AstraSec

Eigenpie

Security Audit Report

September 19, 2024

Contents

1 | Introduction

1.1 About Eigenpie

`Eigenpie` is a re-staking platform for `SubDAO`, providing Liquid Stake Token (`LST`) holders with the ability to re-stake their assets and maximize their profit potential. It achieves this by creating dedicated liquidity restaking for each accepted `LST` on its platform, effectively isolating risks associated with any particular `LST`.

1.2 Audit Scope

	LINK	Base Commit	Final Commit
1	https://github.com/magpiexyz/eigenpie.git	297d1ba	72227d5
2	https://github.com/../withdrawStakedETH	eda5ddb	57d825e
3	https://github.com/magpiexyz/eigenpie/pull/14	263ef2f	9c905b1
4	https://github.com/magpiexyz/eigenpie/pull/15	c3e361a	85c14e5
5	https://github.com/magpiexyz/eigenpie/pull/15	85c14e5	48889f5
6	https://github.com/magpiexyz/eigenpie/pull/19	7210994	f2bd5d0
7	https://github.com/magpiexyz/eigenpie/pull/23	55f6347	c7fd9f6
8	https://github.com/magpiexyz/eigenpie/pull/31	cf8380c	1afbd98
9	https://github.com/magpiexyz/eigenpie/pull/34	2f0e46e	dd57100
10	https://github.com/magpiexyz/eigenpie/pull/35	44b7453	-
11	https://github.com/magpiexyz/eigenpie/pull/43	294fb83	6aec262
12	https://github.com/magpiexyz/eigenpie/pull/47	ff44118	0d8a121
13	https://github.com/magpiexyz/eigenpie/pull/59	fec6782	-
14	https://github.com/magpiexyz/eigenpie/pull/61	1e567f8	0b9aed2
15	https://github.com/magpiexyz/eigenpie/pull/70	5329ee9	402da7a

1.3 Changelog

Version	Date
First Audit	February 14, 2024
Second Audit	March 23, 2024
Third Audit	April 8, 2024
Forth Audit	April 12, 2024
Fifth Audit	April 19, 2024
Sixth Audit	May 3, 2024
Seventh Audit	May 23, 2024
Eighth Audit	May 25, 2024
Nineth Audit	June 12, 2024
Tenth Audit	June 18, 2024
Eleventh Audit	July 6, 2024
Twelfth Audit	July 18, 2024
Thirteenth Audit	September 3, 2024
Fourteenth Audit	September 10, 2024
Fifteenth Audit	September 17, 2024

2 | Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the `Eigenpie` project. Throughout this audit, we identified several issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	-	-	-	-
High	9	-	-	9
Medium	3	1	-	2
Low	2	-	-	2
Informational	-	-	-	-
Undetermined	-	-	-	-

3 | Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Description
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3   Vulnerability Details

[H-1] Revised Pre-Deposit Logic in EigenpieStaking::depositAsset()

Target	Category	IMPACT	LIKELIHOOD	STATUS
EigenpieStaking.sol EigenpiePreDepositHelper.sol	Business Logic	High	High	🔗Addressed

The `EigenpieStaking::depositAsset()` function serves as a mechanism for users to deposit supported `LST` (e.g., `ankrETH`, `cbETH`, etc.) and the corresponding `mLRT-LST` token is minted. During the pre-deposit phase of the protocol, users deposit underlying token into the `EigenpieStaking` contract (line 177). However, the corresponding `mLRT-LST` token is not immediately minted for them (lines 166 - 168). Instead, they need to wait until the current pre-deposit cycle is concluded. Upon claiming (line 84), the `mLRT-LST` token will then be minted and allocated to the users (line 93). This may result in the `totalSupply` of the `mLRT-LST` token is not updated in time, which is crucial for calculating the `mLRT-LST/LST` exchange rate. Consequently, it will lead to inaccuracy in the exchange rate calculation.

EigenpieStaking::depositAsset()

```
145  function depositAsset(
146      address asset,
147      uint256 depositAmount,
148      uint256 minRec,
149      address referral
150  )
151      external
152      whenNotPaused
153      nonReentrant
154      onlySupportedAsset(asset)
155  {
156      // checks
157      if (depositAmount == 0  depositAmount < minAmountToDeposit) {
158          revert InvalidAmountToDeposit();
159      }

161      if (depositAmount > getAssetCurrentLimit(asset)) {
162          revert MaximumDepositLimitReached();
163      }

165      uint256 mintedAmount;
166      if (isPreDeposit) {
167          (mintedAmount,) = getMLRTAmountToMint(asset, depositAmount);
168          IEigenpiePreDepositHelper(eigenpiePreDepositHelper).feedUserDeposit(msg.
                 sender, asset, mintedAmount);
169      } else {
```

```
170          // mint receipt
171          mintedAmount = _mintMLRT(asset, depositAmount);
172      }
173      if (mintedAmount < minRec) {
174          revert MinimumAmountToReceiveNotMet();
175      }

177      IERC20(asset).safeTransferFrom(msg.sender, address(this), depositAmount);

179      emit AssetDeposit(msg.sender, asset, depositAmount, referral);
180  }
```

EigenpiePreDepositHelper::userClaim()

```
84  function userClaim(uint256[] calldata _cycles, address[] calldata _assets)
        external nonReentrant {
85      for (uint256 i = 0; i < _cycles.length; i++) {
86          if (!claimmableCycles[_cycles[i]]) revert ClaimCycleNotStarted();
87          for (uint256 j = 0; j < _assets.length; j++) {
88              bytes32 cycleUserKey = this._getCycleUserKey(_cycles[i], msg.sender)
                    ;
89              UserInfo storage user = userInfo[cycleUserKey][_assets[j]];
90              uint256 amount = user.amount - user.claimed;
91              if (amount > 0) {
92                  address receipt = eigenpieConfig.mLRTReceiptByAsset(_assets[j]);
93                  IMintableERC20(receipt).mint(msg.sender, amount);
94                  user.claimed += amount;
95                  emit Claim(msg.sender, _assets[j], amount, _cycles[i]);
96              }
97          }
98      }
99  }
```

Remediation   Ensure the `totalSupply` of the `mLRT-LST` token is updated in time.

[H-2] Improper exchangeRate Precision in PriceProvider::updateMLRTPrice(address)

Target	Category	IMPACT	LIKELIHOOD	STATUS
PriceProvider.sol	Business Logic	High	High	🔗Addressed

The `PriceProvider::updateMLRTPrice(address)` function is utilized to update the `mLRT-LST/LST` exchange rate for the specified asset. The exchange rate is derived from the current state of the corresponding pool. During our examination of the exchange rate calculation logic, it is apparent

that there is a loss of precision for the result. Given this, we suggest to improve its implementation as below: `uint256 exchangeRate = totalLST * 1 ether / receiptSupply` (line 69).

Moreover, to mitigate potential front-run attacks, we recommend adding access control to this function and execute transactions for updating the exchange rate through private RPC (e.g., `flashbot`).

PriceProvider::updateMLRTPrice(address)

```
54  /// @notice updates mLRT-LST/LST exchange rate
55  /// @dev calculates based on stakedAsset value received from eigen layer
56  /// @param asset the asset for which exchange rate to update
57  function updateMLRTPrice(address asset) external {
58      address mLRTReceipt = eigenpieConfig.mLRTReceiptByAsset(asset);
59      uint256 receiptSupply = IMLRT(mLRTReceipt).totalSupply();

61      if (receiptSupply == 0) {
62          IMLRT(mLRTReceipt).updateExchangeRateToLST(1 ether);
63          return;
64      }

66      address eigenStakingAddr = eigenpieConfig.getContract(EigenpieConstants.
            EIGENPIE_STAKING);
67      uint256 totalLST = IEigenpieStaking(eigenStakingAddr).getTotalAssetDeposits(
            asset);

69      uint256 exchangeRate = totalLST / receiptSupply;

71      _checkNewRate(mLRTReceipt, exchangeRate);

73      IMLRT(mLRTReceipt).updateExchangeRateToLST(exchangeRate);
74  }
```

Remediation Correct the implementation of the `PriceProvider::updateMLRTPrice(address)` function as above mentioned.

[H-3] Improper Implementation of PriceProvider::updateMLRTPrice(address, uint256)

Target	Category	IMPACT	LIKELIHOOD	STATUS
PriceProvider.sol	Business Logic	High	High	🔗Addressed

As part of its intended functionality, the `PriceProvider::updateMLRTPrice(address, uint256)` function is employed by the privileged account to manually adjust the exchange rate based on off-chain calculations, thereby optimizing gas usage. However, thorough examination of its implementation,

we observed that it lacks any form of access control and does not actually modify the exchange rate, which clearly deviates from the intended design.

PriceProvider::updateMLRTPrice(address, uint256)

```
76  /// @notice updates mLRT-LST/LST exchange rate manually for gas fee saving
77  /// @dev calculates based on stakedAsset value received from eigen layer
78  /// @param asset the asset for which exchange rate to update
79  /// @param newExchangeRate the new exchange rate to update
80  function updateMLRTPrice(address asset, uint256 newExchangeRate) external {
81      address mLRTReceipt = eigenpieConfig.mLRTReceiptByAsset(asset);

83      _checkNewRate(mLRTReceipt, newExchangeRate);

85      emit ExchangeRateUpdate(asset, mLRTReceipt, newExchangeRate);
86  }
```

Remediation   Apply necessary access control and properly update the exchange rate.

[H-4] Revised CleanUp of Withdrawal Schedules

Target	Category	IMPACT	LIKELIHOOD	STATUS
EigenpieWithdrawManager.sol	Business Logic	High	High	🔗Addressed

The EigenpieWithdrawManager contract exclusively manages the queuing and withdrawal processes of tokens, specifically LSTs (Liquid Staking Tokens). Users queue for the withdrawals of specific assets and proceed with the withdrawal after the withdrawal period expires. Then the withdrawals are removed from the schedules list.

While reviewing the withdraw logic, we notice that the claimed withdrawals may not be correctly removed from the schedules list. There are two key points of concern. Firstly, the `userWithdrawAsset()` function does not accurately count the total number of claimed withdrawals that can be removed (i.e., `claimedWithdrawalSchedulesPerAsset`), but only counts the number of new claimed withdrawals in the current transaction (line 157). This leads to the `_cleanUpWithdrawalSchedules()` function being unable to correctly determine whether to proceed with the removal based on the threshold check (`withdrawalscheduleCleanUp`).

EigenpieWithdrawManager::userWithdrawAsset()

```
139  function userWithdrawAsset(address[] memory assets) external nonReentrant {
140    uint256[] memory claimedWithdrawalSchedules = new uint256[](assets.length);

142    for (uint256 i = 0; i < assets.length; i++) {
```

```
143        bytes32 userToAsset = userToAssetKey(msg.sender, assets[i]);
144        WithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset];

146        uint256 totalClaimedAmount;
147        uint256 burnAmount;
148        uint256 claimedWithdrawalSchedulesPerAsset;

150        for (uint256 j = 0; j < schedules.length; j++) {
151            WithdrawalSchedule storage schedule = schedules[j];
152            if (block.timestamp >= schedule.endTime) {
153                uint256 availableToClaim = schedule.queuedWithdrawLSTAmt;

155                if (availableToClaim >= schedule.claimedAmt) {
156                    ...
157                    claimedWithdrawalSchedulesPerAsset++;
158                }
159            }
160        }

162        claimedWithdrawalSchedules[i] = claimedWithdrawalSchedulesPerAsset;

164        if (totalClaimedAmount > 0) {...}
165    }

167    _cleanUpWithdrawalSchedules(assets, claimedWithdrawalSchedules);
168 }
```

Secondly, a redundant for-loop (line 272) is added in `_cleanUpWithdrawalSchedules()` which brings the possibility of claimed withdrawals being removed from the schedules list repeatedly.

EigenpieWithdrawManager::_cleanUpWithdrawalSchedules()

```
262  function _cleanUpWithdrawalSchedules(
263     address[] memory assets,
264     uint256[] memory clamiedWithdrawalSchedules
265  )
266      internal
267  {
268      for (uint256 i = 0; i < assets.length; i++) {
269          bytes32 userToAsset = userToAssetKey(msg.sender, assets[i]);
270          WithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset
                ];

272          for (uint256 j = 0; j < clamiedWithdrawalSchedules.length; j++) {
273              if (clamiedWithdrawalSchedules[j] > 0 && clamiedWithdrawalSchedules[
                    j] >= withdrawalscheduleCleanUp) {
274                  for (uint256 k = 0; k < schedules.length -
                        clamiedWithdrawalSchedules[j]; k++) {
275                      schedules[k] = schedules[k + clamiedWithdrawalSchedules[j]];
```

```
276                    }

278                    while (clamiedWithdrawalSchedules[j] > 0) {
279                        schedules.pop();
280                        clamiedWithdrawalSchedules[j]--;
281                    }
282                }
283            }
284        }
285 }
```

Remediation   Revisit the above mentioned functions to ensure all claimed schedules are accurately counted and properly removed from the schedules list.

[H-5] Reserve ETH for New Validators in NodeDelegator::receive()

Target	Category	IMPACT	LIKELIHOOD	STATUS
NodeDelegator.sol	Business Logic	High	High	🔗Addressed

The NodeDelegator contract accepts the deposit of native token from the EigenpieStaking contract and initializes validators by depositing specified data onto the Beacon Chain. With the validators, they can participate in Native Restaking in EigenLayer.

However, while reviewing the implementation of the `receive()` routine, we notice that all the received native tokens are transferred to the rewarDistributor as rewards (line 48). As a result, there is no native token left for the NodeDelegator to setup validators.

NodeDelegator::receive()

```
40  receive() external payable {
41      // If a payment comes in from the delayed withdrawal router, assume it is
              from the pending unstaked withdrawal
42      // and subtract that amount from the pending amount
43      if (msg.sender == address(eigenPod.delayedWithdrawalRouter())) {
44          ...
45      }

47      address rewarDistributor = eigenpieConfig.getContract(EigenpieConstants.
              EIGENPIE_REWADR_DISTRIBUTOR);
48      TransferHelper.safeTransferETH(rewarDistributor, msg.value);

50      emit RewardsForwarded(rewarDistributor, msg.value);
51  }
```

Remediation   Properly reserve the received staking of native token within the contract to initialize validators.

[H-6] Flawed Exchange Rate Calculation

Target	Category	IMPACT	LIKELIHOOD	STATUS
EigenpieStaking.sol EigenpieWithdrawManager.sol	Business Logic	High	Medium	🔗Addressed

Eigenpie calculates the exchange rate of mLRT by dividing the total amount of deposited LSTs by the `totalSupply`. The total amount of LSTs is obtained by calling `EigenpieStaking::getTotalAssetDeposits()`, which sums up the LSTs in the EigenpieStaking contract, NDCs, and Eigen-Layer but does not account for the LSTs in the EigenpieWithdrawManager contract. In particular, if the EigenpieWithdrawManager contract holds LSTs for users withdrawals, updating the exchange rate before the user's mLRT is burned would result in an incorrect value.

EigenpieStaking::getAssetDistributionData

```
102  function getAssetDistributionData(address asset)
103     public
104     view
105     override
106     onlySupportedAsset(asset)
107     returns (uint256 assetLyingInDepositPool, uint256 assetLyingInNDCs, uint256
            assetStakedInEigenLayer)
108  {
109     assetLyingInDepositPool = TransferHelper.balanceOf(asset, address(this));

111     uint256 ndcsCount = nodeDelegatorQueue.length;
112     for (uint256 i; i < ndcsCount;) {
113         assetLyingInNDCs += TransferHelper.balanceOf(asset, nodeDelegatorQueue[i])
                ;
114         assetStakedInEigenLayer += INodeDelegator(nodeDelegatorQueue[i]).
                getAssetBalance(asset);
115         unchecked {
116             ++i;
117         }
118     }
119  }
```

Remediation   It's recommended to include the amount of LSTs held in the EigenpieWithdraw-Manager contract into the calculation within `EigenpieStaking::getTotalAssetDeposits()`.

[H-7] Incorrect engienpieEnterprise Initialization in MLRTWallet

Target	Category	IMPACT	LIKELIHOOD	STATUS
MLRTWallet.sol	Business Logic	High	High	🔗Addressed

To facilitate the evolution of the `MLRTWallet` contract, it is designed to be beacon-upgradeable. The `MLRTWallet` contract has an `engienpieEnterprise` state variable that references the `EigenpieEnterprise` contract. During our code review, we found the `engienpieEnterprise` state cannot be properly initialized to a valid value in the proxy space because it can only be set in the constructor of `MLRTWallet`.

In the following, we provide a code snippet demonstrating how `engienpieEnterprise` is initialized in the constructor of the `MLRTWallet` contract. Note that the `engienpieEnterprise` is defined as a regular state variable, not `immutable`, which limits its initialization to the `MLRTWallet` contract space only. However, there's no way to initialize it correctly within the proxy space.

As a result, the `engienpieEnterprise` variable lacks a valid value, which disrupts normal functionality.

<div align="center">MLRTWallet.sol</div>

```solidity
22  IEigenpieEnterprise public engienpieEnterprise;

24  constructor(address eigenpieConfigAddr, address engienpieEnterpriseAddr) {
25      UtilLib.checkNonZeroAddress(eigenpieConfigAddr);

27      eigenpieConfig = IEigenpieConfig(eigenpieConfigAddr);
28      engienpieEnterprise = IEigenpieEnterprise(engienpieEnterpriseAddr);

30      _disableInitializers();
31  }
```

Remediation   Update the `engienpieEnterprise` variable to be immutable, or provide a mechanism to initialize it from the proxy.

[H-8] Revisited Logic of EigenpieEnterprise::burnMLRT()

Target	Category	IMPACT	LIKELIHOOD	STATUS
EigenpieEnterprise.sol	Business Logic	High	High	🔗Addressed

The `burnMLRT()` function is designed to burn the `amountToBurn` specified amount of MLRT token. It checks that `amountToBurn` does not exceed the maximum burnable `MLRT` token (line 131), burns the specified `MLRT` token, and updates the `nativeUsed` (or `lstUsed`) variable accordingly (line 133). Upon thorough examination, we notice it incorrectly updates the `nativeUsed` (or `lstUsed`) variable by using

the collateral amount corresponding to the maximum burnable `MLRT` token instead of the `amountToBurn` parameter. This allows a malicious actor to manipulate the `nativeUsed` (or `lstUsed`) variable to zero, enabling them to re-mint more `MLRT` token than permitted.

EigenpieEnterprise::burnMLRT()

```
125  function burnMLRT(address client, address mlrtAsset, uint256 amountToBurn)
          external nonReentrant {
126      address asset = IMLRT(mlrtAsset).underlyingAsset();

128      (uint256 valuedAssetLess, uint256 shouldBurn) = _checkCollateralLess(client,
              asset);

130      if (valuedAssetLess == 0) revert EnoughCollateral();
131      if (amountToBurn > shouldBurn) revert BurnTooMuch();

133      _burnFromWallet(client, asset, valuedAssetLess, amountToBurn);

135      emit BurnMLRTFromWallet(client, asset, valuedAssetLess, amountToBurn);
136  }

138  function _burnFromWallet(
139      address client,
140      address asset,
141      uint256 lessAmount,
142      uint256 shouldBurn
143  ) internal {
144      address receipt = eigenpieConfig.mLRTReceiptByAsset(asset);
145      ClientData storage clientData = allowedClients[client];
146      IMLRT(receipt).burnFrom(clientData.mlrtWallet, shouldBurn);

148      if (asset == EigenpieConstants.PLATFORM_TOKEN_ADDRESS) {
149          if (clientData.mlrtMinted > shouldBurn)
150              clientData.mlrtMinted -= shouldBurn;
151          else
152              clientData.mlrtMinted = 0;

154          if (clientData.nativeUsed > lessAmount)
155              clientData.nativeUsed -= lessAmount;
156          else
157              clientData.nativeUsed = 0;
158      } else {
159          LSTData storage lstData = clientAssetMapping[client][asset];

161          if (lstData.mlrtMinted > shouldBurn)
162              lstData.mlrtMinted -= shouldBurn;
163          else
164              lstData.mlrtMinted = 0;

166          if (lstData.lstUsed > lessAmount)
```

```
167            lstData.lstUsed -= lessAmount;
168        else
169            lstData.lstUsed = 0;
170     }

172     if (totalMintedMlrt[receipt] > shouldBurn)
173        totalMintedMlrt[receipt] -= shouldBurn;
174     else
175        totalMintedMlrt[receipt] = 0;
176 }
```

Remediation   Adjust the function to correctly calculate and update the `nativeUsed` and `lstUsed` variables based on the `amountToBurn` parameter to prevent unauthorized manipulation.

[H-9] Incorrect Operation in ValidatorLib::verifyWithdrawalCredentials()

Target	Category	IMPACT	LIKELIHOOD	STATUS
ValidatorLib.sol	Business Logic	High	High	🔗Addressed

As the Node Delegator contract continues to expand, the contract size increases correspondingly. To address this, the `Magpie` development team utilizes libraries to modularize the reusable code, thereby reducing the overall contract size and optimizing performance. When examining the `verifyWithdrawalCredentials()` function in the `ValidatorLib` contract, we notice that the current implementation returns an incorrect value, and the execution of this function will revert. Specifically, the current implementation decrements the `stakedButNotVerifiedEth` value when it should increment it (line 126). Moreover, this will cause the function to revert because the subtraction operation on `stakedButNotVerifiedEth` will underflow.

ValidatorLib::verifyWithdrawalCredentials()

```
106  function verifyWithdrawalCredentials(
107      IEigenPod eigenPod,
108      uint64 oracleTimestamp,
109      BeaconChainProofs.StateRootProof calldata stateRootProof,
110      uint40[] calldata validatorIndices,
111      bytes[] calldata withdrawalCredentialProofs,
112      bytes32[][] calldata validatorFields
113  ) external returns (uint256 stakedButNotVerifiedEth) {
114      eigenPod.verifyWithdrawalCredentials(
115          oracleTimestamp,
116          stateRootProof,
117          validatorIndices,
118          withdrawalCredentialProofs,
```

```
119          validatorFields
120      );

122      // Decrement the staked but not verified ETH
123      for (uint256 i = 0; i < validatorFields.length; ) {
124          uint64 validatorCurrentBalanceGwei = BeaconChainProofs
125              .getEffectiveBalanceGwei(validatorFields[i]);
126          stakedButNotVerifiedEth -= (validatorCurrentBalanceGwei *
127              EigenpieConstants.GWEI_TO_WEI);

129          unchecked {
130              ++i;
131          }
132      }
133  }
```

Remediation   Correct the operation to increment `stakedButNotVerifiedEth`.

[M-1] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	Medium	Medium	Acknowledged

In the `Eigenpie` protocol, the existence of a series of privileged accounts introduces centralization risks, as they hold significant control and authority over critical operations governing the protocol. In the following, we show the representative function potentially affected by the privileges associated with the privileged accounts.

MLRT::mint()/burnFrom()

```
67  /// @notice Mints EGETH when called by an authorized caller
68  /// @param to the account to mint to
69  /// @param amount the amount of EGETH to mint
70  function mint(address to, uint256 amount) external onlyRole(EigenpieConstants.
        MINTER_ROLE) whenNotPaused {
71      _mint(to, amount);
72  }

74  /// @notice Burns EGETH when called by an authorized caller
75  /// @param account the account to burn from
76  /// @param amount the amount of EGETH to burn
77  function burnFrom(address account, uint256 amount) external onlyRole(
        EigenpieConstants.BURNER_ROLE) whenNotPaused {
78      _burn(account, amount);
79  }
```

Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged accounts. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been confirmed by the team. The multi-sig mechanism will be used to mitigate this issue.

[M-2] Revised assetTotalWithdrawAmt Check

Target	Category	IMPACT	LIKELIHOOD	STATUS
EigenpieWithdrawManager.sol	Coding Practices	Medium	Medium	🔗Addressed

The `EigenpieWithdrawManager::completeAssetWithdrawalFromEigenLayer()` function is responsible for completing asset withdrawals initiated from the EigenLayer. It aims to complete all withdrawals for the given assets in the specified epoch. Therefore, it checks if the total amount of assets to be withdrawn in that epoch (i.e., `assetTotalWithdrawAmt`) can be fulfilled by the current input parameters. However, we notice that the current implementation incorrectly checks the `assetTotalWithdrawAmt`. Specifically, the current implementation requires withdrawing the `assetTotalWithdrawAmt` amount of assets from each NDC (line 228), while the expectation is to withdraw the `assetTotalWithdrawAmt` amount of assets from all NDCs combined.

EigenpieWithdrawManager::withdrawAssetsFromEigenLayer()

```
222  for (uint256 i = 0; i < nodeDelegators.length; i++) {
223     if (nodeToAssets[i].length != nodeToAmount[i].length) {
224         revert LengthMistmatch();
225     }
226     for (uint256 j = 0; j < nodeToAssets[i].length; j++) {
227         bytes32 assetToEpoch = assetEpochKey(nodeToAssets[i][j], epochTime);
228         if (assetTotalWithdrawAmt[assetToEpoch] != nodeToAmount[i][j]) revert
                InvalidWithdrawAmt();
229     }
230  }
```

Remediation Revisit the implementation of `completeAssetWithdrawalFromEigenLayer()` function to improve the check for `assetTotalWithdrawAmt`.

[M-3] Revisited eigenpieEnterprise Configuration in PriceProvider

Target	Category	IMPACT	LIKELIHOOD	STATUS
PriceProvider.sol	Business Logic	High	High	🔗Addressed

The `PriceProvider` contract is designed as an upgradable contract. In this upgrade, a new `eigenpieEnterprise` variable (line 27) is added to store the address of the `EigenpieEnterprise` contract. However, upon thorough examination, we observe there is no interface provided to configure this `eigenpieEnterprise` variable. This oversight means that the `eigenpieEnterprise` address cannot be set or updated after deployment, rendering the new variable effectively unusable.

MLRTWallet.sol

```
20  contract PriceProvider is IPriceProvider, EigenpieConfigRoleChecker,
        Initializable {
21      uint256 public rateIncreaseLimit; // as a protection
22      uint256 public rateChangeWindowLimit; // as a protection

24      mapping(address asset => address priceOracle) public override
            assetPriceOracle;
25      mapping(address receipt => uint256 timestamp) public rateLastUpdate;

27      IEigenpieEnterprise public eigenpieEnterprise;

29      ...
30  }
```

Remediation Implement a setter function to configure the `eigenpieEnterprise` variable.

[L-1] Integration of Non-Standard ERC20 Tokens

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Business Logic	Low	Low	🔗Addressed

Inside the `EigenpieStaking::depositAsset()` function, the statement of `if (!IERC20(asset).transferFrom (msg.sender, address(this), depositAmount)) {revert TokenTransferFailed();}` (line 69) is employed to transfer the user's asset into the `EigenpieStaking` contract. However, in the case of USDT-like token whose transferFrom() lacks a return value, it would lead to a revert. Given this, we recommend employing the widely-used `SafeERC20` library (which serves as a wrapper for ERC20 operations while accommodating a diverse range of non-standard ERC20 tokens) to address this case.

EigenpieStaking::depositAsset()

```
128  function depositAsset(
129      address asset,
130      uint256 depositAmount,
131      uint256 minRec,
132      address referral
```

```
133  )
134      external
135      whenNotPaused
136      nonReentrant
137      onlySupportedAsset(asset)
138  {
139      // checks
140      if (depositAmount == 0  depositAmount < minAmountToDeposit) {
141          revert InvalidAmountToDeposit();
142      }

144      if (depositAmount > getAssetCurrentLimit(asset)) {
145          revert MaximumDepositLimitReached();
146      }

148      if (!IERC20(asset).transferFrom(msg.sender, address(this), depositAmount)) {
149          revert TokenTransferFailed();
150      }

152      // mint receipt
153      uint256 mintedAmount = _mintMLRT(asset, depositAmount);
154      if (mintedAmount < minRec) {
155          revert MinimumAmountToReceiveNotMet();
156      }

158      emit AssetDeposit(msg.sender, asset, depositAmount, referral);
159  }
```

Remediation   Replace `transfer()`/`transferFrom()` with `safeTransfer()`/`safeTransferFrom()`.

[L-2] Improved Logic of MLRTWallet::withdrawFromSwellStaking()

Target	Category	IMPACT	LIKELIHOOD	STATUS
MLRTWallet.sol	Business Logic	Low	High	🔗Addressed

The `withdrawFromSwellStaking()` function is used to withdraw the staked `MLRT` tokens from the `swellSimpleStaking` contract. Typically, only privileged accounts of the wallet can perform this withdrawal operation. However, if the user's collateral is insufficient, anyone can withdraw the excess `MLRT` tokens and burn them. While examining its implementation, we notice that if a malicious actor withdraws the excess `MLRT` tokens without burning them, he can repeatedly withdraw `MLRT` tokens until all tokens are exhausted, which clearly undermines the original design of the protocol.

MLRTWallet::withdrawFromSwellStaking()

```
96   function withdrawFromSwellStaking (address mlrt, uint256 amount) external
         nonReentrant {
97       _checkValidWithdrawCondition(msg.sender, amount);

99       ISimpleStakingERC20 swellSimpleStaking = ISimpleStakingERC20(eigenpieConfig.
             getContract(EigenpieConstants.SWELL_SIMPLE_STAKING));
100      swellSimpleStaking.withdraw(IERC20(mlrt), amount, address(this));

102      emit WithdrawFromSwellStaking(msg.sender, mlrt, amount);
103  }

105  function _checkValidWithdrawCondition(address caller, uint256 amountToWithdraw)
         internal {
106      bool isClient = caller == client;
107      bool isClientOperator = allowedClientOperators[caller];
108      bool isManager = IAccessControl(address(eigenpieConfig)).hasRole(
             EigenpieConstants.MANAGER, caller);
109      // if client or eigenpie manager calling, then all good
110      if (isClient  isClientOperator  isManager) return;
111
112      (, uint256 mlrtLess) = eigenpieEnterprise.nativeRestakedLess(client);
113      if (amountToWithdraw > mlrtLess) revert PublicWithdrarTooMuch();
114  }
```

Remediation   Improve the implementation of the `withdrawFromSwellStaking()`/`withdrawFromZicruit` `()` functions to prevent above-mentioned attack.

4 | Appendix

4.1 About AstraSec

`AstraSec` is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, `AstraSec` maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. `AstraSec`'s comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

4.3   Contact

Name	AstraSec Team
Phone	+86 176 2267 4194
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI