SMART CONTRACT AUDIT REPORT

for

RVOL

Prepared By: Yiqun Chen

PeckShield

July 4, 2021

Document Properties

Client	Ribbon Finance
Title	Smart Contract Audit Report
Target	RVOL
Version	1.0-rc
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	July 4, 2021	Xiaotao Wu	Initial Draft

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 RVOL . . . 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 Incorrect Calculation in Math::cdf()/ncdf() . . . 11

3.2 Improved Corner Case Handling in getTimeWeightedAverageTick() . . . 12

3.3 Inconsistency Between Document and Implementation . . . 14

4 Conclusion **15**

References **16**

1 Introduction

Given the opportunity to review the source code of the RVOL smart contract, 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 RVOL

RVOL (Ribbon Volatility) is a set of Solidity libraries and tools that utilize the Uniswap V3 to make on-chain volatility data accessible. Its goals are to help builders for building on-chain volatility indices, querying realized volatility information, and pricing derivatives and options with on-chain data. The RVOL library intends to have the features of realized volatility oracles for Uniswap v3 pools, options pricing with Black Scholes for Uniswap v3 pools, and index pricing with oracles.

The basic information of RVOL is as follows:

Table 1.1: Basic Information of RVOL

Item	Description
Issuer	Ribbon Finance
Website	https://ribbon.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 4, 2021

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

- https://github.com/ribbon-finance/rvol.git (08a01b3)

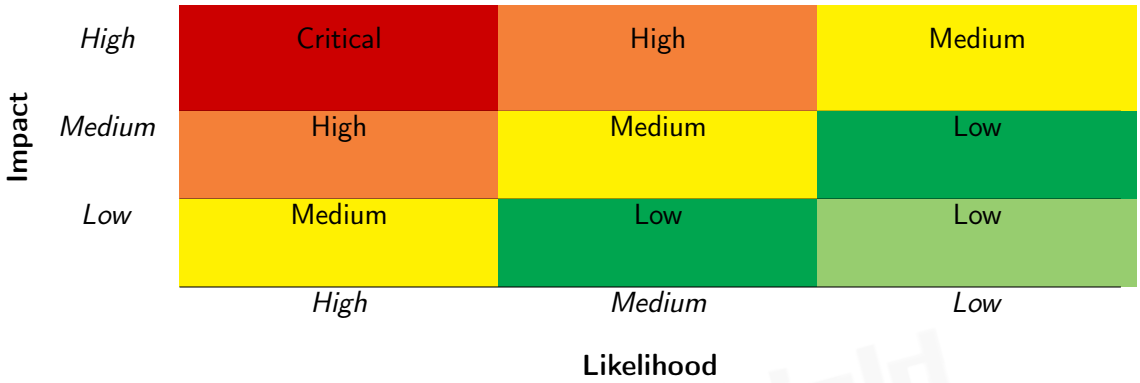
And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/ribbon-finance/rvol.git (869bc1a)

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



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 RVOL 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	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 RVOL Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Calculation in Math::cdf()/ncdf()	Numeric Errors	Fixed
PVE-002	Low	Improved Corner Case Handling in getTimeWeightedAverageTick()	Time And State	Fixed
PVE-003	Informational	Inconsistency Between Document and Implementation	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 Incorrect Calculation in Math::cdf()/ncdf()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Math`
- Category: Numeric Errors [6]
- CWE subcategory: CWE-190 [2]

Description

The `OptionsPremiumPricer` contract provides an external `getPremium()` function to calculate the premium of the provided option using the `Black-Scholes` model. This function internally calls the cumulative distribution functions provided by the `RVOL`'s `Math` library to calculate the `Black-Scholes` for both `PUT` and `CALL` options. To elaborate, we show below these `CDF` routines. It comes to our attention that an incorrect parameter is used to calculate the value for local variable `t1` in the `ncdf()/cdf()` functions and may lead to precision loss of the calculated result. Specifically, as described in [10], the correct coefficient multiplied to the variable `x` should be `2316419`, instead of current `2315419`.

```
245     function ncdf(uint256 x) internal pure returns (uint256) {
246         int256 t1 = int256(1e7 + ((2315419 * x) / FIXED_1));
247         uint256 exp = ((x / 2) * x) / FIXED_1;
248         int256 d = int256((3989423 * FIXED_1) / optimalExp(uint256(exp)));
249         uint256 prob =
250             uint256(
251                 (d *
252                     (3193815 +
253                         ((-3565638 +
254                             ((17814780 +
255                                 ((-18212560 + (13302740 * 1e7) / t1) * 1e7) /
256                                 t1) * 1e7) /
257                             t1) * 1e7) /
258                         t1) *
259                     1e7) / t1
260             );
```

```
261         if (x > 0) prob = 1e14 - prob;
262         return prob;
263     }
264
265     function cdf(int256 x) internal pure returns (uint256) {
266         int256 t1 = int256(1e7 + int256((2315419 * abs(x)) / FIXED_1));
267         uint256 exp = uint256((x / 2) * x) / FIXED_1;
268         int256 d = int256((3989423 * FIXED_1) / optimalExp(uint256(exp)));
269         uint256 prob =
270             uint256(
271                 (d *
272                     (3193815 +
273                         ((-3565638 +
274                             ((17814780 +
275                                 ((-18212560 + (13302740 * 1e7) / t1) * 1e7) /
276                                 t1) * 1e7) /
277                             t1) * 1e7) /
278                         t1) *
279                     1e7) / t1
280             );
281         if (x > 0) prob = 1e14 - prob;
282         return prob;
283     }
```

Listing 3.1: `Math::ncdf()`

Recommendation Use correct parameter in `ncdf()/cdf()` routines.

Status The issue has been fixed by this commit: `869bc1a`.

3.2 Improved Corner Case Handling in getTimeWeightedAverageTick()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VolOracle`
- Category: Time and State [5]
- CWE subcategory: CWE-682 [3]

Description

The `VolOracle` contract provides a public `twap()` function to calculate the time weighted average price for a specified `Uniswap V3` pool for the pool's entire `Uniswap` observation period. This function uses the `Oracle` datas obtained from the `Uniswap V3` pool to compute the time weighted average tick, and further to get the time weighted average price.

While examining the computation, we notice the `getTimeWeightedAverageTick()` implementation can be improved. To elaborate, we show below its code snippet. When there is only one `Oracle` entry stored in the `Uniswap V3` pool, the given `duration` equals `0`, leading to a `division by zero` in the calculation of $\_timeWeightedAverageTick = int24(tickCumulativesDelta/duration)$ (line 193).

Note that this does not result in an incorrect return value from `getTimeWeightedAverageTick()`, but does cause the function to revert unnecessarily when the above corner case occurs. Meanwhile, it is worth mentioning that if there is a `divide by zero`, the execution or the contract call will be thrown by executing the `INVALID` opcode, which by design consumes all of the gas in the initiating call. This is different from `REVERT` and has the undesirable result in causing unnecessary monetary loss.

To address this particular corner case, We suggest to validate the input argument of `duration`, and make `timeWeightedAverageTick()` well defined over its all possible inputs.

```
183     /**
184      * @notice Gets the time weighted average tick
185      * @return timeWeightedAverageTick is the tick which was resolved to be the time-
             weighted average
186      */
187     function getTimeWeightedAverageTick(
188         int56 olderTickCumulative,
189         int56 newerTickCumulative,
190         uint32 duration
191     ) private pure returns (int24 timeWeightedAverageTick) {
192         int56 tickCumulativesDelta = newerTickCumulative - olderTickCumulative;
193         int24 _timeWeightedAverageTick = int24(tickCumulativesDelta / duration);
194 
195         // Always round to negative infinity
196         if (tickCumulativesDelta < 0 && (tickCumulativesDelta % duration != 0))
197             _timeWeightedAverageTick--;
198 
199         return _timeWeightedAverageTick;
200     }
```

Listing 3.2: `VolOracle::getTimeWeightedAverageTick()`

Recommendation Add the argument check for `duration` to avoid the above-mentioned `divide by zero` issue.

Status The issue has been fixed by this commit: `869bc1a`.

3.3 Inconsistency Between Document and Implementation

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DSMath`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

There are a few inconsistent or misleading descriptions in the given repository, which bring unnecessary hurdles to understand and/or maintain the library implementation.

A few example comments can be found in the preceding descriptions of the `DSMath::wdiv()/rdiv()` routines. Specifically, it has been documented that the return result of `DSMath::wdiv()` routine will round to zero if $x * y < WAD/2$ and the return result of `DSMath::rdiv()` routine will round to zero if $x * y < RAY/2$.

However, the current implementation shows that the return result of `DSMath::wdiv()` routine will round to zero if $x * WAD < y/2$ and the return result of `DSMath::rdiv()` routine will round to zero if $x * RAY < y/2$. For comparison, we show the code snippet of `DSMath::wdiv()/rdiv()` below.

```
62      //rounds to zero if x*y < WAD / 2
63      function wdiv(uint256 x, uint256 y) internal pure returns (uint256 z) {
64          z = add(mul(x, WAD), y / 2) / y;
65      }
66
67      //rounds to zero if x*y < RAY / 2
68      function rdiv(uint256 x, uint256 y) internal pure returns (uint256 z) {
69          z = add(mul(x, RAY), y / 2) / y;
70      }
```

Listing 3.3: `DSMath::wdiv()/rdiv()`

Recommendation Ensure the consistency between documents and implementation.

Status The issue has been fixed by this commit: `869bc1a`.

4 Conclusion

In this audit, we have analyzed the RVOL implementation, which is a set of Solidity libraries and tools that utilizes the Uniswap V3 to make on-chain volatility data accessible. Its goal is to help builders to build on-chain volatility indices, query realized volatility information, and price derivatives and options with on-chain data. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

[5] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[6] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.

[10] UCLA. Normal Distribution Function. https://www.math.ucla.edu/~tom/distributions/normal.html?