SMART CONTRACT AUDIT REPORT

for

Eigenpie Protocol

Prepared By: Xiaomi Huang

PeckShield

October 3, 2024

Document Properties

Client	Eigenpie
Title	Smart Contract Audit Report
Target	Eigenpie
Version	1.0.1
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0.1	October 3, 2024	Xuxian Jiang	Post-Final Release #1
1.0	September 30, 2024	Xuxian Jiang	Final Release

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the design document and related source code of the `Eigenpie` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts could potentially be improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Eigenpie

The `Eigenpie` protocol aims to build a `Liquid Restaking` solution for public blockchain networks. Initially inspired from `Kelp DAO`, `Eigenpie` does not mint a single `rsETH` all for supported `LSTs`. Instead, it has isolated `LRTReceiptToken` minted `1:1` for different deposited `LST` The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Eigenpie

Item	Description
Name	Eigenpie
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 3, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/magpiexyz/eigenpie.git (ed5a8dd)

And this is the commit ID after all fixes for the issues found in the audit have been addressed:

- https://github.com/magpiexyz/eigenpie.git (84f022a, fc860a8f)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Eigenpie` protocol smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	2	■■
Informational	0	
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others may involve unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Reward Updated in vlEigenpie::cancelUnlock()	Business Logic	Resolved
PVE-002	Low	Simplified balanceOf()/totalLocked() Logic in vlEigenpie	Coding Practices	Resolved
PVE-003	Low	Improved add/setRewardDestination() Logic in RewardDistributor	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improper Reward Updated in vlEigenpie::cancelUnlock()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `vlEigenpie`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Eigenpie` protocol has a core `vlEigenpie` contract that is used to vote-lock `Eigenpie` tokens. The vote-locked amount is used to calculate the rewards a user may deserve. In the process of examining existing logic to calculate the reward amount, we notice an issue that results from the cancellation of a cool down entry.

In the following, we show below the related `cancelUnlock()` implementation. It has a rather straightforward logic in validating the user input and properly reducing cool down amount being cancelled. However, the user rewards should be calculated before making any adjustment on the cool down amount being cancelled. The reason is that the `rewarder.updateFor()` (line 306) relies on the user vote-locked amount for reward calculation, which should not be affected by the cool down amount being cancelled.

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

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

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

Listing 3.1: `vlEigenpie::cancelUnlock()`

Recommendation Improve the above logic to properly compute the user rewards once a lock in cool down is being cancelled.

Status The issue has been fixed by this commit: `84f022a`.

3.2 Simplified balanceOf()/totalLocked() Logic in vlEigenpie

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `vlEigenpie`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.1, `Eigenpie` has a core `vlEigenpie` contract that is used to vote lock `Eigenpie` tokens. In the process of examining current logic, we notice certain accounting-related routines may be simplified.

To elaborate, we show below an example routine, i.e., `balanceOf()`. This routine is used to count total `Eigenpie` amount held in the contracts, including the locked ones as well as the ones in cool down. With that, we can simplify its function body as `return userDeposits[_user];`.

Moreover, the `totalLocked()` routine is used to query the total amount being locked, excluding the ones in cool down. With that, we can simplify the function body as `return totalSupply()-totalAmountInCoolDown();`.

```
    function balanceOf(address _user) public view override returns (uint256) {
        return getUserTotalLocked(_user) + getUserAmountInCoolDown(_user);
    }

    // total Eigenpie locked, excluding the ones in cool down
    function totalLocked() public view override returns (uint256) {
        return this.totalSupply() - this.totalAmountInCoolDown();
    }

    /// @notice Get the total Eigenpie a user locked, not counting the ones in cool down
    /// @param _user the user
    /// @return _lockAmount the total Eigenpie a user locked, not counting the ones in
        cool down
```

```
    function getUserTotalLocked(
        address _user
    ) public view override returns (uint256 _lockAmount) {
        _lockAmount = userDeposits[_user] - getUserAmountInCoolDown(_user);
    }
```

Listing 3.2: `vlEigenpie::balanceOf()/totalLocked()/getUserTotalLocked()`

Recommendation Revise the above-mentioned routines by simplifying redundant calls or calculations.

Status The issue has been fixed by this commit: `15cc60a`.

3.3 Improved add/setRewardDestination() Logic in RewardDistributor

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardDistributor`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Eigenpie` protocol is no exception. Specifically, if we examine the `RewardDistributor` contract, it has defined a number of protocol-wide risk parameters, such as `reward destinations` and their percentages for reward sharing. In the following, we show the corresponding routines that allow for their changes.

```
    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);
    }
```

Listing 3.3: RewardDistributor::addRewardDestination() and RewardDistributor::setRewardDestination()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, the above routines can be improved by ensuring the sum of all `RewardDest`'s value is no larger than `DENOMINATOR`, not each one. The same issue also affects the same functions in another contract named `WLNodedelegator`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by this commit: `15cc60a`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The Eigenpie protocol has a privileged account (with the role of DEFAULT_ADMIN_ROLE) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure protocol-wide risk parameters and whitelist tokens). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function addNewSupportedAsset(
        address asset,
        address mLRTReceipt,
        uint256 depositLimit
    )
        external
        onlyRole(EigenpieConstants.MANAGER)
    {
        _addNewSupportedAsset(asset, mLRTReceipt, depositLimit);
    }
    ...
    function updateReceiptToken(
        address asset,
        address mLRTReceipt
    )
        external
        onlyRole(EigenpieConstants.DEFAULT_ADMIN_ROLE)
    {
        if (!isSupportedAsset[asset]) {
            revert AssetNotSupported();
        }

        if (asset != IMLRT(mLRTReceipt).underlyingAsset()) revert InvalidAsset();
        mLRTReceiptByAsset[asset] = mLRTReceipt;

        emit ReceiptTokenUpdated(asset, mLRTReceipt);
    }
    ...
    function updateAssetDepositLimit(
        address asset,
        uint256 depositLimit
    )
```

```
        external
        onlyRole(EigenpieConstants.MANAGER)
        onlySupportedAsset(asset)
    {
        depositLimitByAsset[asset] = depositLimit;
        emit AssetDepositLimitUpdate(asset, depositLimit);
    }
    ...
    function updateAssetStrategy(
        address asset,
        address strategy
    )
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
        onlySupportedAsset(asset)
    {
        UtilLib.checkNonZeroAddress(strategy);
        if (assetStrategy[asset] == strategy) {
            revert ValueAlreadyInUse();
        }

        if (
            asset != EigenpieConstants.PLATFORM_TOKEN_ADDRESS && asset != address(
                IStrategy(strategy).underlyingToken())
        ) revert InvalidAsset();

        assetStrategy[asset] = strategy;
        emit AssetStrategyUpdate(asset, strategy);
    }
    ...
    function updateAssetBoost(
        address asset,
        uint256 boost
    )
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
        onlySupportedAsset(asset)
    {
        boostByAsset[asset] = boost;

        emit AssetBoostUpdate(asset, boost);
    }
```

Listing 3.4: Example Privileged Operations in `EigenpieConfig`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated with the use of a multisig as the admin.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Eigenpie` protocol, which aims to build a `Liquid Restaking` solution for public blockchain networks. Initially inspired from `Kelp DAO`, `Eigenpie` does not mint a single `rsETH` all for supported `LSTs`. Instead, it has isolated `LRTReceiptToken` minted `1:1` for different deposited `LST` The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.