SMART CONTRACT AUDIT
Report 5 of 6: Void Energy (VE) Staking

March 14th 2023 | v. 1.0

Security Audit Score

PASS

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

TECHNICAL SUMMARY

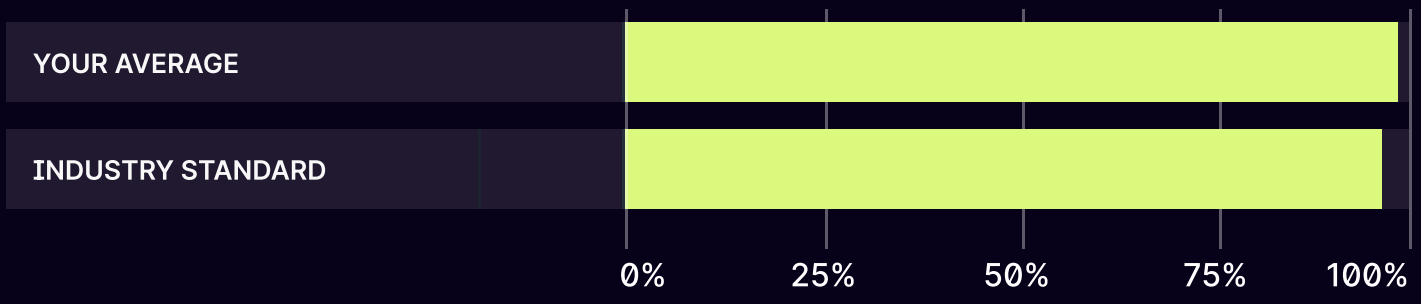
This document outlines the overall security of the TangleSwap smart contract evaluated by the Zokyo Security team.

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

Contract Status

There were 0 critical issues found during the audit. (See in the Complete Analysis, started from 6 page)

Testable Code



100% of the code is testable, 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 but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the IOTA & Shimmer network's fast-paced and rapidly changing environment, we recommend that the TangleSwap team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the TangleSwap repository:
https://github.com/TangleSwap/ve-staking-contract

Last commit: f614fa472f3b61ee9c638afbe004c9d38a547612
Commit of Fix: 980032c90cf61d4aa6dd49df456015a8076084e7

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- TangleswapVEStaking

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrancy attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of TangleSwap smart contract. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

There were three issues with medium severity, two issues with low severity and some informational issues found. The TangleSwap team resolved all issues.

During the audit, necessary fixes were carried out by developers. In order to achieve that, features have been added like: a new formula to calculate rewards, introducing reward debt and coming up with a new way to mutate the state of the contract (i.e. updating the pool).

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. 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

FINDINGS SUMMARY

#	Title	Risk	Status.sol
1	Pre-estimating stakeEndBlock can lead to miscalculated rewards	Medium	Resolved
2	Pending rewards are calculated incorrectly	Medium	Resolved
3	Centralization Risk	Medium	Resolved
4	No validation applied on the constructor arguments	Low	Resolved
5	Revert messages are too long and costly	Low	Resolved
6	Using msg.sender along with _msgSender()	Informational	Resolved
7	Unneeded use of keyword storage	Informational	Resolved
8	Lock Solidity version	Informational	Resolved
9	Unnecessary redundant branches at lines 132, 120	Informational	Resolved

MEDIUM | RESOLVED

Pre-estimating stakeEndBlock can lead to miscalculated rewards

On staking void token , `user.stakeEndBlock` is being estimated in the future according to this

```
user.stakeEndBlock = user.stakeStartBlock + user.lockDuration.div(blockTime);
```

But it is assumed that the `blockTime` is changeable (i.e. depending on the chain contract deployed on) and changing `blockTime` during staking period causes tokenomical issues. Suppose in a scenario in which `blockTime` is increased, while a given investor is having a stake already for a period of one year and the `stakeEndBlock` is estimated to be around that `stakeEndTime` also after one year. If the `blockTime` due to chain parameter changes and admin updating the value, the `stakeEndBlock` shall be driven away after that stakeEndTime in an unpredictable.
To summarize, relying on `blockTime` to estimate how much reward should be taken in the future by estimating `stakeEndBlock` can lead to a misalignment between stakeEndBlock and `stakeEndTime` which in the end leads to a miscalculated reward received.

Recommendation:
Relying on estimating rewards based on a `blockTime` that should be fixed for more than 3 years is the crux of this issue of misalignment of `stakeEndTime` and `stakeEndBlock`. A recommended way to deal with it is to adopt the reward model of `PancakeSwap/SmartChefInitializable`

```
uint256 multiplier = _getMultiplier(lastRewardBlock, block.number);
uint256 cakeReward = multiplier * rewardPerBlock;
accTokenPerShare = accTokenPerShare + (cakeReward * PRECISION_FACTOR) / stakedTokenSupply;
lastRewardBlock = block.number;
```

`block.number` & `lastRewardBlock` get us the actual block count difference not just an estimation, therefore, developers shall need to add a checkpoint like `lastRewardBlock` for each user (i.e. mapping) and calculate reward based on the actual difference.

Pending rewards are calculated incorrectly

Admin can modify `rewardPerBlock` which can lead to issues. The code at line 135, line 185, and line 223 calculates the pending reward based on current value of rewardPerBlock rather than its value at the time of staking and account for changes during the stake period. Currently, any changes to the value of this variable in between are not considered while paying the rewards.

Example: A user stakes for period in range (T, T + 100) when rewardPerBlock was 10. Suppose at T + 20 reward is set to 5 and at T + 50 = 40. Ideally, user should get rewarded as follows:

*[blockCount(T, T + 20) * 10 + blockCount(T+ 21, T+50) * 5 + blockCount(T + 51, T + 100) * 40]*

where blockCount(a,b) is the number of blocks in between the time a and b.

But, contract will reward (T, T+ 100) * whateverOwnerHasSet (at the moment of withdraw) and thus any changes to rewardPerBlock in between gets lost.

Recommendation:
A recommended way to deal with it is to adopt the reward model of PancakeSwap/ SmartChefInitializable. To achieve that assign for each user their own rewardPerBlock, we suggest you something like this code snippet which shall help also in issue#1.

```
function stake () {
    // calculate rewards at this point
    calculateReward(user.localRewardPerBlock, user.lastRewardBlock, ...)

    // update stake for user
    user.localRewardPerBlock = globalRewardPerBlock
    user.lastRewardBlock = block.number
}

function unstake () {
    calculateReward(user.localRewardPerBlock, user.lastRewardBlock, ...)
}
```

MEDIUM | RESOLVED

Centralization Risk

Status: Resolved
Admin enjoys too much authority. The general theme of the repo is that admin has power to call several functions like emergencyRewardWithdraw, recoverToken, general setters/ mutators. Some functions can be more highly severe to be left out controlled by one wallet more than other functions; depending on the intentions behind the project.

Recommendation:
Apply governance / use multisig wallets.

LOW | RESOLVED

No validation applied on the constructor arguments

No require statements to validate inputs of constructor during deployment.

Recommendation:
Add require statements similar to the following:

```
require(address(voidToken) != address(0));
require(minimumStakeableAmount >  0);
require(bonusEndBlock > startBlock )
require(blockTime > 0);
require(rewardPerBlock > 0);
```

LOW | RESOLVED

Revert messages are too long and costly

Error messages in some require statements are long and considered costly in terms of gas.

Recommendation:

- Write shorter error messages
- Use custom errors which is the goto choice for developers since solidity v0.8.4 details about this shown here: soliditylang.

INFORMATIONAL | RESOLVED

Using msg.sender along with _msgSender()

Confusion arises from using `_msgSender()` and `msg.sender` in the same contract. `_msgSender()` here returns `msg.sender` but it is recommended from a coding perspective to make it consistent. It shall be problematic though if `TangleswapVEStaking` is inherited by another contract that overrides `_msgSender()`.

Recommendation:
replace `msg.sender` by `_msgSender()`

INFORMATIONAL | RESOLVED

Unneeded use of keyword storage

Literal `storage` being used in view functions unnecessarily for `userInfo` in `pendingRewards`, `getUserInfo` and `isLockTimeExpired`

Recommendation:
replace with `memory` instead.

INFORMATIONAL | RESOLVED

Lock Solidity version

All contracts, Lock the pragma to a specific version, since not all the EVM compiler versions support all the features, especially the latest ones which are kind of beta versions, So the intended behavior written in code might not be executed as expected. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler, which may have higher risks of undiscovered bugs.

Recommendation:
fix version to 0.8.17

INFORMATIONAL | RESOLVED

Unnecessary redundant branches at lines 132, 120

Description: If statements at line 120 and 132 are repeated unnecessarily.

Recommendation:
Code can be re-arranged to have only 1 branch (if statement) to save some gas.

	TangleswapVEStaking
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

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

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

TangleswapVEStaking

- ✓ Should be deployed correctly (46ms)
- ✓ Should be able to update block time (149ms)
- ✓ Should be able to update start and end blocks (146ms)
- ✓ Should be able to update reward per block (90ms)
- ✓ Should be able to stop reward (51ms)
- ✓ Should update minimum stakeable amount (105ms)
- ✓ Should be to stake tokens (871ms)
- ✓ Should be to unstake tokens (580ms)
- ✓ Should allow restaking (692ms)
- ✓ Should allow emergency withdrawals (306ms)
- ✓ Should be able to emergency reward (90ms)
- ✓ Should allow owner to recover token (146ms)
- ✓ Should be able to get current block
- ✓ Should be able to get user info
- ✓ Should be able to view pending rewards (88ms)
- ✓ Should tell if stake lock time has expired (109ms)

16 passing (8s)

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

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% UNCOVERED LINES
TangleswapVEStaking.sol	100	100	100	100	
FILE	**100**	**100**	**100**	**100**	

We are grateful for the opportunity to work with the TangleSwap team.

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

Zokyo Security recommends the TangleSwap team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo