AstraSec

Eigenpie

Security Audit Report

November 27, 2024

Contents

1 Introduction

1.1 About Eigenpie

Eigenpie is a restaking 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 Source Code

The following source code was reviewed during the audit:

- https://github.com/magpiexyz/eigenpie.git
- CommitID: d5b52b2

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/magpiexyz/eigenpie.git
- CommitID: 135db1d

1.3 Changelog

Version	Date	Description
v1.0	October 13, 2024	Initial Audit
v1.1	November 16, 2024	PR106 (35c5579)

2 Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the Eigenpie protocol. Throughout this audit, we identified a total of 8 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	1	–	–	1
Medium	3	1	–	2
Low	4	1	–	3
Informational	–	–	–	–
Undetermined	–	–	–	–

3 Vulnerability Summary

3.1 Overview

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

- ~~H-1~~ Possible Storage Conflict in EigenpieWithdrawManager
- ~~M-1~~ Revisited ETH Balance Calculation in getETHBalance()
- ~~M-2~~ Unexpected Revert in EigenpieEnterprise::registerReStaking()
- M-3 Potential Risks Associated with Centralization
- ~~L-1~~ Revisited Logic of NodeDelegator::_recordGas()
- ~~L-2~~ Incompatibility with Non-Standard ERC20 Tokens
- L-3 Potential User Asset Loss in EigenpieWithdrawManager::userWithdrawAsset()
- ~~L-4~~ Timely Burn of Excess MLRT in MLRTWallet::bridgeMLRTToZircuit()

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	Acknowledged
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

3.3.1 [H-1] Possible Storage Conflict in EigenpieWithdrawManager

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
EigenpieWithdrawManager.sol	Business Logic	High	High	Addressed

The EigenpieWithdrawManager contract is upgradable. To support ETH withdrawals, a new variable, nonce, was added to the UserWithdrawalSchedule struct. However, introducing a new variable to an existing struct in an upgradable contract creates a storage conflict, as the updated storage layout no longer aligns with the layout from previous versions. This storage slot conflict represents a serious issue for contract upgradeability and may cause undefined behavior in production.

eigenpie - EigenpieWithdrawManager.sol

```
contract EigenpieWithdrawManager is
    IEigenpieWithdrawManager,
    EigenpieConfigRoleChecker,
    PausableUpgradeable,
    ReentrancyGuardUpgradeable
{
    using SafeERC20 for IERC20;

    uint256 public lstWithdrawalDelay; // a buffer time period making sure user able to withdraw the LST unstake by
        // Eigenpie
    uint256 public startTimestamp; // the start timestamp counting epoch
    uint256 public constant EPOCH_DURATION = 7 days;
    uint256 public withdrawalscheduleCleanUp; // the threshold to clean up withdra queue length

    mapping(bytes32 => UserWithdrawalSchedule[]) public withdrawalSchedules; //bytes32 = user + asset
    mapping(bytes32 => WithdrawalSum) public withdrawalSums; // aggregated withdrawal information // bytes32 = asset + epoch
    ...
}
```

Remediation Avoid introducing storage conflict during the upgrade process.

3.3.2 [M-1] Revisited ETH Balance Calculation in getETHBalance()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
AssetManagementLib.sol	Business Logic	High	Low	Addressed

The getEthBalance() function is designed to calculate the total ETH balance held by the NodeDelegator. However, upon review, we identify a logical flaw in the implementation. Specifically, when stakedButNotVerifiedEth is less than absPodOwnerShares (lines 89–91), the function returns 0 without accounting for queuedETHShares. This omission leads to an inaccurate ETH balance calculation, effectively violating the intended design and potentially disrupting dependent functionalities.

eigenpie-feat-egETH-1121 - AssetManagementLib.sol

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

Remediation Refactor the getEthBalance() function to always factor in queuedETHShares when calculating the ETH balance, ensuring that all relevant components are accounted for.

3.3.3 [M-2] Unexpected Revert in EigenpieEnterprise::registerReStaking()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
EigenpieEnterprise.sol	Business Logic	Medium	Medium	Addressed

The registerReStaking() function in the EigenpieEnterprise contract is at risk of reverting because both it and the burnMLRT() (line 169) function it calls are protected by the nonReentrant modifier. Since nonReentrant prevents reentrancy within the same call chain, attempting to call burnMLRT() from within registerReStaking() triggers a revert. This design issue causes registerReStaking() to fail due to the conflicting nonReentrant protections in both functions.

eigenpie-main - EigenpieEnterprise.sol

```
155 function registerReStaking(
156     address underlyingToken,
157     uint256 amountToMintMlt
158 )
159     external
160     nonReentrant
161     onlyAllowedClient
162 {
163     ...
164     // burn excess token minted by client which are no longer restaked
165     address[] memory supportedAssetList = eigenpieConfig.getSupportedAssetList();
166     for(uint256 i=0; i < supportedAssetList.length; i++){
167         (, uint256 mlrtShouldBurn) = restakedLess(msg.sender, supportedAssetList[i]);
168         if(mlrtShouldBurn > 0){
169             burnMLRT(msg.sender, eigenpieConfig.mLRTReceiptByAsset(supportedAssetList[i]), mlrtShouldBurn);
170         }
171     }
172     ...
173 }
```

eigenpie-main - EigenpieEnterprise.sol

```
126 function burnMLRT(address client, address mlrtAsset, uint256 amountToBurn) public nonReentrant {
127     ...
128 }
```

Remediation Refactor an internal function that implements the burn MLRT functionality for the registerReStaking() function to call.

3.3.4 [M-3] Potential Risks Associated with Centralization

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	High	Low	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.

eigenpie-main - PriceProvider.sol

```
// /// @notice updates mLRT-LST/LST exchange rate
// /// @dev calculates based on stakedAsset value received from eigen layer
// /// @param asset the asset for which exchange rate to update
function updateMLRTPrice(address asset) external onlyOracle {
    uint256 exchangeRate = _calculateExchangeRate(asset);
    _checkNewRate(asset, exchangeRate);

    _updateMLRTPrice(asset, exchangeRate);
}

/// @notice updates mLRT-LST/LST exchange rate manually for gas fee saving
/// @dev calculates based on stakedAsset value received from eigen layer
/// @param asset the asset for which exchange rate to update
/// @param newExchangeRate the new exchange rate to update
function updateMLRTPrice(address asset, uint256 newExchangeRate) external onlyOracleAdmin {
    _checkNewRate(asset, newExchangeRate);

    _updateMLRTPrice(asset, newExchangeRate);
}
```

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.

3.3.5 [L-1] Revisited Logic of NodeDelegator::_recordGas()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
NodeDelegator.sol	Business Logic	Low	Low	Addressed

The NodeDelegator contract has an issue in the _recordGas() and _refundGas() functions, where gas refunds may not be handled correctly if the privileged bot account is a contract instead of an EOA. This is due to the use of msg.sender (line 414) in _recordGas() and tx.origin (line 421) in _refundGas() . If a contract bot calls these functions, it will not receive the gas refund as intended, potentially causing unexpected behavior and inefficient gas usage.

eigenpie-main - NodeDelegator.sol

```
411 function _recordGas(uint256 initialGas) internal {
412     uint256 gasSpent = AssetManagementLib.calculateGasSpent(initialGas, eigenpieConfig, tx.gasprice);
413
414     adminGasSpentInWei[msg.sender] += gasSpent;
415     emit GasSpent(msg.sender, gasSpent);
416 }
417
418 function _refundGas() internal returns (uint256) {
419     uint256 gasRefund = AssetManagementLib.calculateAndTransferRefundGas(tx.origin, adminGasSpentInWei[tx.origin]);
420     // reset gas spent by admin
421     adminGasSpentInWei[tx.origin] -= gasRefund;
422
423     emit GasRefunded(tx.origin, gasRefund);
424     return gasRefund;
425 }
```

Remediation Ensure consistent handling of gas refunds, especially for contract callers.

3.3.6 [L-2] Incompatibility with Non-Standard ERC20 Tokens

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
NodeDelegator.sol	Business Logic	Low	Low	Addressed

Although there is a standard ERC-20 specification, many token contracts may not strictly follow this specification or may have additional functionalities beyond it. The following is an example using the maxApproveToEigenStrategyManager() function in the NodeDelegator contract.

This function's purpose is to approve the maximum amount (type(uint256).max) of a specified asset to the Eigenlayer strategy manager. If the specified asset is USDT, there will be a compatibility issue with non-standard ERC20 tokens when this function is executed. This is due to the use of safeApprove() without resetting the current allowance to zero before setting it to type(uint256).max. Since USDT does not allow increasing an allowance from a non-zero value, this can cause the function to fail when the initial allowance for Eigenlayer strategy manager is not zero.

eigenpie-main - NodeDelegator.sol

```
/// @notice Approves the maximum amount of an asset to the eigen strategy manager
/// @dev only supported assets can be deposited and only called by the Eigenpie manager
/// @param asset the asset to deposit
function maxApproveToEigenStrategyManager(address asset)
    external
    override
    onlySupportedAsset(asset)
    onlyEigenpieManager
{
    address eigenlayerStrategyManagerAddress = eigenpieConfig.getContract(EigenpieConstants.EIGEN_STRATEGY_MANAGER);
    IERC20(asset).safeApprove(eigenlayerStrategyManagerAddress, type(uint256).max);
}
```

Remediation To improve compatibility, the maxApproveToEigenStrategyManager() function should first reset the allowance to zero before setting it to the desired value.

3.3.7 [L-3] Potential User Asset Loss in userWithdrawAsset()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
EigenpieWithdrawManager.sol	Business Logic	Low	Low	Acknowledged

By design, when a user wants to withdraw previously deposited LST, they first need to queue their request using userQueuingForWithdraw(). At the end of the epoch, a privileged bot gathers all withdrawal requests and processes a bulk withdrawal from Eigenlayer. Once this is done, users can claim their assets via userWithdrawAsset().

Upon reviewing userWithdrawAsset(), we observe that if the contract lacks sufficient assets during a user's claim, the user only receives the available balance, potentially resulting in asset loss (lines 183–185). This issue arises if the bot fails to promptly withdraw assets from Eigenlayer, leaving the contract underfunded. We recommend enhancing the function to prevent users from claiming if the contract lacks sufficient funds, thereby avoiding potential losses.

eigenpie-main - EigenpieWithdrawManager.sol

```
178 function userWithdrawAsset(address[] memory assets) external nonReentrant {
179     ...
180     for (uint256 i = 0; i < assets.length;) {
181         ...
182         claimedWithdrawalSchedules[i] = claimedWithdrawalSchedulesPerAsset;
183         if(totalClaimedAmount > IERC20(assets[i]).balanceOf(address(this))) {
184             totalClaimedAmount = IERC20(assets[i]).balanceOf(address(this));
185         }
186
187         if (totalClaimedAmount > 0) {
188             IERC20(assets[i]).safeTransfer(msg.sender, totalClaimedAmount);
189             emit AssetWithdrawn(msg.sender, assets[i], totalClaimedAmount);
190         }
191
192         unchecked {
193             ++i;
194         }
195     }
196
197     _cleanUpWithdrawalSchedules(assets, claimedWithdrawalSchedules);
198 }
```

Remediation Improve the function to prevent users from claiming if the contract lacks sufficient funds

3.3.8 [L-4] Timely Burn of Excess MLRT bridgeMLRTToZircuit()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
MLRTWallet.sol	Business Logic	Low	Low	Addressed

The bridgeMLRTToZircuit() function facilitates the cross-chain transfer of MLRT tokens from Ethereum to Zircuit. However, it lacks a check to verify if the client's MLRT holdings exceed their required collateral before initiating the bridge. Without this check, the excess MLRT can be bridged unnecessarily, potentially leading to an imbalance in the client's collateral. Given this, we recommend burning any excess MLRT before initiating the bridge, ensuring that only the necessary MLRT amount is transferred to Zircuit.

Moreover, the depositToZicruit() and depositToSwellStaking() functions share the similar issue.

eigenpie-main - MLRTWallet.sol

```
function bridgeMLRTToZircuit(
    address _mlrt,
    uint256 _amount,
    uint256 _minAmount
) external payable whenNotPaused onlyClientOrAllowedOperator nonReentrant {
    ...
    // Approve the adapter to lock the mLRT token and bridge it
    IERC20(_mlrt).safeApprove(address(mlrtAdapter), _amount);
    mlrtAdapter.bridgeMLRT{value: fee.nativeFee}(
        EigenpieConstants.LZ_ZIRCUIT_DESTINATION_ID,
        0,
        _amount,
        _minAmount,
        mlrtWalletZircuit,
        msg.sender
    );

    emit BridgeMLRTToZircuit(client, msg.sender, _mlrt, _amount);
}
```

Remediation Improve the implementation of these functions as above-mentioned.

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

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI