BLOCKSEC

Security Audit
Report for EigenPie Contracts

Date: April 21, 2025 **Version:** 1.4
Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Magpiexyz
Target	EigenPie Contracts

Version History

Version	Date	Description
1.0	October 23, 2024	First release
1.1	November 28, 2024	Update for egETH withdrawal
1.2	February 13, 2025	Add external pauser functionality and VlRewardQueuer contract
1.3	April 9, 2025	Support hemi chain
1.4	April 21, 2025	Support for the EigenLayer v1.3.0 upgrade

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-04-22 08:51:06+0000

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

This audit focuses on the EigenPie Contracts for Magpiexyz [1]. Eigenpie is the first isolated liquid restaking platform for ETH LSTs, leveraging the infrastructure of EigenLayer and allowing native ETH and ETH LST token holders to earn more. Specifically, only the following contracts in the repository are included in the scope of this audit. Other files are not within the scope of this audit.

- contracts/EigenpieConfig.sol
- contracts/EigenpieEnterprise.sol
- contracts/EigenpiePreDepositHelper.sol
- contracts/EigenpieStaking.sol
- contracts/EigenpieWithdrawManager.sol
- contracts/MLRTWallet.sol
- contracts/MLRTWalletZircuit.sol [2]
- contracts/NodeDelegator.sol
- contracts/RewardDistributor.sol
- contracts/crosschain/Eigenpie.sol
- contracts/vlEigenpie.sol
- contracts/rewards/VlStreamRewarder.sol
- contracts/crosschain/MLRTOFT.sol
- contracts/crosschain/MLRTOFTAdapter.sol
- contracts/crosschain/MLRTCCIPBridge.sol
- contracts/crosschain/RemoteMLRT.sol
- contracts/tokens/MLRT.sol [3]
- contracts/oracles/AnkrETHOracleAdapter.sol
- contracts/oracles/CbETHOracleAdapter.sol
- contracts/oracles/ChainlinkAdapter.sol
- contracts/oracles/ConstantOracleAdapter.sol
- contracts/oracles/ETHxOracleAdapter.sol
- contracts/oracles/LsETHOracleAdapter.sol
- contracts/oracles/MethOracleAdapter.sol
- contracts/oracles/OETHOracleAdapter.sol

[1] https://github.com/magpiexyz/eigenpie

[2] This file was added in commit a062c56129ddfbbc852872698be4ca3e4afd1a34.

[3] This file was renamed to MLRTOFTBridge in commit 19cfb1758a6e64aae46d9ee499374a220a2fbab5.

- contracts/oracles/OsETHOracleAdapter.sol
- contracts/oracles/PriceProvider.sol
- contracts/oracles/RemotePriceProvider.sol
- contracts/oracles/SfrxEthOracleAdapter.sol
- contracts/oracles/SwEthOracleAdapter.sol
- contracts/oracles/WbEthOracleAdapter.sol
- contracts/balancer/MstETHRateProvider.sol
- contracts/balancer/MswETHRateProvider.sol
- contracts/libraries/AssetManagementLib.sol
- contracts/libraries/ValidatorLib.sol
- contracts/utils/TransferHelper.sol
- contracts/utils/UtilLib.sol
- contracts/utils/EigenpieConfigRoleChecker.sol
- contracts/utils/EigenpieConstants.sol
- contracts/utils/external/BeaconChainProofs.sol
- contracts/utils/external/Endian.sol
- contracts/utils/external/Merkle.sol
- contracts/AVSRewardDistributor.sol [4]
- contracts/rewards/VlRewardQueuer.sol [5]
- contracts/MLRTWalletHemi.sol
- contracts/MLRTWalletSideChainBaseUpg.sol
- contracts/ETHProcessor.sol

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
EigenPie Contracts	Version 1	ed5a8dd81e3d45df4b1888f7c4c9fecb78faa4a2
	Version 2	a062c56129ddfbbc852872698be4ca3e4afd1a34
	Version 3	19cfb1758a6e64aae46d9ee499374a220a2fbab5
	Version 4	97d737a6bbe70f0f55437bc3f7113570ca177cd4
	Version 5	35c5579e41749afcff07dccb4de42cf48bbf76ac
	Version 6	135db1dae072f6e35d72120c5e46ca5f0ba451c5
	Version 7	11d84a633f58e80f918a33a9416a1366c8d357b7
	Version 8	ddb9db1a63e577d3e18537666c537c3a51292fdc
	Version 9	f46d44041666587a1ca1b8da9987c4a17f922e49
	Version 10	64e9174f2844b554103a5b0b9933fb3c2e9afe3a
	Version 11	ba50e8ee230488f3ffc101aa51fea17996cb3361
	Version 12	92d5b5c1d4033635d0ba25c0a645625c5fc8289c

[4] This file was added in commit d582b5b9efb6c46771c541c50e7f1d875d8ddc49 to support native token (egETH) withdrawal.

[5] This file was added in commit c254a61c3366806f31abb0d0ea06ca74ca0faadd.

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [6] and Common Weakness Enumeration [7]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

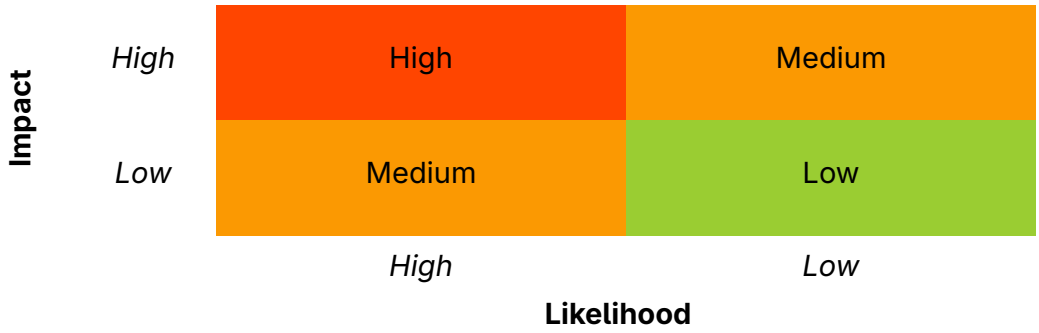
Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.

[6] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[7] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification



- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **twenty-nine** potential security issues. Besides, we have **eleven** recommendations and **three** notes.

- High Risk: 7
- Medium Risk: 16
- Low Risk: 6
- Recommendation: 11
- Note: 3

ID	Severity	Description	Category	Status
1	Medium	Potential unclaimable reward after changing the rewarder contract	DeFi Security	Confirmed
2	High	Incorrect accounting of staked but unverified Ether	DeFi Security	Fixed
3	Medium	Potential removal of unclaimable schedules due to duplicate assets in function `userWithdrawAsset()`	DeFi Security	Fixed
4	Medium	Lack of token approval in function `deposit()`	DeFi Security	Fixed
5	Medium	Potential claim failure due to special logic in token contracts	DeFi Security	Fixed
6	High	Lack of check in function `_getPodShares()`	DeFi Security	Fixed
7	High	Incorrect logic in function `cancelUnlock()`	DeFi Security	Fixed
8	Medium	Potential price provider failure due to negative `podOwnerShares`	DeFi Security	Fixed
9	Medium	Incorrect cross-chain fee refunding	DeFi Security	Fixed
10	Medium	Lack of function `reactivation()` in contract `NodeDelegator`	DeFi Security	Fixed
11	Medium	Inconsistent exchange rate between functions `userQueuingForWithdraw()` and `userWithdrawAsset()`	DeFi Security	Fixed
12	Medium	Potential funds loss or reward rate manipulated	DeFi Security	Confirmed
13	High	Potential inflated rewards from overminting `mLRT` tokens	DeFi Security	Fixed
14	Medium	Unrefunded native tokens in function `depositAsset()`	DeFi Security	Fixed
15	Low	Inconsistent pausing behavior	DeFi Security	Fixed

16	Low	Lack of functions to receive refunded fees	DeFi Security	Fixed
17	Medium	Incorrect return value	DeFi Security	Fixed
18	Medium	Lack of slippage check in function `_debit()`	DeFi Security	Fixed
19	Medium	Incorrect calculation of native token balance	DeFi Security	Fixed
20	High	Potential delayed withdrawal due to incorrect logic	DeFi Security	Fixed
21	High	Potential incorrect accounting for validator slashing	DeFi Security	Confirmed
22	High	Incorrect check of the fund source	DeFi Security	Fixed
23	Low	Lack of function `donateRewards()` in VlRewardQueuer contract	DeFi Security	Fixed
24	Low	Inconsistent user staking state due to untimely sync of user balance	DeFi Security	Confirmed
25	Medium	Lack of cross-chain mLRT burning mechanism	DeFi Security	Confirmed
26	Low	Incorrect check in function `transferExcessETHToStaking()`	DeFi Security	Fixed
27	Low	Unable to update _mLRTWallet due to improper check	DeFi Security	Fixed
28	Medium	Inconsistent withdrawal timing between EigenPie and EigenLayer	DeFi Security	Confirmed
29	Medium	Incompatible slash querying logic in function `getEthBalance()`	DeFi Security	Confirmed
30	-	Add checks on the total weights in reward distribution	Recommendation	Fixed
31	-	Add `view` modifier to function `restakedLess()`	Recommendation	Fixed
32	-	Remove redundant code	Recommendation	Confirmed
33	-	Incorrect logic in function `getFullyUnlock()`	Recommendation	Fixed
34	-	Inconsistent logic in function `addNodeDelegatorContractToQueue()`	Recommendation	Fixed
35	-	Add checks in `advanceCycle()`	Recommendation	Fixed
36	-	Improper check in function `makeBeaconDeposit()`	Recommendation	Fixed
37	-	Remove redundant logic related to deprecated contracts	Recommendation	Fixed
38	-	Fix incorrect parameter for events	Recommendation	Fixed

39	-	Gas optimizations	Recommendation	Fixed
40	-	Redundant code	Recommendation	Fixed
41	-	Potential centralization risks	Note	-
42	-	Lack of gas fee check during cross-chain	Note	-
43	-	Potential inconsistent pausing behavior	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential unclaimable reward after changing the rewarder contract

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The function `getReward()` is responsible for claiming rewards and can only be called by the contract `VlEigenpie`. When the contract `VlEigenpie` updates the rewarder contract, users will no longer be able to claim tokens from the old rewarder contract.

```
394   function setRewarder(address _rewarder) external onlyDefaultAdmin {
395       address oldRewarder = address(rewarder);
396       rewarder = IVLStreamRewarder(_rewarder);
397       emit RewarderUpdated(oldRewarder, _rewarder);
398   }
```

Listing 2.1: contracts/vlEigenpie.sol

Impact Users may potentially lose their rewards if the rewarder contract is changed.

Suggestion Implement a function that allows users to claim rewards from the old rewarder contract after contract `VlEigenpie` sets a new one.

Feedback from the project Acknowledged. In future if such situation arises that we need to change the rewarder, then we will make the old rewarder as legacy rewarder and allow user to claim from legacy as well as current rewarder.

2.1.2 Incorrect accounting of staked but unverified Ether

Severity High

Status Fixed in `Version 4`

Introduced by `Version 1`

Description The contract `NodeDelegator` is responsible for registering validators by making deposits into the Beacon Deposit Contract and verifying withdrawal credentials through the contract `EigenPod`. When a deposit is made for each validator, the function `_makeBeaconDeposit()`

increases the state variable `stakedButNotVerifiedEth` by 32 Ether to represent staked funds that have not yet had their withdrawal credentials verified.

Correspondingly, in the function `verifyWithdrawalCredentials()`, the state variable `stakedButNotVerifiedEth` is decreased to account for validators whose credentials have been verified. However, the function `ValidatorLib.verifyWithdrawCredentials()` returns the effective balance of all validators, which can lead to incorrect accounting if a validator is slashed and its balance falls below 32 Ether before the withdrawal credentials verification process. This mismatch may result in incorrect pricing of the `egETH` token.

```
422   function _makeBeaconDeposit(
423       bytes[] memory publicKeys,
424       bytes[] memory signatures,
425       bytes32[] memory depositDataRoots
426   )
427       internal
428   {
429       ValidatorLib.makeBeaconDeposit(publicKeys, signatures, depositDataRoots, eigenpieConfig,
              address(eigenPod));
430       stakedButNotVerifiedEth += publicKeys.length * EigenpieConstants.DEPOSIT_AMOUNT;
431   }
```

Listing 2.2: contracts/NodeDelegator.sol

```
232   function verifyWithdrawalCredentials(
233       uint64 beaconTimestamp,
234       BeaconChainProofs.StateRootProof calldata stateRootProof,
235       uint40[] calldata validatorIndices,
236       bytes[] calldata validatorFieldsProofs,
237       bytes32[][] calldata validatorFields
238   )
239       external
240       whenNotPaused
241       onlyAllowedBot
242   {
243       uint256 gasBefore = gasleft();
244       stakedButNotVerifiedEth -= ValidatorLib.verifyWithdrawalCredentials(
245           eigenPod, beaconTimestamp, stateRootProof, validatorIndices, validatorFieldsProofs,
                validatorFields
246       );
247       // update the gas spent for RestakeAdmin
248       _recordGas(gasBefore);
249   }
```

Listing 2.3: contracts/NodeDelegator.sol

```
88   function verifyWithdrawalCredentials(
89       IEigenPod eigenPod,
90       uint64 beaconTimestamp,
91       BeaconChainProofs.StateRootProof calldata stateRootProof,
92       uint40[] calldata validatorIndices,
93       bytes[] calldata validatorFieldsProofs,
94       bytes32[][] calldata validatorFields
```

```
 95   )
 96       external
 97       returns (uint256 stakedButNotVerifiedEth)
 98   {
 99       eigenPod.verifyWithdrawalCredentials(
100           beaconTimestamp, stateRootProof, validatorIndices, validatorFieldsProofs,
                validatorFields
101       );
102
103       // Decrement the staked but not verified ETH
104       for (uint256 i = 0; i < validatorFields.length;) {
105           uint64 validatorCurrentBalanceGwei = BeaconChainProofs.getEffectiveBalanceGwei(
                validatorFields[i]);
106           stakedButNotVerifiedEth += (validatorCurrentBalanceGwei * EigenpieConstants.GWEI_TO_WEI
                );
107
108           unchecked {
109               ++i;
110           }
111       }
112   }
```

Listing 2.4: contracts/libraries/ValidatorLib.sol

Impact If validators are slashed before the corresponding withdrawal credentials are verified, the pricing of the `egETH` token could become inaccurate.

Suggestion Ensure that the state variable `stakedButNotVerifiedEth` is decreased by 32 Ether for each validator that has successfully passed the withdrawal credential verification process.

2.1.3 Potential removal of unclaimable schedules due to duplicate assets in function `userWithdrawAsset()`

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 1`

Description The function `userWithdrawAsset()` allows users to withdraw assets that have reached their withdrawal time. When the number of claimed schedules for an asset reaches a threshold (defined by the state variable `withdrawalScheduleCleanUp`), a cleaning process for claimed schedules for that asset is initiated.

However, if a user mistakenly provides duplicate assets as input parameters, the local variable `claimedWithdrawalSchedulesPerAsset` which tracks the number of claimed schedules per asset will be incorrectly incremented. As a result, unclaimed schedules may be prematurely removed by the internal function `_cleanUpWithdrawalSchedules()`, leading to an incorrect state where users cannot access their unclaimed withdrawals.

```
178   function userWithdrawAsset(address[] memory assets) external nonReentrant {
179       uint256[] memory claimedWithdrawalSchedules = new uint256[](assets.length);
180       for (uint256 i = 0; i < assets.length;) {
181           bytes32 userToAsset = userToAssetKey(msg.sender, assets[i]);
```

```
182         UserWithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset];
183
184         uint256 totalClaimedAmount;
185         uint256 claimedWithdrawalSchedulesPerAsset;
186
187         for (uint256 j = 0; j < schedules.length;) {
188             UserWithdrawalSchedule storage schedule = schedules[j];
189
190             // if claimable
191             if (block.timestamp >= schedule.endTime && schedule.claimedAmt == 0) {
192                 claimedWithdrawalSchedulesPerAsset++;
193
194                 schedule.claimedAmt = schedule.queuedWithdrawLSTAmt;
195                 totalClaimedAmount += schedule.queuedWithdrawLSTAmt;
196             } else if (block.timestamp >= schedule.endTime && schedule.claimedAmt == schedule.
                     queuedWithdrawLSTAmt) {
197                 claimedWithdrawalSchedulesPerAsset++;
198             }
199
200             unchecked {
201                 ++j;
202             }
203         }
204
205         claimedWithdrawalSchedules[i] = claimedWithdrawalSchedulesPerAsset;
206
207         if (totalClaimedAmount > 0) {
208             IERC20(assets[i]).safeTransfer(msg.sender, totalClaimedAmount);
209             emit AssetWithdrawn(msg.sender, assets[i], totalClaimedAmount);
210         }
211
212         unchecked {
213             ++i;
214         }
215     }
216
217     _cleanUpWithdrawalSchedules(assets, claimedWithdrawalSchedules);
218 }
```

Listing 2.5: contracts/EigenpieWithdrawManager.sol

```
309 function _cleanUpWithdrawalSchedules(
310     address[] memory assets,
311     uint256[] memory claimedWithdrawalSchedules
312 ) internal {
313     for (uint256 i = 0; i < assets.length;) {
314         bytes32 userToAsset = userToAssetKey(msg.sender, assets[i]);
315         UserWithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset];
316
317         if (claimedWithdrawalSchedules[i] >= withdrawalscheduleCleanUp) {
318             for (uint256 j = 0; j < schedules.length - claimedWithdrawalSchedules[i];) {
319                 schedules[j] = schedules[j + claimedWithdrawalSchedules[i]];
320
```

```
321                 unchecked {
322                     ++j;
323                 }
324             }
325
326             while (claimedWithdrawalSchedules[i] > 0) {
327                 schedules.pop();
328                 claimedWithdrawalSchedules[i]--;
329             }
330         }
331
332         unchecked {
333             ++i;
334         }
335     }
336 }
```

Listing 2.6: contracts/EigenpieWithdrawManager.sol

Impact Users may be unable to claim LSTs due to the unintended removal of unclaimed schedules.

Suggestion Add a check in the function `userWithdrawAsset()` to prevent the input of duplicate assets, ensuring accurate tracking of claimed schedules.

2.1.4 Lack of token approval in function `deposit()`

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the contract `NodeDelegator`, the function `deposit()` is responsible for depositing `SSV` tokens into the contract `SSVNetwork`. The function `transferFrom()` is called to transfer the tokens from the contract `NodeDelegator` to the contract `SSVNetwork`. However, the function `approve()` is not called beforehand to give the contract `SSVNetwork` permission to spend `SSV` tokens from the contract `NodeDelegator`. As a result, the `deposit()` function in the contract `NodeDelegator` cannot function as expected.

This same issue also appears in the contract `WLNodeDelegator` and `MLRTWalletZircuit`, where the function `deposit()` and `bridgeMLRTToEthereum()` faces similar problems due to the missing `approve()` step.

```
368     function deposit(
369         uint64[] memory operatorIds,
370         uint256 amount,
371         ISSVNetworkCore.Cluster memory cluster
372     )
373         external
374         onlyEigenpieManager
375     {
376         address ssvNetwork = eigenpieConfig.getContract(EigenpieConstants.SSVNETWORK_ENTRY);
377         ISSVClusters(ssvNetwork).deposit(address(this), operatorIds, amount, cluster);
```

```
378   }
```

Listing 2.7: contracts/NodeDelegator.sol

```
350   function deposit(
351       uint64[] memory operatorIds,
352       uint256 amount,
353       ISSVNetworkCore.Cluster memory cluster
354   )
355       external
356       onlyClientOrManager
357   {
358       address ssvNetwork = eigenpieConfig.getContract(EigenpieConstants.SSVNETWORK_ENTRY);
359       ISSVClusters(ssvNetwork).deposit(address(this), operatorIds, amount, cluster);
360   }
```

Listing 2.8: contracts/WLNodedelegator.sol

The following code segment shows the corresponding deposit logic from SSV Network [1].

```
1   function deposit(uint256 amount) internal {
2       if (!SSVStorage.load().token.transferFrom(msg.sender, address(this), amount)) {
3           revert ISSVNetworkCore.TokenTransferFailed();
4       }
5   }
```

Listing 2.9: The deposit logic from SSV Network: contracts/libraries/CoreLib.sol

Impact The function `deposit()` in the contract `NodeDelegator` cannot function correctly, leading to a failure in depositing `SSV` tokens into the contract `SSVNetwork`.

Suggestion Update the contract logic to call `approve()` before invoking the function `deposit()` of the contract `SSVNetwork`, ensuring the necessary permissions are granted for the transfer of `SSV` tokens.

2.1.5 Potential claim failure due to special logic in token contracts

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In general, centralized ERC-20 tokens may have `pause` and `whitelist/blacklist` functionalities. However, the function `getReward()` in the contract `VIStreamRewarder` does not allow for separated withdrawal of reward tokens. Therefore, if one of the tokens in the contract is paused, all the users are unable to withdraw the rewards in other tokens.

```
218   function getReward(
219       address _account
220   ) external onlyVlEigenpie returns (bool) {
221       updateFor(_account);
222
```

[1] https://github.com/ssvlabs/ssv-network/blob/main/contracts/libraries/CoreLib.sol

```
223         for (uint256 index = 0; index < rewardTokens.length; ++index) {
224             address rewardToken = rewardTokens[index];
225             _sendReward(_account, rewardToken);
226         }
227         return true;
228     }
```

Listing 2.10: contracts/rewards/VlStreamRewarder.sol

Impact Users may potentially be unable to claim their rewards.

Suggestion Add a function for users that allows them to specify the tokens of reward claims.

2.1.6 Lack of check in function `_getPodShares()`

Severity High

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the function `registerReStaking()`, a registered client can mint `MLRT` based on their staked amount, determined by the current exchange rate. The function calculates the number of shares to mint by checking the staked amount of the client. The internal function `_getPodShares()` returns the `podOwnerShares` of the client, which represents the amount of native tokens staked.

However, there is no check to ensure that the return value of `_getPodShares()` is greater than or equal to zero. If a client has been slashed for malicious behavior, their `podOwnerShares` could be negative (smaller than zero). Casting this negative value from `int256` to `uint256` can result in an extremely large value, falsely reflecting a large number of shares, even though the client does not possess that many. This issue could lead to an incorrect number of `MLRT` tokens being minted.

```
154     function registerReStaking(
155         address underlyingToken,
156         uint256 amountToMintMlt
157     )
158         external
159         nonReentrant
160         onlyAllowedClient
161     {
162         ClientData storage clientData = allowedClients[msg.sender];
163         address receipt;
164         uint256 amountToMint;
165         _updateClientRestakingData(msg.sender, clientData);
166         _checkValidMint(msg.sender, clientData, underlyingToken, amountToMintMlt);
167         (receipt, amountToMint) = _calculateMintAndUpdate(msg.sender, underlyingToken,
                amountToMintMlt);
168
169         if (clientData.mlrtWallet == address(0)) {
170             clientData.mlrtWallet = _deployMLRTWallet(msg.sender, clientData.eigenPod);
171         }
172
```

```
173         IMLRT(receipt).mint(clientData.mlrtWallet, amountToMint);
174         totalMintedMlrt[receipt] += amountToMint;
175
176         emit ClientRegisterRestake(msg.sender, clientData.mlrtWallet, underlyingToken,
                amountToMintMlt, amountToMint);
177     }
```

Listing 2.11: contracts/EigenpieEnterprise.sol

```
352     function _updateClientRestakingData(address client, ClientData storage clientData) internal {
353         (address[] memory underlyingTokens, uint256[] memory underlyingAmounts) =
                getRestakingShares(client);
354
355         uint256 totalStrategies = underlyingTokens.length;
356         for (uint256 i = 0; i < totalStrategies - 1; i++) {
357             clientAssetMapping[client][underlyingTokens[i]].lstRestakedAmount = underlyingAmounts[i
                    ];
358             emit UpdateClientLSTRestakedAmount(client, underlyingTokens[i], underlyingAmounts[i]);
359         }
360
361         clientData.nativeRestakedAmount = underlyingAmounts[totalStrategies - 1];
362         emit UpdateClientNativeRestakedAmount(client, clientData.nativeRestakedAmount);
363     }
```

Listing 2.12: contracts/EigenpieEnterprise.sol

```
109     function getRestakingShares(address client) public view returns (address[] memory, uint256[]
            memory) {
110         uint256 podShares = _getPodShares(client);
111         (address[] memory underlyingTokens, uint256[] memory underlyingAmounts, uint256 assetLength
                ) =
112             _getStrategyShares(client); // The last entry here will be vacant reason being native
                    strategy is not included
113             // in the strategies array
114
115         // Add native strategy (platform token) shares to the array
116         underlyingTokens[assetLength - 1] = EigenpieConstants.PLATFORM_TOKEN_ADDRESS;
117         underlyingAmounts[assetLength - 1] = podShares;
118         return (underlyingTokens, underlyingAmounts);
119     }
```

Listing 2.13: contracts/EigenpieEnterprise.sol

```
395     function _getPodShares(address client) internal view returns (uint256 podShares) {
396         IEigenPodManager eigenPodManager = _getEigenPodManager();
397         return uint256(eigenPodManager.podOwnerShares(client));
398     }
```

Listing 2.14: contracts/EigenpieEnterprise.sol

Impact Clients may be able to mint `MLRT` tokens that do not accurately reflect their actual staked amount.

Suggestion Add a check to ensure that the return value of `_getPodShares()` is greater than or equal to zero before casting `int256` to `uint256`.

2.1.7 Incorrect logic in function `cancelUnlock()`

Severity High

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the function `cancelUnlock()`, the function `updateFor()` is called after the state variable `totalAmountInCoolDown` and `slot.amountInCoolDown` are updated. These two variables indirectly influence the calculation in the function `totalStaked()` in the `VlStreamRewarder` contract, which in turn affects the rewards users can receive. An attacker can exploit this vulnerability to steal all rewards from the `VlStreamRewarder`. The attack steps are as follows:

1. The attacker calls the function `lock()` to obtain rewards, and the lock amount may be a significant portion of the contract`vlEigenpie`.
2. The attacker then calls the function `startUnlock()`. This action leads to a reduction in `totalStaked()`, resulting in an anomalously high `rewardPerToken()` value due to its inverse relationship with `totalStaked()`.
3. After the function `updateFor()` is triggered multiple times for other users, the attacker calls the function `cancelUnlock()`. Since the function `updateFor()` is called after the state variable `totalAmountInCoolDown` is set, when calculating the attacker's rewards, the rewards from the unlock period are also included.
4. As a result, the attacker can extract all rewards.

```
295   function cancelUnlock(
296       uint256 _slotIndex
297   ) external override whenNotPaused nonReentrant {
298       _checkIdexInBoundary(msg.sender, _slotIndex);
299       UserUnlocking storage slot = userUnlockings[msg.sender][_slotIndex];
300
301       _checkInCoolDown(msg.sender, _slotIndex);
302
303       totalAmountInCoolDown -= slot.amountInCoolDown; // reduce amount to cool down accordingly
304       slot.amountInCoolDown = 0; // not in cool down anymore
305
306       if (address(rewarder) != address(0)) rewarder.updateFor(msg.sender);
307
308       emit ReLock(msg.sender, _slotIndex, slot.amountInCoolDown);
309   }
```

Listing 2.15: contracts/vlEigenpie.sol

Impact The attacker can steal all of the rewards intended for other users, leading to significant financial losses.

Suggestion Revise the logic in the function to call `rewarder.updateFor()` before `totalAmountInCoolDown -= slot.amountInCoolDown`. This will ensure that the updates occur in the correct order, preventing exploitation of the reward system.

2.1.8 Potential price provider failure due to negative `podOwnerShares`

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the function `getEthBalance()`, the ETH balance is calculated using the formula `stakedButNotVerifiedEth - uint256(-podOwnerShares)` when `podOwnerShares` is negative. However, once the validator completes the withdrawal credentials verification process, the `stakedButNotVerifiedEth` is set to zero. In this scenario, if the fetched value of `podOwnerShares` is negative, the `getEthBalance()` function will revert due to an underflow.

```
120   function getEthBalance() external view returns (uint256) {
121       // TODO: Once withdrawals are enabled, allow this to handle pending withdraws
122       IEigenPodManager eigenPodManager = AssetManagementLib.getEigenPodManager(eigenpieConfig);
123       return AssetManagementLib.getEthBalance(eigenPodManager, stakedButNotVerifiedEth, address(
              this));
124   }
```

Listing 2.16: contracts/NodeDelegator.sol

```
73    function getEthBalance(
74        IEigenPodManager eigenPodManager,
75        uint256 stakedButNotVerifiedEth,
76        address nodeDelegator
77    )
78        public
79        view
80        returns (uint256)
81    {
82        int256 podOwnerShares = eigenPodManager.podOwnerShares(nodeDelegator);
83        return podOwnerShares < 0
84            ? stakedButNotVerifiedEth - uint256(-podOwnerShares)
85            : stakedButNotVerifiedEth + uint256(podOwnerShares);
86    }
```

Listing 2.17: contracts/libraries/AssetManagementLib.sol

Impact The contract `PriceProvider` will be unable to update `exchangeRate`.

Suggestion Revise the logic in the `getEthBalance()` function to ensure it handles negative `podOwnerShares` correctly.

2.1.9 Incorrect cross-chain fee refunding

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 1`

Description When using CCIP for cross-chain operations, the user's input for `msg.value` is checked. If the `msg.value` exceeds the required fee, it should be refunded. However, due to an implementation error in the condition check, users are unable to receive the excess cross-chain fees.

```
145     if (0 > msg.value - fee) {
146         // Calculate excess funds
147         uint256 excessFunds = msg.value - fee;
148         // Refund excess funds to the sender
149         payable(msg.sender).transfer(excessFunds);
150     }
```

Listing 2.18: contracts/crosschain/MLRTCCIPBridge.sol

Impact Users' funds will not be refunded.

Suggestion Change the condition from `0 > msg.value - fee` to `msg.value - fee > 0` to ensure that excess funds are properly calculated and refunded to the user.

2.1.10 Lack of function `reactivation()` in contract `NodeDelegator`

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the SSV Network, if the SSV tokens are insufficient for a cluster, the cluster can be liquidated by others. After liquidation, the `reactivation()` function should be called to reset the cluster to an active state. However, the current protocol does not provide this functionality, causing the cluster to remain in an inactive state, which prevents it from earning rewards from SSV. The same issue exists in the contract `WLNodeDelegator`.

Impact After liquidation, the protocol will be unable to derive any benefits from SSV, and the protocol would not be able to reactivate the clusters.

Suggestion Add the relevant calls to the function `reactivation()` in the contracts `NodeDelegator` and `WLNodeDelegator` to ensure that clusters can be reset to an active state after liquidation.

2.1.11 Inconsistent exchange rate between functions `userQueuingForWithdraw()` and `userWithdrawAsset()`

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 1`

Description The user's withdrawal process is divided into two steps. First, the user calls the function `userQueuingForWithdraw()` to initiate a withdrawal request. After a waiting period of at least 7 days, the user calls the function `EigenpieWithdrawManager.userWithdrawAsset()` to complete the withdrawal. If the `stEth` balance decreases due to Lido slash, it could result in an insufficient amount of `stETH` tokens in the contract. Ultimately, this may prevent the last user from being able to withdraw their funds.

```
208   if (totalClaimedAmount > 0) {
209       IERC20(assets[i]).safeTransfer(msg.sender, totalClaimedAmount);
210       emit AssetWithdrawn(msg.sender, assets[i], totalClaimedAmount);
211   }
```

Listing 2.19: contracts/EigenpieWithdrawManager.sol

Impact The user may potentially be unable to withdraw their funds.

Suggestion When processing withdrawals, compare `totalClaimAmount` with `asset.balanceOf()` and withdraw the smaller of the two amounts to ensure sufficient balance for user withdrawals.

2.1.12 Potential funds loss or reward rate manipulated

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description When the `rewardRate` decreases and there is only one user locked in `vlEigenpie`, the user can preemptively call `startUnlock()` to prevent the `rewardRate` from being updated, allowing the user to continue receiving rewards at a higher `rewardRate`.

Consider the following scenario:

1. Currently, there is only one user in `vlEigenpie`, and that user has locked only 1 wei. The owner has donated 1000 USD, intending to distribute the rewards over 1000 seconds.
2. After 300 seconds, 300 USD in rewards have been distributed. At this point, the owner wants to donate an additional 100 USD as rewards. Considering the duration (i.e., 1000 seconds) is unchanged while the `queuedReward` is 800 USD now, this action would reduce the `rewardRate`. However, the user called `startUnlock()` before this operation, preventing the `rewardRate` from changing. After the donation, the user immediately calls `cancelUnlock()`.
3. After another 700 seconds, the user receives 1000 USD. However, under normal circumstances, the user should have earned USD rewards amounting to 300 + (100 + 700)/1000 * 700 = 860 USD in these 1000 seconds. This means the user received 140 USD more than they would have normally.

Additionally, when there is only one user staked, there is a potential issue where funds can be locked. After calling `donateRewards()`, the user unlocks the amount they had locked, this will result in the `rewardRate` being non-zero while `VlStreamRewarder.totalStaked()` returns 0. In this case, rewards will begin to be distributed over time, but no one will be able to claim them, ultimately causing some rewards to get stuck in the contract.

```
264   function _provisionReward(uint256 _rewards, address _rewardToken) internal {
265       _rewards = _rewards * DENOMINATOR; // to support small deciaml rewards
266
267       Reward storage rewardInfo = rewards[_rewardToken];
268
269       if (totalStaked() == 0) {
270           rewardInfo.queuedRewards = rewardInfo.queuedRewards + _rewards;
271           return;
272       }
273
274       rewardInfo.rewardPerTokenStored = rewardPerToken(_rewardToken);
275       _rewards = _rewards + rewardInfo.queuedRewards;
```

```
276         rewardInfo.queuedRewards = 0;
277
278         if (block.timestamp >= rewardInfo.periodFinish) {
279             rewardInfo.rewardRate = _rewards / duration;
280         } else {
281             uint256 remaining = rewardInfo.periodFinish - block.timestamp;
282             uint256 leftover = remaining * rewardInfo.rewardRate;
283             _rewards = _rewards + leftover;
284             rewardInfo.rewardRate = _rewards / duration;
285         }
286
287         rewardInfo.lastUpdateTime = block.timestamp;
288         rewardInfo.periodFinish = block.timestamp + duration;
289     }
```

Listing 2.20: contracts/rewards/VlStreamRewarder.sol

Impact Users can manipulate the update of `rewardRate`, leading to an increase in rewards over a period of time. Some reward funds may get stuck in the contract.

Suggestion The `rewardRate` should also be updated when `totalStaked()` is 0. Additionally, a function should be added to either withdraw the funds stuck in the contract or reallocate them back into the rewards for users.

Feedback from the project The code can be quite complicated if we want to handle this only one user in lock who is also being malicious. We will make sure there is like 10 vlEGP locked by the team and will never start unlock or unlock.

2.1.13 Potential inflated rewards from overminting `mLRT` tokens

Severity High

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the contract `EigenpieEnterprise`, whitelisted clients can perform restaking directly on Eigenlayer to obtain shares, which are subsequently used to mint corresponding `mLRT` tokens via the function `registerReStaking()`. These shares are directly recorded in the relevant Eigenlayer contract rather than being issued as tokens. When `EigenpieEnterprise` mints `mLRT` tokens, it reads the shares corresponding to each `LST` token strategy that the client holds in Eigenlayer and mints the corresponding `mLRT` tokens to the client's `MLRTWallet`.

However, the function `registerReStaking()` only checks the restaking shares for one type of `LST` token when determining whether the minting amount exceeds the shares the user possesses. This allows users to mint, withdraw their `LST` tokens, exchange them for another type of `LST` token, and then continue to restake and mint additional `mLRT` tokens. By repeating this process, clients can obtain `mLRT` tokens worth far more than the value of their `LST` holdings, and deposit them in `Zicruit` or `Swell` to earn staking rewards.

Although the protocol allows anyone to withdraw and burn `mLRT` tokens that do not belong to the client by calling the corresponding withdraw function, the rewards during this period are still collected with no loss.

```
154   function registerReStaking(
155       address underlyingToken,
156       uint256 amountToMintMlt
157   )
158       external
159       nonReentrant
160       onlyAllowedClient
161   {
162       ClientData storage clientData = allowedClients[msg.sender];
163       address receipt;
164       uint256 amountToMint;
165       _updateClientRestakingData(msg.sender, clientData);
166       _checkValidMint(msg.sender, clientData, underlyingToken, amountToMintMlt);
167       (receipt, amountToMint) = _calculateMintAndUpdate(msg.sender, underlyingToken,
              amountToMintMlt);
168
169       if (clientData.mlrtWallet == address(0)) {
170           clientData.mlrtWallet = _deployMLRTWallet(msg.sender, clientData.eigenPod);
171       }
172
173       IMLRT(receipt).mint(clientData.mlrtWallet, amountToMint);
174       totalMintedMlrt[receipt] += amountToMint;
175
176       emit ClientRegisterRestake(msg.sender, clientData.mlrtWallet, underlyingToken,
              amountToMintMlt, amountToMint);
177   }
```

Listing 2.21: contracts/EigenpieEnterprise.sol

```
214   function _checkValidMint(
215       address client,
216       ClientData storage clientData,
217       address underlyingToken,
218       uint256 amountToMintMlt
219   )
220       internal
221       view
222   {
223       uint256 quotaLeft;
224       if (underlyingToken != EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
225           LSTData memory lstData = clientAssetMapping[client][underlyingToken];
226           quotaLeft = lstData.lstRestakedAmount - lstData.lstUsed;
227       } else {
228           quotaLeft = clientData.nativeRestakedAmount - clientData.nativeUsed;
229       }
230       if (quotaLeft < amountToMintMlt) {
231           revert AssetNotEnough(quotaLeft, amountToMintMlt);
232       }
233   }
```

Listing 2.22: contracts/EigenpieEnterprise.sol

Impact The client can inflate rewards by minting multiple times of `mLRT` tokens.

Suggestion Revise the validation logic to ensure the sum of all shares the user holds in Eigenlayer does not exceed the number of `mLRT` tokens that can be minted before minting.

2.1.14 Unrefunded native tokens in function `depositAsset()`

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 1`

Description When a user invokes the function `depositAsset()` to stake assets, the asset address needs to be passed as a parameter. However, if the user directly sends native tokens during the invocation while also providing the address of another type of `LST`, the function will transfer the corresponding amount of `LST` tokens from the user's account without returning the native tokens, which is incorrect.

```
143   function depositAsset(
144       address asset,
145       uint256 depositAmount,
146       uint256 minRec,
147       address referral
148   )
149       external
150       payable
151       whenNotPaused
152       nonReentrant
153       onlySupportedAsset(asset)
154   {
155       // checks
156       bool isNative = UtilLib.isNativeToken(asset);
157       if (isNative && msg.value != depositAmount) {
158           revert InvalidAmountToDeposit();
159       }
160
161       if (depositAmount == 0 || depositAmount < minAmountToDeposit) {
162           revert InvalidAmountToDeposit();
163       }
164
165       if (depositAmount > getAssetCurrentLimit(asset)) {
166           revert MaximumDepositLimitReached();
167       }
168
169       uint256 mintedAmount;
170
171       if (isPreDeposit && !isNative) {
172           // only when not native and in pre deposit phase, we don't min receipt token to users
173           address eigenpiePreDepositHelper = eigenpieConfig.getContract(EigenpieConstants.
                  EIGENPIE_PREDEPOSITHELPER);
174           mintedAmount = _mintMLRT(address(eigenpiePreDepositHelper), asset, depositAmount);
175           IEigenpiePreDepositHelper(eigenpiePreDepositHelper).feedUserDeposit(msg.sender, asset,
                  mintedAmount);
176       } else {
177           // mint receipt
```

```
178             mintedAmount = _mintMLRT(msg.sender, asset, depositAmount);
179         }
180
181         if (mintedAmount < minRec) {
182             revert MinimumAmountToReceiveNotMet();
183         }
184
185         if (!isNative) {
186             IERC20(asset).safeTransferFrom(msg.sender, address(this), depositAmount);
187         }
188
189         emit AssetDeposit(msg.sender, asset, depositAmount, referral, mintedAmount, isPreDeposit);
190     }
```

Listing 2.23: contracts/EigenpieStaking.sol

Impact The native tokens of the users will not be refunded if they provided the wrong parameters.

Suggestion Add a check to ensure that the user has provided the correct asset address when invoking the function `depositAsset()`.

2.1.15 Inconsistent pausing behavior

Severity Low

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the contract `RewardDistributor`, the function `forwardRewards()` is decorated with the modifier `whenNotPaused`, preventing it from being called when the contract is paused. Conversely, the function `receive()` lacks this `whenNotPaused` restriction, allowing it to be invoked even when the contract is paused. It creates an inconsistency in the behavior of the contract `RewardDistributor` during pausing.

```
41     receive() external payable nonReentrant {
42         _forwardETH();
43     }
```

Listing 2.24: contracts/RewardDistributor.sol

```
46     function forwardRewards() external payable nonReentrant whenNotPaused onlyEigenpieManager {
47         _forwardETH();
48     }
```

Listing 2.25: contracts/RewardDistributor.sol

Impact The function `_forwardETH()` can still be invoked while the contract is in the paused state.

Suggestion In the paused state, the function `receive()` should not be able to invoke the function `_forwardETH()`.

2.1.16 Lack of functions to receive refunded fees

Severity Low

Status Fixed in `Version 4`

Introduced by `Version 2`

Description In the contract `MLRTWallet`, there are no functions to receive native token transfer, i.e., the functions `receive()` or `fallback()`. However, it is possible for LayerZero to refund native tokens if the cross chain requests fail. Per the documentation [2] of the LayerZero, it is required to implement a fallback or receive function to receive potential refunds from LayerZero.

```
162   function bridgeMLRTToZircuit(
163       address _mlrt,
164       uint256 _amount
165   ) external payable whenNotPaused onlyClientOrAllowedOperator nonReentrant {
166       IMLRTAdapter mlrtAdapter = IMLRTAdapter(
167           eigenpieConfig.getContract(EigenpieConstants.MLRT_ADAPTER)
168       );
169
170       MessagingFee memory fee = mlrtAdapter.getEstimateGasFees(
171           EigenpieConstants.LZ_ZIRCUIT_DESTINATION_ID,
172           0,
173           _amount,
174           _amount,
175           mlrtWalletZircuit
176       );
177
178       // approve to lock MLRT in adapter and mint on destination chain
179       IERC20(_mlrt).safeApprove(address(mlrtAdapter), _amount);
180       mlrtAdapter.bridgeMLRT{value: fee.nativeFee}(
181           EigenpieConstants.LZ_ZIRCUIT_DESTINATION_ID,
182           0,
183           _amount,
184           _amount,
185           mlrtWalletZircuit
186       );
187       emit BridgeMLRTToZircuit(client, msg.sender, _mlrt, _amount);
188   }
```

Listing 2.26: contracts/MLRTWallet.sol

```
48    function bridgeMLRT(
49        uint32 _dstEid,
50        uint128 _dstGasCost,
51        uint256 _amountLD,
52        uint256 _minAmountLD,
53        address _receiver
54    ) external payable nonReentrant{
55        (uint256 amountSentLD, uint256 amountReceivedLD) = _debit(
56            msg.sender,
57            _amountLD,
```

[2] https://docs.layerzero.network/v2/developers/evm/oapp/overview

```
58              _minAmountLD,
59              _dstEid
60          );
61
62          (
63              bytes memory message,
64              bytes memory options
65          ) = _buildCustomMsgAndOptions(
66                  _dstEid,
67                  amountReceivedLD,
68                  _getDestinationGasCost(_dstGasCost),
69                  _receiver
70              );
71
72          MessagingReceipt memory msgReceipt = _lzSend(
73              _dstEid,
74              message,
75              options,
76              MessagingFee(msg.value, 0),
77              payable(msg.sender)
78          );
79
80          emit BridgeMLRT(
81              msg.sender,
82              _dstEid,
83              amountSentLD,
84              amountReceivedLD,
85              msgReceipt.guid
86          );
87      }
```

Listing 2.27: contracts/MLRTOFTAdapter.sol

Impact The refunded fee may not be received.

Suggestion Add the function to receive native tokens (i.e., the `fallback()` or `receive()` function) to receive potential refunded fees.

2.1.17 Incorrect return value

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 3`

Description As commented in the official OFT implementation from LayerZero, the return value of the function `approvalRequired` indicates whether the OFT contract requires approval of the `token()` to send. Therefore, the function `approvalRequired` of the contract `MLRTOFTBridge` should return `true` because remote MLRT tokens must be approved to be burnt due to the usage of the `burnFrom` function.

```
43  /// @notice This function was added to remove the error of missing implementations
44  function approvalRequired() external pure virtual returns (bool) {
```

```
45        return false; // dummy implementation
46    }
```

Listing 2.28: contracts/MLRTOFTBridge.sol

Impact When integrate with other contracts, the `MLRTOFTBridge` may not function properly.

Suggestion Return `true` in the `approvalRequired` function.

2.1.18 Lack of slippage check in function `_debit()`

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 3`

Description In the contract `MLRTOFTBridge`, the `_debit()` function fails to incorporate a slippage check. This omission could lead to scenarios where the actual amount of tokens bridged (i.e.,`_amountLD`) minus the dust tokens during the bridge process, does not exceed the threshold (i.e., `_minAmountLD`) specified in the function parameters.

```
183   function _debit(
184       address _from,
185       uint256 _amountLD,
186       uint256 /*_minAmountLD*/,
187       uint32 /*_dstEid*/
188   ) internal override whenNotPaused returns (uint, uint) {
189       UtilLib.checkNonZeroAddress(_from);
190       if (msg.sender != _from) revert InvalidSender();
191       remoteMLRT.burnFrom(_from, _amountLD);
192       return (_amountLD, _amountLD);
193   }
```

Listing 2.29: contracts/crosschain/MLRTOFTBridge.sol

Impact The slippage is not checked.

Suggestion Invoke `_debitView()` function to ensure the slippage check.

2.1.19 Incorrect calculation of native token balance

Severity Medium

Status Fixed in `Version 6`

Introduced by `Version 5`

Description In the function `getEthBalance()`, to achieve correct accounting for the native token withdrawal process, an additional `queuedETHShares` is used and participated in the calculation. However, this modified calculation is not correct. Specifically, in the case where `podOwnerShares` is negative, when `stakedButNotVerifiedEth < abs(podOwnerShares)` but `stakedButNotVerifiedEth + queuedETHShares > abs(podOwnerShares)`, the function returns incorrect result 0.

```
75  function getEthBalance(
76      IEigenPodManager eigenPodManager,
77      uint256 stakedButNotVerifiedEth,
78      address nodeDelegator,
79      uint256 queuedETHShares
80  )
81      public
82      view
83      returns (uint256)
84  {
85      int256 podOwnerShares = eigenPodManager.podOwnerShares(nodeDelegator);
86      if (podOwnerShares < 0) {
87      // Ensure no underflow when stakedButNotVerifiedEth is 0 and podOwnerShares is negative
88      uint256 absPodOwnerShares = uint256(-podOwnerShares);
89      return stakedButNotVerifiedEth >= absPodOwnerShares
90          ? queuedETHShares + stakedButNotVerifiedEth - absPodOwnerShares
91          : 0;
92      } else {
93          return queuedETHShares + stakedButNotVerifiedEth + uint256(podOwnerShares);
94      }
95  }
```

Listing 2.30: contracts/libraries/AssetManagementLib.sol

Impact The function `getEthBalance()` may return an incorrect value under certain circumstances.

Suggestion Refactor the native token balance calculation logic.

2.1.20 Potential delayed withdrawal due to incorrect logic

Severity High

Status Fixed in `Version 6`

Introduced by `Version 5`

Description The contract `EigenpieWithdrawManager` is updated to accommodate native token withdrawal. Specifically, when a user initiates native token withdrawal, if the current balance of the contract is sufficient, the withdrawal is immediately fulfilled and enters the cooldown period. In contrast, if there is insufficient balance, the withdrawal would require extra delay to wait for sufficient native tokens. In this case, a nonce is generated to queue this withdrawal into the `withdrawQueued` state variable with the corresponding request value.

In the function `userWithdrawAsset()`, it first iterates over all the user withdrawal schedules for an asset, and accumulates the total amount of underlying and MLRT tokens. After all the withdrawal schedules are processed, the function then checks for the validity of the withdrawal for the ETH withdrawal, starting from Line 285. However, **the nonce of the latest request is used to query for the withdrawQueued entry to validate the delay process for this request.**

Therefore, there is a circumstance where a user sends two withdrawal requests (request #1 and request #2). If the withdrawal request #1 is not delayed but the #2 is delayed, it would

result in both of them having to be delayed for an extra amount of time, due to the incorrect nonce used for this process.

```
237   function userWithdrawAsset(address[] memory assets) external whenNotPaused nonReentrant {
238       uint256[] memory claimedWithdrawalSchedules = new uint256[](assets.length);
239
240       // check if there are no duplicate entries in input data
241       for (uint256 i = 0; i < assets.length; i++) {
242           for (uint256 j = i + 1; j < assets.length; j++) {
243               if (assets[i] == assets[j]) {
244                   revert InvalidInput();
245               }
246           }
247       }
248
249       for (uint256 i = 0; i < assets.length;) {
250           bytes32 userToAsset = userToAssetKey(msg.sender, assets[i]);
251           UserWithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset];
252
253           uint256 totalClaimedAmount;
254           uint256 totalEgETHBurnAmount;
255           uint256 claimedWithdrawalSchedulesPerAsset;
256           uint256 nonce;
257
258           for (uint256 j = 0; j < schedules.length;) {
259               UserWithdrawalSchedule storage schedule = schedules[j];
260
261               // if claimmable
262               if (block.timestamp >= schedule.endTime && schedule.claimedAmt == 0) {
263                   claimedWithdrawalSchedulesPerAsset++;
264
265                   schedule.claimedAmt = schedule.queuedWithdrawLSTAmt;
266                   totalClaimedAmount += schedule.queuedWithdrawLSTAmt;
267                   totalEgETHBurnAmount += schedule.receiptMLRTAmt;
268                   nonce = schedule.nonce;
269               } else if (block.timestamp >= schedule.endTime && schedule.claimedAmt == schedule.
                       queuedWithdrawLSTAmt)
270               {
271                   claimedWithdrawalSchedulesPerAsset++;
272               }
273
274               unchecked {
275                   ++j;
276               }
277           }
278
279           claimedWithdrawalSchedules[i] = claimedWithdrawalSchedulesPerAsset;
280           if(assets[i]!= EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
281               if(totalClaimedAmount > IERC20(assets[i]).balanceOf(address(this))) {
282               totalClaimedAmount = IERC20(assets[i]).balanceOf(address(this));
283               }
284           }
285           if (totalClaimedAmount > 0) {
```

```
286             if (assets[i] == EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
287                 bytes32 _withdrawHash = keccak256(abi.encode(nonce, msg.sender));
288                 // Revert if withdrawal is queued and not filled completely
289                 if (
290                     withdrawQueued[_withdrawHash].queued
291                         && withdrawQueued[_withdrawHash].fillAt > ethWithdrawQueue.
                             queuedWithdrawFilled
292                 ) revert QueuedWithdrawalNotFilled();
293
294                 // reduce initial amountToRedeem from claim reserve
295                 ethClaimReserve -= totalClaimedAmount;
296
297                 // burn egETH locked for withdraw request
298                 address receipt = eigenpieConfig.mLRTReceiptByAsset(assets[i]);
299                 IMintableERC20(receipt).burnFrom(address(this), totalEgETHBurnAmount);
300                 TransferHelper.safeTransferETH(msg.sender, totalClaimedAmount);
301             } else {
302                 IERC20(assets[i]).safeTransfer(msg.sender, totalClaimedAmount);
303                 emit AssetWithdrawn(msg.sender, assets[i], totalClaimedAmount);
304             }
305         }
306         unchecked {
307             ++i;
308         }
309     }
310
311     _cleanUpWithdrawalSchedules(assets, claimedWithdrawalSchedules);
312 }
```

Listing 2.31: contracts/EigenpieWithdrawManager.sol

Impact The native token withdrawal process is potentially incorrectly accounted, resulting in some of withdrawal requests being incorrectly delayed or advanced.

Suggestion Revise the native token withdrawal logic.

2.1.21 Potential incorrect accounting for validator slashing

Severity High

Status Confirmed

Introduced by `Version 5`

Description In the updated version of the contract `NodeDelegator`, the contract takes any native token transfer which is times of 32 ETH as the returned funds for validator exits, and the remainder is accounted for as rewards. However, this process does not take validator slashing into account, where the effective balance of a validator may be less than 32 ETH, so the returned funds for validator exits may be less than 32 ETH.

```
444     function _checkAndFillETHWithdrawBuffer(uint256 _amount) internal {
445         address eigenpieWithdrawManager = eigenpieConfig.getContract(EigenpieConstants.
                EIGENPIE_WITHDRAW_MANAGER);
446         // Check the withdraw buffer and fill if below buffer target
```

```
447         uint256 bufferToFill = IEigenpieWithdrawManager(eigenpieWithdrawManager).getWithdrawDeficit
                ();
448         uint256 totalETHBal = currEthBalance + _amount;
449
450         if (bufferToFill > 0) {
451             bufferToFill = (totalETHBal <= bufferToFill) ? totalETHBal : bufferToFill;
452             // fill withdraw buffer from received ETH
453             IEigenpieWithdrawManager(eigenpieWithdrawManager).fillEthWithdrawBuffer{ value:
                    bufferToFill }();
454             emit BufferFilled(bufferToFill);
455         }
456         currEthBalance = (totalETHBal <= bufferToFill) ? 0 : totalETHBal - bufferToFill;
457     }
458
459     function _processETH() internal {
460         address eigenStaking = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_STAKING);
461         address delegationManagerAddr = eigenpieConfig.getContract(EigenpieConstants.
                EIGEN_DELEGATION_MANAGER);
462         // If Eth from Eigenstaking or delegationManagerAddr, then should stay waiting;
463         if (msg.sender == eigenStaking || msg.sender != delegationManagerAddr) {
464             return;
465         }
466
467         uint256 gasRefunded = _refundGas();
468         (uint256 ethShares, uint256 rewards) = _calRewardAmt(msg.value);
469
470         // Forward remaining balance to rewarDistributor.
471         // Any random eth transfer to this contract will also be treated as reward.
472         address rewarDistributor = eigenpieConfig.getContract(EigenpieConstants.
                EIGENPIE_REWADR_DISTRIBUTOR);
473         TransferHelper.safeTransferETH(rewarDistributor, rewards - gasRefunded);
474
475         _checkAndFillETHWithdrawBuffer(ethShares);
476
477         emit RewardsForwarded(rewarDistributor, msg.value);
478     }
479
480     function _calRewardAmt(uint256 _recievedAmt) internal pure returns (uint256 ethShares, uint256
            sendRewards) {
481         sendRewards = _recievedAmt % 32 ether;
482         ethShares = _recievedAmt - sendRewards;
483     }
```

Listing 2.32: contracts/NodeDelegator.sol

Impact Incorrect accounting for validator exits may result in some returned funds for validator exits being accounted for as rewards.

Suggestion Refactor the accounting for the validator exits.

Feedback from the project We will set up alerts for the events of validator slashing. If the reward falls between 2 ETH and less than 32 ETH, it signals that 32 ETH has been slashed. In such cases, we will manually transfer ETH from the contract `EigenPieStaking` and deposit

it into the contract NodeDelegator. From there, we can call an admin function that triggers _checkAndFillETHWithdrawBuffer to transfer the amount back to the contract EigenPieWithdrawManager.

2.1.22 Incorrect check of the fund source

Severity High

Status Fixed in Version 6

Introduced by Version 5

Description In the contract NodeDelegator, it contains an incorrect check for the source of the fund in the function _processETH(). Specifically, no native tokens would be transferred from the contract DelegationManager from EigenLayer, and the native token transfers from the corresponding contract EigenPod from the EigenLayer is not correctly accounted as rewards or funds from validator exits. As a result, the native token processing logic is broken for the contract NodeDelegator.

```
459   function _processETH() internal {
460       address eigenStaking = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_STAKING);
461       address delegationManagerAddr = eigenpieConfig.getContract(EigenpieConstants.
              EIGEN_DELEGATION_MANAGER);
462       // If Eth from Eigenstaking or delegationManagerAddr, then should stay waiting;
463       if (msg.sender == eigenStaking || msg.sender != delegationManagerAddr) {
464           return;
465       }
466
467       uint256 gasRefunded = _refundGas();
468       (uint256 ethShares, uint256 rewards) = _calRewardAmt(msg.value);
469
470       // Forward remaining balance to rewarDistributor.
471       // Any random eth transfer to this contract will also be treated as reward.
472       address rewarDistributor = eigenpieConfig.getContract(EigenpieConstants.
              EIGENPIE_REWADR_DISTRIBUTOR);
473       TransferHelper.safeTransferETH(rewarDistributor, rewards - gasRefunded);
474
475       _checkAndFillETHWithdrawBuffer(ethShares);
476
477       emit RewardsForwarded(rewarDistributor, msg.value);
478   }
```

Listing 2.33: contracts/NodeDelegator.sol

Impact The native token processing logic is broken for the contract NodeDelegator.

Suggestion Modify the condition of the checks on the msg.sender.

2.1.23 Lack of function donateRewards() in VlRewardQueuer contract

Severity Low

Status Fixed in Version 8

Introduced by Version 7

Description In the `VlStreamRewarder` contract, users with `RewardQueuer` permissions can transfer rewards to the contract using the functions `donateRewards()` and `queueNewRewards()`. However, the `VlRewardQueuer` contract, which has the required permissions, only implements the function `queueNewRewards()` and does not include the function `donateRewards()`. As a result, the `donateRewards()` function remains inaccessible, preventing intended reward transfers.

```
201    function donateRewards(address _rewardToken, uint256 _rewards) external nonReentrant
           onlyRewardQueuer {
202        if (!isRewardToken[_rewardToken]) revert InvalidToken();
203
204        IERC20(_rewardToken).safeTransferFrom(msg.sender, address(this), _rewards);
205        _provisionReward(_rewards, _rewardToken);
206        emit RewardQueued(_rewardToken, _rewards);
207    }
```

Listing 2.34: VlStreamRewarder.sol

Impact The function `donateRewards()` is unavailable, which deviates from the expected design.

Suggestion Declare and implement the `donateRewards()` function in the `VlRewardQueuer` contract.

2.1.24 Inconsistent user staking state due to untimely sync of user balance

Severity Low

Status Confirmed

Introduced by `Version 9`

Description Whitelist clients obtain `mLRT` tokens by depositing `LST`/`ETH` tokens into `EigenLayer` via function `registerReStaking()`, which mints `mLRT` based on the deposited amount. When `LST`/`ETH` tokens are withdrawn, the protocol should burn the corresponding `mLRT` using function `burnMLRT()`. However, since function `burnMLRT()` must be triggered manually, failure to do so in time can result in users retaining `mLRT` even after withdrawing their `LST`/`ETH` tokens from `EigenLayer`, leading to an inconsistent staking state between `EigenLayer` and `EigenPie`.

```
123    function burnMLRT(address client, address mlrtAsset, uint256 amountToBurn) public nonReentrant
           whenNotPaused {
124        _burnMLRTInternal(client, mlrtAsset, amountToBurn);
125    }
```

Listing 2.35: EigenpieEnterprise.sol

```
240    function _burnMLRTInternal(address client, address mlrtAsset, uint256 amountToBurn) internal {
241        address asset = IMLRT(mlrtAsset).underlyingAsset();
242        if (mlrtAsset != eigenpieConfig.mLRTReceiptByAsset(asset)) revert InvalidMLRTAsset();
243
244
245        ClientData storage clientData = allowedClients[client];
246        if (!clientData.registered) revert InvalidClient();
247
```

```
248
249         if (msg.sender != clientData.mlrtWallet) {
250             // Update the client restaking data only if the caller is not mLRTWallet. This prevents
                    a double update, as
251             // mLRTWallet already updates the data before calling this function
252             _updateClientRestakingData(client, clientData);
253         }
254
255
256         (uint256 valuedAssetLess, uint256 shouldBurn) = _checkCollateralLess(client, asset);
257
258
259         if (valuedAssetLess == 0) revert EnoughCollateral();
260         if (amountToBurn > shouldBurn) revert BurnTooMuch();
261
262
263         valuedAssetLess = valuedAssetLess * amountToBurn / shouldBurn;
264
265
266         _burnFromWallet(client, asset, valuedAssetLess, amountToBurn);
267
268
269         emit BurnMLRTFromWallet(client, asset, valuedAssetLess, amountToBurn);
270     }
```

Listing 2.36: EigenpieEnterprise.sol

Impact The whitelist clients may hold `mLRT` despite withdrawing assets from `EigenLayer`, potentially causing discrepancies in the protocol's accounting and staking integrity.

Suggestion Enforce timely execution of function `burnMLRT()` to maintain staking state consistency.

Feedback from the project Whenever a whitelisted client withdraws assets from `EigenLayer` but does not burn the excess tokens, the team manually burns them.

2.1.25 Lack of cross-chain mLRT burning mechanism

Severity Medium

Status Confirmed

Introduced by `Version 9`

Description The protocol enables whitelist clients to transfer `mLRT` tokens from their `MLRTWallet` on `Ethereum` to corresponding `MLRTWallet` contracts on `Hemi` and `Zircuit` chains. These `mLRT` tokens are minted based on the client's `LST`/`ETH` deposits in `EigenLayer`. When a client withdraws `LST`/`ETH` tokens from `EigenLayer`, the protocol should burn the corresponding `mLRT` to maintain consistency. However, the `MLRTWallet` contracts on `Hemi` and `Zircuit` do not implement this burn mechanism. As a result, once `mLRT` is bridged, the protocol loses the ability to burn the corresponding `mLRT` when the client retrieves `LST`/`ETH` tokens from `EigenLayer`, which is incorrect.

Impact Whitelist clients can withdraw `LST`/`ETH` tokens while retaining `mLRT` on other chains.

Suggestion Implement a mechanism on Hemi and Zircuit MLRTWallet contracts to burn mLRT when LST/ETH tokens are withdrawn from EigenLayer.

Feedback from the project The team will bridge the excess mLRT from Hemi to Ethereum, then burn the excess mLRT on Ethereum.

2.1.26 Incorrect check in function transferExcessETHToStaking()

Severity Low

Status Fixed in Version 10

Introduced by Version 9

Description In the EigenpieWithdrawManager contract, the privileged function transferExcessETHToStaking() transfers excess ETH by first invoking the function getAvailableToWithdraw() to determine the unallocated ETH balance. If the returned value is greater than _amount, the function transfers _amount to eigenpieStaking. However, the condition does not account for cases where _amount is exactly equal to the available balance, preventing the full withdrawal of excess ETH.

```
483   function transferExcessETHToStaking(uint256 _amount) external onlyDefaultAdmin {
484       address eigenpieStaking = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_STAKING);
485       if(getAvailableToWithdraw() > _amount){
486           TransferHelper.safeTransferETH(eigenpieStaking, _amount);
487           emit ExcessETHTransferredToStaking(_amount);
488       } else { revert InsufficientETH(); }
489   }
```

Listing 2.37: EigenpieWithdrawManager.sol

Impact Unable to fully withdraw excess ETH from the contract.

Suggestion Revise the logic to include the equality condition, ensuring all excess ETH can be transferred.

2.1.27 Unable to update _mLRTWallet due to improper check

Severity Low

Status Fixed in Version 10

Introduced by Version 9

Description In the MLRTWallet contract, the function setMLRTWalletForChain() is used to set the mapping between _destinationChainId and _mLRTWallet. If a _mLRTWallet has already been set for a given _destinationChainId, it cannot be updated again. Considering that _mLRTWallet may become obsolete as the protocol evolves, it would not be possible to update it using setMLRTWalletForChain(). Which is incorrect.

```
343   function setMLRTWalletForChain(uint64 _destinationChainId, address _mLRTWallet) external
          onlyEigenpieEnterprise {
344       UtilLib.checkNonZeroAddress(_mLRTWallet);
345       if (mLRTWalletForChain[_destinationChainId] != address(0)) {
346           revert MLRTWalletForChainAlreadySet();
```

```
347         }
348         mLRTWalletForChain[_destinationChainId] = _mLRTWallet;
349         emit MLRTWalletForChainSet(client, _mLRTWallet, _destinationChainId);
350     }
```

Listing 2.38: MLRTWallet.sol

Impact The protocol is unable to update the `_mLRTWallet` corresponding to the `_destinationChainId`.

Suggestion Revise the logic to ensure that the `_mLRTWallet` can be updated.

2.1.28 Inconsistent withdrawal timing between EigenPie and EigenLayer

Severity Medium

Status Confirmed

Introduced by `Version 11`

Description To accommodate recent changes in `EigenLayer`, `EigenPie` adjusted the timing logic between a user's withdrawal request and the actual withdrawal completion. Specifically, since `EigenLayer` has extended its withdrawal delay by `14` days, `EigenPie` introduced two new variables: `EPOCH_DURATION_SLASHING` and `epochTransitionTimestamp`. This means that if the current timestamp is before `epochTransitionTimestamp`, the user's withdrawal interval is calculated using the old `EPOCH_DURATION`. If the current timestamp is on or after `epochTransitionTimestamp`, the interval is calculated using the new `EPOCH_DURATION_SLASHING`.

However, this approach introduces a potential inconsistency. If a user submits a withdrawal request just one day before `epochTransitionTimestamp`, the function `nextUserWithdrawalTime()` will still calculate the user's withdrawal availability based on the old epoch duration. Given that EigenLayer's upgrade enforces a `14`-day delay for all pending withdrawals, this discrepancy may cause the user to reach the expected withdrawal time in `EigenPie` but still be unable to retrieve their funds from `EigenLayer`, which is incorrect.

```
 90   function nextUserWithdrawalTime() public view returns (uint256) {
 91       // Get next epoch number since withdrawals are processed in the next epoch
 92       uint256 epoch = currentEpoch() + 1;
 93       if (block.timestamp < epochTransitionTimestamp) {
 94           // Before transition: calculate using original EPOCH_DURATION
 95           return startTimestamp + epoch * EPOCH_DURATION + lstWithdrawalDelay;
 96       } else {
 97           // Calculate how many epochs occurred before the transition
 98           uint256 epochsBeforeTransition = (epochTransitionTimestamp - startTimestamp) /
                  EPOCH_DURATION;
 99           // Calculate total time that passed during pre-transition epochs
100           uint256 timeBeforeTransition = epochsBeforeTransition * EPOCH_DURATION;
101           // Calculate remaining epochs after transition point
102           uint256 remainingEpochs = epoch - epochsBeforeTransition;
103           // Calculate additional time using new EPOCH_DURATION_SLASHING
104           uint256 timeAfterTransition = remainingEpochs * EPOCH_DURATION_SLASHING;
105           // Return total time: pre-transition + post-transition + withdrawal delay
106           return startTimestamp + timeBeforeTransition + timeAfterTransition + lstWithdrawalDelay
                  ;
107       }
```

```
108   }
```

Listing 2.39: EigenpieWithdrawManager.sol

```
235   function userQueuingForWithdraw(
236       address asset,
237       uint256 mLRTamount
238   )
239       external
240       whenNotPaused
241       nonReentrant
242       onlySupportedAsset(asset)
243   {
244       address receipt = eigenpieConfig.mLRTReceiptByAsset(asset);
245       uint256 userReceiptBal = IERC20(receipt).balanceOf(msg.sender);
246       if (mLRTamount > userReceiptBal) revert InvalidAmount();
247
248
249       uint256 epochCurr = currentEpoch();
250       bytes32 userToAsset = userToAssetKey(msg.sender, asset);
251       bytes32 assetToEpoch = assetEpochKey(asset, epochCurr);
252
253
254       uint256 rate = IMLRT(receipt).exchangeRateToLST();
255       uint256 withdrawLSTAmt = (rate * mLRTamount) / 1 ether;
256       uint256 userWithdrawableTime = nextUserWithdrawalTime();
257
258
259       UserWithdrawalSchedule memory schedule =
260           UserWithdrawalSchedule(mLRTamount, withdrawLSTAmt, 0, userWithdrawableTime);
261
262
263       if (asset == EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
264           uint256 availableToWithdraw = getAvailableToWithdraw();
265           withdrawRequestNonce++;
266           uint256 nonce = withdrawRequestNonce;
267
268
269           userNonces[msg.sender].push(nonce);
270           nonceToSchedule[nonce] = schedule;
271
272
273           if (withdrawLSTAmt > availableToWithdraw) {
274               // increase the claim reserve to partially fill withdrawRequest with max available
                      in buffer
275               ethClaimReserve += availableToWithdraw;
276               // fill the queue with availableToWithdraw
277               ethWithdrawQueue.queuedWithdrawFilled += availableToWithdraw;
278               // update the queue to fill
279               ethWithdrawQueue.queuedWithdrawToFill += withdrawLSTAmt;
280               // calculate withdrawRequest hash
281               bytes32 withdrawHash = keccak256(abi.encode(nonce, msg.sender));
282
```

```
283
284             withdrawQueued[withdrawHash].queued = true;
285             withdrawQueued[withdrawHash].fillAt = ethWithdrawQueue.queuedWithdrawToFill;
286         } else {
287             // add redeem amount to claimReserve of claim asset
288             ethClaimReserve += withdrawLSTAmt;
289         }
290     } else {
291         WithdrawalSum storage withdrawalSum = withdrawalSums[assetToEpoch];
292         withdrawalSum.assetTotalToWithdrawAmt += withdrawLSTAmt;
293         withdrawalSum.mLRTTotalToBurn += mLRTamount;
294
295
296         withdrawalSchedules[userToAsset].push(schedule);
297     }
298
299
300     IERC20(receipt).safeTransferFrom(msg.sender, address(this), mLRTamount);
301
302
303     emit UserQueuingForWithdrawal(msg.sender, asset, mLRTamount, withdrawLSTAmt, epochCurr,
            userWithdrawableTime);
304 }
```

Listing 2.40: EigenpieWithdrawManager.sol

Impact The user may be unable to withdraw assets at the expected time.

Suggestion Revise the logic to ensure that the withdrawal timing recorded in both `EigenLayer` and `EigenPie` remains consistent.

Feedback from the project The team is aware of this. If someone initiated a withdrawal before the transition, they must wait additional time. We'll display a note on the UI to inform users about this.

2.1.29 Incompatible slash querying logic in function `getEthBalance()`

Severity Medium

Status Confirmed

Introduced by `Version 11`

Description In the latest `EigenLayer` upgrade, the slashing mechanism for users' queued withdrawals has changed. Previously, when a user was slashed during the withdrawal queue period, the associated `podOwnerShares` in their `EigenPod` would become negative, and the slashed amount would be deducted from their withdrawal when executing function `completeQueuedWithdrawal()`. In the updated implementation, `podOwnerShares` is no longer set to a negative value. The actual slashed share amount is now computed during the execution of function `completeQueuedWithdrawal()` based on the slashing factor.

However, the `getEthBalance()` function in the contract `NodeDelegator` contract within the `EigenPie` protocol has not been updated accordingly. It still relies on `podOwnerShares` to infer how many shares were slashed, which is no longer valid under the new logic. Instead, it should

use EigenLayer's view function getQueuedWithdrawal() to accurately determine the final share amount after accounting for slashing.

```
 99   function getEthBalance(
100       IEigenpieConfig eigenpieConfig,
101       uint256 stakedButNotVerifiedEth,
102       address nodeDelegator,
103       uint256 queuedETHShares
104   )
105       public
106       view
107       returns (uint256)
108   {
109       IDelegationManager delegationManager = IDelegationManager(eigenpieConfig.getContract(
              EigenpieConstants.EIGEN_DELEGATION_MANAGER));
110       IStrategy[] memory strategies = new IStrategy[](1);
111       strategies[0] = IStrategy(eigenpieConfig.assetStrategy(EigenpieConstants.
              PLATFORM_TOKEN_ADDRESS));
112
113       (uint256[] memory withdrawableShares,) = delegationManager.getWithdrawableShares(
              nodeDelegator, strategies);
114       uint256 podOwnerShares = withdrawableShares[0];
115
116
117       if (podOwnerShares < 0) {
118           // Case 1: stakedButNotVerifiedEth >= absPodOwnerShares
119           if (stakedButNotVerifiedEth >= podOwnerShares) {
120               return queuedETHShares + stakedButNotVerifiedEth - podOwnerShares;
121           }
122           // Case 2: stakedButNotVerifiedEth < absPodOwnerShares
123           uint256 remainingDeficit = podOwnerShares - stakedButNotVerifiedEth;
124           // Check if queuedETHShares can cover the remaining deficit
125           return queuedETHShares >= remainingDeficit ? queuedETHShares - remainingDeficit : 0;
126       } else {
127           return queuedETHShares + stakedButNotVerifiedEth + uint256(podOwnerShares);
128       }
129   }
```

Listing 2.41: AssetManagementLib.sol

Impact The ETH balance displayed in NodeDelegator is inaccurate.

Suggestion Update function getEthBalance() to reference function getQueuedWithdrawal() for accurate share calculation.

Feedback from the project The team has decided not to implement slashing with their operator at this time. Instead, the plan is to allow other operators to opt in first, assess the associated risks and rewards, and then reconsider slashing after a month.

2.2 Additional Recommendation

2.2.1 Add checks on the total weights in reward distribution

Status Fixed in `Version 4`

Introduced by `Version 1`

Description The contract `RewardDistributor` distributes native token rewards to multiple destinations based on the configured reward distribution. However, there is no check to ensure that the total weights assigned to the reward destinations equal to `DENOMINATOR`.

```
50  function _forwardETH() internal {
51      uint256 balance = address(this).balance;
52      if (balance == 0) {
53          return;
54      }
55
56      uint256 length = rewardDests.length;
57
58      for (uint256 i; i < length;) {
59          RewardDestinations memory dest = rewardDests[i];
60          uint256 toSendAmount = balance * dest.value / EigenpieConstants.DENOMINATOR;
61
62          if (dest.needWrap) {
63              // TODO will need to handle wrap as Weth
64              // TODO will need to check if isAddress and queue reward to rewarder
65          } else {
66              TransferHelper.safeTransferETH(dest.to, toSendAmount);
67          }
68
69          unchecked {
70              ++i;
71          }
72      }
73  }
```

Listing 2.42: contracts/RewardDistributor.sol

```
79  function addRewardDestination(
80      uint256 _value,
81      address _to,
82      bool _isAddress,
83      bool _needWrap
84  )
85      external
86      onlyDefaultAdmin
87  {
88      if (_value > EigenpieConstants.DENOMINATOR) revert InvalidFeePercentage();
89      UtilLib.checkNonZeroAddress(_to);
90
91      rewardDests.push(RewardDestinations({ value: _value, to: _to, isAddress: _isAddress,
            needWrap: _needWrap }));
92      emit RewardDestinationAdded(rewardDests.length - 1, _value, _to, _isAddress, _needWrap);
```

```
 93    }
 94
 95    function setRewardDestination(
 96        uint256 _index,
 97        uint256 _value,
 98        address _to,
 99        bool _isAddress,
100        bool _needWrap
101    )
102        external
103        onlyDefaultAdmin
104    {
105        if (_index >= rewardDests.length) revert InvalidIndex();
106        if (_value > EigenpieConstants.DENOMINATOR) revert InvalidFeePercentage();
107        UtilLib.checkNonZeroAddress(_to);
108
109        RewardDestinations storage dest = rewardDests[_index];
110        dest.value = _value;
111        dest.to = _to;
112        dest.isAddress = _isAddress;
113        dest.needWrap = _needWrap;
114        emit RewardDestinationUpdated(_index, _value, _to, _isAddress, _needWrap);
115    }
116
117    function removeRewardDestination(uint256 _index) external onlyDefaultAdmin {
118        if (_index >= rewardDests.length) revert InvalidIndex();
119
120        for (uint256 i = _index; i < rewardDests.length - 1; i++) {
121            rewardDests[i] = rewardDests[i + 1];
122        }
123        rewardDests.pop();
124        emit RewardDestinationRemoved(_index);
125    }
```

Listing 2.43: contracts/RewardDistributor.sol

Suggestion Check the total weights when changing the reward distribution configuration.

2.2.2 Add `view` modifier to function `restakedLess()`

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the contract `MLRTWallet`, the function `restakedLess()` invokes the function `restakedLess()` of the contract `EigenpieEnterprise` to query the staked amount of a specified client. This invocation does not alter the contract's state and simply reads data stored in the contract. However, the function `restakedLess()` has not been marked as `view`, so users must pay gas to invoke `restakedLess()`, which is incorrect.

```
76    function restakedLess(address underlyingToken) external returns (uint256 ethLess, uint256
          shouldBurn) {
77        return eigenpieEnterprise.restakedLess(client, underlyingToken);
```

```
78    }
```

Listing 2.44: contracts/MLRTWallet.sol

```
97    function restakedLess(
98        address client,
99        address underlyingToken
100   )
101       external
102       view
103       override
104       returns (uint256 underlyingLessAmount, uint256 mlrtShouldBurn)
105   {
106       return _checkCollateralLess(client, underlyingToken);
107   }
```

Listing 2.45: contracts/EigenpieEnterprise.sol

Suggestion Use the `view` modifier for the function `restakedLess()`.

2.2.3 Remove redundant code

Status Confirmed

Introduced by `Version 1`

Description Users can pay a penalty through the function `forceUnLock()` to forcibly exit their locked positions. The invoke to the internal function `_checkInCoolDown()` is redundant. Specifically, the previous check has already ensured that the user has not unlocked, so no further checks are needed within this branch.

Similarly, the function `setEigenPod()` in the contract `EigenpieEnterprise` is also redundant. The function `getPod()` retrieves the corresponding `EigenPod` based on the client passed in. According to the implementation of the contract `EigenPodManager`, if the `EigenPod` corresponding to the client is `address(0)`, it will calculate and return the `EigenPod` address based on the client's address. Additionally, when the function `updateAllowedClient()` registers the client, it simultaneously sets the address of the `EigenPod`, so there is no scenario where the function `setEigenPod()` needs to be invoked separately.

In addition, the function `setEigenPod()` and the global variable `eigenPod` in the contract `MLRTWallet` are also redundant.

```
312   function forceUnLock(
313       uint256 _slotIndex
314   ) external whenNotPaused nonReentrant {
315       _checkIdexInBoundary(msg.sender, _slotIndex);
316       UserUnlocking storage slot = userUnlockings[msg.sender][_slotIndex];
317
318       // Check if the slot is already unlocked (amountInCoolDown == 0) and revert if so
319       if (slot.amountInCoolDown == 0) {
320           revert UnlockedAlready();
321       }
322
323       uint256 penaltyAmount = 0;
```

```
324         uint256 amountToUser = slot.amountInCoolDown; // Default to the full amount
325
326         _claimFromRewarder(msg.sender);
327
328         // If the current time is not beyond the slot's endTime, then there's penalty.
329         if (block.timestamp < slot.endTime) {
330             _checkInCoolDown(msg.sender, _slotIndex);
331
332             (penaltyAmount, amountToUser) = expectedPenaltyAmount(_slotIndex);
333         }
334
335         _unlock(slot.amountInCoolDown);
336
337         IERC20(Eigenpie).safeTransfer(msg.sender, amountToUser);
338         totalPenalty += penaltyAmount;
339
340         slot.amountInCoolDown = 0;
341         slot.endTime = block.timestamp;
342
343         emit ForceUnLock(msg.sender, _slotIndex, amountToUser, penaltyAmount);
344     }
```

Listing 2.46: contracts/vlEigenpie.sol

```
203     function setEigenPod(address client) external {
204         ClientData storage clientData = allowedClients[client];
205         UtilLib.checkNonZeroAddress(clientData.mlrtWallet);
206
207         _setEigenPod(client, clientData);
208     }
```

Listing 2.47: contracts/EigenpieEnterprise.sol

```
365     function _setEigenPod(address client, ClientData storage clientData) internal returns (address
            eigenPod) {
366         eigenPod = _fetchEigenPod(client);
367
368         if (eigenPod != address(0) && clientData.eigenPod == address(0)) {
369             registeredPod[eigenPod] = true;
370             clientData.eigenPod = eigenPod;
371             _updateMLRTWalletEigenPod(clientData, eigenPod);
372             emit EigenPodSet(client, eigenPod);
373         }
374     }
```

Listing 2.48: contracts/EigenpieEnterprise.sol

```
376     function _fetchEigenPod(address client) internal view returns (address) {
377         IEigenPodManager eigenPodManager = _getEigenPodManager();
378         return address(eigenPodManager.getPod(client));
379     }
```

Listing 2.49: contracts/EigenpieEnterprise.sol

```
192   function setEigenPod(address _eigenpod) external onlyEigenpieEnterprise {
193       UtilLib.checkNonZeroAddress(_eigenpod);
194       eigenPod = _eigenpod;
195       emit EigenPodUpdated(client, _eigenpod);
196   }
```

Listing 2.50: contracts/MLRTWallet.sol

```
25   address public eigenPod;
```

Listing 2.51: contracts/MLRTWallet.sol

Suggestion Remove this redundant code.

Feedback from the project Removed the redundant code in the contract `vlEigenpie` but want to keep the code in the contracts `EigenpieEnterprise` and `MLRTWallet`.

2.2.4 Incorrect logic in function `getFullyUnlock()`

Status Fixed in `Version 4`

Introduced by `Version 1`

Description The function `getFullyUnlock()` is used to query the amount of `EigenPie` tokens that a specified user can currently unlock. Specifically, if the current `block.timestamp` is greater than or equal to `userUnlockings[_user][i].endTime`, the tokens should be considered unlocked. However, in the current implementation, the condition does not include the "equal to" case, which is incorrect.

```
136   function getFullyUnlock(
137       address _user
138   ) public view override returns (uint256 unlockedAmount) {
139       uint256 length = getUserUnlockSlotLength(_user);
140       for (uint256 i; i < length; i++) {
141           if (
142               userUnlockings[_user][i].amountInCoolDown > 0 &&
143               block.timestamp > userUnlockings[_user][i].endTime
144           ) unlockedAmount += userUnlockings[_user][i].amountInCoolDown;
145       }
146   }
```

Listing 2.52: contracts/vlEigenpie.sol

Suggestion Revise the logic by replacing it with `block.timestamp >= userUnlockings[_user][i].endTime`.

2.2.5 Inconsistent logic in function `addNodeDelegatorContractToQueue()`

Status Fixed in `Version 4`

Introduced by `Version 1`

Description The protocol owner can invoke the function `addNodeDelegatorContractToQueue()` to add a new `NodeDelegator` to the protocol. The `maxNodeDelegatorLimit` sets the upper limit

for the number of NodeDelegator. However, in the current check, the protocol does not account for whether the input parameters contain duplicate NodeDelegator, while the loop logic does take into consideration the possibility of duplicates. This inconsistency may cause the function to not function as expected.

```
227   function addNodeDelegatorContractToQueue(address[] calldata nodeDelegatorContracts) external
          onlyDefaultAdmin {
228       uint256 length = nodeDelegatorContracts.length;
229       if (nodeDelegatorQueue.length + length > maxNodeDelegatorLimit) {
230           revert MaximumNodeDelegatorLimitReached();
231       }
232
233       for (uint256 i; i < length;) {
234           UtilLib.checkNonZeroAddress(nodeDelegatorContracts[i]);
235
236           // check if node delegator contract is already added and add it if not
237           if (isNodeDelegator[nodeDelegatorContracts[i]] == 0) {
238               nodeDelegatorQueue.push(nodeDelegatorContracts[i]);
239               isNodeDelegator[nodeDelegatorContracts[i]] = 1;
240           }
241
242           unchecked {
243               ++i;
244           }
245       }
246
247       emit NodeDelegatorAddedinQueue(nodeDelegatorContracts);
248   }
```

Listing 2.53: contracts/EigenpieStaking.sol

Suggestion Revise the logic by moving the check for the upper limit of NodeDelegator to after the loop.

2.2.6 Add checks in advanceCycle()

Status Fixed in Version 4

Introduced by Version 1

Description In the contract EigenpiePreDepositHelper, the admin is able to invoke the function setCycleClaimable() to make the currentCycle claimable, allowing users to claim the corresponding mLRT tokens. When it's time to move to the next cycle, the admin will invoke the advanceCycle() function to increment the currentCycle by 1. However, if the function advanceCycle() is invoked first, entering the next cycle, the previous cycle cannot be set as claimable, resulting in users being unable to withdraw their assets.

```
157   /// @notice Sets the current cycle as claimable or not.
158   function setCycleClaimable(bool _isClaim) external onlyDefaultAdmin {
159       claimableCycles[currentCycle] = _isClaim;
160       emit CycleModified(_isClaim, currentCycle);
161   }
162
```

```
163   /// @notice Advances to the next cycle.
164   function advanceCycle() external onlyDefaultAdmin {
165       currentCycle++;
166   }
```

Listing 2.54: contracts/EigenpiePreDepositHelper.sol

Suggestion Add a check to ensure `currentCycle` is claimable before entering into the next cycle.

2.2.7 Improper check in function `makeBeaconDeposit()`

Status Fixed in `Version 4`

Introduced by `Version 1`

Description In the `ValidatorLib` library, the function `makeBeaconDeposit()` takes input parameters of beacon deposit data for multiple validators and makes deposits into the Beacon Deposit Contract. However, when checking the number of maximum validators against the preset constant `MAX_VALIDATORS`, the larger-than-or-equal-to (`>=`) condition is used, which is incorrect.

```
26   function makeBeaconDeposit(
27       bytes[] memory publicKeys,
28       bytes[] memory signatures,
29       bytes32[] memory depositDataRoots,
30       IEigenpieConfig eigenpieConfig,
31       address eigenPod
32   )
33       external
34   {
35       // sanity checks
36       uint256 count = depositDataRoots.length;
37       if (count == 0) revert INodeDelegator.AtLeastOneValidator();
38       if (count >= EigenpieConstants.MAX_VALIDATORS) {
39           revert INodeDelegator.MaxValidatorsInput();
40       }
41       if (publicKeys.length != count) {
42           revert INodeDelegator.PublicKeyNotMatch();
43       }
44       if (signatures.length != count) {
45           revert INodeDelegator.SignaturesNotMatch();
46       }
47   }
```

Listing 2.55: contracts/libraries/ValidatorLib.sol

Suggestion Replace the condition to check for maximum validators to larger-than (`>`).

2.2.8 Remove redundant logic related to deprecated contracts

Status Fixed in `Version 4`

Introduced by `Version 1`

Description According to the EigenLayer documentation, the contract DelayedWithdrawalRouter will be deprecated after the PEPE upgrade. After the upgrade, the gas refund logic in the following code logic is disabled as it requires the msg.sender to be DelayedWithdrawalRouter. This behavior cannot function properly after the upgrade of the EigenLayer.

```
55  receive() external payable {
56      address eigenStaking = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_STAKING);
57      // If Eth from Eigenstaking, then should stay waiting to be restaked;
58      if (msg.sender == eigenStaking) {
59          return;
60      }
61
62      uint256 gasRefunded;
63      address dwr = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_DWR);
64      // If Eth from dwr, then is partial withdraw of CL reward
65      if (msg.sender == dwr && adminGasSpentInWei[tx.origin] > 0) {
66          gasRefunded = _refundGas();
67
68          // If no funds left, return
69          if (msg.value == gasRefunded) {
70              return;
71          }
72      }
73      // Forward remaining balance to rewarDistributor.
74      // Any random eth transfer to this contract will also be treated as reward.
75      address rewarDistributor = eigenpieConfig.getContract(EigenpieConstants.
            EIGENPIE_REWADR_DISTRIBUTOR);
76      TransferHelper.safeTransferETH(rewarDistributor, msg.value - gasRefunded);
77
78      emit RewardsForwarded(rewarDistributor, msg.value);
79  }
```

Listing 2.56: contracts/NodeDelegator.sol

Suggestion Remove the redundant logic related to DelayedWithdrawalRouter.

2.2.9 Fix incorrect parameter for events

Status Fixed in Version 4

Introduced by Version 3

Description For the contracts MLRTOFTBridge and MLRTOFTAdapter, the parameters for the event BridgeMLRT is incorrectly assigned as the _refundAddress.

```
32  event BridgeMLRT(
33      address indexed from,
34      uint32 indexed dstEid,
35      uint256 amountSent,
36      uint256 amountReceived,
37      bytes32 indexed guid
38  );
```

Listing 2.57: contracts/MLRTOFTBridge.sol

```
109    emit BridgeMLRT(
110        _refundAddress,
111        _dstEid,
112        amountSentLD,
113        amountReceivedLD,
114        msgReceipt.guid
115    );
```

Listing 2.58: contracts/MLRTOFTBridge.sol

Suggestion Fix the incorrect variable for the event `BridgeMLRT`.

2.2.10 Gas optimizations

Status Fixed in `Version 6`

Introduced by `Version 5`

Description For the There are several locations where the code can be optimized for gas optimizations and clearer logic.

1. In the following code segment, the local variables `availableWithdraw` can be moved inside the if clause, and the queued variable can be removed.

```
197    uint256 availableToWithdraw = getAvailableToWithdraw();
198    bool queued = false;
199    uint256 nonce;
200
201    if (asset == EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
202        if (withdrawLSTAmt > availableToWithdraw) {
203            withdrawRequestNonce++;
204            // increase the claim reserve to partially fill withdrawRequest with max
                   available in buffer
205            ethClaimReserve += availableToWithdraw;
206            // fill the queue with availableToWithdraw
207            ethWithdrawQueue.queuedWithdrawFilled += availableToWithdraw;
208            // update the queue to fill
209            ethWithdrawQueue.queuedWithdrawToFill += withdrawLSTAmt;
210            // calculate withdrawRequest hash
211            bytes32 withdrawHash = keccak256(abi.encode(withdrawRequestNonce, msg.sender));
212
213            withdrawQueued[withdrawHash].queued = true;
214            withdrawQueued[withdrawHash].fillAt = ethWithdrawQueue.queuedWithdrawToFill;
215            queued = true;
216
217            nonce = withdrawRequestNonce;
218        } else {
219            // add redeem amount to claimReserve of claim asset
220            ethClaimReserve += withdrawLSTAmt;
221        }
222    } else {
223        WithdrawalSum storage withdrawalSum = withdrawalSums[assetToEpoch];
224        withdrawalSum.assetTotalToWithdrawAmt += withdrawLSTAmt;
225        withdrawalSum.mLRTTotalToBurn += mLRTamount;
```

```
226    }
```

Listing 2.59: contracts/EigenpieWithdrawManager.sol

2. In the following code segment, the calculation of the local variable `currEthBalance` can reuse the result of the local variable `bufferToFill`.

```
444    function _checkAndFillETHWithdrawBuffer(uint256 _amount) internal {
445        address eigenpieWithdrawManager = eigenpieConfig.getContract(EigenpieConstants.
               EIGENPIE_WITHDRAW_MANAGER);
446        // Check the withdraw buffer and fill if below buffer target
447        uint256 bufferToFill = IEigenpieWithdrawManager(eigenpieWithdrawManager).
               getWithdrawDeficit();
448        uint256 totalETHBal = currEthBalance + _amount;
449
450        if (bufferToFill > 0) {
451            bufferToFill = (totalETHBal <= bufferToFill) ? totalETHBal : bufferToFill;
452            // fill withdraw buffer from received ETH
453            IEigenpieWithdrawManager(eigenpieWithdrawManager).fillEthWithdrawBuffer{ value:
                   bufferToFill }();
454            emit BufferFilled(bufferToFill);
455        }
456        currEthBalance = (totalETHBal <= bufferToFill) ? 0 : totalETHBal - bufferToFill;
457    }
```

Listing 2.60: contracts/NodeDelegator.sol

Suggestion Refactor the corresponding logic.

2.2.11 Redundant code

Status Fixed in `Version 12`

Introduced by `Version 11`

Description The function `_getPodShares()` is a view function that returns the amount of native tokens a specified client has deposited into `EigenLayer`. It internally invokes `getWithdrawableShares()` from `EigenLayer` to fetch the client's native token balance. Since `getWithdrawableShares()` is guaranteed to return a non-negative value under the current implementation, the check for whether `podShares` is less than zero is redundant.

```
253    function getWithdrawableShares(
254        address staker,
255        IStrategy[] memory strategies
256    )
257        external
258        view
259        returns (
260            uint256[] memory withdrawableShares,
261            uint256[] memory depositShares
262        );
```

Listing 2.61: IDelegationManager.sol

```
439   function _getPodShares(address client) internal view returns (uint256 podShares) {
440       IDelegationManager delegationManager = IDelegationManager(eigenpieConfig.getContract(
              EigenpieConstants.EIGEN_DELEGATION_MANAGER));
441       IStrategy[] memory strategies = new IStrategy[](1);
442       strategies[0] = IStrategy(eigenpieConfig.assetStrategy(EigenpieConstants.
              PLATFORM_TOKEN_ADDRESS));
443       (uint256[] memory withdrawableShares,) = delegationManager.getWithdrawableShares(client,
              strategies);
444       podShares = withdrawableShares[0];
445       if (podShares < 0) {
446           return 0;
447       }
448       return podShares;
449   }
```

Listing 2.62: EigenpieEnterprise.sol

Suggestion Remove this redundant code.

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description The protocol includes several privileged functions, such as function `updateExchangeRateCeiling()` and `updatePriceAdapterFor()`. If the owner's private key is lost or maliciously exploited, it could potentially cause losses to users.

Feedback from the Project We're using multisig as owner to govern our contracts.

2.3.2 Lack of gas fee check during cross-chain

Introduced by Version 1

Description In `MLRTOFT.bridgeMLRT()` and `MLRTOFTAdapter.bridgeMLRT()`, there is no check on the user's input for msg.value, which may result in the transaction failing when executing the message content on the target chain.

Feedback from the Project We plan on leaving that up to the user to use the function `getEstimateGasFees()` before interacting with the `bridgeMLRT()` function.

2.3.3 Potential inconsistent pausing behavior

Introduced by Version 1

Description There is a potential maintenance problem that once the contract `RewardDistributor`, the reward distribution from the contract `NodeDelegator` would revert. Therefore, the pausing should only happen in very rare cases.

```
41    receive() external payable nonReentrant {
42        _forwardETH();
43    }
44
45    // TODO, will have to handle ERC20 if reward in LST form
46    function forwardRewards() external payable nonReentrant whenNotPaused onlyEigenpieManager {
47        _forwardETH();
48    }
```

Listing 2.63: contracts/RewardDistributor.sol

Feedback from the Project If we ever need to pause the contract, we will pause both deposits and withdrawals as well. Additionally, since we automate the reward claiming process, we can pause that too, meaning reward claims will also be halted. In short, everything will pause simultaneously if needed.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS