SMART CONTRACT AUDIT REPORT

for

Rift Protocol

Prepared By: Yiqun Chen

PeckShield

January 31, 2022

Document Properties

Client	Rift Finance
Title	Smart Contract Audit Report
Target	Rift
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 31, 2022	Xuxian Jiang	Final Release
1.0-rc	January 23, 2022	Xuxian Jiang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4

1.1 About Rift 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings 9

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results 11

3.1 Improved Gas Efficiency of Vault::__updateDepositRequests() 11

3.2 Vault Incompatibility With Deflationary Tokens 12

3.3 Trust Issue of Admin Keys 14

4 Conclusion 16

References 17

1 Introduction

Given the opportunity to review the design document and related source code of the `Rift` 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 Rift

The `Rift` protocol provides a synergistic way for users with different motivations to achieve their bespoke investment goals together. Specifically, it allows users who are liquidity motivated (e.g. `DAO` s), to pair deposits with users who are profit motivated so that the first user can achieve deepened liquidity for their asset, and the second user can receive all of the returns from the position. The protocol consists of vaults - each accepting deposits for two assets. Vaults can interact with any `UniswapV2` style `AMMs` to deepen liquidity for any `ERC20`. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	Rift Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 31, 2022

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

- https://github.com/Rift-Finance/rift-protocol.git (7378c61)

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

- https://github.com/Rift-Finance/rift-protocol.git (f6674df)

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 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

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

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 1.4: 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 `Rift` 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	1	🟦
Low	1	🟦
Informational	1	🟦
Total	3	

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 refer to 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 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 we identify the following issues (shown in Table 2.1) for improvement, including 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings of Rift Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Gas Efficiency of Vault::__updateDepositRequests()	Coding Practices	Fixed
PVE-002	Informational	Vault Incompatibility With Deflationary Tokens	Business Logic	Confirmed
PVE-003	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.

3 | Detailed Results

3.1 Improved Gas Efficiency of Vault::__updateDepositRequests()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In `Rift`, users can deposit into a vault with the chosen `token0` or `token1`. And the protocol contract updates the respective accounting balances in a so-called `lazy accounting` scheme. By this scheme, the user deposits are queued up and will be included in the next epoch. While reviewing this scheme, we notice a potential gas optimization opportunity.

To elaborate, we show below the related `__updateDepositRequests()` handler. This handler is designed to update the deposit requests with any new deposit amount. If the epoch of the request has passed, the deposit amount is flushed to the `balanceDay0` state for normalized `lazy accounting`. While examining the above handler, we notice there exist repeated storage read operations on `req.amount` (lines 138, 141, and 146), which can be cached upon the first read and simply reused later! By doing so, we can save the storage reads from statements located at lines 141 and 146.

```
130     function __updateDepositRequests(
131         AssetData storage assetData,
132         uint256 currEpoch,
133         uint256 _depositAmount
134     ) private returns (uint256 newBalanceDay0) {
135         Request storage req = assetData.depositRequests[msg.sender];

137         uint256 balance = assetData.balanceDay0[msg.sender];
138         uint256 newReqAmount = req.amount;
```

```
        // If they have a prior request
        if (req.amount > 0 && req.epoch < currEpoch) {
            // and if it was from a prior epoch
            // we now know the exchange rate at that epoch,
            // so we can add to their balance
            uint256 conversionRate = assetData.epochToRate[req.epoch];
            balance += (req.amount * RAY) / conversionRate;

            newReqAmount = 0;
        }

        if (_depositAmount > 0) {
            // if they don't have a prior request, store this one (if this is a non-zero
                deposit)
            newReqAmount += _depositAmount;
            req.epoch = currEpoch;
        }
        req.amount = newReqAmount;

        return balance;
    }
```

Listing 3.1: `Vault::__updateDepositRequests())`

Recommendation Avoid repeated storage reads on the same location for improved gas efficiency.

Status The issue has been fixed in the following commit: ba140c9.

3.2 Vault Incompatibility With Deflationary Tokens

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Vault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `Rift` protocol, the `Vault` contract is designed to take users' assets and deliver corresponding yields. In particular, one interface, i.e., `depositToken0()/depositToken1()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdrawToken0()/withdrawToken1(0)`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, the contract using the `safeTransferFrom()` routine to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function depositToken0(uint256 _amount) external payable override whitelisted
        whenNotPaused nonReentrant {
        if (isEthVault) {
            IWETH(address(token0)).deposit{ value: msg.value }();
            _depositAccounting(token0Data, msg.value, TOKEN0);
        } else {
            require(msg.value == 0, "NOT_ETH_VAULT");
            token0.safeTransferFrom(msg.sender, address(this), _amount);
            _depositAccounting(token0Data, _amount, TOKEN0);
        }
    }

    /// @notice schedules a deposit of the TOKEN1 into the ceiling tranche
    /// @param _amount the amount of the TOKEN1 to schedule-deposit
    function depositToken1(uint256 _amount) external override whitelisted whenNotPaused
        nonReentrant {
        token1.safeTransferFrom(msg.sender, address(this), _amount);
        _depositAccounting(token1Data, _amount, TOKEN1);
    }
```

Listing 3.2: `Vault::depositToken0()/depositToken1()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Rift` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Rift` protocol, there is a privileged account with `GOVERN_ROLE` that plays a critical role in governing and regulating the system-wide operations (e.g., assigning other roles, configuring protocol parameters, and initiating the next epoch). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function rescueTokens(address[] calldata tokens, uint256[] calldata amounts)
        external
        override
        nonReentrant
        onlyGuardian
        whenPaused
    {
        require(tokens.length == amounts.length, "INVALID_INPUTS");

        for (uint256 i = 0; i < tokens.length; i++) {
            uint256 amount = amounts[i];
            if (tokens[i] == address(0)) {
                amount = (amount == 0) ? address(this).balance : amount;
                (bool success, ) = msg.sender.call{ value: amount }("");
                require(success, "TRANSFER_FAILED");
            } else {
                amount = (amount == 0) ? IERC20Upgradeable(tokens[i]).balanceOf(address(
                    this)) : amount;
                IERC20Upgradeable(tokens[i]).safeTransfer(msg.sender, amount);
            }
        }
        emit FundsRescued(msg.sender);
    }
    function unstakeLiquidity() external override nonReentrant onlyGuardian whenPaused {
        _unstakeLiquidity();
    }
```

Listing 3.3: `Vault::unstakeLiquidity()/rescueTokens()`

If the privileged account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed 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 confirmed with the team. For the time being, the team has confirmed that they will exercise care to delegate responsibilities to independent roles, such as `Pauser`, `Guardian`, `Strategist`, `Governor`, so that each responsibility is siloed. For example, rescuing funds can only be done by the `Guardian`, but in order to execute this, the Pauser must have paused the contracts. Only the `Governor` can set parameters like the protocol fee. Only the strategist can move the vault to the next epoch, etc. Each of these roles will be managed independently so that a single party does not have control over protocol-wide operations. Additionally, the `Governor` role (which is the only role with the ability to add roles) will be managed by a multisig.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Rift` protocol, which aims to offer a way for `DAOs` to attract liquidity for their tokens. The current code base is well organized and 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.