RotoHive Audit

03.08.2018

Fabio Hildebrand, CISA, CISSP
fabiohildebrand@gmail.com

Overview

I have been reached by Valencian Digital Group to audit the smart contracts developed for RotoHive. RotoHive is a fantasy sports platform, consisting of an ERC20 token (ROTO) with additional functionality in order to allow users to stake tokens in tournaments. Users are then rewarded in ROTO (users that did not stake ROTO) and ETH (if a user staked ROTO in the tournament) based on the accuracy of their predictions.

Scope

The following contracts were in scope for the audit (https://github.com/RotoHive/contracts, commit `41467a56fe239f9e7a340d0dc0298a4f16dda732`):

- ERC20.sol
- RotoBasic.sol
- RotoManager.sol
- RotoToken.sol
- SafeMath.sol
- StandardToken.sol

The contracts were tested against known vulnerabilities, the ERC20 standard and the specifications provided (refer to Annex I).

Major Deficiencies

1. Trustlessness

The contracts are highly dependent on a central entity. Although this is a valid design decision, this level of control is generally not accepted as valid for a truly decentralised application. The owner of the contract is able to perform the following actions:

- Owner of `RotoToken` can set himself as manager, fact that allows him to stake ROTO on behalf of any user, reward and destroy ROTO from users.
- Owner of `RotoManager` contract can reward, destroy or even reward with Eth any account, irrespective of the result from the Tournament (in the current version the tournament results are not input in the smart contracts).
- Owner can close a Tournament before performing payouts, fact that would freeze the stakes of users.
- There is a check if the balance of the manager contract is larger than the `ethPrize` of a Tournament in its creation, but the owner can bypass the prize by paying more, less, paying the wrong user or even not paying at all.
- Owner can change the token contract and manager contracts at any time, even with Tournaments open / payouts pending.

Recommendation

The contracts should enforce the rules of the game in a complete manner. Users should submit results along with staking, and the outcome of the tournament updated on chain, allowing for transparent calculation of payouts (pull payments are also recommended, and would scatter the gas cost to users instead of relying on the owner posting a transaction for each user that staked ROTO in a tournament).

All these facts require the user of the dapp to fully trust the maintainer, much like a centralized app. This is fine if it is a design decision, but I strongly encourage clear communication of this fact to users.

If the owner restricted functions are to be called by different people / infrastructure, the creation of one key pair per unique user is advisable, as it reduces the impact if one of the keys is compromised (refer to recent Bancor case).

Lastly, the contracts do not allow for change of the owner, fact that would prevent the only possibility of recovery if the owner key is compromised.

2. Token and stake accounting can become inconsistent

The token ledger is kept by the `RotoToken` contract (and the inherited contracts with ERC20 functionality). The stakes ledger, however, is maintained by the `RotoManager` contract.

When a token is staked, it is subtracted from the users balance in the token contract and the stake amount kept in a struct along with Tournament data in RotoManager. When the user is rewarded through `rewardRoto` function, he is rewarded based on the value informed by the caller (owner). Since all calculations are performed off chain, incorrect payouts would throw the ROTO token in an inconsistent state where `totalSupply != sum of all balances from users +  sum of all staked tokens`.

This can also be caused by the closing of a tournament without performing payouts, and if the manager contract is upgraded with ROTO staked in it.

Recommendation

The ROTO `totalSupply` have to reflect the total amount of tokens available. In this context it should equal the sum of balances of all users plus the sum of all staked values at all times.

If the logic of the dapp involves the owner minting/burning tokens, this should be reflected in the `totalSupply` variable.

Both minting tokens indefinitely and the inconsistent state are a no-go if listing the ROTO token in an exchange is a goal.

Minor Deficiencies

3. RotoBasic and RotoToken contracts have an empty fallback function, accepting Ether sent with no data

The RotoBasic contract has an empty fallback function `function() public payable {}` allowing any user to send Ether to the contract. The `RotoToken` has the same function in place without the need for one, as all Ether sent to contract is in fact lost forever, due to the fact that there aren't functions that allow the withdrawal of funds.

Recommendation

If the intention is to send Ether for prizes, I would strongly encourage the use of `createTournament` and setting the `msg.value` as the prize. This would avoid confusion, while ensuring funds for the payouts are available and match exactly the prizes.

The fallback function should be removed from `RotoToken`.

If the fallback function is absolutely needed, the data size of the message should be verified, and transactions with Ether calling for functions with unknown signature (i.e.: a user with a different version of the front end) should be reverted.

4. Some functions would consume less gas if set to external

Functions meant to be only accessed by external contracts/addresses, such as `releaseRoto` and `rewardRoto` from `rotoManager`, can benefit from using the external visibility as this generates significant gas saving (link)

5. Lock Pragma (Solidity compiler version)

Contracts should be deployed with the same compiler version and flags that they have been tested the most with. 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. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.

Conclusion

Two major deficiencies were found, along with three minor deficiencies. The contracts were well coded and prevented attempts of performing overflows, reentrancy attacks, denial of service among other attacks. The ERC20 transferFrom frontrunning attack is mitigated using the current best practice.

There is a good test coverage, and the contracts are upgradeable, as defined in the specifications.

The contracts have, however, opportunities to improve in the trustlessness of their design, how access is managed and payouts are calculated.

Annex I - Specifications

Below the smart contracts' specifications as provided by the client:

OVERVIEW

The solidity contracts are designed to handle the creation and closing of tournaments; the staking of ROTO alongside user submissions; the rewarding of 'good' submissions, and destruction of poor submission(in the event that they staked). The RotoManager contract handles all functionality relating to the tournament, handling the aforementioned tournament functionality. Whilst the RotoToken functions handles the accounting portions of the tournament(on top of it's function as a ERC20 Token).

RotoManager Contract

The RotoManager contract was designed to be replaceable, and thus upgradeable. It's separation for the token was intentionally made as an abstraction layer between the tournaments and token management. As part of the RotoHive application, users will have the ability to create a rankings of fantasy football players every week. Alongside those rankings, the users have the option to stake ROTO as a part of their submissions. This is taken into account when deciding how each user will be rewarded based on their submissions(which is calculated using the PPR scoring standard).

The users who perform well, and staked their submissions, will receive their staked ROTO as well as an amount of ETH proportional to their performance.

If a user staked ROTO, and performs mediocrely, they will get their ROTO back, but no ETH.

If a user stakes ROTO, and performs poorly, they will lose their staked ROTO. Which will go back into the prize pool for future tournaments.

If a user doesn't stake ROTO, but perform well. They will receive ROTO as a reward.

RotoToken Contract

The RotoToken Contract is a ERC20 Compliant Token based on the OpenZeppelin implementation with a few additional functions, and variables. These functions(and variables) are designed to handle RotoManager contract upgrades, as well as the accounting portions of the tournament related functions used in RotoManager.