SMART CONTRACT AUDIT REPORT

for

Rhombus Protocol

Prepared By: Xiaomi Huang

PeckShield

June 5, 2024

Document Properties

Client	Rhombus
Title	Smart Contract Audit Report
Target	Rhombus
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	June 5, 2024	Xuxian Jiang	Final Release
1.0-rc	June 2, 2023	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 `Rhombus` 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 can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Rhombus Protocol

`Rhombus Protocol` is a robust money market to provide seamless lending and borrowing with accessible liquidity to the users on `KAIA` network. The platform aims to democratize access to liquidity, serving not only `DeFi` users but also protocols through `Liquidity Link`. One of the major challenges for `DeFi` protocols today is bootstrapping initial liquidity and consistently sourcing it for growth. Many protocols struggle to access liquidity in money markets due to a lack of composability, limiting their growth potential. While the `DeFi` landscape continuously evolves to support diverse borrowing use cases, the architecture of money markets has not kept pace. This forces protocols to independently source liquidity, which is often limited. Consequently, protocols spend a significant portion of their native token supply on incentives to obtain liquidity. This approach is unsustainable in the long run and leads to liquidity fragmentation. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Rhombus

Item	Description
Name	Rhombus
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 5, 2024

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

- https://github.com/rhombusprotocol/rhombus-contracts.git (142283a)

And here is the commit ID after fixes for the issues found in the audit have been checked in:

- https://github.com/rhombusprotocol/rhombus-contracts.git (2023cfe)

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 classified into four categories accordingly, 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 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.

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 1.3: 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

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 implementation of the `Rhombus` protocol. 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. 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 that 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 Rhombus Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Reward Harvesting Logic in RewardDistributor	Business Logic	Resolved
PVE-002	Low	Incorrect Pool Distribution APY Calculation in RewardDistributor	Business Logic	Resolved
PVE-003	Low	Improved Protocol Parameter Validation in RewardDistributor	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

PeckShield

3 | Detailed Results

3.1 Improper Reward Harvesting Logic in RewardDistributor

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

Description

To incentivize protocol users, the Rhombus protocol has the built-in RewardDistributor contract. While examining the reward-harvesting logic, we notice an issue that does not properly reward protocol users.

In the following, we show the implementation of the related harvest() routine. As the name indicates, this routine is designed to harvest rewards for the given position (specified by the input posId). While it does properly calculate the rewards, we notice the final reward distribution does not include the accumulated posInfo.pendingAmount. As a result, if the position is updated by another user, the rewards will only be accumulated in posInfo.pendingAmount and the owning user may not receive any reward.

```
    function harvest(address rewardToken, address lendingPool, uint256 posId) external
        onlyReporter returns (uint256) {
        notifySupplyIndexInternal(rewardToken, lendingPool);
        notifyBorrowIndexInternal(rewardToken, lendingPool);

        RewardMarketState storage marketState = rewardMarketState[rewardToken][
            lendingPool];
        RewardPosPoolState storage posInfo = rewardPosPoolState[rewardToken][posId][
            lendingPool];

        uint256 accumulatedSupplyReward = marketState.supplyAccPerShare * posInfo.
            collateralAmountStored / 1e26;

        uint256 pendingSupplyReward = accumulatedSupplyReward - posInfo.
            supplierRewardDebt;
```

```
        posInfo.supplierRewardDebt = accumulatedSupplyReward;

        uint256 accumulatedBorrowReward = marketState.borrowAccPerShare * posInfo.
            borrowSharesStored / 1e26;

        uint256 pendingBorrowReward = accumulatedBorrowReward - posInfo.
            borrowerRewardDebt;

        posInfo.borrowerRewardDebt = accumulatedBorrowReward;

        uint256 totalPendingReward = pendingSupplyReward + pendingBorrowReward;

        if (totalPendingReward > 0) {
            address posOwner = IERC721(posManager).ownerOf(posId);
            _require(posOwner != address(0), Errors.ZERO_VALUE);
            SafeERC20.safeTransfer(IERC20(rewardToken), posOwner, totalPendingReward);
            emit RewardGranted(rewardToken, posOwner, totalPendingReward);

            return totalPendingReward;
        }

        return 0;
    }
```

Listing 3.1: `RewardDistributor::harvest()`

Recommendation Revisit the above routine to properly distribute user rewards.

Status This issue has been fixed by the following commit: `2023cfe`.

3.2 Incorrect Pool Distribution APY Calculation in RewardDistributor

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardDistributor`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `Rhombus` protocol has a built-in `RewardDistributor` contract to incentivize protocol users. This contract contains a helper routine `calculatePoolDistributionAPY()` to report the expected `APY`s for protocol users. Our analysis shows this routine computes an incorrect `APY`.

```
388     function calculatePoolDistributionAPY(address lendingPool, address rewardToken)
389         public
390         view
391         returns (uint256 supplyApy, uint256 borrowApy)
392     {
393         {
394             IRBSOracle priceCalculator = IRBSOracle(oracle);
395             RewardMarketState memory marketState = rewardMarketState[rewardToken][
                    lendingPool];
396             address underlyingToken = ILendingPool(lendingPool).underlyingToken();
397             uint8 underlyingTokenDecimals = IERC20Metadata(underlyingToken).decimals();
398             uint8 rewardTokenDecimals = IERC20Metadata(rewardToken).decimals();
399
400             uint256 annualSupplyRewardInUSD = marketState.supplySpeed * 365 days
401                 * priceCalculator.getPrice_e36(rewardToken) / 10 ** (18 -
                        rewardTokenDecimals);
402             uint256 supplyInUSD = ILendingPool(lendingPool).totalAssets()
403                 * priceCalculator.getPrice_e36(underlyingToken) / 10 ** (18 -
                        underlyingTokenDecimals);
404             supplyApy = annualSupplyRewardInUSD > 0 ? annualSupplyRewardInUSD /
                    supplyInUSD : 0;
405
406             uint256 annualBorrowRewardInUSD = marketState.borrowSpeed * 365 days
407                 * priceCalculator.getPrice_e36(rewardToken) / 10 ** (18 -
                        rewardTokenDecimals);
408
409             uint256 borrowInUSD = ILendingPool(lendingPool).totalDebt() *
                    priceCalculator.getPrice_e36(underlyingToken)
410                 / 10 ** (18 - underlyingTokenDecimals);
411
412             borrowApy = annualBorrowRewardInUSD > 0 ? annualBorrowRewardInUSD /
                    borrowInUSD : 0;
413         }
414     }
```

Listing 3.2: `RewardDistributor::calculatePoolDistributionAPY()`

To elaborate, we show above the implementation of this `calculatePoolDistributionAPY()` routine. It has a rather straightforward logic in computing the supply APY and the borrow APY. However, current implementation incorrectly interprets the token decimals. Specifically, the `annualSupplyRewardInUSD` calculation is currently computed as `marketState.supplySpeed * 365 days * priceCalculator.getPrice_e36(rewardToken)/ 10 ** (18 - rewardTokenDecimals)` (lines 400 – 401), which should be as follows: `marketState.supplySpeed * 365 days * priceCalculator.getPrice_e36(rewardToken)/ 10 ** (18 + rewardTokenDecimals)`. Similarly, the the `supplyInUSD` calculation should be corrected as `ILendingPool(lendingPool).totalAssets()* priceCalculator.getPrice_e36(underlyingToken)/ 10 ** (18 + underlyingTokenDecimals)`. Note the calculations of `annualBorrowRewardInUSD` and `borrowInUSD` share the same issue.

Recommendation Revise the above routine to properly compute the reward distribution APYs.

Status This issue has been fixed by the following commit: `2023cfe`.

3.3 Improved Protocol Parameter Validation 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 `Rhombus` protocol is no exception. Specifically, if we examine the `RewardDistributor` contract, it has defined a number of protocol-wide risk parameters, such as `posManager` and `oracle`. In the following, we show the corresponding routines that allow for their changes.

```
    function initialize(address _acm, address _posManager, address _oracle) public
        initializer {
        __UnderACM_init(_acm);

        posManager = _posManager;
        oracle = _oracle;

        approvedReporter[msg.sender] = true;
        approvedReporter[posManager] = true;
        approvedReporter[oracle] = true;
    }

    function setPosManager(address posManager_) public onlyGovernor {
        posManager = posManager_;
    }

    function setOracle(address oracle_) public onlyGovernor {
        oracle = oracle_;
    }
```

Listing 3.3: RewardDistributor::initialize()/setPosManager()/setOracle()

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. For example, the update of `posManager` and/or `oracle` should be accompanied with the related update on `approvedReporter`, in either disabling previous entity's reporting role or enabling the new entity's reporting role.

Recommendation Validate any changes regarding these system-wide parameters and ensure the cascading impacts are properly applied.

Status This issue has been fixed by the following commit: `2023cfe`.

3.4 Trust Issue of Admin Keys

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

Description

In the `Rhombus` protocol, there is a privileged administrative account (with the `GOVERNOR` role). The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Config` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setPoolConfig(address _pool, PoolConfig calldata _config) external onlyGuardian {
        __poolConfigs[_pool] = _config;
        emit SetPoolConfig(_pool, _config);
    }

    /// @inheritdoc IConfig
    function setCollFactors_e18(uint16 _mode, address[] calldata _pools, uint128[] calldata _factors_e18)
        external
        onlyGovernor
    {
        _require(_mode != 0, Errors.INVALID_MODE);
        _require(_pools.length == _factors_e18.length, Errors.ARRAY_LENGTH_MISMATCHED);
        _require(AddressArrayLib.isSortedAndNotDuplicate(_pools), Errors.NOT_SORTED_OR_DUPLICATED_INPUT);
        EnumerableSet.AddressSet storage collTokens = __modeConfigs[_mode].collTokens;
        for (uint256 i; i < _pools.length; i = i.uinc()) {
            _require(_factors_e18[i] <= ONE_E18, Errors.INVALID_FACTOR);
            collTokens.add(_pools[i]);
            __modeConfigs[_mode].factors[_pools[i]].collFactor_e18 = _factors_e18[i];
        }
        emit SetCollFactors_e18(_mode, _pools, _factors_e18);
    }

    /// @inheritdoc IConfig
```

```
    function setBorrFactors_e18(uint16 _mode, address[] calldata _pools, uint128[] calldata _factors_e18)
        external
        onlyGovernor
    {
        _require(_mode != 0, Errors.INVALID_MODE);
        _require(_pools.length == _factors_e18.length, Errors.ARRAY_LENGTH_MISMATCHED);
        _require(AddressArrayLib.isSortedAndNotDuplicate(_pools), Errors.NOT_SORTED_OR_DUPLICATED_INPUT);
        EnumerableSet.AddressSet storage borrTokens = __modeConfigs[_mode].borrTokens;
        for (uint256 i; i < _pools.length; i = i.uinc()) {
            borrTokens.add(_pools[i]);
            _require(_factors_e18[i] >= ONE_E18, Errors.INVALID_FACTOR);
            __modeConfigs[_mode].factors[_pools[i]].borrFactor_e18 = _factors_e18[i];
        }
        emit SetBorrFactors_e18(_mode, _pools, _factors_e18);
    }

    /// @inheritdoc IConfig
    function setModeStatus(uint16 _mode, ModeStatus calldata _status) external onlyGuardian {
        _require(_mode != 0, Errors.INVALID_MODE);
        __modeConfigs[_mode].status = _status;
        emit SetModeStatus(_mode, _status);
    }

    /// @inheritdoc IConfig
    function setMaxHealthAfterLiq_e18(uint16 _mode, uint64 _maxHealthAfterLiq_e18) external onlyGuardian {
        _require(_mode != 0, Errors.INVALID_MODE);
        _require(_maxHealthAfterLiq_e18 > ONE_E18, Errors.INPUT_TOO_LOW);
        __modeConfigs[_mode].maxHealthAfterLiq_e18 = _maxHealthAfterLiq_e18;
        emit SetMaxHealthAfterLiq_e18(_mode, _maxHealthAfterLiq_e18);
    }

    /// @inheritdoc IConfig
    function setWhitelistedWLps(address[] calldata _wLps, bool _status) external onlyGovernor {
        for (uint256 i; i < _wLps.length; i = i.uinc()) {
            whitelistedWLps[_wLps[i]] = _status;
        }
        emit SetWhitelistedWLps(_wLps, _status);
    }
```

Listing 3.4: Example Privileged Operations in `Config`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, 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 plan to transfer the privileged account to a multi-sig account.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the Rhombus protocol, which is a robust money market to provide seamless lending and borrowing with accessible liquidity to the users on KAIA network. The platform aims to democratize access to liquidity, serving not only DeFi users but also protocols through Liquidity Link. One of the major challenges for DeFi protocols today is bootstrapping initial liquidity and consistently sourcing it for growth. Many protocols struggle to access liquidity in money markets due to a lack of composability, limiting their growth potential. While the DeFi landscape continuously evolves to support diverse borrowing use cases, the architecture of money markets has not kept pace. This forces protocols to independently source liquidity, which is often limited. Consequently, protocols spend a significant portion of their native token supply on incentives to obtain liquidity. This approach is unsustainable in the long run and leads to liquidity fragmentation. 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.

PeckShield