SMART CONTRACT AUDIT REPORT

for

Revert Compoundor

Prepared By: Xiaomi Huang

PeckShield

August 16, 2022

Document Properties

Client	Revert Finance
Title	Smart Contract Audit Report
Target	Revert Compoundor
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 16, 2022	Jing Wang	Final Release
1.0-rc	August 11, 2022	Jing Wang	Release Candidate

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 Revert Compoundor design document and related smart contract source code, 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 Revert Compoundor

Revert Compoundor is a protocol to automate compounding of liquidity provider fees for positions in Uniswap v3. It automates the fee-compounding process by allocating a fixed percentage of the fees to incentivize protocol participants (compoundors) to pay for the gas costs incurred in performing the required contract interactions.The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Revert Compoundor

Item	Description
Issuer	Revert Finance
Website	https://revert.finance/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 16, 2022

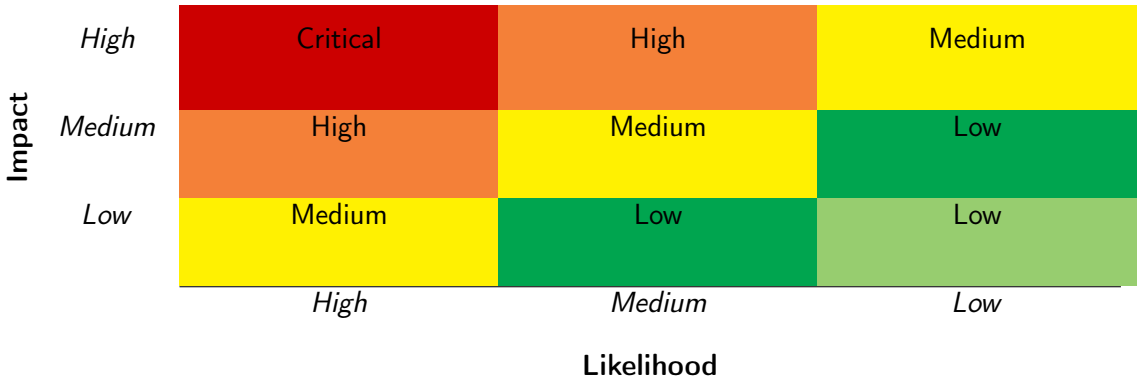
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the Revert Compoundor protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/revert-finance/compoundor/blob/main/contracts/Compoundor.sol (cee8623)

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 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

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

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 Revert Compoundor 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	0	
Low	2	■■
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 2 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Revert Compoundor Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Immutable States If Only Set at Constructor()	Coding Practices	Confirmed
PVE-002	Low	Incompatibility with Deflationary Tokens	Business Logic	Confirmed
PVE-003	Low	Trust Issue of Admin Keys	Coding Practices	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 Immutable States If Only Set at Constructor()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Compoundor`
- Category: Coding Practices [5]
- CWE subcategory: CWE-561 [2]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variables defined in `Compoundor`. If there is no need to dynamically update these key state variables, e.g., `nonfungiblePositionManager`, `swapRouter`, `weth` and `factory`, they can be declared as `immutable` for gas efficiency.

```
26    contract Compoundor is ICompoundor, ReentrancyGuard, Ownable, Multicall {
27        ...
28        // wrapped native token address
29        address override public weth;

31        // uniswap v3 components
32        IUniswapV3Factory public override factory;
33        INonfungiblePositionManager public override nonfungiblePositionManager;
```

```
34        ISwapRouter public override swapRouter;
```

Listing 3.1: Compoundor.sol

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status This issue has been confirmed. The team clarifies that they will adpat it for next version.

3.2 Incompatibility with Deflationary Tokens

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

Description

In the `Revert Compoundor` protocol, the contract is designed to collect fees from the `Uniswap v3` positions and add liquidity to the position for compounding. In particular, one interface, i.e., `autoCompound()`, accepts asset transfer-in and records the owner's token balance. Another interface, i.e, `_withdrawBalanceInternal()`, allows the user to withdraw the asset. For the above two operations, i.e., `autoCompound()` and `_withdrawBalanceInternal()`, the contract makes the use of `safeTransfer()` routine to transfer assets into or out of contract. This routine works as expected with standard ERC20 tokens: namely the contract's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
129     function autoCompound(AutoCompoundParams memory params)
130         override
131         external
132         nonReentrant
133         returns (uint256 reward0, uint256 reward1, uint256 compounded0, uint256
                compounded1)
134     {
135         require(ownerOf[params.tokenId] != address(0), "!found");
136
137         AutoCompoundState memory state;
138
139         // collect fees
140         (state.amount0, state.amount1) = nonfungiblePositionManager.collect(
141             INonfungiblePositionManager.CollectParams(params.tokenId, address(this),
                    type(uint128).max, type(uint128).max)
142         );
143         ...
```

```
144         _setBalance(state.tokenOwner, state.token0, state.amount0.sub(compounded0).sub(
                state.amount0Fees));
145         _setBalance(state.tokenOwner, state.token1, state.amount1.sub(compounded1).sub(
                state.amount1Fees));
146         ...
147     }

149     function _withdrawBalanceInternal(address token, address to, uint256 balance,
            uint256 amount) internal {
150         require(amount <= balance, "amount>balance");
151         accountBalances[msg.sender][token] = accountBalances[msg.sender][token].sub(
                amount);
152         emit BalanceRemoved(msg.sender, token, amount);
153         SafeERC20.safeTransfer(IERC20(token), to, amount);
154         emit BalanceWithdrawn(msg.sender, token, to, amount);
155     }
```

Listing 3.2: `Compoundor::autoCompound()/_withdrawBalanceInternal()`

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, such as `autoCompound()` and `_withdrawBalanceInternal()`, 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 contract 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 will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `collect()` is called 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 `Compoundor` for support.

Recommendation Check the balance before and after the `collect()` call to ensure the book-keeping amount is accurate.

Status This issue has been confirmed. The team clarifies that they won't support deflationary tokens.

3.3 Trust Issue of Admin Keys

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

Description

In the Revert Compoundor protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the protocol-wide operations (e.g., setting parameters). 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 owner account and its related privileged accesses in current contract.

To elaborate, we show the setTWAPConfig() routine from the Compoundor() contract. This function allows the owner account to set the max amount of ticks the current tick may differ from the Oracle TWAP tick to allow swaps when protect the swap from price manipulation.

```
81      function setTWAPConfig(uint32 _maxTWAPTickDifference, uint32 _TWAPSeconds) external
            override onlyOwner {
82          maxTWAPTickDifference = _maxTWAPTickDifference;
83          TWAPSeconds = _TWAPSeconds;
84          emit TWAPConfigUpdated(msg.sender, _maxTWAPTickDifference, _TWAPSeconds);
85      }
```

Listing 3.3: Compoundor::setTWAPConfig()

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged owner 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 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 team has transferred ownership of the compoundor contracts to a 24 hour time-lock controlled by a team multisig.

4 Conclusion

In this audit, we have analyzed the Revert Compoundor design and implementation. Revert Compoundor is a protocol to automate compounding of liquidity provider fees for positions in Uniswap v3. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.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.