SMART CONTRACT AUDIT REPORT

for

Ribbon Treasury

Prepared By: Yiqun Chen

PeckShield

January 31, 2022

Document Properties

Client	Ribbon Finance
Title	Smart Contract Audit Report
Target	Ribbon Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Liu, Xuxian Jiang
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-rc1	January 22, 2022	Xuxian Jiang	Release Candidate #1

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 Ribbon Treasury . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 6

2 Findings 10
- 2.1 Summary . . . 10
- 2.2 Key Findings . . . 11

3 Detailed Results 12
- 3.1 Potential OOG With Flooded Tiny Depositors . . . 12
- 3.2 Improved Logic in _removeDepositor() . . . 13
- 3.3 Improved NatSpec Comments of transferAsset() . . . 14

4 Conclusion 16

References 17

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Ribbon Treasury 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 Ribbon Treasury

Ribbon Finance is building on-chain option vaults that use smart contracts to automate various options strategies. Users can simply deposit their assets into a smart contract and will automatically start running a specific options strategy. The audited Ribbon Treasury is a Ribbon's Theta Vault product but catered to DAOs. Note that Theta Vault is a yield-generating strategy, which runs a covered call strategy to earn yield on a weekly basis through writing out of the money covered calls and collecting the premiums.

The basic information of Ribbon Treasury is as follows:

Table 1.1: Basic Information of Ribbon Finance

Item	Description
Issuer	Ribbon Finance
Website	https://ribbon.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. And this audit only covers the following two contracts: `RibbonTreasuryVault` and `VaultLifecycleTreasury`.

- https://github.com/ribbon-finance/ribbon-v2.git (d94e37a)

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

- https://github.com/ribbon-finance/ribbon-v2.git (696b3b9)

1.2 About PeckShield

PeckShield Inc. [7] 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 the OWASP Risk Rating Methodology [6]:

- 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 checklist of 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) [5], 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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 Logic	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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Ribbon Treasury`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 logic, 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 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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Ribbon Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential OOG With Flooded Tiny Depositors	Business Logic	Fixed
PVE-002	Low	Improved Logic in _removeDepositor()	Coding Practices	Fixed
PVE-003	Informational	Improved NatSpec Comments of transferAsset()	Coding Practices	Fixed

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 Potential OOG With Flooded Tiny Depositors

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: RibbonTreasuryVault
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

The Ribbon Finance protocol develops new on-chain option vaults that allow for automating various options strategies. The audited RibbonTreasuryVault contract contains a helper routine _distributePremium() to distribute the premium to depositor addresses. Our analysis shows that this routine may suffer from a potential out-of-gas execution when there is a huge list of depositors.

In the following, we show below the implementation of this helper routine. As the name indicates, it distributes the collected premium to the current depositors based on their deposited amount. However, it is possible to flood a huge list of tiny depositors and each only deposits a tiny amount (e.g., 1 WEI). By doing so, the premium distribution routine will essentially run out-of-gas, leading to a denial-of-service situation and causing potential disruption to conclude option sales.

```
    function _distributePremium(IERC20 token, uint256 amount) internal {
        // Distribute to depositor address
        address[] storage _depositors = depositorsArray;
        uint256[] memory _amounts = new uint256[](_depositors.length);
        uint256 totalSupply = totalSupply();

        for (uint256 i = 0; i < _depositors.length; i++) {
            // Distribute to depositors proportional to the amount of
            // shares they own
            address depositorAddress = _depositors[i];
            _amounts[i] = shares(depositorAddress).mul(amount).div(totalSupply);

            token.safeTransfer(depositorAddress, _amounts[i]);
```

```
        }

        emit DistributePremium(
            amount,
            _amounts,
            _depositors,
            vaultState.round - 1
        );
    }
```

Listing 3.1: `RibbonTreasuryVault::_distributePremium()`

Recommendation Set up a minimum deposit amount to mitigate the above `out-of-gas` (`OOG`) situation.

Status The issue has been fixed by this commit: `b0984e0`.

3.2 Improved Logic in _removeDepositor()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RibbonTreasuryVault`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.1, to properly distribute the collected premium, the `Ribbon Treasury` protocol keeps track of a list of current depositors and provides related routines (`_addDepositor` and `_removeDepositor()`) to manage this list. Our analysis on the depositor removal logic can be improved.

To elaborate, we show below the `_removeDepositor()` routine. This routine aims to maintain the order of current depositors by firstly locating the index of the given depositor for removal and then shifting the subsequent depositors to fill the removed index. This logic may unavoidably incur gas cost, which may be significantly increased when the deposit list is sufficiently long. We can choose to ignore the depositor order and apply a `swap-and-pop` approach, i.e., we can choose to switch the removed index with the last element in the list and then pop up the last "switched" element at the end.

```
    function _removeDepositor(address excludeDepositor) internal {
        uint256 DepositorListLength = depositorsArray.length;
        require(depositorsMap[excludeDepositor], "Depositor does not exist");

        depositorsMap[excludeDepositor] = false;

```

```
435         for (uint256 i = 0; i < DepositorListLength; i++) {
436             if (excludeDepositor == depositorsArray[i]) {
437                 for (uint256 j = i; j < (DepositorListLength - 1); j++) {
438                     depositorsArray[j] = depositorsArray[j + 1];
439                 }
440             }
441         }
442         depositorsArray.pop();
443     }
```

Listing 3.2: `RibbonTreasuryVault::_removeDepositor()`

Moreover, the `for`-loop (line 435) in the above routine can be further optimized by setting up the upper bound to `DepositorListLength-1`, instead of the current `DepositorListLength`.

Recommendation Apply the above optimizations to reduce the gas cost and eliminate possible `OOG` issue.

Status The issue has been fixed by this commit: `e9dcf8f`.

3.3 Improved NatSpec Comments of transferAsset()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RibbonTreasuryVault`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

`Solidity`-based contracts can use a special form of comments to provide rich documentation for functions and return variables. This special form is named the `Ethereum Natural Language Specification Format (NatSpec)`. The `Ribbon Treasury` protocol provides detailed and helpful `NatSpec` comments. When reviewing the provided comments, we notice a function's comment is not consistent with its current implementation.

In particular, we show below the related function `transferAsset()`. This function has a rather straightforward logic in making a transfer of the supported `vaultParams.asset`. It comes to our attention that the preceding comments indicate the support of `either an ETH transfer or ERC20 transfer`.

```
    /**
     * @notice Helper function to make either an ETH transfer or ERC20 transfer
     * @param recipient is the receiving address
     * @param amount is the transfer amount
```

```
     */
    function transferAsset(address recipient, uint256 amount) internal {
        address asset = vaultParams.asset;
        IERC20(asset).safeTransfer(recipient, amount);
    }
```

Listing 3.3: `RibbonTreasuryVault::transferAsset()`

Recommendation Be consistent in the function logic and the associated comments.

Status The issue has been fixed by this commit: `6db8b79`.

4 Conclusion

In this audit, we have analyzed the `Ribbon Treasury` design and implementation. The system presents a unique, robust offering as a decentralized protocol for automating various options strategies. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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