Teneo

SMART CONTRACT AUDIT

ZOKYO.

December 6th, 2021 | v. 1.0

PASS

Zokyo Security Team has concluded that this smart contract passes security qualifications

SCORE

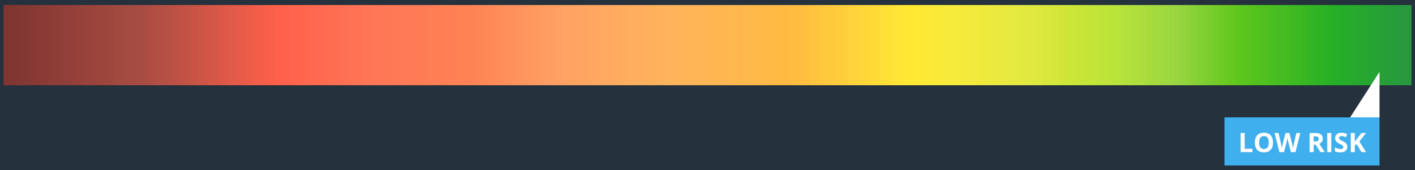
95

TECHNICAL SUMMARY

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

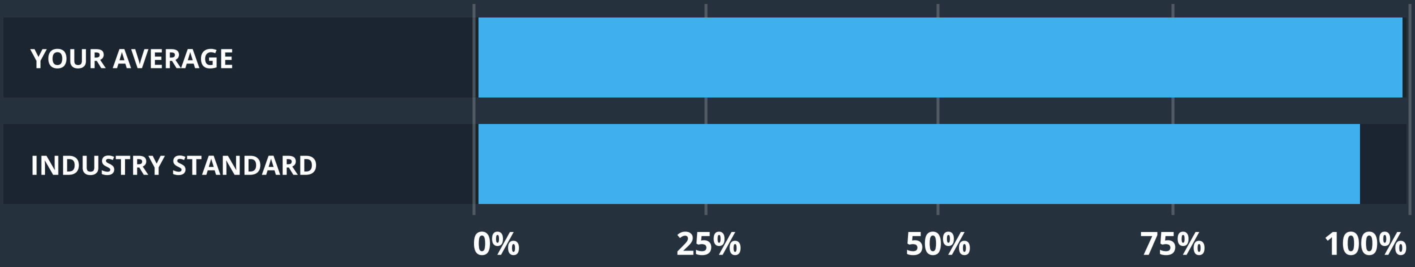
The scope of this audit was to analyze and document the Teneo Finance smart contract codebase for quality, security, and correctness.

Contract Status



There were no critical issues found during the audit.

Testable Code



The testable code is 100%, 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 Teneo Finance 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 Teneo Finance repository.

Repository:
https://github.com/TeneoFinance/contracts/commit/a9749ab5e3aee310010ae2691bd8ccd820510145

Contracts:

- ReflowStaking.

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

- Implements and adheres to existing Token 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 gas, 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 Teneo Finance 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.

2 Cross-comparison with other, similar smart contracts by industry leaders.

3 Testing contract logic against common and uncommon attack vectors.

4 Thorough, manual review of the codebase, line-by-line.

SUMMARY

The Zokyo team has conducted a security audit of the given codebase. The contracts provided for an audit are well written and structured. All the findings within the auditing process are presented in this document.

During the auditing process, our auditor's team found 1 issue with low severity and 1 informational issue. These issues were not fixed.

Based on the results of the audit, we can give a score of 95 and state that the audited contracts are fully production-ready.

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

● 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

Rounding of rewards

LOW | UNRESOLVED

Because solidity doesn't support operations with floating point it rounding values up or down. For this reason user's could lose part of their reward tokens. Amount of lost reward tokens depends on the staked amount.

Recommendation:
Change the logic of calculation.

```
function _calcReflow(uint256 pid) internal view returns (uint256, uint256, uint256) {
        if (_poolInfo[pid].stakeAmount > 0) {
            uint256 period;
            uint256 lastRewardSecond = block.timestamp;

            if (lastRewardSecond <= _poolInfo[pid].rewardEndTime) {
                // If the pool is not expired yet, the period will be calculated "normally"
                // The actual timestamp gets subtracted by the last time a reflow happened. (claim, deposit,
withdraw, any funding)
                period = lastRewardSecond - _poolInfo[pid].lastRewardSecond;
            } else {
                // If the pool expires, the reward end time (pool end timestamp) is subtracted by the last reward
second
                // The reason is: when a pool gets funded after the pool expires, it is also possible to fund.
                // For that the reward end time is increased and here calculats the rewards.
                period = _poolInfo[pid].rewardEndTime - _poolInfo[pid].lastRewardSecond;
                lastRewardSecond = _poolInfo[pid].rewardEndTime;
            }

            // calculate how many rewards are not calculated yet (also calculate in the residual if one exists)
            uint256 rewardPerPeriod = period * _poolInfo[pid].rewardPerSecond + _poolInfo[pid].rewardReserve;
            // calculate the rewards per share (the shares are divided by 10**12 to make the calculation easier)
            uint256 uncalculated = rewardPerPeriod / _poolInfo[pid].stakeAmount +
_poolInfo[pid].actualRewardPerShareAndSecond;
            // residuals are stored for the next time
            uint256 rest = rewardPerPeriod % _poolInfo[pid].stakeAmount;
          return (uncalculated, rest, lastRewardSecond);
        } else {
            return (_poolInfo[pid].actualRewardPerShareAndSecond, _poolInfo[pid].rewardReserve,
_poolInfo[pid].lastRewardSecond);
        }
    }
```

Unused requirements

INFORMATIONAL | UNRESOLVED

This requirement is not used because in front of him is another requirement which does the same work “checking that the user doesn’t try to withdraw more than he has.”. Also in your contract impossible situation then the user withdraws the amount of staked tokens which he has and at the same time withdraws more than pool has.

```
require(_allowances[sender][_msgSender()] >= amount, "transferFrom: transfer amount exceeds
allowance");
```

Recommendation:
Remove requirement:

```
require (wAmount / CORRECT_DIVISOR <= _poolInfo[pid].stakeAmount
```

Or swap it with requirement:

```
require (wAmount / CORRECT_DIVISOR <= _userInfo[pid][msg.sender].stakeAmount

function _withdraw(uint256 pid, uint256 wAmount) internal {
        require(wAmount / CORRECT_DIVISOR > 0, "_withdraw: Sorry, but too few to withdraw. (min. 10**12
wei)");
        require(wAmount % CORRECT_DIVISOR == 0, "_withdraw: Sorry, but for the correct calculation you only
can withdraw without residual. (amount % 10 ** 12 == 0)");
        require (wAmount / CORRECT_DIVISOR <= _userInfo[pid][msg.sender].stakeAmount, "_withdraw: Not
enough stakes in your account");
```

```
        require (wAmount / CORRECT_DIVISOR <= _poolInfo[pid].stakeAmount, "_withdraw: Not enough stakes
in the pool");
        require (_getWithdrawPossibleIn(pid, msg.sender) == 0, "_withdraw: Withdraw time is not expired");

        // saves the so far earned rewards (by withdrawing the basis for the calculation changes)
        _saveRewards(pid, 0);

        _poolInfo[pid].poolT.stakedToken.safeTransfer(msg.sender, wAmount);
        // save the new staked amount of the pool (the saved amount is divided by 10**12)
        _poolInfo[pid].stakeAmount = _poolInfo[pid].stakeAmount - (wAmount / CORRECT_DIVISOR);
 // save the new staked amount of a user (the saved amount is divided by 10**12)
        _userInfo[pid][msg.sender].stakeAmount = _userInfo[pid][msg.sender].stakeAmount - (wAmount /
CORRECT_DIVISOR);

        if (_userInfo[pid][msg.sender].stakeAmount == 0) {
            _poolInfo[pid].userCount -= 1;
        }
    }
```

	ReflowStaking
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 Zokyo Security team

As part of our work assisting Teneo Finance 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 Teneo Finance contract requirements for details about issuance amounts and how the system handles these.

Code Coverage

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	UNCOVERED LINES
staking\	100.00	96.97	100.00	100.00	
ReflowStaking.sol	100.00	96.97	100.00	100.00	
All files	**100.00**	**96.97**	**100.00**	**100.00**	

Test Results

```
Contract: ReflowStaking
  Methods
    initialize
      ✓ Check for correct deploy of ReflowStaking contract (459ms)
    setupPool
      ✓ Must fail if staked/reward token address equal to zero address (1756ms)
      ✓ Must fail if daoWallet address equal to zero address (249ms)
      ✓ Must fail if claimFee bigger than 50% (275ms)
      ✓ Must fail if staticWithdrawTime equals to 0 (290ms)
      ✓ Must setup first pool correctly with 2 different tokens, mode - 0 (1630ms)
    getAPY
      ✓ Must return total rewards per year if stake amount == 0 (445ms)
    ✓ Must return total rewards per year if stake amount != 0, mod 0 (1405ms)
  changeRewardPerSecond
    ✓ Must fail if user try to set RPS to 0 (541ms)
    ✓ Must fail to set new RPS without increasing period and depositing (591ms)
    ✓ Must change RPS correctly with increasing period (1398ms)
  fundAndChangeRPS
    ✓ Must fail if try to set RPS to 0 (692ms)
    ✓ Must fail to set new RPS without increasing period and depositing (640ms)
    ✓ Must change RPS correctly with increasing period (1987ms)
  fundIncreasePeriod
    ✓ Must increase period of rewardEndTime correctly (3139ms)
  setClaimFee
    ✓ Must fail if caller isn't owner (774ms)
    ✓ Must fail if new claim value is bigger than 50% (767ms)
    ✓ Must set new claim fee correctly (669ms)
  deposit
    ✓ Must fail if user try to stake less than 1*10**12 wei (987ms)
    ✓ Must fail if pool is not filled by reward tokens (705ms)
    ✓ Must deposit staked token correctly if depositAmount divisible by correctDivisor (2922ms)
    ✓ Must deposit staked token correctly if depositAmount isn’t divisible by CorrectDivisor (1646ms)
    ✓ Must deposit 2 times from one user (3487ms)
  fundIncreaseRPS
    ✓ Must fail if increase RPS without deposit and rewardEndTime (1264ms)
    ✓ Must increase RPS without deposit (2891ms)
    ✓ Must increase RPS correctly (3660ms)
    ✓ Must increase RPS correctly at the end of the pool (3352ms)
  getClaimableAmount
    ✓ Must get claimable amount from pool with mod 0 (3324ms)
    ✓ Must get claimable amount from pool with mod 1 (2963ms)
    ✓ Must get claimable amount from pool with mod 1 with big lock time (3297ms)
  claim
    ✓ Must fail if try to claim more than user can (2732ms)
    ✓ Must claim correctly from pool with mod 0 (9562ms)
    ✓ Must claim from pool with another mod (4062ms)
    ✓ Must claim after pool has ended (6315ms)
    ✓ Must fail if user has claimed all his reward already (5552ms)
    ✓ Must claim from pool with another mod (4062ms)
  withdraw
    ✓ Must fail if user try to withdraw less than 1*10^12 wei (3450ms)
```

```
      ✓ Must fail if withdraw amount isn’t divisible by 1*10^12 (3632ms)
      ✓ Must fail if user try to withdraw more than he has (3609ms)
      ✓ Must fail if user try to withdraw at lock period (3627ms)
      ✓ Must withdraw correctly from pool with mod 0 (10799ms)
      ✓ Must withdraw correctly, pool with mod 1, before rewards creating (10635ms)
      ✓ Must withdraw correctly, pool with mod 1, after rewards creating with lock time
      ✓ Must withdraw and increase RPS correctly, pool with mod 1, after rewards creating
        with lock time 11900ms)
      ✓ Must withdraw not all tokens correctly (8935ms)
      ✓ Must withdraw all tokens at the end of the pool correctly (11330ms)
    redeem
      ✓ Must withdraw and claim rewards correctly, mod 0, before lock time (76644ms)
      ✓ Must withdraw and claim rewards correctly, mod 0, after lock time (54816ms)
      ✓ Must withdraw and claim rewards correctly, mod 0, before lock time
    getWithdrawPossibleIn
      ✓ Must return 0 if pool has mod 1 (7760ms)
      ✓ Must return correct value if lock period hasn't passed, mod 0 (11366ms)
      ✓ Must return 0 if lock period has passed, mod 0 (22279ms)

  49 passing (7m)
```

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

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

ZOKYO.