



SMART CONTRACT AUDIT REPORT

for

WhiteRock Finance



Prepared By: Xiaomi Huang

PeckShield
December 31, 2024

Document Properties

Client	Whiterock Finance
Title	Smart Contract Audit Report
Target	WhiteRock Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 31, 2024	Xuxian Jiang	Final Release
1.0-rc	December 16, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About WhiteRock Finance	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	9
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Revisited _maxWalletSize Enforcement in White/arbWhite	12
3.2	Improved Validation of Protocol Parameters	13
3.3	Improved Constructor Logic in White/arbWhite	14
3.4	Trust Issue of Admin Keys	15
4	Conclusion	17
	References	18

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the WhiteRock 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 WhiteRock Finance

WhiteRock Finance is a real-world asset protocol designed to tokenize economic rights to traditional financial assets like stocks, bonds, property, options, and derivatives. By bridging traditional finance with blockchain, WhiteRock unlocks global liquidity and simplifies access to equities, fixed income, and other investments. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of WhiteRock Finance

Item	Description
Name	Whiterock Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 31, 2024

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

- <https://github.com/ZephyrETH/hacken-whiterock.git> (62a9a40)

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

- <https://github.com/ZephyrETH/hacken-whiterock.git> (0ceb143, 3e2ac34)

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

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

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

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.

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.

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.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the WhiteRock Finance 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	2	
Low	2	
Informational	0	
Total	4	

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 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key WhiteRock Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited <code>_maxWalletSize</code> Enforcement in <code>White/arbWhite</code>	Business Logic	Resolved
PVE-002	Low	Improved Validation of Protocol Parameters	Coding Practices	Resolved
PVE-003	Low	Improved Constructor Logic in <code>White/arbWhite</code>	Coding Practices	Resolved
PVE-004	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 Revisited `_maxWalletSize` Enforcement in `White/arbWhite`

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `White`, `arbWhite`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

To facilitate the token decentralization, the `WhiteRock` protocol has a key parameter `_maxWalletSize`, which enforces the maximum holding balance for each individual wallet. In the process of examining the enforcement of this specific parameter, we notice current logic may need to revisit.

To elaborate, we show below the `_transfer()` function. As the name indicates, this function implements the basic transfer functionality with necessary fee charged, except for excluded entities (e.g., owner, gateway and router). However, the enforcement of `_maxWalletSize` can be validated at the end of the function, not with the pre-condition when the token is being sold (line 265).

```

256     function _transfer(address from, address to, uint256 amount) private {
257         require(from != address(0), "ERC20: transfer from the zero address");
258         require(to != address(0), "ERC20: transfer to the zero address");
259         uint256 taxAmount = 0;
260         if (from != owner() && to != owner()) {
261             require(!blacklisted[from] && !blacklisted[to]);
262             if (from != customGatewayAddress && to != customGatewayAddress) {
263                 taxAmount = amount.mul(tax).div(100);
264
265                 if (from == uniswapV2Pair && to != address(uniswapV2Router) && !
                     _isExcludedFromFee[to]) {
266                     require(balanceOf(to) + amount <= _maxWalletSize, "Exceeds the
                         maxWalletSize.");
267                 }
268             }
269         }
270     }

```

```

271     _balances[from] = _balances[from].sub(amount);
272     _balances[to] = _balances[to].add(amount.sub(taxAmount));
273
274     emit Transfer(from, to, amount.sub(taxAmount));
275     if (taxAmount > 0) {
276         _balances[address(this)] = _balances[address(this)].add(taxAmount);
277         emit Transfer(from, address(this), taxAmount);
278     }
279 }

```

Listing 3.1: White::_transfer()

Recommendation Revise the above logic to properly enforce the `_maxWalletSize` argument.

Status This issue has been fixed in the following commit: 5d82b39.

3.2 Improved Validation of Protocol Parameters

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: White, arbWhite
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The WhiteRock protocol is no exception. Specifically, if we examine the White contract, it has defined a number of protocol-wide risk parameters, such as `tax` and `_maxWalletSize` (Section 3.1). In the following, we show the corresponding routines that allow for their changes.

```

281     function updateFee(uint256 tax_) external onlyOwner {
282         tax = tax_;
283     }
284     ...
285     function updateMaxWalletSize(uint256 maxWalletSize_) external onlyOwner {
286         _maxWalletSize = maxWalletSize_;
287     }

```

Listing 3.2: White::updateFee() and White::updateMaxWalletSize()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, current `updateFee()` can be improved by further enforcing the following requirement, i.e., `require(tax_ <= 100, "Exceeds max tax fee.");`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status This issue has been fixed in the following commit: 5d82b39.

3.3 Improved Constructor Logic in White/arbWhite

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: White, arbWhite
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, the arbWhite contract is instantiated as a proxy with actual logic contracts in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers()`; Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```

129     function initialize() public initializer {
130         tax = 0;
131         _maxWalletSize = _tTotal.div(50);
132
133         __Ownable_init();
134
135         _taxWallet = payable(_msgSender());
136         _balances[_msgSender()] = 1;
137
138         _isExcludedFromFee[owner()] = true;
139         _isExcludedFromFee[address(this)] = true;
140         _isExcludedFromFee[_taxWallet] = true;
141
142         _name = unicode"WhiteRock";
143         _symbol = "WHITE";
144
145         emit Transfer(address(0), _msgSender(), 1);
146     }

```

Listing 3.3: arbWhite::initialize()

Moreover, the above `initialize()` routine can be improved by also initializing the inherited contract `UUPSUpgradeable` with the call of `__UUPSUpgradeable_init()`.

Recommendation Improve the above-mentioned constructor routine in affected contracts.

Status This issue has been fixed in the following commit: [5d82b39](#).

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [\[4\]](#)
- CWE subcategory: CWE-287 [\[2\]](#)

Description

In the WhiteRock Finance protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters and pause/unpause the protocol). 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 account and their related privileged accesses in current contracts.

```

281     function updateFee(uint256 tax_) external onlyOwner {
282         tax = tax_;
283     }
284
285     function updateName(string memory name_) external onlyOwner {
286         _name = name_;
287     }
288
289     function updateSymbol(string memory symbol_) external onlyOwner {
290         _symbol = symbol_;
291     }
292
293     function setArbitrumGateway(address address_) external onlyOwner {
294         customGatewayAddress = address_;
295     }
296
297     function setArbitrumRouter(address address_) external onlyOwner {
298         routerAddress = address_;
299     }
300
301     function withdrawETH(uint256 amount) external onlyOwner {
302         _taxWallet.transfer(amount);
303     }
304

```

```
305     function withdrawTokens() external onlyOwner {  
306         IERC20(address(this)).transfer(msg.sender, balanceOf(address(this)));  
307     }
```

Listing 3.4: Example Privileged Functions in the `White` Contract

If the privileged `owner` account is managed by a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 issue has been mitigated as the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `WhiteRock Finance` protocol, which is a real-world asset protocol designed to tokenize economic rights to traditional financial assets like stocks, bonds, property, options, and derivatives. By bridging traditional finance with blockchain, `WhiteRock` unlocks global liquidity and simplifies access to equities, fixed income, and other investments. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. <https://cwe.mitre.org/data/definitions/837.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>.