SMART CONTRACT AUDIT REPORT

for

Synclub Liquid Staking

Prepared By: Xiaomi Huang

PeckShield

June 30, 2023

Document Properties

Client	Synclub
Title	Smart Contract Audit Report
Target	LSD BNB
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	June 30, 2023	Xuxian Jiang	Post Release #1
1.0	May 10, 2023	Xuxian Jiang	Final Release
1.0-rc	May 6, 2023	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

Given the opportunity to review the design document and related smart contract source code of the Synclub's Liquid Staking 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 Synclub

Synclub plans to implement the Liquid Staking module for BNB, which allows users to stake their BNB and acquire rewards. And at the same time, they could get an interest-bearing CoD, named SnBNB, which could be used as collateral in many DeFi protocols to borrow assets, and could be used by LPs to yield higher rewards. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of LSD BNB

Item	Description
Target	LSD BNB
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 30, 2023

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

- https://github.com/agiledev624/synclub-contracts.git (b559bd7)

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

- https://github.com/agiledev624/synclub-contracts.git (8703458)

And here is the list of deployed addresses:

- `Deployer` = 0x0403f7d7cfb1cd871ee762236bd96e6b602fffdb
- `ProxyAdmin` = 0x8ce30a8d13d6d729708232aa415d7da46a4fa07b

- `SnBnb Proxy` = 0xB0b84D294e0C75A6abe60171b70edEb2EFd14A1B
- `SnBnb Logic` = 0xaF8DC8A33B60173693590BD867d571D88501CF81

- `SnStakeManager Proxy` = 0x1adB950d8bB3dA4bE104211D5AB038628e477fE6
- `SnStakeManager Logic` = 0xf1068e9393DC7C07bD127e5765aDFa9116762C9c

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*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

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 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

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
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.

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 Synclub's Liquid Staking protocol, implementation. 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	3	🟦🟦🟦
Low	1	🟦
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 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 3 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key LSD BNB Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Adherence of The Checks-Effects-Interactions Pattern	Coding Practices	Resolved
PVE-002	Medium	Improved Validation in requestWithdraw()	Business Logic	Resolved
PVE-003	Medium	Revisited undelegate() Logic	Business Logic	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 Suggested Adherence of The Checks-Effects-Interactions Pattern

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SnStakeManager`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the `Uniswap/Lendf.Me` hack [10].

We notice an occasion where the `checks-effects-interactions` principle is violated. Using the `SnStakeManager` as an example, the `delegate()` function (see the code snippet below) is provided to externally call an external contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 150) starts before effecting the update on internal state (line 151), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `delegate()` function. Note that there may be no harm caused to current protocol. However, it is still suggested to follow the known `checks-effects-interactions` best practice.

```
134    function delegate()
```

```
        external
        payable
        override
        whenNotPaused
        onlyRole(BOT)
        returns (uint256 _amount)
    {
        uint256 relayFee = IStaking(nativeStaking).getRelayerFee();
        uint256 relayFeeReceived = msg.value;
        _amount = amountToDelegate - (amountToDelegate % TEN_DECIMALS);

        require(relayFeeReceived >= relayFee, "Insufficient RelayFee");
        require(availableReserveAmount >= reserveAmount, "Insufficient Reserve Amount");
        require(_amount + reserveAmount >= IStaking(nativeStaking).getMinDelegation(), "Insufficient Deposit Amount");
        // delegate through native staking contract
        IStaking(nativeStaking).delegate{value: _amount + msg.value + reserveAmount}(bcValidator, _amount);
        amountToDelegate = amountToDelegate - _amount;
        totalDelegated += _amount;

        emit Delegate(_amount);
        emit DelegateReserve(reserveAmount);
    }
```

Listing 3.1: `SnStakeManager::delegate()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice.

Status The issue has been fixed by this commit: `458e01a`.

3.2 Improved Validation in requestWithdraw()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SnStakeManager`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The `Synclub's Liquid Staking` protocol has a core `SnStakeManager` contract that allows users to stake and unstake. While reviewing the current unstaking logic, we notice the related implementation needs to be improved.

To elaborate, we show below the core `requestWithdraw()` routine. As the name indicates, this routine allows staking users to unstake. However, it restricts the available `totalBnbToWithdraw` to be no larger than `totalDelegated`, which somehow excludes the `amountToDelegate` amount. This may put unnecessary restrictions on staking users. For example, suppose the following case of having one single user `Alice` who just deposited 100 `BNB` (via `deposit()`) and there is no `delegate()` yet. `Alice` found out that it is now impossible to withdraw the staked 100 `BNB` back even the funds are not actually delegated yet.

```
    function requestWithdraw(uint256 _amountInSnBnb)
        external
        override
        whenNotPaused
    {
        require(_amountInSnBnb > 0, "Invalid Amount");

        totalSnBnbToBurn += _amountInSnBnb;
        uint256 totalBnbToWithdraw = convertSnBnbToBnb(totalSnBnbToBurn);
        require(
            totalBnbToWithdraw <= totalDelegated,
            "Not enough BNB to withdraw"
        );

        userWithdrawalRequests[msg.sender].push(
            WithdrawalRequest({
                uuid: nextUndelegateUUID,
                amountInSnBnb: _amountInSnBnb,
                startTime: block.timestamp
            })
        );

        IERC20Upgradeable(snBnb).safeTransferFrom(
            msg.sender,
            address(this),
            _amountInSnBnb
        );
        emit RequestWithdraw(msg.sender, _amountInSnBnb);
    }
```

Listing 3.2: `SnStakeManager::requestWithdraw()`

Recommendation Revisit the unstake logic to ensure user funds can be fully withdrawn in all possible cases.

Status The issue has been fixed by this commit: `458e01a`.

3.3 Revisited undelegate() Logic

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SnStakeManager
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Synclub's Liquid Staking protocol provides an undelegate() function to allow the authorized bots to undelegate the requested amount of BNB for withdrawal. While examining the undelegate logic, we notice a required validation needs to be revisited.

In particular, we show below its implementation. It has a rather straightforward logic in calculating and validating the requested withdrawal amount in current batch and then making the actual undelegate request. It comes to our attention that the validation enforces the following requirement: require(reserveAmount >= IStaking(nativeStaking).getDelegated(address(this), bcValidator) (line 288), which needs to be revised as require(_amount + reserveAmount <= IStaking(nativeStaking).getDelegated(address(this), bcValidator)). The purpose here is to ensure the total delegated amount should be sufficient to satisfy this withdrawal request.

```
270     function undelegate()
271         external
272         payable
273         override
274         whenNotPaused
275         onlyRole(BOT)
276         returns (uint256 _uuid, uint256 _amount)
277     {
278         uint256 relayFee = IStaking(nativeStaking).getRelayerFee();
279         uint256 relayFeeReceived = msg.value;

281         require(relayFeeReceived >= relayFee, "Insufficient RelayFee");

283         _uuid = nextUndelegateUUID++; // post-increment : assigns the current value
                first and then increments
284         uint256 totalSnBnbToBurn_ = totalSnBnbToBurn; // To avoid Reentrancy attack
285         _amount = convertSnBnbToBnb(totalSnBnbToBurn_);
286         _amount -= _amount % TEN_DECIMALS;

288         require(reserveAmount >= IStaking(nativeStaking).getDelegated(address(this),
                bcValidator),
289              "Insufficient Delegate Amount");
290         require(
291             _amount + reserveAmount >= IStaking(nativeStaking).getMinDelegation(),
292             "Insufficient Withdraw Amount"
```

```
        );

        uuidToBotUndelegateRequestMap[_uuid] = BotUndelegateRequest({
            startTime: 0,
            endTime: 0,
            amount: _amount,
            amountInSnBnb: totalSnBnbToBurn_
        });

        totalDelegated -= _amount;
        totalSnBnbToBurn = 0;

        ISnBnb(snBnb).burn(address(this), totalSnBnbToBurn_);

        // undelegate through native staking contract
        IStaking(nativeStaking).undelegate{value: msg.value}(bcValidator, _amount +
            reserveAmount);

        emit UndelegateReserve(reserveAmount);
    }
```

Listing 3.3: `SnStakeManager::undelegate()`

Recommendation Revise the above-mentioned routine to properly handle the undelegate logic.

Status The issue has been fixed by this commit: `458e01a`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `SnStakeManager`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Synclub's Liquid Staking` protocol, there is a privileged `manager` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters and assign other roles). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setReserveAmount(uint256 amount) external override onlyManager {
        reserveAmount = amount;
        emit SetReserveAmount(amount);
    }

    function proposeNewManager(address _address) external override onlyManager {
        require(manager != _address, "Old address == new address");
        require(_address != address(0), "zero address provided");

        proposedManager = _address;

        emit ProposeManager(_address);
    }

    function setBotRole(address _address) external override onlyManager {
        require(_address != address(0), "zero address provided");

        _setupRole(BOT, _address);

        emit SetBotRole(_address);
    }

    function revokeBotRole(address _address) external override onlyManager {
        require(_address != address(0), "zero address provided");

        _revokeRole(BOT, _address);

        emit RevokeBotRole(_address);
    }

    /// @param _address - Beck32 decoding of Address of Validator Wallet on Beacon Chain with ‘0x‘ prefix
    function setBCValidator(address _address)
        external
        override
        onlyManager
    {
        require(bcValidator != _address, "Old address == new address");
        require(_address != address(0), "zero address provided");

        bcValidator = _address;

        emit SetBCValidator(_address);
    }
```

Listing 3.4: Example Privileged Operations in SnStakeManager

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be 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 system parameters, which directly undermines the assumption of the protocol design.

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 by the team. The team intends to have a multi-sig account to manage the admin key.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Synclub's Liquid Staking` protocol, which allows users to stake their `BNB` and acquire rewards. And at the same time, users could get an interest-bearing `CoD`, named `SnBNB`, which could be used as collateral in many DeFi protocols to borrow assets, and could be used by `LPs` to yield higher rewards. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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.

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.