SMART CONTRACT AUDIT

ZOKYO.

June 10th 2022 | v. 1.0

PASS

Zokyo's Security Team has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

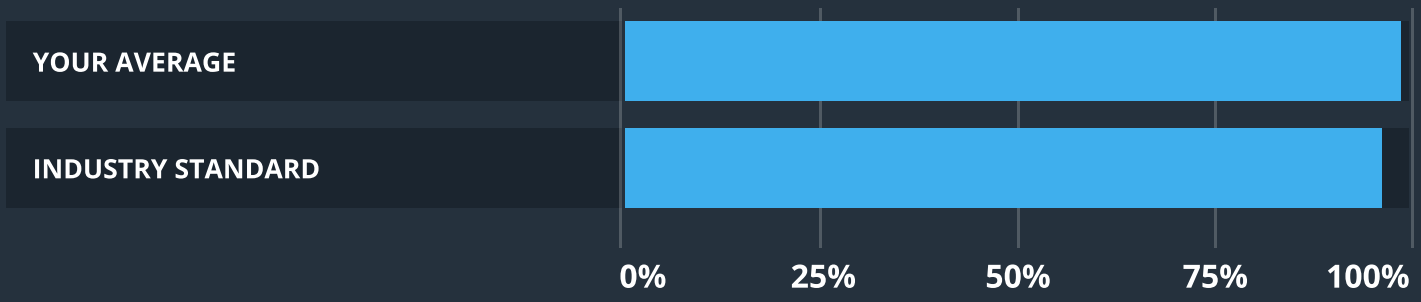
TECHNICAL SUMMARY

This document outlines the overall security of the TrustSwap smart contracts, evaluated by Zokyo's Blockchain Security team.

The scope of this audit was to analyze and document the TrustSwap smart contract codebase for quality, security, and correctness.

Contract Status

Testable Code



The testable code is 99%, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the Ethereum network's fast-paced and rapidly changing environment, we at Zokyo recommend that the TrustSwap team put in place a bug bounty program to encourage further and active analysis of the smart contract.

TABLE OF CONTENTS

AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the TrustSwap repository.

Repository - https://github.com/trustswap/locked-staking-contracts

Initial commit: b7653562957337c29ed649c3f0558c1e065a5808
Last reviewed commit - 45f24a41fb305f98737983bddb491095893d45a5

Within the scope of this audit Zokyo auditors have reviewed the following contract(s):

- LockedStaking.sol

Throughout the review process, care was taken to ensure that the contract:

- Implements and adheres to existing standards appropriately and effectively;
- Documentation and code comments match logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices in efficient use of resources, without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the latest vulnerabilities;
- Whether the code meets best practices in code readability, etc.

Zokyo's Security Team has followed best practices and industry-standard techniques to verify the implementation of TrustSwap smart contracts. To do so, the code is reviewed line-by-line by our smart contract developers, documenting any issues as they are discovered. Part of this work includes writing a unit test suite using the Truffle testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

1	Due diligence in assessing the overall code quality of the codebase.	3	Testing contract logic against common and uncommon attack vectors.
2	Cross-comparison with other, similar smart contracts by industry leaders.	4	Thorough, manual review of the codebase, line-by-line.

EXECUTIVE SUMMARY

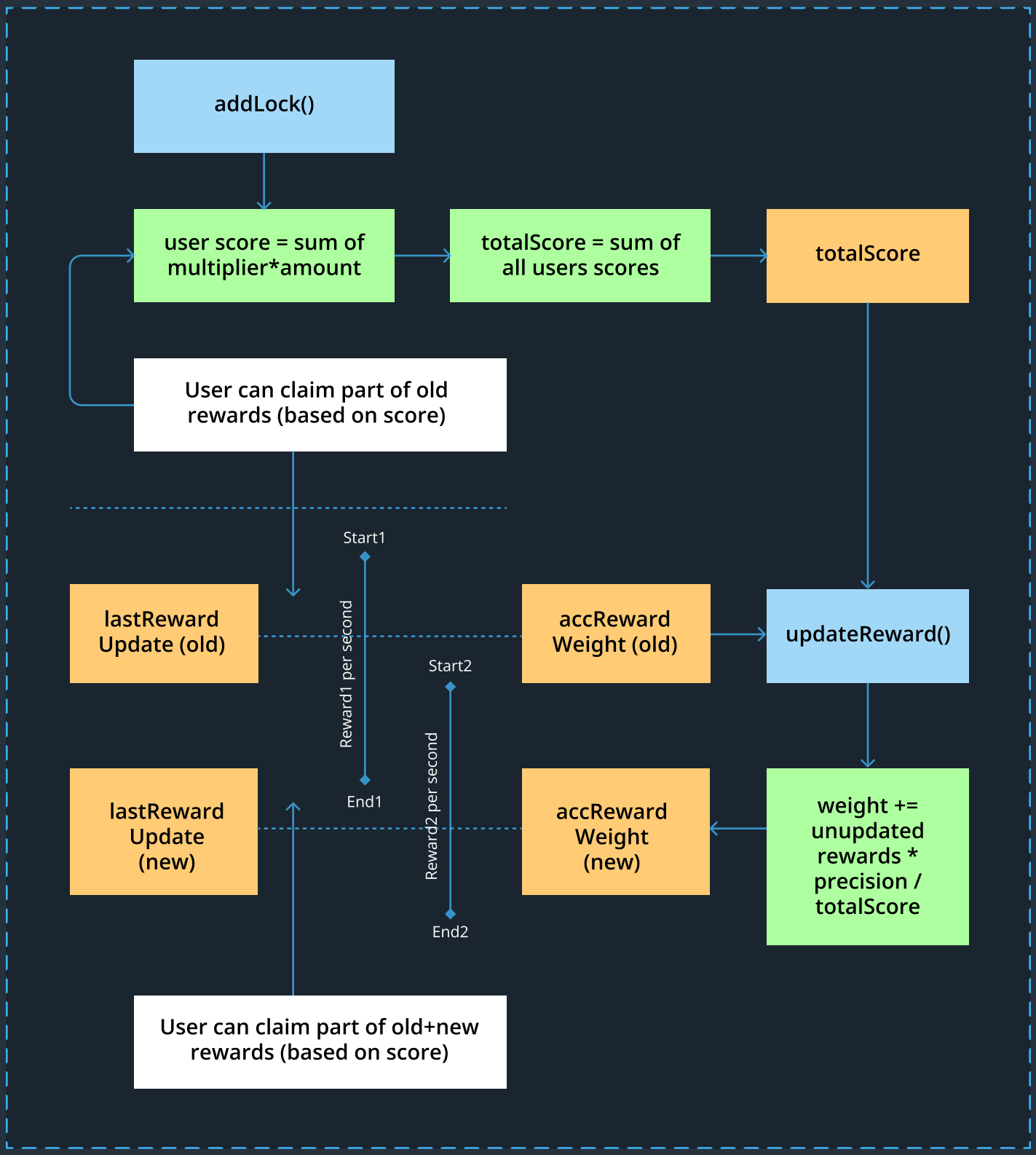
During the audit, staking protocol of TrustSwap was carefully checked. Staking is designed to lock swap tokens and earn reward in the form of same swap token. Staking functionality was verified to operate correctly. It was checked, that users' funds cannot get stuck on contract, users are able to withdraw their funds after lock period.

Additional functionalities of the contract, such as autocompounding and updating of locks were also successfully checked by auditors.

Overall, no critical issues were found. All found issues were verified and resolved by TrustSwap team. The code is well-organised and tested. Code was additionally tested by auditors to verify that all functions work as intended.

PROTOCOL OVERVIEW

LOCKS CREATION AND REWARD EARNING



STRUCTURE AND ORGANIZATION OF DOCUMENT

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. Issues tagged “Verified” contain unclear or suspicious functionality that either needs explanation from the Customer’s side or it is an issue that the Customer disregards as an issue. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

MEDIUM | VERIFIED

SafeERC20 should be used.

SwapToken is used with regular transfer() and transferFrom() methods from the OpenZeppeling IERC20 interface. Though, in general, the ERC20 token may have not return value for these methods (see USDT implementation) and lead to the failure of calls and contract blocking. Thus the SafeERC20 library should be used or the token should be verified to implement to implement return values for transfer() and transferFrom() methods

Recommendation:
Verify the swapToken used to implement the OpenZeppelin version of IERC20 interface or use SafeERC20 library.

From client: Only swap tokens would be used, which interface returns true or false.

MEDIUM | RESOLVED

Iteration through storage array.

LockedStaking.sol: getRewardsWeight(), line 362.
Iteration through a storage array might consume more gas than allowed per transaction, thus function will always revert, preventing the protocol from correct operation. Issue is marked as medium, because only the owner can add reward tokens and control an amount of elements in an array.

Recommendation:
Either add an ability to remove expired rewards or add a maximum limit of rewards, which can be added to array “rewards”.

Post audit: Maximum limit was added.

LOW | RESOLVED

Users are able to create an unlimited amount of locks.

LockedStaking.sol: function addLock().
There is no limitation on the amount of locks for each user. Each lock is added to the user's array of locks. In case user has created too many locks, this can cause to revert when iterating through all user's locks (function getUserClaimable(): Line 256, function getUsersTotalScore(): Line 273) and prevent user from withdrawing his rewards or locked tokens. Issue is marked as low, since there is no impact to overall protocol and other users.

Recommendation:
Either add a limitation on amount of locks per user, or add a minimum boundary for locked amount when creating locks(So that user is not able to add a lot of lock with low dusting locked amount(like several weis for example)).

INFORMATIONAL | RESOLVED

Variables lack validation.

LockedStaking.sol: function addReward().
Time-dependent parameters should be validated. It is a common practice to verify that time-dependent parameters are greater than the current block's timestamp and that the "end" parameter is greater than the "start" parameter of any time frame. Although, the correctness of "start" and "end" variables are verified indirectly in line 103 (when subtract start from end), such error might be unclear. And also it still leaves room for the adding rewards with the timeline in the past - and in such way it will be left unnoticeable for the rewards weight update.
Variable "amountPerSecond" should be validated not to be zero. It may be validated indirectly in the IERC.transferFrom() call (through 0 token transfer validation), but such a check may be missing for the token. Issue is marked as info since only the owner is able to call this function and also rewards can be updated with function updateReward().

Recommendation:
Validate that parameter "start" is greater than block.timestamp, "end" is greater than "start" and "amountPerSecond" is greater than 0.

INFORMATIONAL | RESOLVED

Public functions can be marked as external.

Functions initialize(), getRewardsLength(), getLockInfo(), getUserLocks(), getLockLength(), getRewards(), addReward(), updateReward(), addLock(), compound(), updateLockAmount(), updateLockDuration(), getUserClaimable(), claim(), unlock(), getUnlockedRewards() can be marked external, since they are not called within other functions of the contract. Usage of external functions decrease gas spendings compared to public functions.

Recommendation:
Mark the following functions as external.

INFORMATIONAL | RESOLVED

getUserClaimable() can be simplified.

getUserClaimable() function can be simplified with the usage of internal calculateUserClaimable() function.

Recommendation:
The function can be simplified.

INFORMATIONAL | RESOLVED

Undocumented logic.

addLock() function creates new lock, though it is undocumented, that this function also provides lock of the claimable rewards (if the user has such). Neither function name nor parameters give info about this approach. It will be better for the code quality to add the natspec about such behavior or/and add a scheme to the documentation of the “autocompounding” approach

Recommendation:
Increase documentation quality about the “autocompounding” approach.

LockedStaking.sol	
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by TrustSwap team

As part of our work assisting TrustSwap in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Truffle testing framework.

Tests were based on the functionality of the code, as well as a review of the TrustSwap contract requirements for details about issuance amounts and how the system handles these.

```
Contract: LockedStaking
    ✓ multiplier function (393ms)
    ✓ getters (207ms)
    ✓ create new pool (466ms)
    ✓ create multiple pools (718ms)
    ✓ update pool (7333ms)
    ✓ add lock (207ms)
    ✓ update lock amount (418ms)
    ✓ update lock duration (449ms)
    ✓ unlock (329ms)
    ✓ claim (277ms)
    ✓ 10 users + 2 pools (2130ms)
  11 passing (26s)
```

FILE	% STMTS	% BRANCH	% FUNCS
LockedStaking.sol	72.11	43.24	87.5
All files	**72.11**	**43.24**	**87.5**

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Secured team

As part of our work assisting TrustSwap in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Truffle testing framework.

Tests were based on the functionality of the code, as well as a review of the TrustSwap contract requirements for details about issuance amounts and how the system handles these.

Contract: LockedStaking

LockedStaking initializing:
- ✓ If SwapToken address = 0 error ZeroAddress()
- ✓ If precision = 0 days error ZeroPrecision()

Normal work getters:
- ✓ Correct .getRewardsLength()
- ✓ Correct .getLockInfo()
- ✓ Correct .getUserLocks()
- ✓ Correct .getLockLength()
- ✓ Correct .getRewards()
- ✓ Correct .getUserClaimable()
- ✓ Correct .getRewardsWeight() if current time < reward start
- ✓ Correct .getRewardsWeight() if current time > reward start

Adding and update rewards frames:
- ✓ Correct adding reward frame
- ✓ Should revert if amount per second equal 0
- ✓ Should revert if time parameters are invalid
- ✓ Should revert if maximum amount of rewards added
- ✓ If sender has no enough balance when adding error TransferFailed()
- ✓ Correct update reward frame
- ✓ Crash if updating reward not exist
- ✓ If sender has no enough balance when updating error TransferFailed()

Work with locks (create, update, compound, unlock, claim):
- ✓ If amount == 0 error AmountIsZero()
- ✓ If duration < 30 days error DurationOutOfBounds()
- ✓ If duration > 1825 days error DurationOutOfBounds()
- ✓ Correct adding lock
- ✓ Should revert if added maximum locks amount exceeded
- ✓ Correct adding lock if claimable > 0

✓ If claimable > amount and contract has no enough balance error TransferFailed()
✓ If claimable = 0 and user has no enough balance error TransferFailed()
✓ Correct update lock amount
✓ If updating lock amount = 0 error AmountIsZero()
✓ Correct update lock amount if user claimable > 0
✓ If contract has no enough balance and claimable > 0 update lock amount error TransferFailed()
✓ If sender has no enough balance and claimable = 0 update lock amount error TransferFailed()
✓ Correct update lock duration
✓ Correct update lock duration if claimable > 0
✓ If claimable > 0 and contract has no enough balance when update lock duration error TransferFailed()
✓ If update lock to smaller duration multiplier error UpdateToSmallerMultiplier()
✓ If update lock with smaller then was end time error MustProlongLock()
✓ Correct unlock if user claimable = 0
✓ Correct unlock if user claimable > 0
✓ If user claimable > 0 and contract has no enough balance when unlock error TransferFailed()
✓ In unlock if lock still active error LockStillActive()
✓ If user claimable > 0 correct compound
✓ If user claimable = 0 when compound error NothingToClaim()
✓ Correct claim if user claimable > 0
✓ If user claimable = 0 when claim error NothingToClaim()
✓ If contract has no enough balance when claim error TransferFailed()

42 passing (11s)

FILE	% STMTS	% BRANCH	% FUNCS
LockedStaking.sol	99.33	95.59	100
All files	**99.33**	**95.59**	**100**

We are grateful to have been given the opportunity to work with the TrustSwap team.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

Zokyo's Security Team recommends that the TrustSwap team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

ZOKYO.