BLOCKSEC

Security Audit Report for EigenPie Contracts

Date: November 28, 2024 **Version:** 1.1

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

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-11-29 02:15:15+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

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]

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

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

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

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 [5] and Common Weakness Enumeration [6]. 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.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

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:

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

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

- **Undetermined** No response yet.
- **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-two** potential security issues. Besides, we have **ten** recommendations and **three** notes.

- High Risk: 7
- Medium Risk: 13
- Low Risk: 2
- Recommendation: 10
- 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	-	Add checks on the total weights in reward distribution	Recommendation	Fixed
24	-	Add `view` modifier to function `restakedLess()`	Recommendation	Fixed
25	-	Remove redundant code	Recommendation	Confirmed
26	-	Incorrect logic in function `getFullyUnlock()`	Recommendation	Fixed
27	-	Inconsistent logic in function `addNodeDelegatorContractToQueue()`	Recommendation	Fixed
28	-	Add checks in `advanceCycle()`	Recommendation	Fixed
29	-	Improper check in function `makeBeaconDeposit()`	Recommendation	Fixed
30	-	Remove redundant logic related to deprecated contracts	Recommendation	Fixed
31	-	Fix incorrect parameter for events	Recommendation	Fixed
32	-	Gas optimizations	Recommendation	Fixed
33	-	Potential centralization risks	Note	-
34	-	Lack of gas fee check during cross-chain	Note	-
35	-	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.

```
function setRewarder(address _rewarder) external onlyDefaultAdmin {
    address oldRewarder = address(rewarder);
    rewarder = IVLStreamRewarder(_rewarder);
    emit RewarderUpdated(oldRewarder, _rewarder);
}
```

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.

```
function _makeBeaconDeposit(
    bytes[] memory publicKeys,
    bytes[] memory signatures,
    bytes32[] memory depositDataRoots
)
    internal
{
    ValidatorLib.makeBeaconDeposit(publicKeys, signatures, depositDataRoots, eigenpieConfig,
        address(eigenPod));
    stakedButNotVerifiedEth += publicKeys.length * EigenpieConstants.DEPOSIT_AMOUNT;
}
```

Listing 2.2: contracts/NodeDelegator.sol

```
function verifyWithdrawalCredentials(
    uint64 beaconTimestamp,
    BeaconChainProofs.StateRootProof calldata stateRootProof,
    uint40[] calldata validatorIndices,
    bytes[] calldata validatorFieldsProofs,
    bytes32[][] calldata validatorFields
)
    external
    whenNotPaused
    onlyAllowedBot
{
    uint256 gasBefore = gasleft();
    stakedButNotVerifiedEth -= ValidatorLib.verifyWithdrawalCredentials(
        eigenPod, beaconTimestamp, stateRootProof, validatorIndices, validatorFieldsProofs,
            validatorFields
    );
    // update the gas spent for RestakeAdmin
    _recordGas(gasBefore);
}
```

Listing 2.3: contracts/NodeDelegator.sol

```
function verifyWithdrawalCredentials(
    IEigenPod eigenPod,
    uint64 beaconTimestamp,
    BeaconChainProofs.StateRootProof calldata stateRootProof,
    uint40[] calldata validatorIndices,
    bytes[] calldata validatorFieldsProofs,
    bytes32[][] calldata validatorFields
)
    external
    returns (uint256 stakedButNotVerifiedEth)
{
    eigenPod.verifyWithdrawalCredentials(
        beaconTimestamp, stateRootProof, validatorIndices, validatorFieldsProofs,
            validatorFields
    );

    // Decrement the staked but not verified ETH
    for (uint256 i = 0; i < validatorFields.length;) {
        uint64 validatorCurrentBalanceGwei = BeaconChainProofs.getEffectiveBalanceGwei(
            validatorFields[i]);
        stakedButNotVerifiedEth += (validatorCurrentBalanceGwei * EigenpieConstants.GWEI_TO_WEI
            );

        unchecked {
            ++i;
        }
    }
}
```

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.

```
function userWithdrawAsset(address[] memory assets) external nonReentrant {
    uint256[] memory claimedWithdrawalSchedules = new uint256[](assets.length);
    for (uint256 i = 0; i < assets.length;) {
        bytes32 userToAsset = userToAssetKey(msg.sender, assets[i]);
        UserWithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset];

        uint256 totalClaimedAmount;
        uint256 claimedWithdrawalSchedulesPerAsset;

        for (uint256 j = 0; j < schedules.length;) {
            UserWithdrawalSchedule storage schedule = schedules[j];

            // if claimable
            if (block.timestamp >= schedule.endTime && schedule.claimedAmt == 0) {
                claimedWithdrawalSchedulesPerAsset++;

                schedule.claimedAmt = schedule.queuedWithdrawLSTAmt;
                totalClaimedAmount += schedule.queuedWithdrawLSTAmt;
            } else if (block.timestamp >= schedule.endTime && schedule.claimedAmt == schedule.
                queuedWithdrawLSTAmt) {
                claimedWithdrawalSchedulesPerAsset++;
            }

            unchecked {
                ++j;
            }
        }

        claimedWithdrawalSchedules[i] = claimedWithdrawalSchedulesPerAsset;

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

        unchecked {
            ++i;
        }
    }

    _cleanUpWithdrawalSchedules(assets, claimedWithdrawalSchedules);
}
```

Listing 2.5: contracts/EigenpieWithdrawManager.sol

```
function _cleanUpWithdrawalSchedules(
    address[] memory assets,
    uint256[] memory claimedWithdrawalSchedules
) internal {
    for (uint256 i = 0; i < assets.length;) {
        bytes32 userToAsset = userToAssetKey(msg.sender, assets[i]);
        UserWithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset];

        if (claimedWithdrawalSchedules[i] >= withdrawalscheduleCleanUp) {
            for (uint256 j = 0; j < schedules.length - claimedWithdrawalSchedules[i];) {
                schedules[j] = schedules[j + claimedWithdrawalSchedules[i]];

                unchecked {
                    ++j;
                }
            }

            while (claimedWithdrawalSchedules[i] > 0) {
                schedules.pop();
                claimedWithdrawalSchedules[i]--;
            }
        }

        unchecked {
            ++i;
        }
    }
}
```

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.

```
function deposit(
    uint64[] memory operatorIds,
    uint256 amount,
    ISSVNetworkCore.Cluster memory cluster
)
    external
    onlyEigenpieManager
{
    address ssvNetwork = eigenpieConfig.getContract(EigenpieConstants.SSVNETWORK_ENTRY);
    ISSVClusters(ssvNetwork).deposit(address(this), operatorIds, amount, cluster);
}
```

Listing 2.7: contracts/NodeDelegator.sol

```
function deposit(
    uint64[] memory operatorIds,
    uint256 amount,
    ISSVNetworkCore.Cluster memory cluster
)
    external
    onlyClientOrManager
{
    address ssvNetwork = eigenpieConfig.getContract(EigenpieConstants.SSVNETWORK_ENTRY);
    ISSVClusters(ssvNetwork).deposit(address(this), operatorIds, amount, cluster);
}
```

Listing 2.8: contracts/WLNodedelegator.sol

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

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

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

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 `VlStreamRewarder` 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.

```
function getReward(
    address _account
) external onlyVlEigenpie returns (bool) {
    updateFor(_account);

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

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.

```
function registerReStaking(
    address underlyingToken,
    uint256 amountToMintMlt
)
    external
    nonReentrant
    onlyAllowedClient
{
    ClientData storage clientData = allowedClients[msg.sender];
    address receipt;
    uint256 amountToMint;
    _updateClientRestakingData(msg.sender, clientData);
    _checkValidMint(msg.sender, clientData, underlyingToken, amountToMintMlt);
    (receipt, amountToMint) = _calculateMintAndUpdate(msg.sender, underlyingToken,
        amountToMintMlt);

    if (clientData.mlrtWallet == address(0)) {
        clientData.mlrtWallet = _deployMLRTWallet(msg.sender, clientData.eigenPod);
    }

    IMLRT(receipt).mint(clientData.mlrtWallet, amountToMint);
    totalMintedMlrt[receipt] += amountToMint;

    emit ClientRegisterRestake(msg.sender, clientData.mlrtWallet, underlyingToken,
        amountToMintMlt, amountToMint);
}
```

Listing 2.11: contracts/EigenpieEnterprise.sol

```
function _updateClientRestakingData(address client, ClientData storage clientData) internal {
    (address[] memory underlyingTokens, uint256[] memory underlyingAmounts) =
        getRestakingShares(client);

    uint256 totalStrategies = underlyingTokens.length;
    for (uint256 i = 0; i < totalStrategies - 1; i++) {
        clientAssetMapping[client][underlyingTokens[i]].lstRestakedAmount = underlyingAmounts[i
            ];
        emit UpdateClientLSTRestakedAmount(client, underlyingTokens[i], underlyingAmounts[i]);
    }
```

```

        clientData.nativeRestakedAmount = underlyingAmounts[totalStrategies - 1];
        emit UpdateClientNativeRestakedAmount(client, clientData.nativeRestakedAmount);
    }
```

Listing 2.12: contracts/EigenpieEnterprise.sol

```
    function getRestakingShares(address client) public view returns (address[] memory, uint256[] memory) {
        uint256 podShares = _getPodShares(client);
        (address[] memory underlyingTokens, uint256[] memory underlyingAmounts, uint256 assetLength) =
            _getStrategyShares(client); // The last entry here will be vacant reason being native strategy is not included
            // in the strategies array

        // Add native strategy (platform token) shares to the array
        underlyingTokens[assetLength - 1] = EigenpieConstants.PLATFORM_TOKEN_ADDRESS;
        underlyingAmounts[assetLength - 1] = podShares;
        return (underlyingTokens, underlyingAmounts);
    }
```

Listing 2.13: contracts/EigenpieEnterprise.sol

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

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 contractvlEigenpie.

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.

```
function cancelUnlock(
    uint256 _slotIndex
) external override whenNotPaused nonReentrant {
    _checkIdexInBoundary(msg.sender, _slotIndex);
    UserUnlocking storage slot = userUnlockings[msg.sender][_slotIndex];

    _checkInCoolDown(msg.sender, _slotIndex);

    totalAmountInCoolDown -= slot.amountInCoolDown; // reduce amount to cool down accordingly
    slot.amountInCoolDown = 0; // not in cool down anymore

    if (address(rewarder) != address(0)) rewarder.updateFor(msg.sender);

    emit ReLock(msg.sender, _slotIndex, slot.amountInCoolDown);
}
```

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.

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

Listing 2.16: contracts/NodeDelegator.sol

```
function getEthBalance(
    IEigenPodManager eigenPodManager,
    uint256 stakedButNotVerifiedEth,
    address nodeDelegator
)
    public
    view
    returns (uint256)
{
    int256 podOwnerShares = eigenPodManager.podOwnerShares(nodeDelegator);
    return podOwnerShares < 0
        ? stakedButNotVerifiedEth - uint256(-podOwnerShares)
        : stakedButNotVerifiedEth + uint256(podOwnerShares);
}
```

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.

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

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.

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

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.

```
function _provisionReward(uint256 _rewards, address _rewardToken) internal {
    _rewards = _rewards * DENOMINATOR; // to support small deciaml rewards

    Reward storage rewardInfo = rewards[_rewardToken];

    if (totalStaked() == 0) {
        rewardInfo.queuedRewards = rewardInfo.queuedRewards + _rewards;
        return;
    }

    rewardInfo.rewardPerTokenStored = rewardPerToken(_rewardToken);
    _rewards = _rewards + rewardInfo.queuedRewards;
    rewardInfo.queuedRewards = 0;

    if (block.timestamp >= rewardInfo.periodFinish) {
        rewardInfo.rewardRate = _rewards / duration;
    } else {
        uint256 remaining = rewardInfo.periodFinish - block.timestamp;
        uint256 leftover = remaining * rewardInfo.rewardRate;
        _rewards = _rewards + leftover;
        rewardInfo.rewardRate = _rewards / duration;
    }

    rewardInfo.lastUpdateTime = block.timestamp;
    rewardInfo.periodFinish = block.timestamp + duration;
}
```

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.

```
function registerReStaking(
    address underlyingToken,
    uint256 amountToMintMlt
)
    external
    nonReentrant
    onlyAllowedClient
{
    ClientData storage clientData = allowedClients[msg.sender];
    address receipt;
    uint256 amountToMint;
    _updateClientRestakingData(msg.sender, clientData);
    _checkValidMint(msg.sender, clientData, underlyingToken, amountToMintMlt);
```

```
        (receipt, amountToMint) = _calculateMintAndUpdate(msg.sender, underlyingToken,
            amountToMintMlt);

        if (clientData.mlrtWallet == address(0)) {
            clientData.mlrtWallet = _deployMLRTWallet(msg.sender, clientData.eigenPod);
        }

        IMLRT(receipt).mint(clientData.mlrtWallet, amountToMint);
        totalMintedMlrt[receipt] += amountToMint;

        emit ClientRegisterRestake(msg.sender, clientData.mlrtWallet, underlyingToken,
            amountToMintMlt, amountToMint);
    }
```

Listing 2.21: contracts/EigenpieEnterprise.sol

```
    function _checkValidMint(
        address client,
        ClientData storage clientData,
        address underlyingToken,
        uint256 amountToMintMlt
    )
        internal
        view
    {
        uint256 quotaLeft;
        if (underlyingToken != EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
            LSTData memory lstData = clientAssetMapping[client][underlyingToken];
            quotaLeft = lstData.lstRestakedAmount - lstData.lstUsed;
        } else {
            quotaLeft = clientData.nativeRestakedAmount - clientData.nativeUsed;
        }
        if (quotaLeft < amountToMintMlt) {
            revert AssetNotEnough(quotaLeft, amountToMintMlt);
        }
    }
```

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.

```
function depositAsset(
    address asset,
    uint256 depositAmount,
    uint256 minRec,
    address referral
)
    external
    payable
    whenNotPaused
    nonReentrant
    onlySupportedAsset(asset)
{
    // checks
    bool isNative = UtilLib.isNativeToken(asset);
    if (isNative && msg.value != depositAmount) {
        revert InvalidAmountToDeposit();
    }

    if (depositAmount == 0 || depositAmount < minAmountToDeposit) {
        revert InvalidAmountToDeposit();
    }

    if (depositAmount > getAssetCurrentLimit(asset)) {
        revert MaximumDepositLimitReached();
    }

    uint256 mintedAmount;

    if (isPreDeposit && !isNative) {
        // only when not native and in pre deposit phase, we don't min receipt token to users
        address eigenpiePreDepositHelper = eigenpieConfig.getContract(EigenpieConstants.
            EIGENPIE_PREDEPOSITHELPER);
        mintedAmount = _mintMLRT(address(eigenpiePreDepositHelper), asset, depositAmount);
        IEigenpiePreDepositHelper(eigenpiePreDepositHelper).feedUserDeposit(msg.sender, asset,
            mintedAmount);
    } else {
        // mint receipt
        mintedAmount = _mintMLRT(msg.sender, asset, depositAmount);
    }

    if (mintedAmount < minRec) {
        revert MinimumAmountToReceiveNotMet();
    }

    if (!isNative) {
        IERC20(asset).safeTransferFrom(msg.sender, address(this), depositAmount);
    }

    emit AssetDeposit(msg.sender, asset, depositAmount, referral, mintedAmount, isPreDeposit);
}
```

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.

```
receive() external payable nonReentrant {
    _forwardETH();
}
```

Listing 2.24: contracts/RewardDistributor.sol

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

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.

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

```
function bridgeMLRTToZircuit(
    address _mlrt,
    uint256 _amount
) external payable whenNotPaused onlyClientOrAllowedOperator nonReentrant {
    IMLRTAdapter mlrtAdapter = IMLRTAdapter(
        eigenpieConfig.getContract(EigenpieConstants.MLRT_ADAPTER)
    );

    MessagingFee memory fee = mlrtAdapter.getEstimateGasFees(
        EigenpieConstants.LZ_ZIRCUIT_DESTINATION_ID,
        0,
        _amount,
        _amount,
        mlrtWalletZircuit
    );

    // approve to lock MLRT in adapter and mint on destination chain
    IERC20(_mlrt).safeApprove(address(mlrtAdapter), _amount);
    mlrtAdapter.bridgeMLRT{value: fee.nativeFee}(
        EigenpieConstants.LZ_ZIRCUIT_DESTINATION_ID,
        0,
        _amount,
        _amount,
        mlrtWalletZircuit
    );
    emit BridgeMLRTToZircuit(client, msg.sender, _mlrt, _amount);
}
```

Listing 2.26: contracts/MLRTWallet.sol

```
function bridgeMLRT(
    uint32 _dstEid,
    uint128 _dstGasCost,
    uint256 _amountLD,
    uint256 _minAmountLD,
    address _receiver
) external payable nonReentrant{
    (uint256 amountSentLD, uint256 amountReceivedLD) = _debit(
        msg.sender,
        _amountLD,
        _minAmountLD,
        _dstEid
    );

    (
        bytes memory message,
        bytes memory options
    ) = _buildCustomMsgAndOptions(
            _dstEid,
            amountReceivedLD,
            _getDestinationGasCost(_dstGasCost),
            _receiver
        );

    MessagingReceipt memory msgReceipt = _lzSend(
        _dstEid,
        message,
        options,
        MessagingFee(msg.value, 0),
        payable(msg.sender)
    );

    emit BridgeMLRT(
        msg.sender,
        _dstEid,
        amountSentLD,
        amountReceivedLD,
        msgReceipt.guid
    );
}
```

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.

```
/// @notice This function was added to remove the error of missing implementations
function approvalRequired() external pure virtual returns (bool) {
    return false; // dummy implementation
}
```

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.

```
function _debit(
    address _from,
    uint256 _amountLD,
    uint256 /*_minAmountLD*/,
    uint32 /*_dstEid*/
) internal override whenNotPaused returns (uint, uint) {
    UtilLib.checkNonZeroAddress(_from);
    if (msg.sender != _from) revert InvalidSender();
    remoteMLRT.burnFrom(_from, _amountLD);
    return (_amountLD, _amountLD);
}
```

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.

```
function getEthBalance(
    IEigenPodManager eigenPodManager,
    uint256 stakedButNotVerifiedEth,
    address nodeDelegator,
    uint256 queuedETHShares
)
    public
    view
    returns (uint256)
{
    int256 podOwnerShares = eigenPodManager.podOwnerShares(nodeDelegator);
    if (podOwnerShares < 0) {
    // Ensure no underflow when stakedButNotVerifiedEth is 0 and podOwnerShares is negative
    uint256 absPodOwnerShares = uint256(-podOwnerShares);
    return stakedButNotVerifiedEth >= absPodOwnerShares
```

```
            ? queuedETHShares + stakedButNotVerifiedEth - absPodOwnerShares
            : 0;
    } else {
        return queuedETHShares + stakedButNotVerifiedEth + uint256(podOwnerShares);
    }
}
```

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]);
```

```
        UserWithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset];

        uint256 totalClaimedAmount;
        uint256 totalEgETHBurnAmount;
        uint256 claimedWithdrawalSchedulesPerAsset;
        uint256 nonce;

        for (uint256 j = 0; j < schedules.length;) {
            UserWithdrawalSchedule storage schedule = schedules[j];

            // if claimmable
            if (block.timestamp >= schedule.endTime && schedule.claimedAmt == 0) {
                claimedWithdrawalSchedulesPerAsset++;

                schedule.claimedAmt = schedule.queuedWithdrawLSTAmt;
                totalClaimedAmount += schedule.queuedWithdrawLSTAmt;
                totalEgETHBurnAmount += schedule.receiptMLRTAmt;
                nonce = schedule.nonce;
            } else if (block.timestamp >= schedule.endTime && schedule.claimedAmt == schedule.queuedWithdrawLSTAmt)
            {
                claimedWithdrawalSchedulesPerAsset++;
            }

            unchecked {
                ++j;
            }
        }

        claimedWithdrawalSchedules[i] = claimedWithdrawalSchedulesPerAsset;
        if(assets[i]!= EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
            if(totalClaimedAmount > IERC20(assets[i]).balanceOf(address(this))) {
            totalClaimedAmount = IERC20(assets[i]).balanceOf(address(this));
            }
        }
        if (totalClaimedAmount > 0) {
            if (assets[i] == EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
                bytes32 _withdrawHash = keccak256(abi.encode(nonce, msg.sender));
                // Revert if withdrawal is queued and not filled completely
                if (
                    withdrawQueued[_withdrawHash].queued
                        && withdrawQueued[_withdrawHash].fillAt > ethWithdrawQueue.queuedWithdrawFilled
                ) revert QueuedWithdrawalNotFilled();

                // reduce initial amountToRedeem from claim reserve
                ethClaimReserve -= totalClaimedAmount;

                // burn egETH locked for withdraw request
                address receipt = eigenpieConfig.mLRTReceiptByAsset(assets[i]);
                IMintableERC20(receipt).burnFrom(address(this), totalEgETHBurnAmount);
                TransferHelper.safeTransferETH(msg.sender, totalClaimedAmount);
            } else {
                IERC20(assets[i]).safeTransfer(msg.sender, totalClaimedAmount);
                emit AssetWithdrawn(msg.sender, assets[i], totalClaimedAmount);
            }
        }
        unchecked {
            ++i;
        }
    }

    _cleanUpWithdrawalSchedules(assets, claimedWithdrawalSchedules);
}
```

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.

```
function _checkAndFillETHWithdrawBuffer(uint256 _amount) internal {
    address eigenpieWithdrawManager = eigenpieConfig.getContract(EigenpieConstants.
         EIGENPIE_WITHDRAW_MANAGER);
    // Check the withdraw buffer and fill if below buffer target
    uint256 bufferToFill = IEigenpieWithdrawManager(eigenpieWithdrawManager).getWithdrawDeficit
         ();
    uint256 totalETHBal = currEthBalance + _amount;

    if (bufferToFill > 0) {
        bufferToFill = (totalETHBal <= bufferToFill) ? totalETHBal : bufferToFill;
        // fill withdraw buffer from received ETH
        IEigenpieWithdrawManager(eigenpieWithdrawManager).fillEthWithdrawBuffer{ value:
             bufferToFill }();
        emit BufferFilled(bufferToFill);
    }
    currEthBalance = (totalETHBal <= bufferToFill) ? 0 : totalETHBal - bufferToFill;
}

function _processETH() internal {
    address eigenStaking = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_STAKING);
```

```
        address delegationManagerAddr = eigenpieConfig.getContract(EigenpieConstants.
            EIGEN_DELEGATION_MANAGER);
        // If Eth from Eigenstaking or delegationManagerAddr, then should stay waiting;
        if (msg.sender == eigenStaking || msg.sender != delegationManagerAddr) {
            return;
        }

        uint256 gasRefunded = _refundGas();
        (uint256 ethShares, uint256 rewards) = _calRewardAmt(msg.value);

        // Forward remaining balance to rewarDistributor.
        // Any random eth transfer to this contract will also be treated as reward.
        address rewarDistributor = eigenpieConfig.getContract(EigenpieConstants.
            EIGENPIE_REWADR_DISTRIBUTOR);
        TransferHelper.safeTransferETH(rewarDistributor, rewards - gasRefunded);

        _checkAndFillETHWithdrawBuffer(ethShares);

        emit RewardsForwarded(rewarDistributor, msg.value);
    }

    function _calRewardAmt(uint256 _recievedAmt) internal pure returns (uint256 ethShares, uint256
         sendRewards) {
        sendRewards = _recievedAmt % 32 ether;
        ethShares = _recievedAmt - sendRewards;
    }
```

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.

```
function _processETH() internal {
    address eigenStaking = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_STAKING);
    address delegationManagerAddr = eigenpieConfig.getContract(EigenpieConstants.
        EIGEN_DELEGATION_MANAGER);
    // If Eth from Eigenstaking or delegationManagerAddr, then should stay waiting;
    if (msg.sender == eigenStaking || msg.sender != delegationManagerAddr) {
        return;
    }

    uint256 gasRefunded = _refundGas();
    (uint256 ethShares, uint256 rewards) = _calRewardAmt(msg.value);

    // Forward remaining balance to rewarDistributor.
    // Any random eth transfer to this contract will also be treated as reward.
    address rewarDistributor = eigenpieConfig.getContract(EigenpieConstants.
        EIGENPIE_REWADR_DISTRIBUTOR);
    TransferHelper.safeTransferETH(rewarDistributor, rewards - gasRefunded);

    _checkAndFillETHWithdrawBuffer(ethShares);

    emit RewardsForwarded(rewarDistributor, msg.value);
}
```

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.

2.2 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.

```
function _forwardETH() internal {
    uint256 balance = address(this).balance;
    if (balance == 0) {
        return;
    }

    uint256 length = rewardDests.length;

    for (uint256 i; i < length;) {
        RewardDestinations memory dest = rewardDests[i];
        uint256 toSendAmount = balance * dest.value / EigenpieConstants.DENOMINATOR;

        if (dest.needWrap) {
            // TODO will need to handle wrap as Weth
            // TODO will need to check if isAddress and queue reward to rewarder
        } else {
            TransferHelper.safeTransferETH(dest.to, toSendAmount);
        }

        unchecked {
            ++i;
        }
    }
}
```

Listing 2.34: contracts/RewardDistributor.sol

```
function addRewardDestination(
    uint256 _value,
    address _to,
    bool _isAddress,
    bool _needWrap
)
    external
    onlyDefaultAdmin
{
    if (_value > EigenpieConstants.DENOMINATOR) revert InvalidFeePercentage();
    UtilLib.checkNonZeroAddress(_to);

    rewardDests.push(RewardDestinations({ value: _value, to: _to, isAddress: _isAddress,
        needWrap: _needWrap }));
    emit RewardDestinationAdded(rewardDests.length - 1, _value, _to, _isAddress, _needWrap);
}

function setRewardDestination(
    uint256 _index,
    uint256 _value,
    address _to,
    bool _isAddress,
    bool _needWrap
)
    external
    onlyDefaultAdmin
{
    if (_index >= rewardDests.length) revert InvalidIndex();
    if (_value > EigenpieConstants.DENOMINATOR) revert InvalidFeePercentage();
    UtilLib.checkNonZeroAddress(_to);

    RewardDestinations storage dest = rewardDests[_index];
    dest.value = _value;
    dest.to = _to;
    dest.isAddress = _isAddress;
    dest.needWrap = _needWrap;
    emit RewardDestinationUpdated(_index, _value, _to, _isAddress, _needWrap);
}

function removeRewardDestination(uint256 _index) external onlyDefaultAdmin {
    if (_index >= rewardDests.length) revert InvalidIndex();

    for (uint256 i = _index; i < rewardDests.length - 1; i++) {
        rewardDests[i] = rewardDests[i + 1];
    }
    rewardDests.pop();
    emit RewardDestinationRemoved(_index);
}
```

Listing 2.35: 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.

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

Listing 2.36: contracts/MLRTWallet.sol

```
function restakedLess(
    address client,
    address underlyingToken
)
    external
    view
    override
    returns (uint256 underlyingLessAmount, uint256 mlrtShouldBurn)
{
    return _checkCollateralLess(client, underlyingToken);
}
```

Listing 2.37: 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.

```
function forceUnLock(
    uint256 _slotIndex
) external whenNotPaused nonReentrant {
    _checkIdexInBoundary(msg.sender, _slotIndex);
    UserUnlocking storage slot = userUnlockings[msg.sender][_slotIndex];

    // Check if the slot is already unlocked (amountInCoolDown == 0) and revert if so
    if (slot.amountInCoolDown == 0) {
        revert UnlockedAlready();
    }

    uint256 penaltyAmount = 0;
    uint256 amountToUser = slot.amountInCoolDown; // Default to the full amount

    _claimFromRewarder(msg.sender);

    // If the current time is not beyond the slot's endTime, then there's penalty.
    if (block.timestamp < slot.endTime) {
        _checkInCoolDown(msg.sender, _slotIndex);

        (penaltyAmount, amountToUser) = expectedPenaltyAmount(_slotIndex);
    }

    _unlock(slot.amountInCoolDown);

    IERC20(Eigenpie).safeTransfer(msg.sender, amountToUser);
    totalPenalty += penaltyAmount;

    slot.amountInCoolDown = 0;
    slot.endTime = block.timestamp;

    emit ForceUnLock(msg.sender, _slotIndex, amountToUser, penaltyAmount);
}
```

Listing 2.38: contracts/vlEigenpie.sol

```
function setEigenPod(address client) external {
    ClientData storage clientData = allowedClients[client];
    UtilLib.checkNonZeroAddress(clientData.mlrtWallet);

    _setEigenPod(client, clientData);
}
```

Listing 2.39: contracts/EigenpieEnterprise.sol

```
function _setEigenPod(address client, ClientData storage clientData) internal returns (address eigenPod) {
    eigenPod = _fetchEigenPod(client);

    if (eigenPod != address(0) && clientData.eigenPod == address(0)) {
        registeredPod[eigenPod] = true;
        clientData.eigenPod = eigenPod;
        _updateMLRTWalletEigenPod(clientData, eigenPod);
        emit EigenPodSet(client, eigenPod);
    }
}
```

Listing 2.40: contracts/EigenpieEnterprise.sol

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

Listing 2.41: contracts/EigenpieEnterprise.sol

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

Listing 2.42: contracts/MLRTWallet.sol

```
address public eigenPod;
```

Listing 2.43: 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.

```
function getFullyUnlock(
    address _user
) public view override returns (uint256 unlockedAmount) {
    uint256 length = getUserUnlockSlotLength(_user);
    for (uint256 i; i < length; i++) {
        if (
            userUnlockings[_user][i].amountInCoolDown > 0 &&
            block.timestamp > userUnlockings[_user][i].endTime
        ) unlockedAmount += userUnlockings[_user][i].amountInCoolDown;
    }
}
```

Listing 2.44: 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.

```
function addNodeDelegatorContractToQueue(address[] calldata nodeDelegatorContracts) external
     onlyDefaultAdmin {
    uint256 length = nodeDelegatorContracts.length;
    if (nodeDelegatorQueue.length + length > maxNodeDelegatorLimit) {
        revert MaximumNodeDelegatorLimitReached();
    }

    for (uint256 i; i < length;) {
        UtilLib.checkNonZeroAddress(nodeDelegatorContracts[i]);

        // check if node delegator contract is already added and add it if not
        if (isNodeDelegator[nodeDelegatorContracts[i]] == 0) {
            nodeDelegatorQueue.push(nodeDelegatorContracts[i]);
            isNodeDelegator[nodeDelegatorContracts[i]] = 1;
        }

        unchecked {
            ++i;
        }
    }

    emit NodeDelegatorAddedinQueue(nodeDelegatorContracts);
}
```

Listing 2.45: 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.

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

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

Listing 2.46: 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.

```
function makeBeaconDeposit(
    bytes[] memory publicKeys,
    bytes[] memory signatures,
    bytes32[] memory depositDataRoots,
    IEigenpieConfig eigenpieConfig,
    address eigenPod
)
    external
{
    // sanity checks
    uint256 count = depositDataRoots.length;
    if (count == 0) revert INodeDelegator.AtLeastOneValidator();
    if (count >= EigenpieConstants.MAX_VALIDATORS) {
        revert INodeDelegator.MaxValidatorsInput();
    }
    if (publicKeys.length != count) {
        revert INodeDelegator.PublicKeyNotMatch();
    }
    if (signatures.length != count) {
        revert INodeDelegator.SignaturesNotMatch();
    }
}
```

Listing 2.47: 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.

```
receive() external payable {
    address eigenStaking = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_STAKING);
    // If Eth from Eigenstaking, then should stay waiting to be restaked;
    if (msg.sender == eigenStaking) {
        return;
    }

    uint256 gasRefunded;
    address dwr = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_DWR);
    // If Eth from dwr, then is partial withdraw of CL reward
    if (msg.sender == dwr && adminGasSpentInWei[tx.origin] > 0) {
        gasRefunded = _refundGas();

        // If no funds left, return
        if (msg.value == gasRefunded) {
            return;
        }
    }
    // Forward remaining balance to rewarDistributor.
    // Any random eth transfer to this contract will also be treated as reward.
    address rewarDistributor = eigenpieConfig.getContract(EigenpieConstants.
         EIGENPIE_REWADR_DISTRIBUTOR);
    TransferHelper.safeTransferETH(rewarDistributor, msg.value - gasRefunded);

    emit RewardsForwarded(rewarDistributor, msg.value);
}
```

Listing 2.48: 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`.

```
event BridgeMLRT(
    address indexed from,
    uint32 indexed dstEid,
    uint256 amountSent,
    uint256 amountReceived,
    bytes32 indexed guid
);
```

Listing 2.49: contracts/MLRTOFTBridge.sol

```
emit BridgeMLRT(
    _refundAddress,
    _dstEid,
    amountSentLD,
    amountReceivedLD,
    msgReceipt.guid
);
```

Listing 2.50: 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.

```
uint256 availableToWithdraw = getAvailableToWithdraw();
bool queued = false;
uint256 nonce;

if (asset == EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
    if (withdrawLSTAmt > availableToWithdraw) {
        withdrawRequestNonce++;
        // increase the claim reserve to partially fill withdrawRequest with max available in buffer
        ethClaimReserve += availableToWithdraw;
        // fill the queue with availableToWithdraw
        ethWithdrawQueue.queuedWithdrawFilled += availableToWithdraw;
        // update the queue to fill
        ethWithdrawQueue.queuedWithdrawToFill += withdrawLSTAmt;
        // calculate withdrawRequest hash
        bytes32 withdrawHash = keccak256(abi.encode(withdrawRequestNonce, msg.sender));

        withdrawQueued[withdrawHash].queued = true;
        withdrawQueued[withdrawHash].fillAt = ethWithdrawQueue.queuedWithdrawToFill;
        queued = true;

        nonce = withdrawRequestNonce;
    } else {
        // add redeem amount to claimReserve of claim asset
        ethClaimReserve += withdrawLSTAmt;
    }
} else {
    WithdrawalSum storage withdrawalSum = withdrawalSums[assetToEpoch];
    withdrawalSum.assetTotalToWithdrawAmt += withdrawLSTAmt;
    withdrawalSum.mLRTTotalToBurn += mLRTamount;
}
```

Listing 2.51: 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`.

```
function _checkAndFillETHWithdrawBuffer(uint256 _amount) internal {
    address eigenpieWithdrawManager = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_WITHDRAW_MANAGER);
    // Check the withdraw buffer and fill if below buffer target
    uint256 bufferToFill = IEigenpieWithdrawManager(eigenpieWithdrawManager).getWithdrawDeficit();
    uint256 totalETHBal = currEthBalance + _amount;

    if (bufferToFill > 0) {
        bufferToFill = (totalETHBal <= bufferToFill) ? totalETHBal : bufferToFill;
        // fill withdraw buffer from received ETH
        IEigenpieWithdrawManager(eigenpieWithdrawManager).fillEthWithdrawBuffer{ value:
            bufferToFill }();
        emit BufferFilled(bufferToFill);
    }
    currEthBalance = (totalETHBal <= bufferToFill) ? 0 : totalETHBal - bufferToFill;
}
```

Listing 2.52: contracts/NodeDelegator.sol

Suggestion Refactor the corresponding logic.

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.

```
receive() external payable nonReentrant {
    _forwardETH();
}

// TODO, will have to handle ERC20 if reward in LST form
function forwardRewards() external payable nonReentrant whenNotPaused onlyEigenpieManager {
        _forwardETH();
    }
```

Listing 2.53: 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