45dfe0852ce02ac8f383bf266192a1fdcf91d19`, link to commit

[3] full commit hash: `30f5381cfd6e7614bae6d7170d6b3f86c31319a7`, link to commit

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Audit Report | Symbiotic: Rewards V2, January 12, 2026.

Appendix B: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

B.1. Detectors

```
wake detect unused-using-for

[WARNING][HIGH] Unused contract in using-for directive [unused-using-for]
  13 /// @notice Contract for processing protocol fees.
  14 abstract contract ProtocolFees is OwnableUpgradeable, IProtocolFees {
  15     using SafeERC20 for IERC20;
❱ 16     using Math for uint256;
  17
  18     /* CONSTANTS */
  19
src/contracts/ProtocolFees.sol
```

Figure 1. Unused using-for directives

```
wake detect unused-public-functionality

[INFO][HIGH] Function is declared as public but could be external since it has no internal references [unused-public-functi
  45
  46     /* PUBLIC FUNCTIONS */
  47
❱ 48     function initialize() public initializer {}
  49
  50     /// @inheritdoc ICuratorRegistry
  51     function getCuratorAt(address vault, uint48 timestamp, bytes memory hint) public view returns (address) {
src/contracts/CuratorRegistry.sol

[INFO][HIGH] Function is declared as public but could be external since it has no internal references [unused-public-functi
  48     function initialize() public initializer {}
  49
  50     /// @inheritdoc ICuratorRegistry
❱ 51     function getCuratorAt(address vault, uint48 timestamp, bytes memory hint) public view returns (address) {
  52         return address(uint160(_curatorRegistryStorage()._curators[vault].upperLookupRecent(timestamp, hint)));
  53     }
  54
src/contracts/CuratorRegistry.sol

[INFO][HIGH] Function is declared as public but could be external since it has no internal references [unused-public-functi
  53     }
  54
  55     /// @inheritdoc ICuratorRegistry
❱ 56     function getCurator(address vault) public view returns (address) {
  57         return address(uint160(_curatorRegistryStorage()._curators[vault].latest()));
  58     }
  59
src/contracts/CuratorRegistry.sol

[INFO][HIGH] Function is declared as public but could be external since it has no internal references [unused-public-functi
  58     }
  59
  60     /// @inheritdoc ICuratorRegistry
❱ 61     function setCurator(address vault, address curator) public {
  62         if (!IRegistry(VAULT_FACTORY).isEntity(vault)) {
  63             revert NotVault();
  64         }
src/contracts/CuratorRegistry.sol
```

Figure 2. Unused public functionality

Appendix C: Wake Arena Findings

This section lists vulnerabilities identified by Wake Arena, an LLM-powered audit tool used for AI-assisted vulnerability discovery during the audit. Wake Arena leverages large language models to understand code context and reason about complex contract behavior, complementing manual review.

C.1. Discovered Findings

The following table contains true-positive findings identified by Wake Arena. These findings are included regardless of whether they were also discovered independently by auditors during manual review.

Finding title	Severity	Reported	Discoverer
L1: `protocolFee` ignores disabled default fee	Low	1.0	Wake Arena, Auditor
L3: Missing zero-address check in `setCurator` locks role and fees	Low	1.0	Wake Arena, Auditor
W4: Division-by-zero in `VaultSnapshotRewards.claimOperatorFees`	Warning	1.0	Wake Arena

Table 4. Table of findings identified by Wake Arena

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz