



# SMART CONTRACT AUDIT REPORT

for

## Rabbit Finance - Leveraged Bull/Bear



Prepared By: Yiqun Chen

PeckShield  
August 6, 2021

## Document Properties

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

## Version Info

Version	Date	Author(s)	Description
1.0	August 6, 2021	Xuxian Jiang	Final Release
1.0-rc	August 5, 2021	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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Rabbit Finance . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	6
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Lack of Emitting Meaningful Events . . . . .	11
3.2	Redundant Code Removal . . . . .	12
3.3	Lack Of Slippage Control in Liquidation . . . . .	13
<b>4</b>	<b>Conclusion</b>	<b>15</b>
	<b>References</b>	<b>16</b>

# 1 | Introduction

Given the opportunity to review the design document and related source code of the the leveraged bull/bear support in the Rabbit Finance 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 Rabbit Finance

Rabbit Finance is a leveraged trading protocol based on deposit and borrowing functions. The key features of Rabbit Finance include deposit and borrowing, leveraged yield farming and leveraged trading, and will support options, synthetic assets, and NFT trading functions in the future. It is designed as an evolutionary improvement of earlier offerings with the goal of continuously improving the utilization of deposit users' funds. New application scenarios of borrowing and leverage farming are continuously discovered and explored. The audited implementation adds new leveraged bull/bear support in addition to earlier improvements with additional workers and strategies.

The basic information of Rabbit Finance is as follows:

Table 1.1: Basic Information of Rabbit Finance

Item	Description
Issuer	Rabbit Finance
Website	<a href="https://rabbitfinance.io/">https://rabbitfinance.io/</a>
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 6, 2021

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

- [https://github.com/RabbitFinanceProtocol/rabbit\\_finance\\_bsc.git](https://github.com/RabbitFinanceProtocol/rabbit_finance_bsc.git) (2f240a4)

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

- [https://github.com/RabbitFinanceProtocol/rabbit\\_finance\\_bsc.git](https://github.com/RabbitFinanceProtocol/rabbit_finance_bsc.git) (7f59041)

## 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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 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 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.

## 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 1.3: 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

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit



Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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 `Rabbit` 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	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 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 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 Audit Findings of Rabbit Finance Protocol

ID	Severity	Title	Category	Status
PVE-001	Informational	<a href="#">Lack of Emitting Meaningful Events</a>	Coding Practices	Fixed
PVE-002	Low	<a href="#">Redundant Code Removal</a>	Coding Practices	Fixed
PVE-003	Low	<a href="#">Lack Of Slippage Control in Liquidation</a>	Time and State	Confirmed

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 Lack of Emitting Meaningful Events

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: FixedEUR
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [1]

#### 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 `Governable` contract as an example. While examining the events that reflect the `gov` dynamics, we notice there is a lack of emitting related events that reflect important state changes. Specifically, when the `gov` is being changed, there is no respective event being emitted to reflect the transfer of `gov` (line 664).

```
42     function setPendingGovernor(address _pendingGovernor) external onlyGov {  
43         pendingGovernor = _pendingGovernor;  
44     }  
45  
46     /// @dev Accept to become the new governor. Must be called by the pending governor.  
47     function acceptGovernor() external {  
48         require(msg.sender == pendingGovernor, 'not the pending governor');  
49         pendingGovernor = address(0);  
50         governor = msg.sender;  
51     }
```

Listing 3.1: `Governable::setPendingGovernor()/acceptGovernor()`

**Recommendation** Properly emit the related `NewGov` event when the `gov` is being updated.

**Status** The issue has been fixed in the following commit: 7f59041.

## 3.2 Redundant Code Removal

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [1]

### Description

The `Rabbit` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Ownable`, to facilitate its code implementation and organization. For example, the `LeverageGoblin` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `LeverageGoblin` contract, it contains a number of events that are emitted to reflect various protocol dynamics. However, there is an event `Reinvest` that is not used in current protocol. Therefore, this event can be safely removed.

```

700 contract LeverageGoblin is Governable, ReentrancyGuardUpgradeSafe, Goblin {
701     /// @notice Libraries
702     using SafeToken for address;
703     using SafeMath for uint;
704
705     /// @notice Events
706     event Reinvest(address indexed caller, uint reward, uint bounty);
707     event AddShare(uint indexed id, uint share);
708     event RemoveShare(uint indexed id, uint share);
709     event Liquidate(uint256 indexed id, address bullTokenAddress, uint256 bullAmount,
710                    address debtToken, uint256 liqAmount);
711     ...
711 }
```

Listing 3.2: The `LeverageGoblin` contract

In addition, the contract `SimpleERCFund` is inherited from another contract `Operator`, which seems unnecessary as there is no operator-related operations.

**Recommendation** Consider the removal of the redundant code to simplify existing contracts.

**Status** The issue has been fixed in the following commit: 7f59041.

### 3.3 Lack Of Slippage Control in Liquidation

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [4]
- CWE subcategory: CWE-682 [2]

#### Description

The leveraged bull/bear support in the Rabbit Finance protocol comes with a new strategy `_LeverageStrategyLiquidate()` to facilitate potential liquidation operations. Our analysis shows that the current implementation lacks an effective slippage control that occurs in liquidation.

```

846     function liquidate(uint256 id, address /*user*/, address borrowToken) override
      external onlyOperator nonReentrant {
847         bool isBorrowBNB = borrowToken == address(0);
848         // 1. Convert the position back to bull tokens and use liquidate strategy.
849         _removeShare(id, borrowToken);

851         address reversedToken = getReversedToken(borrowToken);
852         uint256 reversedBalance;
853         if(reversedToken != address(0)){
854             reversedBalance = reversedToken.myBalance();
855             reversedToken.safeTransfer(address(liqStrat), reversedBalance);
856         }else{
857             reversedBalance = address(this).balance;
858         }

860         liqStrat.execute{value:address(this).balance}(address(0),
861             borrowToken, uint256(0), uint256(0), abi.encode(uint256(0)));

863         // 2. transfer borrowToken and user want back to goblin.
864         uint256 tokenLiquidate;
865         if (isBorrowBNB){
866             tokenLiquidate = address(this).balance;
867             SafeToken.safeTransferETH(msg.sender, tokenLiquidate);
868         } else {
869             tokenLiquidate = borrowToken.myBalance();
870             borrowToken.safeTransfer(msg.sender, tokenLiquidate);
871         }
872         emit Liquidate(id, reversedToken, reversedBalance, borrowToken, tokenLiquidate);
873     }

```

Listing 3.3: LeverageGoblin::liquidate()

To elaborate, we show above the related `liquidate()` routine. We notice the resulting token swap for liquidation is routed to `IUniswapV2Router02` and the actual swap operation `swapExactTokensForTokens`

(c) essentially does not specify any effective restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the TWAP or time-weighted average price of UniswapV2. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

**Recommendation** Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

**Status** The issue has been confirmed by the team.



## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the leveraged bull/bear support in the Rabbit protocol, which is a leveraged trading protocol based on deposit and borrowing functions. The system continues the innovative design and makes it distinctive and valuable when compared with current lending/yield farming offerings. The current code base is well organized and 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-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [2] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [3] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [4] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.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](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [7] PeckShield. PeckShield Inc. <https://www.peckshield.com>.