Audit Report

THORWallet Distributor and Staking Smart Contracts

Produced for Defi Suisse AG

Produced by Dominik Spicher, 21 Analytics AG

26.11.2021

Contents

1 Executive Summary

We have found the staking and distributor smart contracts to be of high quality. The audit has uncovered no issues of high severity, no issues of medium severity, and two issues of low severity. We also list four minor suggestions of how the contracts could be improved.

2 Scope

The review was performed on the Git revision `18866702078f4a87623e34b6386c221195524ae9` of the `THORWallet/smartcontract` repository[1].

In particular, the review pertained to the following smart contract source files, specified here with their `sha256` hashes:

```
distributor.sol     4e817aecb2d77d6f43655a017705f66f72ce709cd91f8e2300ba5d450dcf290a
staking.sol         cdc00d325463ee42c66576d2aa409368ed48e7358b927dd6282b8fad24b3ff16
```

and the following auxiliary test and specification files:

```
distributor-test.js 4e9d014ecc5db50c8ede921dc71f6c2ec498b6c729dd2f6811fa0e106766ffe1
staking-test.js     b851f7cf61ea21a881e7a0ad9634fa3498baa6705b775e05515ca6d0ed037930
```

Imported contracts from well-known libraries such as Open-Zeppelin were not part of the review.

The review was constrained to the Solidity source files, low-level assembly code generated thereof was not inspected.

3 Methodology

The review consisted of the following steps:

- Check for compliance with smart contract development best practices
- Check for compliance with specification, where available and applicable
- Manual inspection and analysis of the smart contract and test code
- Usage of static analysis tools

[1] https://github.com/THORWallet/smartcontract

4 System description

4.1 Contracts

4.1.1 Distributor

The distributor smart contract provides functionality where a large set of user-claimable token amounts can be committed to via a merkle tree root. Users can then submit merkle-proofs and claim the respective amount of tokens, which is transferred by the Distributor contract itself. The merkle root can be updated by the contract owner. The smart contract keeps track of claimed amounts to support the use-case where a merkle root update leads to an increased allowance for a particular user. The user can then submit the new proof with the newly increased total amount, and receive the remaining token difference. The amounts in the Distributor contract are stored without decimals to reduce gas costs. Therefore, amounts in the distributor contract need to be multiplied by 10**18 to receive TGT amounts.

4.1.2 Staking

The staking smart contract provides functionality for users to deposit tokens into the smart contract, and earn staking rewards denominated in a dedicated, global reward token. The staking reward is proportional to both the number of tokens locked up, as well as to the lockup duration. First, the contract owner has to create a liquidity pool dedicated to a particular ERC20-compatible token. Subsequently, users can call a function which will transfer the ERC20 token to the staking contract. Reward tokens are paid out either when the user's amount of staked ERC20 tokens changes (to simplify the accounting of rewards a user is eligible for), or upon the invocation of a dedicated "harvest" function by the user. The contract also contains functionality for a user to emergency-withdraw his locked tokens, forgoing any staking reward he would be eligible to.

4.2 Roles

Both contracts have standard contract owners who get privileged access to a subset of functions. Owners are initially set to the contract deployment account. The Staking ownership can be both transferred and denounced (leaving the contract in an ownerless state), whereas Distributor ownership can only be transferred.

5 Best practices checklist

Non-adherence to the characteristics listed in the below list does not represent a security issue itself. However, following best practices is a good indicator of care and attention to detail, it

allows the review process to be more focused and efficient, thus making it more likely for issues to be uncovered.

- ✓ The code was provided as a Git repository
- ○ There exists specification, covering the most important aspects of smart contract functionality (only the issuance schedule is specified)
- ✓ The development process is understandable through well-delineated, atomic commits
- ✓ Code duplication is minimal
- ✓ The smart contract source files are provided in an unflattened manner
- ✓ The code compiles with a recent Solidity version
- ✓ The code is consistently formatted
- ✓ Code comments are in line with the associated code (mostly)
- ✓ There are tests
- ✓ Tests are easy to run
- ✓ Tests provide a reasonably high coverage (somewhat, see Appendix A)
- ✓ The code is well documented
- ✓ There is no commented code
- ✓ There is no unused code
- ○ The code follows standard Solidity naming conventions

6 Findings

We rank our findings according to their perceived severity (high, medium, low), where an issue's severity is understood to be the product of its likelihood of being triggered and the impact of its consequences. Where deemed appropriate, we also report issues that are not security-relevant per-se but impact things like transaction-cost effectiveness or user experience.

See section 6.4 for suggestions for improvements which have no discernible impact on any of the above.

6.1 High severity

No issues found.

6.2 Medium severity

No issues found.

6.3 Low severity

6.3.1 Potential re-entrancy vulnerability in the staking `harvest()` function

The harvesting function pays out any pending reward with (L250)

```
rewardToken.safeTransferFrom(rewardOwner, to, pendingReward)
```

and updates the `rewardDebt` variable (which was used to compute the pending reward) afterwards to the new value (L252). This is a potential re-entrancy vulnerability, if `rewardToken.safeTransferFrom` includes any mechanism for a receiving contract to take flow control, for example via ERC223 or ERC677 callbacks.

Regarding the two reference contracts mentioned in the staking source file, the `thorstarter` contract[2] correctly applies effects before token contract interactions, whereas the `goose-contracts` contract[3] doesn't contain the `harvest()` function.

The test suite suggests that the TGT token will be the reward token in the deployed system. If so, then this vulnerability should not materialize because TGT contains no functionality where contract callbacks are invoked upon token transfer. Therefore this issue is deemed low severity. Nevertheless, the staking implementation could be made more robust against future changes which accidentally introduce this vulnerability. To this end, there seems to be no downside to making the `harvest()` function `nonReentrant`, similar to `deposit()` and `withdraw()`, as they are not called by each other.

6.3.2 Unintended identical unit tests

The two staking tests "test deposit in same blocks 1" and "test deposit in same blocks 2" are identical, whereas their name suggests that they should test different cases.

The same holds true for "test withdraw in same blocks 1" and "test withdraw in same blocks 2", which are identical except for a single ineffectual line order change.

6.4 Recommendations

In this section we list suggestions for minor improvements that are not deemed severe enough to be called an issue, or whose resolution requires ambiguous tradeoffs.

[2] https://github.com/Thorstarter/thorstarter-contracts/blob/main/contracts/Staking.sol
[3] https://github.com/goosedefi/goose-contracts/blob/master/contracts/MasterChefV2.sol

6.4.1 Comparison to boolean constant

The staking `addPool()` function contains (L106):

```
poolExistence[_lpToken] == false
```

This is needlessly verbose and can be simplified to

```
!poolExistence[_lpToken]
```

6.4.2 Public function that could benefit from being external

Public functions that take array arguments can be made more gas-efficient by marking them external, due to differing memory management. This is the case for the `claimableAmount()` function in the distributor contract. Note that since this is a `view` function, this will only make a difference if the function is called by other smart contracts.

6.4.3 Unused potential for an effective staking integration test

The staking rewards calculation is quite involved and involves multiple levels of allotting shares to entitled entities: Rewards are split up among pools according to their allocation points, as well as inside a pool between users according to the number of tokens they have staked. On top of that is more logic to update book-keeping variables through time. However, all of this machinery is set up to respect the `rewardPerBlock` variable set in the constructor. This affords a great opportunity for an integration test where the invariant

```
sumOfRewards == blocks * rewardPerBlock
```

can be checked against multiple users depositing into and withdrawing from multiple pools randomly.

6.4.4. Confusing naming of and documentation on the `rewardDebt` variable

The staking contract variable `rewardDebt` is confusingly named and documented. The reference to debt suggests that this is a notional amount being owed by some party to another. Furthermore, the documentation on L22 explains `rewardDebt` as “The amount of token entitled to the user”. This is incorrect. Indeed, the variable is used to decrease, not increase, the amount of staking reward claimable by the user (L151, L198, L225, L248).

Instead, the variable indicates how much staking reward needs to be subtracted at the next harvesting event under the current amount of tokens staked by the user. This is confirmed by the fact that a harvesting event (and therefore an updating of `rewardDebt`) is always enforced

whenever the amount of staked tokens changes due to a deposit (L208) or withdrawal (L234) event. It would be an improvement if this subtle behaviour is documented and reflected in the variable name, for example by using “`rewardOffset`” instead of “`rewardDebt`”.

7 Limitations

Even though the code has been reviewed carefully on a best-effort basis, undiscovered issues can not be excluded. This report does not consist in a guarantee that no undeclared issues remain, nor should it be interpreted as such.

Appendix A: Test Coverage[4]

File	% Stmts	% Branch	% Funcs	% Lines
contracts/				
distributor.sol	70.83	35	71.43	70.83
staking.sol	58.33	53.33	43.75	58.33

[4] Generated with the “solidity-coverage" hardhat plugin (version 0.7.16)