SMART CONTRACT AUDIT REPORT

for

SyncBank IDO

Prepared By: Xiaomi Huang

PeckShield

June 8, 2022

Document Properties

Client	SyncBank
Title	Smart Contract Audit Report
Target	SyncBank IDO
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 8, 2022	Xuxian Jiang	Final Release
1.0-rc1	June 6, 2023	Xuxian Jiang	Release Candidate #1

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 SyncBank IDO 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 SyncBank

SyncBank is a lending protocol, built on the zkSync Era scaling solution for Ethereum Layer 2. Its non-custodial lending platform gives users full control over their funds and offers competitive interest rates through a decentralized market that eliminates intermediaries. The audited WhitelistSbSale is the IDO contract code and only whitelisted users can participate in the IDO. The basic information of the audited contract is as follows:

Table 1.1: Basic Information of SyncBank IDO

Item	Description
Name	SyncBank
Website	https://syncbank.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 8, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of sub-directories and this audit covers only the whitelistSbSale contract.

- https://github.com/syncbank/syncbank.git (3977ae1)

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

- https://github.com/syncbank/syncbank.git (78aa475)

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).

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 the 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.

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

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 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. 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 `WhitelistSbSale` 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	2	■■
Low	1	■
Informational	0	
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 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key SyncBank IDO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistent commitmentCap Update Upon Price Adjustment	Business Logic	Resolved
PVE-002	Medium	Possible MAXIMUM_COMMIT_ETH/WHALE_MAXIMUM_COMMIT_ETH Bypass	Business Logic	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides 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 Inconsistent commitmentCap Update Upon Price Adjustment

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WhitelistSbSale`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `WhitelistSbSale` contract implements the essential `IDO` functions for the `SyncBank` protocol. As part of the `IDO` functions, it enforces a number of protocol-wide parameters. For example, it allows the owner or deployer to specify the sale price as well as the intended `commitmentCap`, which is computed as `commitmentCap = totalTokens.mul(_tokenPrice).div(1e18)`. In the meantime, the protocol allows the owner to dynamically update the token price. However, our analysis shows the related `commitmentCap` is not updated when the token price is changed.

To elaborate, we show below the `setTokenPrice()` function. It implements a straightforward logic in updating the token sale price. While it indeed validates the sale has not started, it does not accordingly update the intended `commitmentCap`.

```
    function setTokenPrice(uint256 _tokenPrice) external onlyOwner {
        require(_tokenPrice > 0, "SbSale: tokenPrice must be greater than 0");
        require(marketStatus.commitmentsTotal == 0, "SbSale: Sale has already started");
        tokenPrice = _tokenPrice;
    }
```

Listing 3.1: `WhitelistSbSale::setTokenPrice()`

Recommendation Revise the above `setTokenPrice()` function to properly update the `commitmentCap` as well. In the meantime, it is suggested to emit the related events to reflect the changes of these protocol-wide parameters.

Status The issue has been fixed by this commit: `78aa475`.

3.2 Possible MAXIMUM_COMMIT_ETH/WHALE_MAXIMUM_COMMIT_ETH Bypass

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `WhitelistSbSale`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `WhitelistSbSale` contract is no exception. Specifically, if we examine the `WhitelistSbSale` contract, it has defined a number of protocol-wide risk parameters, such as `MAXIMUM_COMMIT_ETH` and `WHALE_MAXIMUM_COMMIT_ETH`. These two parameters indicate the maximum commited ETHs that may be allowed in the `IDO` during the limit period. However, while examining the enforcement of these parameters, we notice they might be bypassed.

In the following, we use the first parameter `MAXIMUM_COMMIT_ETH` as the example and show the related `_addCommitment()` routine that violates its enforcement. Notice this routine is invoked when a new commitment is being made. Suppose it is still in the limit period, and the new commitment, if successful, will make the following condition true: `limitCommitPeriod()&& _commitment.add (marketStatus.commitmentsTotal)> marketInfo.commitmentCap` (line 266). In this case, the variable `canCommitmentAmount` computes the maximum allowed commit from the current user. However, it is currently computed as `marketInfo.commitmentCap.sub(marketStatus.commitmentsTotal)`, which fails to consider earlier commitments that may be made by the same user. As a result, an user may commit multiple times to exceed the `MAXIMUM_COMMIT_ETH` cap.

```
254     function _addCommitment(address payable _addr, uint256 _commitment, address
            _referral) private {
255         require(block.timestamp >= marketInfo.startTime && block.timestamp <= marketInfo
                .endTime, "SbSale: outside presale hours");
256         require(!marketStatus.finalized, "SbSale: has been finalized");
257 
258         if (limitCommitPeriod() && _commitment.add(marketStatus.commitmentsTotal) <=
                marketInfo.commitmentCap) {
259             require(commitments[_addr] < MAXIMUM_COMMIT_ETH, "SbSale: exceed maximum
                    commit eth");
260             if (commitments[_addr].add(_commitment) >= MAXIMUM_COMMIT_ETH) {
```

```
                uint256 _canCommitmentAmount = MAXIMUM_COMMIT_ETH.sub(commitments[_addr
                    ]);
                uint256 _refundAmount = _commitment.sub(_canCommitmentAmount);
                _commitment = _canCommitmentAmount;
                _safeTransferETH(_addr, _refundAmount);
            }
        } else if (limitCommitPeriod() && _commitment.add(marketStatus.commitmentsTotal)
             > marketInfo.commitmentCap){
            uint256 _canCommitmentAmount = marketInfo.commitmentCap.sub(marketStatus.
                commitmentsTotal);
            if (_canCommitmentAmount > MAXIMUM_COMMIT_ETH) {
                _canCommitmentAmount = MAXIMUM_COMMIT_ETH;
            }
            uint256 _refundAmount = _commitment.sub(_canCommitmentAmount);
            _commitment = _canCommitmentAmount;
            _safeTransferETH(_addr, _refundAmount);
        } else {
            if (_commitment.add(marketStatus.commitmentsTotal) > marketInfo.
                commitmentCap) {
                uint256 _canCommitmentAmount = marketInfo.commitmentCap.sub(marketStatus
                    .commitmentsTotal);
                uint256 _refundAmount = _commitment.sub(_canCommitmentAmount);
                _commitment = _canCommitmentAmount;
                _safeTransferETH(_addr, _refundAmount);
            }
        }

        uint256 newCommitment = commitments[_addr].add(_commitment);
        require(newCommitment >= MINIMUM_COMMIT_ETH, "SbSale: less than minimum
            commitment amount");
        commitments[_addr] = newCommitment;
        marketStatus.commitmentsTotal = marketStatus.commitmentsTotal.add(_commitment);
        emit AddedCommitment(_addr, _commitment, _referral);
    }
```

Listing 3.2: WhitelistSbSale::_addCommitment()

Recommendation Revise the above routine to ensure the `MAXIMUM_COMMIT_ETH` parameter is properly honored. Similarly, the same issue occurs to the `_addWhaleCommitment()` routine regarding the `WHALE_MAXIMUM_COMMIT_ETH` enforcement.

Status The issue has been fixed by this commit: `78aa475`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: WhitelistSbSale
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the WhitelistSbSale contract, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the IDO-wide operations (e.g., configure various settings, adjust the token price, as well as update the whitelist/whale list). It also has the privilege to control or govern the flow of assets within the IDO contracts. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setTokenPrice(uint256 _tokenPrice) external onlyOwner {
        require(_tokenPrice > 0, "SbSale: tokenPrice must be greater than 0");
        require(marketStatus.commitmentsTotal == 0, "SbSale: Sale has already started");
        tokenPrice = _tokenPrice;
    }

    function setTreasury(address payable _treasury) external onlyOwner {
        require(_treasury != address(0), "SbSale: treasury is the zero address");
        treasury = _treasury;
        emit SaleTreasuryUpdated(_treasury);
    }

    function setWhitelist(address _addr, bool isWhiteUser) external onlyOwner {
        whitelist[_addr] = isWhiteUser;
    }

    function setWhitelists(address[] calldata _addrs, bool isWhiteUser) external
        onlyOwner {
        for (uint256 i = 0; i < _addrs.length; i++) {
            whitelist[_addrs[i]] = isWhiteUser;
        }
    }

    function setWhale(address _addr, bool isWhale) external onlyOwner {
        whales[_addr] = isWhale;
    }

    function setWhales(address[] calldata _addrs, bool isWhale) external onlyOwner {
        for (uint256 i = 0; i < _addrs.length; i++) {
            whales[_addrs[i]] = isWhale;
        }
```

```
    }
```

Listing 3.3: Example Privileged Operations in `WhitelistSbSale`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make these privileges explicit to the participating users.

Status The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `WhitelistSbSale` contract, an IDO contract for the `SyncBank` protocol. `SyncBank` is a lending protocol, built on the `zkSync Era` scaling solution for `Ethereum Layer 2`. Its non-custodial lending platform gives users full control over their funds and offers competitive interest rates through a decentralized market that eliminates intermediaries. The audited `WhitelistSbSale` contract ensures only whitelisted users can participate in the IDO. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[3] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.