SMART CONTRACT AUDIT REPORT

for

SyncSwap

Prepared By: Xiaomi Huang

PeckShield
July 17, 2023

Document Properties

Client	SyncSwap
Title	Smart Contract Audit Report
Target	SyncSwap
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 17, 2023	Luck Hu	Final Release
1.0-rc1	July 13, 2023	Luck Hu	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 `SyncSwap` 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 audited protocol 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 SyncSwap

`SyncSwap` is a seamless decentralized exchange (`DEX`) on the `zkSync Era` network. Powered by zero-knowledge technology, `SyncSwap` targets to build a one-stop-shop `DeFi` hub that is totally seamless and easy to use with innovative features. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of SyncSwap

Item	Description
Name	SyncSwap
Website	http://syncswap.xyz
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 17, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. The audit scope covers the following contracts: 1) `./contracts/SyncSwapVoter.sol` 2) `./contracts/SyncSwapGauge.sol` 3) `./contracts/SyncSwapBribe.sol` 4) `./contracts/VortexToken.sol` 5) `./contracts/VortexDividends.sol`.

- https://github.com/syncswap/vortex-contracts (40f1523)

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

- https://github.com/syncswap/vortex-contracts (234a369)

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 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

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) [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 1.4: 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 `SyncSwap` 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	4	■■■■
Low	0	
Undetermined	1	■
Total	5	

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 4 medium-severity vulnerabilities and 1 undetermined issue.

Table 2.1: Key SyncSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Voter Account in SyncSwapVoter::vote()	Business Logic	Fixed
PVE-002	Undetermined	Suggested _beforeTokenTransfer() Call in ERC20Permit2::transferFrom()	Business Logic	Fixed
PVE-003	Medium	New Voting Power Validation in VortexToken::_deallocate()	Business Logic	Fixed
PVE-004	Medium	Improper Staked Amount in SyncSwapGauge::supplyRewards()	Business Logic	Fixed
PVE-005	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 Voter Account in SyncSwapVoter::vote()

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

Description

In the `SyncSwap` protocol, the `SyncSwapVoter` contract provides users with the ability to cast their voting powers among pools. In particular, the protocol operator can vote on behalf of the user. While examining the vote casting on behalf of the user by the operator, we notice the vote is wrongly casted for the operator itself with the voting power of the operator.

To elaborate, we show below the related code snippet of the `SyncSwapVoter::vote()` function. As the name indicates, it is used by the user to cast its voting power among different pools with different weights. In particular, if the function is called by the operator, it can vote on behalf of the user using the voting power of the user.

```
    function vote(
        address account,
        bool updateLastVoted,
        address[] calldata _pools,
        uint[] calldata weights,
        uint totalWeight
    ) external override nonReentrant ensuresVoting(account, updateLastVoted) {
        _castVotes(msg.sender, _pools, weights, totalWeight);
    }

    function _castVotes(address _account, address[] memory _pools, uint[] memory
        _weights, uint _totalWeight) private returns (uint _usedVotes) {
        // Updates `rewardPerVote` first with current total votes.
        uint _rewardPerVote = _updateRewardPerVote();
        uint _minDistributeAmount = minDistributeAmount;
```

```
381
382         // Resets previous votes first.
383         _resetVotes(_account, _rewardPerVote, _minDistributeAmount);
384
385         uint _votingPowers = IVoteAggregator(voteAggregator).getVotingPowers(_account);
386         uint n = _pools.length;
387         require(n <= maxVotesCount, "Too many votes");
388         uint _usedWeight;
389         ...
390     }
```

Listing 3.1: `Interaction::withdraw()`

However, it comes to our attention that it always uses the `msg.sender` (line 374) as the account to invoke the `_castVotes()` routine, where it will cast vote for the given user account. As a result, if the `msg.sender` is an operator, it will cast vote for the operator itself while not the given user account. Our analysis shows that it should use the input `account` parameter (line 368) as the `_account` parameter (line 377) to invoke the `_castVotes()` routine.

Recommendation Use the input `account` parameter of the `vote()` routine as the `_account` parameter to invoke the `_castVotes()` routine.

Status This issue has been fixed in the following commit: 55eee62d.

3.2 Suggested _beforeTokenTransfer() Call in ERC20Permit2::transferFrom()

- ID: PVE-002
- Severity: Undetermined
- Likelihood: High
- Impact: N/A
- Target: `ERC20Permit2`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `SyncSwap` protocol, the `VortexToken` contract allows the `SYNC` token to be converted to `veSYNC` and redeemed back with vesting duration. The `veSYNC` is used to calculate the voting power in the voter. The transfer of `veSYNC` tokens is controlled by a `Whitelister` contract and only whitelisted transfer is allowed. While reviewing the transfer of `veSYNC`, we notice the `Whitelister` check can be bypassed via the `ERC20Permit2::transferFrom()` routine.

To elaborate, we show below the related code snippets of the `ERC20Permit2::transferFrom()`/`VortexToken::_beforeTokenTransfer()` routines. The `VortexToken` contract inherits from the `ERC20Permit2` contract where the token transfer logic is implemented. The token transfer check is performed in the

VortexToken::_beforeTokenTransfer() routine. However, in the ERC20Permit2::transferFrom() routine, it does not properly invoke the _beforeTokenTransfer() routine. As a result, the transfer check is bypassed and the token transfer can be successfully triggered.

Our analysis shows that it should invoke the _beforeTokenTransfer() routine in the ERC20Permit2::transferFrom() to apply the transfer check.

```
function transferFrom(address _from, address _to, uint _amount) public virtual override
    returns (bool) {
    uint256 _allowed = allowance[_from][msg.sender]; // Saves gas for limited approvals.
    if (_allowed != type(uint).max) {
        allowance[_from][msg.sender] = _allowed - _amount;
    }

    balanceOf[_from] -= _amount;
    // Cannot overflow because the sum of all user balances can't exceed the max uint256
        value.
    unchecked {
        balanceOf[_to] += _amount;
    }

    emit Transfer(_from, _to, _amount);
    return true;
}
```

Listing 3.2: ERC20Permit2::transferFrom()

```
    function _beforeTokenTransfer(address from, address to, uint amount) internal view
        override {
        require(
            from == address(0)  from == address(this)  to == address(this)
            IVortexTokenWhitelister(whitelister).isTransferWhitelisted(msg.sender, from,
                to, amount),
            "Not allowed"
        );
    }
```

Listing 3.3: VortexToken::_beforeTokenTransfer()

Status This issue has been fixed in the following commit: 55eee62d.

3.3 New Voting Power Validation in VortexToken::_deallocate()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VortexToken`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `SyncSwap` protocol, the voting power of a user can be calculated based on the `veSYNC` token amount owned by the user and the `veSYNC` token amount that is allocated to the submodules by the user. When the voting power changes, there is a need to check if the new voting power is able to accommodate the used votes. While examining the deallocation of `veSYNC` from a submodule, we notice there is a lack of validation for the new voting power after the possible excess amount of `veSYNC` is burnt.

In the following, we show the code snippet of the `VortexToken::_deallocate()` routine. As the name indicates, it is used to deallocate the input `amount` of `veSYNC` from the submodule and receive the input `deallocateAmount` of `veSYNC` from the submodule. After deducting the excess amount of `veSYNC` (line 340) and the deallocation fee (line 345) from the `amount`, the user finally receives `(deallocateAmount - fee)` amount of `veSYNC`. So the voting power of the user decreases by `amount - (deallocateAmount - fee)`.

However, we notice it does not properly check if the new voting power is big enough to back the used votes. As a result, the used votes are still effective though the new voting power becomes smaller than the used votes.

```
329    function _deallocate(address account, address submodule, uint amount, uint
           deallocateAmount) private returns (uint) {
330        require(amount != 0);
331        require(_submodules.contains(submodule), "Not submodule");
332
333        // 1 Updates allocated amount.
334        _userBalances[account].allocatedAmount -= amount;
335        userAllocations[account][submodule] -= amount;
336
337        totalAllocatedAmount -= amount;
338
339        // 2 Calculates excess and fee amount.
340        uint excessAmount = amount - deallocateAmount; // `fee + excessAmount`, see
               below
341        uint fee;
342
```

```
        uint _deallocationFee = deallocationFees[submodule];
        if (_deallocationFee != 0) {
            fee = deallocateAmount * _deallocationFee / 1e5;
            excessAmount += fee;
        }

        // 3 Unlocks user's veSYNC.
        _transfer(address(this), account, deallocateAmount - fee);

        // 4 Burns excess and fee amount.
        if (excessAmount != 0) {
            totalSyncAmount -= excessAmount;

            _burn(address(this), excessAmount);
            IERC20(syncToken).safeTransfer(treasury, excessAmount);

            unchecked {
                totalExcessAmount += excessAmount; // overflow is allowed
            }
        }

        // 5 Notifies the base module.
        address _baseModule = baseModule;
        if (_baseModule != address(0)) {
            IVortexBaseModule(_baseModule).onDeallocate(msg.sender, account, submodule,
                amount, deallocateAmount, fee);
        }

        emit Deallocate(msg.sender, account, submodule, amount, deallocateAmount, fee);
        return deallocateAmount - fee;
    }
```

Listing 3.4: `VortexToken::_deallocate()`

Recommendation Properly check if the new voting power is big enough to back the used votes after the excess amount of `veSYNC` tokens are burnt, or recast the new voting power per the used weights.

Status This issue has been fixed in the following commit: 55eee62d.

3.4 Improper Staked Amount in SyncSwapGauge::supplyRewards()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SyncSwapGauge`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `SyncSwap` protocol, the `SyncSwapGauge` contract allows users to stake share tokens to earn rewards. The staked amount will be adjusted with the voter boost and the adjusted staked amount will be used to calculate the earned rewards. While reviewing the update of the rewards in the `SyncSwapGauge::supplyRewards()` routine, we notice it uses the original total staked amount, not the adjusted total staked amount.

In the following, we show the related code snippet of the `SyncSwapGauge::supplyRewards()` routine. As the name indicates, it is used to supply new rewards to the `SyncSwapGauge` contract. After the new rewards are supplied, the reward rate will be updated. Before the new reward rate is taken into effect, the accrued reward for each share (`rewardPerShare`) should be updated per the current reward rate by calling `_update(totalStaked, address(0), 0, false)` (line 389).

```
374     function supplyRewards(address token, uint amount) external override
            onlyVoterOrOwner notEmergency returns (bool, uint) {
375         // 1 Checks for reward rate.
376         uint _rewardDuration = IVoter(voter).rewardDuration();
377         uint _rewardRate = amount / _rewardDuration; // `suppliedRewardRate` here
378         if (_rewardRate == 0) {
379             return (false, 0);
380         }

382         // 2 Checks for reward token.
383         require(isRewardToken[token], "Not reward");

385         // 3 Transfers reward tokens from sender.
386         amount = _safeTransferFrom(token, msg.sender, amount);

388         // 4 Updates rewards in current reward rate.
389         _update(totalStaked, address(0), 0, false);

391         // 5 Updates reward rate and amount.
392         RewardData storage data = _rewardData[token];
393         uint _rewardAmount = data.rewardAmount;

395         if (_rewardAmount == 0) {
```

```
            // 5-A Starts a new round of reward.
            data.rewardRate = _rewardRate;
            data.rewardAmount = amount;
        } else {
            // 5-B Extends the current round of reward.
            uint newRewardAmount = _rewardAmount + amount;
            _rewardRate = newRewardAmount / _rewardDuration; // reused as `newRewardRate`

            data.rewardRate = _rewardRate;
            data.rewardAmount = newRewardAmount;
        }

        // 6 Updates snapshot.
        unchecked {
            totalRewards[token] += amount; // overflow is allowed
        }
        rewardSnapshots[token].push(RewardSnapshot(block.timestamp, _rewardAmount,
            amount, _rewardRate));

        // 7 Notifies rewarder.
        address _rewarder = rewarder;
        if (_rewarder != address(0)) {
            IRewarder(_rewarder).onSupplyRewards(msg.sender, token, amount);
        }

        emit SupplyRewards(msg.sender, token, _rewardAmount, amount, _rewardRate);
        return (true, amount);
    }
```

Listing 3.5: SyncSwapGauge::supplyRewards()

However, it comes to our attention that it uses the original total staked amount, i.e., `totalStaked`, to update the `rewardPerShare`, not the adjusted total staked amount, i.e., `adjustedTotalStaked`. As a result, the calculated `rewardPerShare` is not accurate. Our analysis shows that it should use the `adjustedTotalStaked` to update the `rewardPerShare`, i.e., `_update(adjustedTotalStaked, address(0), 0, false)`.

Recommendation Revisit the `SyncSwapGauge::supplyRewards()` routine to update the `rewardPerShare` per the `adjustedTotalStaked`.

Status This issue has been fixed in the following commit: 55eee62d.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `SyncSwap` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., pause/remove a gauge). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `SyncSwapVoter` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the `owner` is privileged to add/remove operator who can vote on behalf of the user, pause/unpause/remove a pool, set the claimable reward amount, set the committed reward amount, set the total reward amount, set the total reward cap, set the voting delay between two vote operations from the same user, set the `whitelister` which provides the whitelisted pools that can be used to create gauges, set the `bribeFactory` which is used to create bribe for each pool, set the `voteAggregator` which provides the voting power of each user, set the reward rate, set any address as a gauge, etc.

```
    function setOperator(address operator, bool status) external onlyOwner {
        require(operator != address(0), "Invalid target");

        if(status) {
            _operators.add(operator);
        } else {
            _operators.remove(operator);
        }

        emit SetOperator(operator, status);
    }

    function setPoolPaused(address pool, bool isPaused) external override onlyOwner {
        require(isPoolPaused[pool] != isPaused, "Already set");
        isPoolPaused[pool] = isPaused;
        emit SetPoolPaused(pool, isPaused);
    }

    function setClaimable(address pool, uint _claimable) external override onlyOwner {
        _poolData[pool].claimable = _claimable;
        emit SetClaimable(pool, _claimable);
    }

    function setCommittedRewardAmount(uint _committedRewardAmount) external onlyOwner {
```

```
        committedRewardAmount = _committedRewardAmount;
        emit SetCommittedRewardAmount(_committedRewardAmount);
    }

    function setTotalRewardAmount(uint _totalRewardAmount) external onlyOwner {
        totalRewardAmount = _totalRewardAmount;
        emit SetTotalRewardAmount(_totalRewardAmount);
    }

    function setTotalRewardCap(uint _totalRewardCap) external onlyOwner {
        totalRewardCap = _totalRewardCap;
        emit SetTotalRewardCap(_totalRewardCap);
    }
    ...

    function setHook(address _hook) external onlyOwner {
        hook = _hook;
        emit SetHook(_hook);
    }

    function setBoost(address _boost) external onlyOwner {
        boost = _boost;
        emit SetBoost(_boost);
    }

    function setMaxVotesCount(uint _maxVotesCount) external onlyOwner {
        maxVotesCount = _maxVotesCount;
        emit SetMaxVotesCount(_maxVotesCount);
    }
```

Listing 3.6: Example Privileged Operations in `SyncSwapVoter`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account 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 Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And 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 mitigated as the team confirmed the `owner` will be transferred to multi-sig and eventually the governance contract in the future.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `SyncSwap` protocol, which is a seamless decentralized exchange (`DEX`) on the `zkSync Era` network. Powered by zero-knowledge technology, `SyncSwap` targets to build a one-stop-shop `DeFi` hub that is totally seamless and easy to use with innovative features. The current code base is well structured and neatly organized. Those 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-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.