



SMART CONTRACT AUDIT REPORT

for

Lista ListaStakeManager



Prepared By: Xiaomi Huang

PeckShield
May 7, 2024

Document Properties

Client	Lista
Title	Smart Contract Audit Report
Target	Lista ListaStakeManager
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 7, 2024	Xuxian Jiang	Final Release
1.0-rc	May 1, 2024	Xuxian Jiang	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	4
1.1	About ListaStakeManager	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	Timely Credit Contract Update in ListaStakeManager	11
3.2	Simplified compoundRewards() Logic in ListaStakeManager	12
3.3	Trust Issue Of Admin Keys	13
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `ListaStakeManager` 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 ListaStakeManager

This audit covers the specific stake manager contract, i.e., `ListaStakeManager`. The stake manager allows users to stake native `BNB` to mint `SLisBnb`. The staked `BNB` will be later delegated (via authorized `bots`) to chosen validators. The contract also takes care of the stake reward collection and distribution, which can be later claimed by staking users. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Audited Contracts

Item	Description
Target	Lista <code>ListaStakeManager</code>
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 7, 2024

In the following, we show the Git repository of reviewed files and the commit hash values used in this audit. This audit only covers the `ListaStakeManager` contract.

- <https://github.com/lista-dao/synclub-contracts.git> (87189aa)

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

- <https://github.com/lista-dao/synclub-contracts.git> (7cca610)

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 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 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

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

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) [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.

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 Logics	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 `ListaStakeManager` 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 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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Lista ListaStakeManager Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely Credit Contract Update in ListaStakeManager	Business Logic	Resolved
PVE-002	Informational	Simplified compoundRewards() Logic in ListaStakeManager	Coding Practice	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 Timely Credit Contract Update in ListaStakeManager

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

Description

To facilitate the user interaction, the `ListaStakeManager` contract has the built-in support of managing multiple validators. In the process of examining the multi-validator management, we notice current implementation can be improved.

To elaborate, we show below the `initialize()` routine that configures the very first validator. However, it comes to our attention that it does not timely synchronizes with the associated credit contracts, i.e., `syncCredits(bscValidator, false)`. Moreover, the stake manager contract also provides a privileged routine `setBSCValidator()` so that we can flexibly set the operator address of the validator initially delegated to. However, there is a need to remove the credit contract associated with the replaced validator before adding the credit contract associated with the new validator.

```
85     function initialize(  
86         address _slisBnb,  
87         address _admin,  
88         address _manager,  
89         address _bot,  
90         uint256 _synFee,  
91         address _revenuePool,  
92         address _validator  
93     ) external override initializer {  
94         __AccessControl_init();  
95         __Pausable_init();  
  
97         require(  

```

```

98         ((_slisBnb != address(0)) &&
99         (_admin != address(0)) &&
100         (_manager != address(0)) &&
101         (_validator != address(0)) &&
102         (_revenuePool != address(0)) &&
103         (_bot != address(0))),
104         "zero address provided"
105     );
106     require(_synFee <= TEN_DECIMALS, "_synFee must not exceed (100%)");

108     _setRoleAdmin(BOT, DEFAULT_ADMIN_ROLE);
109     _setupRole(DEFAULT_ADMIN_ROLE, _admin);
110     _setupRole(BOT, _bot);

112     manager = _manager;
113     slisBnb = _slisBnb;
114     bscValidator = _validator;
115     synFee = _synFee;
116     revenuePool = _revenuePool;

118     emit SetManager(_manager);
119     emit SetBSCValidator(bscValidator);
120     emit SetRevenuePool(revenuePool);
121     emit SetSynFee(_synFee);
122 }

```

Listing 3.1: ListaStakeManager::initialize()

Recommendation Strengthen the above-mentioned routines to timely update credit contracts associated with supported validators.

Status The issue has been resolved as the team considers it is part of the design.

3.2 Simplified compoundRewards() Logic in ListaStakeManager

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ListaStakeManager
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The ListaStakeManager contract has a flexible support to compound stake rewards. While examining the reward-compounding logic, we notice the related implementation can be improved.

Specifically, we show below the implementation of the related routine, i.e., compoundRewards(). It has a rather straightforward logic in collecting the reward and adding the reward (after fee deduction)

back to the delegate. We notice it validates the presence of new reward (line 916), which can be revised as `require(totalBNBInValidators > totalDelegated + totalFee, "No new fee to compound");`.

```

907     function compoundRewards()
908     external
909     override
910     whenNotPaused
911     onlyRole(BOT)
912     {
913         require(totalDelegated > 0, "No funds delegated");
914
915         uint256 totalBNBInValidators = getTotalBnbInValidators();
916         require(totalBNBInValidators >= totalDelegated && totalBNBInValidators -
            totalDelegated > totalFee, "No new fee to compound");
917         uint256 totalProfit = totalBNBInValidators - totalDelegated - totalFee;
918         uint256 fee = 0;
919         if (synFee > 0) {
920             fee = totalProfit * synFee / TEN_DECIMALS;
921             totalFee += fee;
922         }
923         uint256 totalUserProfit = totalProfit - fee;
924
925         totalDelegated += totalUserProfit;
926
927         emit RewardsCompounded(fee);
928     }

```

Listing 3.2: ListaStakeManager::compoundRewards()

Recommendation Improve the above `compoundRewards()` routine with a simplified logic.

Status This issue has been fixed in the following commits: 02a24fa and 092f34c.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: ListaStakeManager
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `ListaStakeManager` contract, there is a privileged account (with the assigned `DEFAULT_ADMIN_ROLE`) that plays a critical role in governing and regulating the contract-wide operations (e.g., manage validator, update fee, and execute privileged operations). In the following, we show the representative functions potentially affected by the privilege of this account.

```

556     function setBSCValidator(address _address)
557         external
558         override
559         onlyManager
560     {
561         require(bscValidator != _address, "Old address == new address");
562         require(_address != address(0), "zero address provided");
563
564         bscValidator = _address;
565         syncCredits(bscValidator, false);
566
567         emit SetBSCValidator(_address);
568     }
569     ...
570     function setSynFee(uint256 _synFee)
571         external
572         override
573         onlyRole(DEFAULT_ADMIN_ROLE)
574     {...}
575
576     function setRedirectAddress(address _address)
577         external
578         override
579         onlyRole(DEFAULT_ADMIN_ROLE)
580     {...}
581
582
583     function setRevenuePool(address _address)
584         external
585         override
586         onlyRole(DEFAULT_ADMIN_ROLE)
587     {...}
588
589     function whitelistValidator(address _address)
590         external
591         override
592         onlyRole(DEFAULT_ADMIN_ROLE)
593     {...}

```

Listing 3.3: Example Privileged Operations in `ListaStakeManager`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive vault parameters, which directly undermines the assumption of the vault design.

In the meantime, the vault contract makes use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

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 The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the privileged account.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the specific stake manager contract, i.e., `ListaStakeManager`. The stake manager allows users to stake native `BNB` to mint `SLisBnb`. The staked `BNB` will be later delegated (via authorized `bots`) to chosen validators. The contract also takes care of the stake reward collection and distribution, which can be later claimed by staking users. identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-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>.