SMART CONTRACT AUDIT REPORT

for

Ribbon Swap

Prepared By: Patrick Liu

Hangzhou, China

March 11, 2022

Document Properties

Client	Ribbon Finance
Title	Smart Contract Audit Report
Target	Ribbon Swap
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 11, 2022	Xiaotao Wu	Final Release
1.0-rc	March 8, 2022	Xiaotao Wu	Release Candidate

Contact

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

Name	Patrick Liu
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Ribbon Swap` feature, 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Ribbon Swap

`Ribbon Finance` is building on-chain option vaults that use smart contracts to automate various options strategies. In running these strategies, `Ribbon Finance` conducts an open auction every week to sell options minted by `Ribbon`'s `Theta Vault`. The `Swap` contract is part of `Ribbon`'s new auction architecture and it serves as a settlement layer between `Ribbon`'s `Theta Vault` and winning bidders. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Ribbon Swap

Item	Description
Name	Ribbon Finance
Website	https://www.ribbon.finance/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 11, 2022

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

- https://github.com/ribbon-finance/ribbon-v2/blob/master/contracts/utils/Swap.sol (aa5c069)

And this 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/blob/master/contracts/utils/Swap.sol (dd92785)

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 the 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 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

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

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. 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 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Ribbon Swap` smart contract. 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	1	■
Medium	1	■
Low	0	
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 high-severity vulnerability, 1 medium-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Ribbon Swap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Lack Of swapId Verification In Swap::settleOffer()	Business Logic	Resolved
PVE-002	Informational	Meaningful Events For Important State Changes	Coding Practices	Resolved
PVE-003	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 Lack Of swapId Verification In Swap::settleOffer()

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

Description

The `Swap` contract provides an external `settleOffer()` function for the swap offer creators to settle the swap offering by iterating through the bids. While examining the `settleOffer()` routine, we notice the current logic is not implemented properly.

To elaborate, we show below its code snippet. It comes to our attention that there is a lack of verification for the `swapId` of the bids before calling the helper function `_swap()` to execute the `ERC20` swap (line 178). Thus the `Swap` contract has the vulnerability that a bid may not be settled by the correct `swapId` that this bid requires. A malicious actor can exploit this to withdraw `biddingToken` from bid signers by simply creating an swap offer with worthless `oToken`.

```
150     /**
151      * @notice Settles the swap offering by iterating through the bids
152      * @param swapId unique identifier of the swap offer
153      * @param bids bids for swaps
154      */
155     function settleOffer(uint256 swapId, Bid[] calldata bids)
156         external
157         override
158         nonReentrant
159     {
160         Offer storage offer = swapOffers[swapId];

162         address seller = offer.seller;
163         require(
164             seller == msg.sender,
```

```
            "Only seller can settle or offer doesn't exist"
        );
        require(offer.availableSize > 0, "Offer fully settled");

        uint256 totalSales;
        OfferDetails memory offerDetails;
        offerDetails.seller = seller;
        offerDetails.oToken = offer.oToken;
        offerDetails.biddingToken = offer.biddingToken;
        offerDetails.minPrice = offer.minPrice;
        offerDetails.minBidSize = offer.minBidSize;

        for (uint256 i = 0; i < bids.length; i++) {
            _swap(offerDetails, offer, bids[i]);
            totalSales += bids[i].sellAmount;
        }

        bool fullySettled = offer.availableSize == 0;

        // Deduct the initial 1 wei offset if offer is fully settled
        offer.totalSales += totalSales - (fullySettled ? 1 : 0);

        if (fullySettled) {
            emit SettleOffer(swapId);
        }
    }
```

Listing 3.1: `Swap::settleOffer()`

Recommendation Add verification for the `swapId` of the bids.

Status This issue has been fixed in the following commit: `dd92785`.

3.2 Meaningful Events For Important State Changes

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Swap
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the Swap contract as an example. While examining the event that reflect the Swap dynamics, we notice there is a lack of emitting related event to reflect important state change. Specifically, when the setFee() is being called, there is no corresponding event being emitted to reflect the occurrence of setFee().

```
    /**
     * @notice Sets the referral fee for a specific referrer
     * @param referrer is the address of the referrer
     * @param fee is the fee in percent in 2 decimals
     */
    function setFee(address referrer, uint256 fee) external onlyOwner {
        require(referrer != address(0), "Referrer cannot be the zero address");
        require(fee < MAX_PERCENTAGE, "Fee exceeds maximum");

        referralFees[referrer] = fee;
    }
```

Listing 3.2: Swap::setFee()

Recommendation Properly emit the related event when the above-mentioned function is being invoked.

Status This issue has been fixed in the following commit: dd92785.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Swap`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Ribbon Swap` contract, there is a privileged account, i.e., `owner`. This account plays a critical role in governing and regulating the `Swap` contract.

In the following, we show the code snippet of the function that potentially affected by the privileges of the `owner` account. The `setFee()` function allows the `owner` to set the referral fee for a specific referrer. If this value is set too large, most of the `biddingToken` will be send to the referrer instead of the swap offer creator when a swap offer creator settles the swap offer.

```
    /**
     * @notice Sets the referral fee for a specific referrer
     * @param referrer is the address of the referrer
     * @param fee is the fee in percent in 2 decimals
     */
    function setFee(address referrer, uint256 fee) external onlyOwner {
        require(referrer != address(0), "Referrer cannot be the zero address");
        require(fee < MAX_PERCENTAGE, "Fee exceeds maximum");

        referralFees[referrer] = fee;
    }
```

Listing 3.3: `Swap::setFee()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the 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 this privilege explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to privileged accounts explicit to `Ribbon V2` protocol users.

Status This issue has been mitigated in the following commit: `dd92785`.

4 Conclusion

In this audit, we have analyzed the Ribbon Swap design and implementation. The system presents a unique, robust offering as a decentralized money market protocol with both secure lending and synthetic stablecoins. Ribbon V2 is the next version for Ribbon's Theta Vault product. It brings several major improvements to the vault and makes the vault operations decentralized. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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.