BlockSec

Security Audit Report for Eigenpie Contracts

Date: Jan 27, 2024

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2024.01.28 13:15:10 +08'00'

Contents

Report Manifest

Item	Description
Client	Magpie
Target	Eigenpie Contracts

Version History

Version	Date	Description
1.0	Jan 27, 2024	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. 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

The target of this audit is the code repository [1] for the Eigenpie Contracts. Eigenpie is a sophisticated SubDAO crafted by Magpie [2], designed to offer Liquid Restaking Services through the framework of EigenLayer.

The auditing process is iterative. Specifically, we will 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. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
Eigenpie Contracts	Version 1	ea28ff1d62c134e95d13bc064ecb71995a394033
	Version 2	ba43688f44f0a5498d690817bf503f9e1efd314e
	Version 3	7ca7638a2da93ab828b0cf91d1ab592612dda148

1.2 Disclaimer

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

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

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

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

[2] https://www.magpiexyz.io/

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
* Access control
* 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 [3] and Common Weakness Enumeration [4]. 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

Impact \ Likelihood	*High*	*Low*
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:

- **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.

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

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

Chapter 2 Findings

In total, we find **two** potential issues. Besides, we also have **three** notes.

- Low Risk: 2
- Note: 3

ID	Severity	Description	Category	Status
1	Low	Potential failure to handle transfers of non-compliant ERC-20 tokens	Software Security	Fixed
2	Low	Potential griefing attack by donating to node delegator	Software Security	Acknowledged
4	-	Potential centralization risk	Note	-
5	-	Potential front-running risk	Note	-
6	-	Non-withdrawable shares are a feature by design	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential failure to handle transfers of non-compliant ERC-20 tokens

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the current implementation, the `EigenpieStaking` and the `NodeDelegator` contracts directly call the `transfer` and `transferFrom` functions of ERC-20 tokens. However, if a token does not strictly follow the ERC-20 standard (for example, the USDT token), the check on the return value may fail. It is recommended to use the `SafeERC20` library provided by OpenZeppelin.

```
function transferBackToEigenpieStaking(
    address asset,
    uint256 amount
)
    external
    whenNotPaused
    nonReentrant
    onlySupportedAsset(asset)
    onlyLRTManager
{
    address eigenpieStaking = eigenpieConfig.getContract(EigenpieConstants.EIGENPIE_STAKING);

    if (!IERC20(asset).transfer(eigenpieStaking, amount)) {
        revert TokenTransferFailed();
    }
}
```

Listing 2.1: NodeDelegator.sol

```
function transferAssetToNodeDelegator(
    uint256 ndcIndex,
    address asset,
    uint256 amount
)
    external
    nonReentrant
    onlyLRTManager
    onlySupportedAsset(asset)
{
    address nodeDelegator = nodeDelegatorQueue[ndcIndex];
    if (!IERC20(asset).transfer(nodeDelegator, amount)) {
        revert TokenTransferFailed();
    }
}
```

Listing 2.2: EigenpieStaking.sol

Impact Token transfers may fail due to tokens that do not strictly follow the ERC-20 standard.

Suggestion Use the `SafeERC20` library provided by OpenZeppelin.

2.1.2 Potential griefing attack by donating to node delegator

Severity Low

Status Acknowledged

Introduced by `Version 2`

Description In the `EigenpieStaking` contract, the amount of minted shares is determined by the deposit amount and the current exchange rate of the `receipt` token.

```
function getMLRTAmountToMint(
    address asset,
    uint256 amount
)
    public
    view
    returns (uint256 mLRTAmountToMint, address mLRTReceipt)
{
    address receipt = eigenpieConfig.mLRTReceiptByAsset(asset);

    uint256 rate = IMLRT(receipt).exchangeRateToLST();

    return (amount * 1 ether / rate, receipt);
}
```

Listing 2.3: EigenpieStaking.sol

However, this exchange rate is derived from the ratio of total deposits to the total supply of the `receipt` token within the `EigenpieStaking` contract.

```
function updateMLRTPrice(address asset) external {
    address mLRTReceipt = eigenpieConfig.mLRTReceiptByAsset(asset);
```

```
        uint256 receiptSupply = IMLRT(mLRTReceipt).totalSupply();

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

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

        uint256 exchangeRate = totalLST / receiptSupply;

        _checkNewRate(mLRTReceipt, exchangeRate);

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

Listing 2.4: PriceProvider.sol

```
    function getTotalAssetDeposits(address asset) public view override returns (uint256
        totalAssetDeposit) {
        (uint256 assetLyingInDepositPool, uint256 assetLyingInNDCs, uint256 assetStakedInEigenLayer
            ) =
            getAssetDistributionData(asset);
        return (assetLyingInDepositPool + assetLyingInNDCs + assetStakedInEigenLayer);
    }
```

Listing 2.5: EigenpieStaking.sol

```
    function getAssetDistributionData(address asset)
        public
        view
        override
        onlySupportedAsset(asset)
        returns (uint256 assetLyingInDepositPool, uint256 assetLyingInNDCs, uint256
            assetStakedInEigenLayer)
    {
        assetLyingInDepositPool = IERC20(asset).balanceOf(address(this));

        uint256 ndcsCount = nodeDelegatorQueue.length;
        for (uint256 i; i < ndcsCount;) {
            assetLyingInNDCs += IERC20(asset).balanceOf(nodeDelegatorQueue[i]);
            assetStakedInEigenLayer += INodeDelegator(nodeDelegatorQueue[i]).getAssetBalance(asset)
                ;
            unchecked {
                ++i;
            }
        }
    }
```

Listing 2.6: EigenpieStaking.sol

Consequently, a malicious user could potentially donate tokens to manipulate the exchange rate and carry out a *griefing attack*. In a worst-case scenario, users might be required to deposit an excessive number of tokens to receive a mere 1 wei of shares. This could severely impair the contract's functionality and result in substantial losses for users.

Impact The design of the current share minting logic is subject to griefing attacks.

Suggestion Revise the share minting logic.

Feedback from the Project This issue can be avoided if the team seed some initial liquidity.

2.2 Notes

2.2.1 Potential centralization risk

Introduced by Version 1

Description There are some potential centralization risks in the project. Specifically, the manager of the EigenpieStaking contract is able to withdraw arbitrary assets and manage funds in permitted node delegators. Additionally, the node delegator array can be arbitrarily configured by the manager.

```
function transferAssetToNodeDelegator(
    uint256 ndcIndex,
    address asset,
    uint256 amount
)
    external
    nonReentrant
    onlyLRTManager
    onlySupportedAsset(asset)
{
    address nodeDelegator = nodeDelegatorQueue[ndcIndex];
    if (!IERC20(asset).transfer(nodeDelegator, amount)) {
        revert TokenTransferFailed();
    }
}
```

Listing 2.7: EigenpieStaking.sol

```
function depositAssetIntoStrategy(address asset)
    external
    override
    whenNotPaused
    nonReentrant
    onlySupportedAsset(asset)
    onlyLRTManager
{
    address strategy = eigenpieConfig.assetStrategy(asset);
    if (strategy == address(0)) {
        revert StrategyIsNotSetForAsset();
    }

    IERC20 token = IERC20(asset);
    address eigenlayerStrategyManagerAddress = eigenpieConfig.getContract(EigenpieConstants.
        EIGEN_STRATEGY_MANAGER);

```

```
        uint256 balance = token.balanceOf(address(this));

        emit AssetDepositIntoStrategy(asset, strategy, balance);

        IEigenStrategyManager(eigenlayerStrategyManagerAddress).depositIntoStrategy(IStrategy(
            strategy), token, balance);
    }
```

Listing 2.8: NodeDelegator.sol

Feedback from the Project All administration roles and owners for all the contracts will be governed by Multisig wallets.

2.2.2 Potential front-running risk

Introduced by Version 2

Description The exchangeRateToLST variable of a receipt token is updated periodically, which introduces the risk of front-running. If the exchange rates are updated to front-run user deposits, users are at risk of receiving fewer receipt tokens, potentially resulting in losses.

```
    function updateExchangeRateToLST(uint256 _newRate) external onlyPriceProvider {
        exchangeRateToLST = _newRate;

        emit LSTExchangeRateUpdated(msg.sender, _newRate);
     }
```

Listing 2.9: EigenpieStaking.sol

Feedback from the Project The team will use private rpc to avoid this kind of risk. Besides, the minRec parameter of the depositAsset function mitigates the problem.

2.2.3 Non-withdrawable shares are a feature by design

Introduced by Version 1

Description The EigenpieStaking contract mints shares for user deposits. The current implementation has no withdrawal mechanism to redeem the shares. This is an expected behavior.