PoolTogether

Date	February 2021
Lead Auditor	Shayan Eskandari
Co-auditors	Sergii Kravchenko

1 Executive Summary

This report presents the results of our engagement with **Pool Together** to review POOL governance token, A fork of the Uniswap merkle distributor, and Pool liquidity mining program.

The review was conducted over 5 days, from **February 1, 2021** to **February 5,2021** by **Shayan Eskandari** and **Sergii Kravchenko**. A total of 10 person-days were spent.

2 Scope

This is a best-effort review on the following repositories (in order of priority):

- Liquidity Mining Contract - `TokenFaucet.sol` - Commit hash `956b9e9dfd41dacd4040c08b5061354cc11897fc` .
- Pool Together Governance Contracts which is a fork of Uniswap’s governance contracts “with a few changes” - Commit hash `6750ca9974740e4123189ab6df10b2e8ff422c46` . To review the changes and deployment approach
- Merkle Distributor which is an unmodified fork of Uniswap Merkle Distributor - Commit hash `ec5ab6fb55791fdab0a1f20fdf1201c72b902965`.
- Employee Vesting Contracts, which is an unmodified fork of Uniswap Employee Vesting Contracts.

3 System Overview

3.1 Liquidity Mining Contract

- `TokenFaucet.sol` Disburses a token at a fixed rate per second to holders of another token. This contract allows users to receive token based on their proportion of `measure` tokens balance in the faucet (e.g. Drip governance token to Liquidity Providers). The main focus on this review has been on this contract. For more details review Findings section.

3.2 Pool Together Governance Contracts

A fork of the Uniswap governance contracts to which ownership of the prize pools will be transferred.

Changes from Uniswap fork

- **GovernorAlpha**
  - Require **1%** of POOL Token votes in support of a proposal in order for a quorum to be reached and for a vote to succeed (Uniswap is 4%)
  - Require **0.1%** of POOL token votes in order for a voter to become a proposer (Uniswap is 1%)
  - Changes to the constructor and the TimeLock contract initial setup
- **Token (POOL)**
  - totalSupply is 10 million POOL (Uniswap is 1 billon UNI)
  - Replace `Uni` with `Pool` on all error messages and variable names
- Minor changes to **Timelock** contract, mainly name changes and constant implementation nuances.
- Only naming changes to **TreasuryVester** contract (`Uni` -> `Pool`).
- Note that the fee structure that are present in Uniswap version, are removed from PoolTogether fork.

3.3 Deployment Process

As requested, we reviewed the deployment script in `deploy/deploy.js`, which uses *hardhat* as the main framework for deployment. Here’s the process of how it deploys currently:

- Prior to the execution of the deploy script, the following (contract) addresses need to be known:
  - `MultiSig` wallet address. This address will initially hold all Pool tokens
  - All vested employee addresses (and the number of their vested shares)
- *Deployer* will be the main address of the HDWallet. Deployer has these roles:
  - Temporary *minter* of Pool token contract, and can change the minter to any desired address
  - `hermes` temporary admin of the governance contract

1. The script uses environment variables for storing private key and the required API keys:
   - `HDWALLET_MNEMONIC` for the deployer address
   - Connecting to the Ethereum network is done via Infura `INFURA_API_KEY`
2. Deployment of governance contract `GovernorAlpha` (*GovernorZero*) with deployer as *hermes*

2. Deployment of governance contract `GovernorAlpha` (GovernorZero) with deployer as hermes

3. Deployment of timelock contract `Timelock` with *2 days* delay, and the governance contract address as admin. The admin can add other admins and change the timelock delay.

4. Using the address of timelock contract, deployer calls the governance contract to set the TimeLock address and nullify *hermes*

5. The minter of Pool token is changed from *deployer* to *timelock* contract

    - Note that if this transaction fails to execute successfully, the deployer will keep the minter role on the Pool contract. An addition check to verify the minter address is suggested.

6. Deployment of the vesting contracts `TreasuryVester` with *two years* vesting period:

    1. Deploying the treasury vesting contracts with timelock contract address as the Treasury address
    2. Deploying the employees vesting contracts with their hardcoded token numbers and addresses
        - Due to the way the script is written, the employees might have slightly different vesting periods (`recentBlock.timestamp + 600` as recentBlock is queried every time and might differ from the previous ones)

A few notes on the deployment script:

- Move all constants to top of the script code, to make it clear when reading the script (e.g. `twoYearsInSeconds` instead of 172800 in the code)
- The employee names for vesting are clear when explained, it might make sense to include these numbers in the token distribution documentation.
- A check after each step to verify the set values is suggested.
- Suggesting to use a unified `vestingCliff` for all employees to prevent any timing issues in the future.

3.4 Merkle Distributor

The main solidity contract `MerkleDistributor.sol` is an unmodified copy of Uniswap Merkle Distributor. This contract will allow users to claim governance tokens allocated to them retroactively.

4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were reviewed by the audit team.

- Anyone can create a new TokenFaucet using the Factory contract. However the tokens used could be malicious or using a totalSupply (Uint256 as in default ERC20 standard) might overflow in some functions (Uint112), which could cause issues. This should be explicitly mentioned in the documentation and warn users when using any UI component for the factory. This is mainly important if a new Prize Pool is introduced to the system in which behaves differently from the tested tokens.

5 Findings

Each issue has an assigned severity:

- `Minor` issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- `Medium` issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- `Major` issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- `Critical` issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 TokenFaucet refill can have an unexpected outcome `Medium`

Description

The `TokenFaucet` contract can only disburse tokens to the users if it has enough balance. When the contract is running out of tokens, it stops dripping.

code/pool-contracts/contracts/token-faucet/TokenFaucet.sol:L119-L138

```
uint256 assetTotalSupply = asset.balanceOf(address(this));
uint256 availableTotalSupply = assetTotalSupply.sub(totalUnclaimed);
uint256 newSeconds = currentTimestamp.sub(lastDripTimestamp);
uint256 nextExchangeRateMantissa = exchangeRateMantissa;
uint256 newTokens;
uint256 measureTotalSupply = measure.totalSupply();

if (measureTotalSupply > 0 && availableTotalSupply > 0 && newSeconds > 0) {
  newTokens = newSeconds.mul(dripRatePerSecond);
  if (newTokens > availableTotalSupply) {
    newTokens = availableTotalSupply;
  }
  uint256 indexDeltaMantissa = measureTotalSupply > 0 ? FixedPoint.calculateMantissa(newTokens, measur
  nextExchangeRateMantissa = nextExchangeRateMantissa.add(indexDeltaMantissa);

  emit Dripped(
    newTokens
  );
}
```

The owners of the faucet can decide to refill the contract so it can disburse tokens again. If there’s been a lot of time since the faucet was drained, the `lastDripTimestamp` value can be far behind the `currentTimestamp`. In that case, the users can instantly withdraw some amount (up to all the balance) right after the refill.

Recommendation

To avoid uncertainty, it’s essential to call the `drip` function before the refill. If this call is made in a separate transaction, the owner should make sure that this transaction was successfully mined before sending tokens for the refill.

5.2 Gas Optimization on transfers Minor

Description

In TokenFaucet, on every transfer `_captureNewTokensForUser` is called twice. This function does a few calculations and writes the latest UserState to the storage. However, if `lastExchangeRateMantissa == exchangeRateMantissa`, or in other words, two transfers happen in the same block, there are no changes in the newToken amounts, so there is an extra storage store with the same values.

Examples

`deltaExchangeRateMantissa` will be 0 in case two transfers (no matter from or to) are in the same block for a user.

`/pool-contracts/contracts/token-faucet/TokenFaucet.sol`

```
uint256 deltaExchangeRateMantissa = uint256(exchangeRateMantissa).sub(userState.lastExchangeRateManti
   uint128 newTokens = FixedPoint.multiplyUintByMantissa(userMeasureBalance, deltaExchangeRateMantiss
   userStates[user] = UserState({
     lastExchangeRateMantissa: exchangeRateMantissa,
     balance: uint256(userState.balance).add(newTokens).toUint128()
   });
```

Recommendation

Return without storage update if `lastExchangeRateMantissa == exchangeRateMantissa`, or by another method if `deltaExchangeRateMantissa == 0`. This reduces the gas cost for active users (high number of transfers that might be in the same block)

5.3 Handle transfer tokens where `from == to` Minor

Description

In TokenFaucet, when calling `beforeTokenTransfer` it should also be optimized when `to == from`. This is to prevent any possible issues with internal accounting and token drip calculations.

`/pool-contracts/contracts/token-faucet/TokenFaucet.sol`

```
...
 if (token == address(measure) && from != address(0)) {  //add && from  = to
      drip();
...
```

Recommendation

As ERC20 standard, `from == to` can be allowed but check in `beforeTokenTransfer` that if `to == from`, then do not call `_captureNewTokensForUser(from);` again.

5.4 Redundant/Duplicate checks Minor

Description

There are a few checks (require) in TokenFaucet that are redundant and/or checked twice.

Examples

- `_dripRatePerSecond > 0` checked twice, no need to check it in `initialize` `pool-contracts/contracts/token-faucet/TokenFaucet.sol`

  ```
  require(_dripRatePerSecond > 0, "TokenFaucet/dripRate-gt-zero");
  asset = _asset;
  measure = _measure;
  setDripRatePerSecond(_dripRatePerSecond);
  ```

  ```
  function setDripRatePerSecond(uint256 _dripRatePerSecond) public onlyOwner {
  require(_dripRatePerSecond > 0, "TokenFaucet/dripRate-gt-zero");
  ```

- `lastDripTimestamp == uint32(currentTimestamp)` and `newSeconds > 0` are basically the same check.
- `measureTotalSupply` can never be < 0, as in the if statement enforces that `/pool-contracts/contracts/token-faucet/TokenFaucet.sol#L111-L117`

  ```
  function drip() public returns (uint256) {
  uint256 currentTimestamp = _currentTime();

  // this should only run once per block.
  if (lastDripTimestamp == uint32(currentTimestamp)) {
    return 0;
  }
  ...
  uint256 newSeconds = currentTimestamp.sub(lastDripTimestamp);
  ...
  if (measureTotalSupply > 0 && availableTotalSupply > 0 && newSeconds > 0) {
  ...
      uint256 indexDeltaMantissa = measureTotalSupply > 0 ? FixedPoint.calculateMantissa(newTokens,
  ```

Recommendation

Remove the redundant checks to reduce the code size and complexity.

5.5 Unnecessary use of upgradability Fixed

Resolution

These contracts are part of OpenZeppelin Contracts Upgradeable.

Description

Libraries such as `SafeMath` and `SafeCast` should not be upgradable as they should be used as pure functions.

Upgradable libraries used in TokenFaucet contract:

- SafeMathUpgradeable
- SafeCastUpgradeable
- IERC20Upgradeable

Recommendation

Remove the upgradability functionality from any part of the system that is unnecessary, as they add complexity and centralization power to the admins.

Appendix 1 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

Request a Security Review Today

Get in touch with our team to request a quote for a smart contract audit.

CONTACT US

AUDITS FUZZING SCRIBBLE BLOG TOOLS RESEARCH ABOUT CONTACT CAREERS PRIVACY POLICY

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.

POWERED BY CONSENSYS