Lido Vault Middleware Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Stalin

Immeas

December 19, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Lido Defi-interface provides an ERC4626-compliant middleware layer that wraps external ERC4626 vaults (e.g., Morpho strategies) and adds fee harvesting, reward distribution, inflation-attack defenses, and a robust emergency withdrawal/recovery flow. The system is structured as a reusable `Vault` core, an `EmergencyVault` extension.

The Vault overrides all standard **preview functions** (e.g., `previewDeposit`). These functions simulate the pending fee harvest, ensuring that the values displayed in user interfaces match the exact execution results—accounting for fee dilution—down to the last wei.

The system utilizes a **High Water Mark** model with synchronous fee harvesting to ensure self-contained economics without reliance on external keepers capturing any immediate surplus value generated in the underlying protocol (e.g., share appreciation or airdrops) is recognized as profit and distributed as free yield.

To handle critical failures such as hacks or **Emergency Withdrawal Insolvency**, the system distinguishes between temporary operational pauses and permanent solvency crises.

The architecture enforces a **one-way state transition** for solvency events. Upon a confirmed loss, the vault enters *Emergency Mode* to evacuate remaining funds, and then permanently transitions to *Recovery Mode*. Once in this state, deposits are disabled forever. Users can only exit via `redeem`, which switches to a mathematical snapshot: it distributes the recovered assets strictly **pro-rata** (`recoveryAssets / recoverySupply`). This ensures that any **Implicit Loss** is shared equally and transparently among all remaining shareholders.

Key Components

1. `Vault` **(Base Layer):** Handles standard ERC4626 accounting, access control, inflation protection via virtual offsets, and performance fee harvesting.
2. `EmergencyVault` **(Safety Layer):** Extends the base vault with "Emergency Mode" (pulling funds to safety) and "Recovery Mode" (pro-rata distribution of recovered assets).
3. `ERC4626Adapter` **(Integration Layer):** A concrete implementation that forwards capital into target strategies (e.g., Aave, MetaMorpho) while respecting the constraints of the parent layers.

4. `RewardDistributor`: A dedicated contract (typically set as the treasury) that redeems accumulated shares and distributes assets to recipients based on immutable allocation percentages.

5 Audit Scope

The audit scope was limited to:

```
src/adapters/ERC4626Adapter.sol
src/EmergencyVault.sol
src/RewardDistributor.sol
src/Vault.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Lido Vault Middleware smart contracts provided by Lido. In this period, a total of 13 issues were found.

During the audit, we identified 5 low severity issue, with the remainder being informational and gas optimizations.

The low severity issues are a series of enhacements to make the system more robust and compatible with extertnal vaults as well as to follow the ERC4626 standard.

The informational issues encompass a series of non-critical enhancements focused on robustness, clarity, and maintainability.

Summary

Project Name	Lido Vault Middleware
Repository	defi-interface
Commit	99fd2b2c64c3...
Fix Commit	fc15b104d385...
Audit Timeline	Dec 15th - Dec 19th, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	5
Informational	6
Gas Optimizations	2
Total Issues	13

Summary of Findings

[L-1] `ERC4626Adapter::maxMint` reverts for uncapped target vaults	Resolved
[L-2] Non-compliant events emitted on vault deposits and withdrawals	Resolved
[L-3] Griefing attack on depositors by manipulating the exchange rate during `recoveryMode` via a donation of `TARGET_VAULT`s shares in between `emergencyMode` and `recoveryMode`	Resolved
[L-4] `EmergencyVault::activateRecovery` can be DoS by a reverting `TARGET_VAULT` calls	Resolved
[L-5] `ERC4626Adapter::maxMint` doesn't consider pending fees to be harvested which leads to under-calculating the real shares that can be minted	Resolved
[I-1] Use named mappings to explicitly denote the purpose of keys and values	Resolved
[I-2] In Solidity don't initialize to default values	Resolved
[I-3] `Vault::decimals` does not reflect correct decimals when `OFFSET` is used	Resolved
[I-4] `EmergencyVault::activateRecovery` NatSpec references wrong event name	Resolved
[I-5] `nonReentrant` is not the first modifier	Resolved
[I-6] Not including `_decimalsOffset` when calculating the fee shares	Acknowledged
[G-1] Use `ReentrancyGuardTransient` for faster `nonReentrant` modifiers	Resolved
[G-2] Use more efficient method of reading recipient account and basis points	Resolved

7 Findings

7.1 Low Risk

7.1.1 `ERC4626Adapter::maxMint` reverts for uncapped target vaults

Description: `ERC4626Adapter::maxMint` forwards `TARGET_VAULT.maxDeposit(address(this))` into `_convertToShares`:

```
function maxMint(address /* user */ ) public view override returns (uint256) {
    if (paused() || emergencyMode) return 0;
    uint256 maxAssets = TARGET_VAULT.maxDeposit(address(this));
    return _convertToShares(maxAssets, Math.Rounding.Floor);
}
```

The EIP-4626 standard for `maxMint/maxDeposit` states that:

> MUST return `2 ** 256 - 1` if there is no limit on the maximum amount of assets that may be deposited.

Standard ERC4626 implementations (like OpenZeppelin's), `maxDeposit` / `maxMint` therefore return `type(uint256).max` as a default. Passing this value into `_convertToShares` causes `Math.mulDiv` to overflow and revert, so `maxMint` itself reverts instead of returning a valid upper bound.

Impact: `ERC4626Adapter::maxMint` reverts for vaults without any cap which disagrees with the EIP-4626 standard that `maxMint` "MUST NOT revert.".

Proof of Concept: Add the following test to `ERC4626Adapter.MaxDeposit.t.sol`:

```
function test_MaxMintReverts() public {
    vm.expectRevert(stdError.arithmeticError);
    vault.maxMint(alice);
}
```

Recommended Mitigation: Add a check for the "unbounded" result from the target vault and avoid feeding `type(uint256).max` into `_convertToShares`. For example:

```
function maxMint(address /* user */ ) public view override returns (uint256) {
    if (paused() || emergencyMode) return 0;

    uint256 maxAssets = TARGET_VAULT.maxDeposit(address(this));
    if (maxAssets == type(uint256).max) {
        // Underlying vault is effectively uncapped: propagate this instead of converting
        return type(uint256).max;
    }

    return _convertToShares(maxAssets, Math.Rounding.Floor);
}
```

Lido: Fixed in commit `af57eb5`

Cyfrin: Verified. Suggested fix implemented.

7.1.2 Non-compliant events emitted on vault deposits and withdrawals

Description: `Vault` emits custom `Deposited` and `Withdrawn` events in `Vault::deposit/mint` and `withdraw/redeem`, while EIP-4626 specifies standard `Deposit` and `Withdraw` event names.

Impact: The implementation is non-conformant at the event level and may break tooling or integrations that rely on the canonical ERC-4626 events for indexing or accounting.

Recommended Mitigation: Emit standard `Deposit` and `Withdraw` events with the exact EIP-4626 signatures instead of the custom `Deposited` / `Withdrawn`.

Lido: Fixed in commit `52217ad`

Cyfrin: Verified. Correct EIP-4626 events are not emitted.

7.1.3 Griefing attack on depositors by manipulating the exchange rate during `recoveryMode` via a donation of `TARGET_VAULT`s shares in between `emergencyMode` and `recoveryMode`

Description: During the activation of the recovery mode is executed a call to harvest fees, which, in the scenario that it detects any profit since the last update to `lastTotalAssets` would mint more shares. But, given the design of the system, where once the `recoveryMode` is enabled:

- It is no longer possible to withdraw from the TARGET_VAULT
- Harvesting fees consider the LidoVault's holdings on the TARGET_VAULT as part of the totalAssets, despite any leftover TARGET_VAULT's shares being no redeemable from that point onwards.

The exchange rate for the vault once the recoveryMode kicks in is based on the actual balance of underlyingToken on the LidoVault and the totalSupply at the moment of the recovery mode activation.

That setup allows for a griefing attack where the execution to activate the recovery mode is front-run, and TARGET_VAULT's shares are donated into the LidoVault. This donation will effectively increase the totalAssets, tricking the system into thinking that there are profits to charge fees on, as such, minting new shares, which effectively dilutes the exchange rate compared to the actual underlyingTokens on the LidoVault's balance.

```
    function activateRecovery() external virtual onlyRole(EMERGENCY_ROLE) nonReentrant {
        if (recoveryMode) revert RecoveryModeAlreadyActive();
        if (!emergencyMode) revert EmergencyModeNotActive();

        //@audit => The donation of TARGET_VAULT shares causes more shares to be minted
        _harvestFees();

        uint256 actualBalance = IERC20(asset()).balanceOf(address(this));
        if (actualBalance == 0) revert InvalidRecoveryAssets(actualBalance);

        uint256 supply = totalSupply();
       ...

        recoveryAssets = actualBalance;
        recoverySupply = supply;
        recoveryMode = true;

        emit RecoveryModeActivated(actualBalance, supply, protocolBalance, implicitLoss);
    }

    function convertToAssets(uint256 shares) public view virtual override returns (uint256) {
        //@audit => exchange rate during recovery mode no longer considers the TARGET_VAULT's shares
        ↪  worth in underlying token.
        if (recoveryMode) {
            return shares.mulDiv(recoveryAssets, recoverySupply, Math.Rounding.Floor);
        }
        return super.convertToAssets(shares);
    }
```

Impact: The recovery exchange rate can be manipulated, effectively causing depositors to recover fewer tokens than they could've otherwise gotten.

Given that this grief attack requires the "attacker" to incur a loss, the probability is low; nevertheless, the impact is considerable, given that depositors would incur a loss of assets.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Vault} from "src/Vault.sol";
```

```
import "./ERC4626AdapterTestBase.sol";

contract ERC4626AdapterPoCs is ERC4626AdapterTestBase {

    function test_PoC_manipulateShareRatioOnRecoveryMode() public {
        //@audit-info => The mitigation would be to swap `_harvestFees()` to `emergencyWithdraw()` and
        ↪  add a function to allow Governance withdrawing from vault once recoveryMode is enabled!
        uint256 depositAmount = 100e6;
        vault.setRewardFee(2000);

        vm.prank(alice);
        vault.deposit(depositAmount, alice);

        vault.emergencyWithdraw();
        assertEq(vault.totalAssets(),depositAmount);

        uint256 aliceAssetsDuringEmergency = vault.convertToAssets(vault.balanceOf(alice));
        assertEq(aliceAssetsDuringEmergency, depositAmount);

        uint256 snapshot = vm.snapshot();
        {
            //@audit-info => A donation to the LidoVault of TARGET_VAULT's shares
            vm.startPrank(bob);
            usdc.approve(address(targetVault), depositAmount);
            targetVault.deposit(depositAmount, address(vault));
            vm.stopPrank();

            vault.activateRecovery();
            assertEq(depositAmount, vault.recoveryAssets());

            //@audit => Manipulation -> depositor gets less assets that could've otherwise got
            uint256 aliceAssetsOnRecoveryMode = vault.convertToAssets(vault.balanceOf(alice));
            assertTrue(aliceAssetsDuringEmergency > aliceAssetsOnRecoveryMode);

            emit log_named_uint("aliceAssetsDuringEmergency: ", aliceAssetsDuringEmergency);
            emit log_named_uint("aliceAssetsOnRecoveryMode: ", aliceAssetsOnRecoveryMode);
        }

        vm.revertTo(snapshot);

        vault.activateRecovery();

        //@audit => No manipulation -> depositor gets the correct exchange rate during recoveryMode
        uint256 aliceAssetsOnRecoveryMode = vault.convertToAssets(vault.balanceOf(alice));
        assertEq(aliceAssetsOnRecoveryMode, aliceAssetsDuringEmergency);
    }
}
```

Recommended Mitigation:

1. Consider harvesting the fees during the emergency withdrawal process, rather than during the activation of recovery mode.
2. Consider allowing the `ERC4626Adapter::recoverERC20` function to sweep TARGET_VAULT's leftover tokens once the `recoveryMode` is enabled.

This is a more defensive strategy to protect users' funds by prioritizing the preservation of the expected exchange rate based on the actual underlyingTokens on the LidoVault and deferring the potential gains in fees as a secondary action by sweeping any leftover TARGET_VAULT's shares.

Lido Fixed in commit 4fd0eb7 and ee29862

Cyfrin: Verified. `_harvestFees` is now called during the activation of the `emergencyMode`. Any leftover `TARGET_VAULT` shares can now be recovered by governance when `recoveryMode` has been enabled via the `recoverERC20`.

7.1.4 `EmergencyVault::activateRecovery` can be DoS by a reverting `TARGET_VAULT` calls

Description: `EmergencyVault::activateRecovery` does external calls to the `TARGET_VAULT` through `_harvestFees()` (which in turn calls `totalAssets()` then `_getProtocolBalance()`) and `_getProtocolBalance()` directly. `_getProtocolBalance()` performs external view calls to `TARGET_VAULT.balanceOf(address(this))` and `TARGET_VAULT.convertToAssets(...)`, which can revert if the target vault is compromised/upgradeable/misbehaving. As a result the recovery activation path can be DoSed by a reverting target vault, even when the adapter already holds recoverable assets locally.

Impact: In an incident where `TARGET_VAULT` becomes untrusted and its view functions revert, the vault may be unable activate `recoveryMode`. This can lock any assets already recovered to the vault contract, since users cannot redeem under the recovery flow until `recoveryMode` is set.

Recommended Mitigation: Refactor `activateRecovery()` to avoid relying on external target vault calls that can revert in incident scenarios. Remove `_getProtocolBalance()` from `activateRecovery()` (it's only for event info) and let the fee harvesting be done manually after calls to `emergencyWithdraw` (if the `emergencyMode` restriction is removed or limited to only `EMERGENCY_ROLE`). Alternatively, add `_harvestFees()` at the end of `emergencyWithdraw()`.

Lido: Fixed in commit ee29862

Cyfrin: Verified. `_harvestFees()` moved to `emergencyWithdraw()` and a `try/catch` added around the call to `_getProtocolBalance`.

7.1.5 `ERC4626Adapter::maxMint` doesn't consider pending fees to be harvested which leads to under-calculating the real shares that can be minted

Description: The ERC4626Adapter::maxMint function computes the maximum quantity of shares that the Vault may mint by converting the maximum depositable assets in the underlying TARGET_VAULT into corresponding vault shares. However, this conversion process does not account for any pending fees.

As a consequence, the returned share amount underestimates the actual maximum mintable shares on the Vault. Specifically, upon harvesting the pending fees, additional shares are minted, thereby increasing the total supply and the effective conversion rate from assets to shares. This results in a post-harvest scenario where a greater number of shares can be minted for the same quantity of deposited assets than what `maxMint` initially indicates.

Impact: `maxMint` won't accurately report the actual maximum number of shares that can be minted.

Proof of Concept: Add the next test to `ERC4626Adapter.MaxDeposit.t.sol` test file

```
    function test_PoC_MaxMint_DoesNotConsiderPendingYield() public {
        uint256 yield = 50_00e6;
        targetVault.setLiquidityCap(500_000e6);

        _seedVaults(yield);

        //@audit-info => Vault has pending yield

        uint256 snapshot = vm.snapshot();
            uint256 maxDeposit = vault.maxDeposit(alice);
            vm.prank(alice);
            vault.deposit(maxDeposit, alice);
            assertEq(vault.maxDeposit(alice), 0);
            assertEq(vault.maxMint(alice), 0);
        vm.revertTo(snapshot);

        //@audit-info => Given that Vault has pending yield, maxMint() is not accurate and will mint
        ↪  less shares than the actual maxMint post harvesting fees
        uint256 maxMintShares = vault.maxMint(alice);
        vm.prank(alice);
        vault.mint(maxMintShares, alice);
        assertGt(vault.maxDeposit(alice), 0);
        assertGt(vault.maxMint(alice), 0);
```

```
        //@audit-info => After attempting to mint the maxShares reported by the vault (and fees have
        ↪  been harvested during the mint), a second mint is possible when it shouldn't be because the
        ↪  previous mint was supposed to mint the max
        maxMintShares = vault.maxMint(alice);
        vm.prank(alice);
        vault.mint(maxMintShares, alice);
        assertEq(vault.maxDeposit(alice), 0);
        assertEq(vault.maxMint(alice), 0);
    }

    function _seedVaults(uint256 yield) internal {
        vm.prank(alice);
        vault.deposit(100_000e6, alice);

        vm.startPrank(bob);
        usdc.approve(address(targetVault), 100_000e6);
        targetVault.deposit(100_000e6, bob);
        vault.deposit(100_000e6, alice);
        vm.stopPrank();

        // mint yield to targetVault
        usdc.mint(address(targetVault), yield);
    }
```

Recommended Mitigation: Consider calculating the amount of shares by taking into account the pending fees to be harvested, similar to how the `previewMint` and `previewDeposit` functions do.

Lido: Fixed in commit fc15b10.

Cyfrin: Verified. `maxMint` calls `previewDeposit` forwarding the `maxAssets` that can be deposited on the `TARGET_VAULT`. `previewDeposit` correctly accounts for any pending fees, meaning the calculated number of shares for the `maxAssets` correctly represents the actual maximum shares that can be minted post-harvesting pending fees.

7.2 Informational

7.2.1 Use named mappings to explicitly denote the purpose of keys and values

Description: Use named mappings to explicitly denote the purpose of keys and values:

```
RewardDistributor.sol
52:     mapping(address => bool) private recipientExists;
```

Lido: Fixed in commit 4898c26.

Cyfrin: Verified.

7.2.2 In Solidity don't initialize to default values

Description: In Solidity don't initialize to default values:

```
RewardDistributor.sol
143:         uint256 totalBps = 0;
145:         for (uint256 i = 0; i < recipients_.length; i++) {
230:         for (uint256 i = 0; i < recipientsLength; i++) {
```

Lido: Fixed in commit 4898c26 for `totalBps`.

Cyfrin: Verified.

7.2.3 `Vault::decimals` does not reflect correct decimals when `OFFSET` is used

Description: `Vault::decimals` simply returns the asset decimals:

```
function decimals() public view virtual override(ERC20, ERC4626) returns (uint8) {
    return IERC20Metadata(asset()).decimals();
}
```

This is different to the OpenZeppelin ERC4626 implementation, where they account for any decimal offset:

```
function decimals() public view virtual override(IERC20Metadata, ERC20) returns (uint8) {
    return _underlyingDecimals + _decimalsOffset();
}
```

Impact: Although the issue will not cause any calculation errors it will cause the exchange rate and share tokens to look strange.

Recommended Mitigation: Consider replicating the OpenZeppelin implementation and adding the `OFFSET` to the decimals:

```
- return IERC20Metadata(asset()).decimals();
+ return IERC20Metadata(asset()).decimals(); + OFFSET;
```

Lido: FIxed in commit 9ce9c0a

Cyfrin: Verified. `OFFSET` is now added to the decimals.

7.2.4 `EmergencyVault::activateRecovery` NatSpec references wrong event name

Description: The NatSpec for `EmergencyVault::activateRecovery` states that it emits `RecoveryActivated`:

```
*      Emits RecoveryActivated(actualBalance, totalSupply, protocolBalance, implicitLoss)
```

While it actually emits: `RecoveryModeActivated`

```
emit RecoveryModeActivated(actualBalance, supply, protocolBalance, implicitLoss);
```

Consider changing the NatSpec to refer the correct event name.

Lido: Fixed in commit 3d89267

Cyfrin: Verified.

7.2.5 `nonReentrant` is not the first modifier

Description: `EmergencyVault::emergencyWithdraw` and `EmergencyVault::activateRecovery` place `nonReentrant` as the second modifier rather than first. To protect against reentrancy in other modifiers, the `nonReentrant` modifier should be the first modifier in the list of modifiers.

Lido: Fixed in commit 3d89267

Cyfrin: Verified.

7.2.6 Not including `_decimalsOffset` when calculating the fee shares

Description: Formulas converting assets to shares across the codebase utilize the virtual supply to prevent rate manipulation. However, when calculating the number of shares to be minted for accrued fees, the conversion formula does not call `_calculateFeeShares()` and omits `_decimalsOffset()` when calculating the fee shares.

Add the next PoC to `ERC4626Adapter.Fees.t.sol`.

```
    function test_PoC_InflateRatioViaFees() public {
        vm.prank(alice);
        uint256 alice_receivedShares = vault.deposit(1, alice);

        emit log_named_uint("totalAssets", vault.totalAssets());
        emit log_named_uint("totalSupply", vault.totalSupply());

        emit log_named_uint("assets per wei of share", vault.convertToAssets(alice_receivedShares));

        usdc.mint(address(targetVault), 1_000e18); //
        vault.harvestFees();

        emit log_named_uint("totalSupply", vault.totalSupply());
        emit log_named_uint("assets per wei of share", vault.convertToAssets(alice_receivedShares));

        vm.prank(bob);
        uint256 bob_receivedShares = vault.deposit(100e18, bob);

        uint256 bobAssetsBeforeRedeem = usdc.balanceOf(bob);

        vm.prank(bob);
        vault.redeem(bob_receivedShares, bob, bob);

        uint256 bobAssetsAfterRedeem = usdc.balanceOf(bob);

        emit log_named_uint("bob assets withdrawn", bobAssetsAfterRedeem
         - bobAssetsBeforeRedeem);
        assertTrue(bobAssetsAfterRedeem - bobAssetsBeforeRedeem > 99e18);

        // @audit //
        // With current formula, bob withdraws: 99999880924647933225 [9.999e19])

        // With formula using _decimalsOffset(): val: 99999803787934739453 [9.999e19])

        // @audit => Difference is neglegible for the required amount to donate to inflate the ratio //

    }
```

Lido: Acknowledged. Precision loss doesn't exceed one wei, no impact to the value received by the fee receiver.

7.3 Gas Optimization

7.3.1 Use `ReentrancyGuardTransient` for faster `nonReentrant` modifiers

Description: Use ReentrancyGuardTransient for faster `nonReentrant` modifiers:

```
Vault.sol
10:import {ReentrancyGuard} from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
32:abstract contract Vault is ERC4626, ERC20Permit, AccessControl, ReentrancyGuard, Pausable {
```

Lido: Fixed in commit 3d89267.

Cyfrin: Verified.

7.3.2 Use more efficient method of reading recipient account and basis points

Description: In `RewardsDistributor::getRecipient` reduce gas cost 791 -> 716 by:

```
function getRecipient(uint256 index) external view returns (address account, uint256 basisPoints) {
    Recipient storage recipient = recipients[index];
    (account, basisPoints) = (recipient.account, recipient.basisPoints);
}
```

In `RewardsDistributor::distribute` reduce gas costs 59254 -> 59172 by:

```
for (uint256 i = 0; i < recipientsLength; i++) {
    Recipient storage recipient = recipients[i];
    (address account, uint256 basisPoints) = (recipient.account, recipient.basisPoints);

    uint256 amount = (balance * basisPoints) / MAX_BASIS_POINTS;

    if (amount > 0) {
        tokenContract.safeTransfer(account, amount);
        emit RecipientPaid(account, token, amount);
    }

    totalAmount += amount;
}
```

Proof of Concept: To verify in `RewardDistributor.t.sol`:

1) In function `test_ReplaceRecipient_Succeeds` add snapshot after last call:

```
function test_ReplaceRecipient_Succeeds() public {
    RewardDistributor distributor = _deployDefaultDistributor();
    address newRecipient = makeAddr("newRecipient");

    (address oldRecipient,) = distributor.getRecipient(0);

    vm.expectEmit(true, true, true, true);
    emit RewardDistributor.RecipientReplaced(0, oldRecipient, newRecipient);

    vm.prank(admin);
    distributor.replaceRecipient(0, newRecipient);

    (address updatedRecipient,) = distributor.getRecipient(0);
+   vm.snapshotGasLastCall("RewardsDistributor", "getRecipient");
    assertEq(updatedRecipient, newRecipient);
}
```

2) In function `test_Distribute_DistributesAccordingToBps` add snapshot after first call:

```
function test_Distribute_DistributesAccordingToBps() public {
```

```
    RewardDistributor distributor = _deployDefaultDistributor();
    uint256 amount = 10_000e6;
    asset.mint(address(distributor), amount);

    address[] memory recipients = new address[](2);
    recipients[0] = recipientA;
    recipients[1] = recipientB;

    uint256[] memory expectedAmounts = new uint256[](2);
    expectedAmounts[0] = (amount * 4_000) / MAX_BPS;
    expectedAmounts[1] = (amount * 6_000) / MAX_BPS;

    vm.recordLogs();
    vm.prank(admin);
    distributor.distribute(address(asset));
+   vm.snapshotGasLastCall("RewardsDistributor", "distribute");

    Vm.Log[] memory entries = vm.getRecordedLogs();
    // snip remaining code...
```

3) Run the test contract: `forge test --match-contract RewardDistributorTest`
4) Examine the gas snapshots: `more snapshots/RewardsDistributor.json`
5) After making the recommended changes, execute 3) and 4) again

Lido: Fixed in commit 4898c26.

Cyfrin: Verified.