SMART CONTRACT AUDIT REPORT

for

Eigenpie Protocol

Prepared By: Xiaomi Huang

PeckShield

May 2, 2024

Document Properties

Client	Eigenpie
Title	Smart Contract Audit Report
Target	Eigenpie
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 2, 2024	Xuxian Jiang	Final Release
1.0-rc	April 23, 2024	Xuxian Jiang	Release Candidate #1

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	May 2, 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 (49102c9)

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 (0e499b1)

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

Likelihood

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	1	■
Informational	1	■
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, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Withdrawal Schedule Cleanup in EigenpieWithdrawManager	Business Logic	Resolved
PVE-002	Low	Simplified Logic in EigenpieWithdraw-Manager	Coding Practices	Resolved
PVE-003	Undetermined	Improved Caller Validation in NodeDel-egator	Security Features	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 Incorrect Withdrawal Schedule Cleanup in EigenpieWithdrawManager

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

Description

The `Eigenpie` protocol builds a `Liquid Restaking` solution that offers liquidity to illiquid assets deposited into restaking platforms. While examining existing logic to withdraw previous stakes, we notice it has an incorrect implementation when clearing withdrawal schedule.

In the following, we show below the related `_cleanUpWithdrawalSchedules()` implementation. It has a rather straightforward logic in having two `for`-loops to iterate current withdrawal schedules and clear previous ones. The first `for`-loop iterates the given asset list and the second `for`-loop evaluates previous withdrawal schedules. However, it comes to our attention that the adjustment of previous schedules makes use of the wrong index `claimedWithdrawalSchedules[j]` (line 313), which should be `claimedWithdrawalSchedules[i]`.

```
301     function _cleanUpWithdrawalSchedules(
302         address[] memory assets,
303         uint256[] memory claimedWithdrawalSchedules
304     )
305         internal
306     {
307         for (uint256 i = 0; i < assets.length;) {
308             bytes32 userToAsset = userToAssetKey(msg.sender, assets[i]);
309             UserWithdrawalSchedule[] storage schedules = withdrawalSchedules[userToAsset
                    ];
310
311             if (claimedWithdrawalSchedules[i] >= withdrawalscheduleCleanUp) {
```

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

                    unchecked {++j;}
                }

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

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

Listing 3.1: `EigenpieWithdrawManager::_cleanUpWithdrawalSchedules()`

Recommendation Improve the staking logic to ensure the given `minRec` restriction is honored.

Status The issue has been fixed by this commit: `0e499b1`.

3.2 Simplified Logic in EigenpieWithdrawManager

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

Description

The `Eigenpie` protocol has a core `EigenpieWithdrawManager` contract that is in charge of processing pool withdraws. In the process of examining current logic, we notice certain routines can be simplified.

To elaborate, we show below an example routine, i.e., `nextUserWithdrawalTime()`, from the `nextUserWithdrawalTime` contract. This routine is used to get timestamp to unstake LST from the protocol if a new makes a queue withdraw request now. However, we notice that it calls `this.currentEpoch()` (line 73) to query the next epoch number. This inter-contract call can be revised to be an internal call `currentEpoch()`. Apparently, we can redefine it from being `external` to `public`.

```
72     function nextUserWithdrawalTime() external view returns (uint256) {
73         return startTimestamp + (this.currentEpoch() + 1) * EPOCH_DURATION +
               lstWithdrawalDelay;
74     }
75
```

```
    // to get current epoch number
    function currentEpoch() external view returns (uint256) {
        return (block.timestamp - startTimestamp) / EPOCH_DURATION + 1;
    }
```

Listing 3.2: `EigenpieWithdrawManager::nextUserWithdrawalTime()`

Note another routine `userQueuingForWithdraw()` shares the same issue.

Recommendation Revise the above-mentioned routines by replacing cross-contract self calls with internal function calls.

Status The issue has been fixed by this commit: `0e499b1`.

3.3 Improved Caller Validation in NodeDelegator

- ID: PVE-003
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `NodeDelegator`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The `Eigenpie` protocol also features a core `NodeDelegator` contract to handle the depositing of user assets into strategies. In the process of examining the interaction with `EigenLayer`, we notice a public function is not guarded to validate the caller.

To elaborate, we show below the implementation of this function, i.e., `completeAssetWithdrawalFromEigenLayer()`. This function is supposed to be a return call from `EigenLayer` to complete user withdrawal requests. However, it currently does not validate the caller, which may open doors to unwanted invocation with arbitrary arguments. To avoid the unintended calls with `EigenLayer`, we suggest to validate the caller in this public function.

```
    function completeAssetWithdrawalFromEigenLayer(
        IDelegationManager.Withdrawal calldata withdrawal,
        IERC20[] calldata tokens,
        uint256 middlewareTimesIndex,
        bool receiveAsTokens
    )
        external
        whenNotPaused
        nonReentrant
    {
        uint256[] memory beforeAmounts = new uint256[](tokens.length);
```

```
        for (uint256 i = 0; i < tokens.length;) {
            beforeAmounts[i] = tokens[i].balanceOf(address(this));

            unchecked {++i;}
        }

        address delegationManagerAddr = eigenpieConfig.getContract(EigenpieConstants.
            EIGEN_DELEGATION_MANAGER);
        IDelegationManager(delegationManagerAddr).completeQueuedWithdrawal(
            withdrawal, tokens, middlewareTimesIndex, receiveAsTokens
        );

        address eigenpieWithdrawManager = eigenpieConfig.getContract(EigenpieConstants.
            EIGENPIE_WITHDRAW_MANAGER);
        for (uint256 i = 0; i < tokens.length;) {
            uint256 afterBalance = tokens[i].balanceOf(address(this));
            IERC20(tokens[i]).safeTransfer(eigenpieWithdrawManager, afterBalance -
                beforeAmounts[i]);

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

Listing 3.3: `NodeDelegator::completeAssetWithdrawalFromEigenLayer()`

Recommendation Improve the above-mentioned routine by validating its caller.

Status This issue has been resolved as it is part of intended design.

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);
    }

    /// @dev Adds a new supported asset
    /// @param asset Asset address
    /// @param mLRTReceipt MLRT receipt
    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);
    }

    /// @dev private function to add a new supported asset
    /// @param asset Asset address
    /// @param depositLimit Deposit limit for the asset
    function _addNewSupportedAsset(address asset, address mLRTReceipt, uint256
        depositLimit) private {
        UtilLib.checkNonZeroAddress(asset);
        UtilLib.checkNonZeroAddress(mLRTReceipt);

        if (isSupportedAsset[asset]) {
            revert AssetAlreadySupported();
        }

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

        isSupportedAsset[asset] = true;
        mLRTReceiptByAsset[asset] = mLRTReceipt;

        supportedAssetList.push(asset);
        depositLimitByAsset[asset] = depositLimit;

        boostByAsset[asset] = EigenpieConstants.DENOMINATOR;

        emit AddedNewSupportedAsset(asset, mLRTReceipt, depositLimit);
```

```
    }

    /// @dev Updates the deposit limit for an asset
    /// @param asset Asset address
    /// @param depositLimit New deposit limit
    function updateAssetDepositLimit(
        address asset,
        uint256 depositLimit
    )
        external
        onlyRole(EigenpieConstants.MANAGER)
        onlySupportedAsset(asset)
    {
        depositLimitByAsset[asset] = depositLimit;
        emit AssetDepositLimitUpdate(asset, depositLimit);
    }

    /// @dev Updates the strategy for an asset
    /// @param asset Asset address
    /// @param strategy New strategy address
    function updateAssetStrategy(
        address asset,
        address strategy
    )
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
        onlySupportedAsset(asset)
    {
        UtilLib.checkNonZeroAddress(strategy);
        if (assetStrategy[asset] == strategy) {
            revert ValueAlreadyInUse();
        }

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

        assetStrategy[asset] = strategy;
        emit AssetStrategyUpdate(asset, strategy);
    }
```

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.