SOMISH

Blockchain Labs

Smart Contract Audit Report
Coin Breeder DAO

August 27th, 2020

Index

Summary

Audit Report prepared by Somish Blockchain Labs for CoinBreederDAO smart contracts.

Client Documents

The following document(s) were provided by the client for the purpose of the audit:

- Code_description.pdf

Process and Delivery

The following process was adopted for conducting this audit:

1. Understanding Project Functionality
    - Documents highlighted under the “Client Documents” section were referred to understand the project functionality.
2. Running Static analysis Tools
    - We used Slither & Solgraph. Relevant findings from the static analysis tools have been included in the audit report.
3. Conducting Manual Review
    - Two (2) independent experts performed an unbiased and isolated audit of the code after the static analysis tools.
    - Code was manually reviewed along with writing test cases wherever applicable to identify and confirm attacks
4. Preparing Audit Report
    - Finally the audit report was prepared with relevant findings from the above 3 steps - understanding, static analysis and the manual review.
    - The debrief between the 2 auditors took place on August 26th, 2020, and the final results are presented here.

Audited Files

The following contract(s) were covered during the audit:

- BREE.sol
- BREE_STAKE_FARM.sol
- ERC20contract.sol
- Owned.sol
- SafeMath.sol

Notes

Audit was performed on contracts which are deployed on the Ropsten Test Network and pragma version ^0.6.0.

Intended Behavior

Token Name: CBDAO
Symbol: BREE
Decimals: 18
Max Capped Supply: 10,000,000 BREE (10 million)

Token is mintable & burnable. Smart contracts are divided into two. BREE.sol and BREE_STAKE_FARM.sol. These two have been separated for the purpose of easy-upgrade of the farm & stake contract later on, without going through a token swap. Users are required to 'approve' the BREE_STAKE_FARM contract to spend their governance assets (for farming) and spend BREE (for staking) on each respective token contract.

Main Functions: Staking & Farming
Interest rates are in the form of APY (yearly rates) and rewards for staking and farming are both accumulated on a seconds basis.

Staking

- Users can stake their balance by locking it for a default period
- Staking rewards are accumulated until the default period ends and it should stop accumulating once the staking period is over.
- Users can claim their accumulated rewards immediately, whenever they wish to do so. When claiming, users are required to pay default fees in ETH. These fees are automatically transferred to the owner of the contract.
- Staking period & rates are fixed at time of initially staking. This means, even if the staking period and rates change while they are actively staking, it won't affect them. Unless, they add more to their staked balance and the new rates & period will be applied.
- Staking period should reset to Day 1 should a user decide to add more to his/her already actively staking balance.

User Flow

1. User approves the BREE_STAKE_FARM contract to spend BREE tokens on the BREE token contract
2. User interacts with the BREE_STAKE_FARM contract (STAKE function)
3. Balance is deposited & locked for the default period
4. User starts accumulating rewards on the token contract (pendingReward). Rewards are calculated every second and interest rates are set as yearly rates (APY). For example, if a user deposits 1000 tokens at 40% APY and the staking period is 180 seconds, by the end of the staking period, the user should have accumulated:
   [(40/31536000)*(1000/100)]*(180) = 0.02283105 BREE (pendingReward)
   - 40 = interest rates in a year (APY)
   - 31536000 = seconds in a year
   - 1000 = user's staked balance
   - 180 = total seconds staked

Write functions for Staking

- **ClaimReward** = claims the user's accumulated staking rewards (users have to pay default fees in ETH which is automatically sent to the owner of the contract)
- **ClaimStakedTokens** = withdraws the user's staked balance (only available when staking period ends)
- **STAKE**= to stake BREE balance
- **addToStake** = used to add more balance to already actively staking balance - this resets the user's staking period to day 1 and should there have been a rate/period change for staking, the new rates/period should apply upon addToStake
- **changeStakingRate** = owner only function to change staking rates (yearly interest rates, 40% APY is therefore, 40.)
- **setClaimFee** = owner only function to change pendingReward claiming fees (ETH)
- **setStakingPeriod** = owner only function to change default staking period (in seconds)
- **whiteList** = owner only function to whitelist users to be able to stake their tokens

Read functions for Staking

- **isUserWhitelisted** = to check if an address is whitelisted for staking or note (true/false)
- **lastStakedOn** = to check the latest time when user has initiated staking (unix timestamp format)
- **latestStakingRate** = shows the latest staking rates
- **pendingReward** = shows total accumulated staking rewards of an address
- **stakeClaimFee** = shows the latest rewards claiming fees in WEI format (ETH)
- **stakingPeriod** = shows the latest default staking period (in seconds)
- **stakingTimeLeft** = shows the staking time left of an address - if there's any active
- **totalRewards** = shows the total combined number of 'claimed/withdrawn' staking rewards of all addresses

- **totalStakeRewardsClaimedTillToday** = shows the total staking rewards 'claimed' by an address until today
- **yourActiveStake** = shows the total actively staking balance of an address
- **yourStakingPeriod** = shows the initial stacking period for an address - implemented to allow users to keep stacking as per the staking period when they first initiated staking, without being affected by a changed/new staking period. This is however, ignored, if the user adds more to his/her actively staking balance as the new/changed staking period will be applied with the staking period reset to day 1.
- **yourStakingRate** = shows the initial staking rate for an address - implemented to allow users to keep staking as usual as per the staking rate when they first initiated staking, without being affected by a changed/new staking rate. This is however, ignored, if the user adds more to his/her actively staking balance as the new staking rate will be applied.
- **yourTotalStakesTillToday** = shows the user's total staked amounts until today (active + expired combined)

Farming

Users can farm BREE tokens with other erc20 tokens. These tokens have their own farming rates for each of them respectively.
Tokens can only be added/deleted by the owner of the contract.

User Flow

1. User approves the BREE_STAKE_FARM contract to spend their erc20 tokens on their respective token contracts
2. User starts farming by filling in the token contract address and amounts on the 'FARM' function
3. User's erc20 token balance is deposited onto the BREE_STAKE_FARM contract and it starts farming BREE tokens
4. Farmed yield is accumulated on a seconds basis in yearly interest rates (works same as staking).
For example, if a user deposits 1000 erc20 tokens for a period of 180 seconds and the interest rates are 40% APY, the user should receive:
[(40/31536000)*(1000/100)]*(180) = 0.02283105 BREE (Farmed Yield)
40 = interest rates in a year
31536000 = seconds of a year
1000 = user's staked balance
180 = seconds since first deposited
5. Users are free to stop farming & withdraw their erc20 tokens at any time. Users are also free to claim their accumulated yield at any time - when claiming yield, they are required to pay default fees in ETH and these fees are automatically transferred to the owner of the contract.
6. pendingYield (accumulated yield) should fluctuate as with the changing farming rates for each token. Meaning, a user's accumulated yield can increase/decrease continuously as the farming

rates change, until the user decides to claim/withdraw them. It serves as a 'mini-game' for farmers to guess/predict/bet on when the best time would be to collect their yield.

Write functions for Farming

- **FARM** = allows a user to start farming BREE tokens with supported ERC20 tokens by filling in amounts & token contract address
- **YIELD** = used to collect pending Yield / accumulated yield - users are required to pay default set fees in ETH and this is automatically transferred to the owner of the contract
- **addToFarm** = used to add more farming balance to an already actively farming balance. Enter erc20 token contract address & amounts.
- **addToken** = owner-only function to add new erc20 tokens which can farm BREE
- **changeFarmingRate** = owner-only function to change farming rate of a particular erc20 which has already been added
- **removeToken** = owner-only function to remove erc20 tokens which can farm BREE
- **setYieldCollectionFee** = owner-only function to change yield collection fees (ETH)
- **withdrawFarmedTokens** = used to stop farming & withdraw an actively farming balance
  Read Functions for Farming
- **activeFarmDeposit** = shows the user's actively deposited amounts which is used to farm BREE
- **lastFarmedOn** = shows the user's most recent deposit time for farming (in unix timestamp format)
- **pendingYield** = shows the user's accumulated & claimable yield (on a seconds basis)
- **tokens** = used to check if an erc20 token is supported for farming BREE - shows true/false and rates in APY
- **totalFarmingRewards** = shows the user's total collected/withdrawn yield
- **yourTotalFarmingTillToday** = shows the total amounts deposited for farming for a particular erc20 token, until today (includes active+past deposits)

Issues Found

Critical

1. Users can get interest on BREE token for more than stake period.

The **YIELD()** function allows the user to yield BREE tokens. This means a user can stake using **STAKE()** function and farm using **YIELD()** function.

Since the **YIELD()** function does not have a check on staking period, the user can withdraw interest for period>stake period. This is contradictory to the requirement that Staking rewards should be accumulated until the default period ends and it should stop accumulating once the staking period is over.

Recommendation
Consider adding a check to disallow BREE token via function **YIELD()**

2. Removing tokens allowed for farming not handled properly.

A check is present in functions **withdrawFarmedTokens()** and **YIELD()** to disallow users from withdrawing non-whitelisted tokens. The contract also contains a function **removeToken()** which allows the owner to remove a token from the list of whitelisted tokens. This means that tokens that were once allowed and later removed, will be permanently locked inside the contract.

Eg, USDT was once allowed for farming and users participated in farming via the function **FARM**(). The owner now decides to disallow USDT and invokes function **removeToken().** All users who initially participated in USDT would now, not be able to withdraw their tokens.
Recommendation
Restriction of whiteslisted tokens should be there on function **farm().** However, it is suggested that once the tokens are there in the contract, the users should be allowed to withdraw, even if further farming on that token is not allowed.

3. Owner has too many privileges

Owner address, currently set as the deployer of contract, has access to most of the critical functions which are not suited for a decentralized platform. There are several control issues:

a. **setStakeFarmingContract():** Allows owners to change the BREE_STAKE_FARM smart contract. This means allowing any smartContract to mint tokens for any user.
b. **changeFarmingRate() :** Allows the owner to change farming rate to any value, add his tokens to farm at the new rate and again reduce the rate. This gives the owner the privilege to get more yield.
c. **changeStakingRate() :** Allows the owner to change staking rate to any value, stake his tokens at the new rate and again reduce the rate. This gives the owner the privilege to get more stake returns.
d. **setStakingPeriod():** Allows the owner to change the stake rate to any value, add his tokens to satke at the new period and again change the rate. This gives the owner the privilege to get stake according to the time he wants.
e. **setClaimFee():** Allow the owner to change the claim fee rate to any,add his token to stake with less claim fee. This gives the owner privilege to claim reward on very less claim fee.
f. **changeFarmingRate():** Allows the owner to change farming rate to any value, farm his tokens at the new rate and again reduce the rate. This gives the owner the privilege to get more yield returns.
g. **mint() :** Allows the owner to mint new BREE tokens to any ethereum address.

Recommendation
Since contract BREE_STAKE_FARM holds data as well as function, consider deploying BREE_STAKE_FARM as an upgradable smart contract, using Open Zepplin's proxy technique. Also, consider giving ownership rights to a governance contract.

4. Withdraw FarmedTokens and YIELD accept Bree as the token address. This will result in early withdrawal.

When a user stakes the Bree token using **STAKE()** function the specific amount of token is locked in the smart contract for the given 'stakingPeriod' variable value. But stacked tokens can be withdrawn from the **withdrawFarmedTokens()** function before stakingPeriod ends.

Recommendation
Add a check that Bree tokens are not allowed in the **withdrawFarmedTokens**() and **YIELD**() function.

5. Exposed function allowing anyone to set yieldCollection Fee.

The function **setYieldCollectionFee()** in BREE_STAKE_FARM.sol has no access restrictions. This allows anyone to set a yield collection fee.

Recommendation

We would recommend adding a modifier to restrict access.

Major

1. Use SafeMath for all token operations

SafeMath is not used in function **mintTokens()**,**transfer()**,**YIELD()**,**withdrawFarmedTokens()**, **ClaimStakedTokens()**,**ClaimReward()**,**_newDeposit()** and **_addToExisting()** (in **BREE.sol, BREE_STAKE_FARM.sol**)**,** which may possibly lead to integer overflow.

Minor

1. Use separate functions for updating allowance

Follow the standard ERC20 pattern for increasing and decreasing allowance amount by using separate functions, which resolves the frontrun attacks.
Example of a frontrun attack:
For instance, consider Bob has given an allowance of 100 tokens to Alice, but after some time, Bob makes a transaction for updating the allowance to 90. Now, if Alice frontruns Bob's transaction with more gas and transfers 90 tokens, then the updated allowance will still be 90 tokens. Ideally, the allowance should have been set to 0 as Alice has already transferred 90 tokens.

To handle this situation, standard ERC20's decreaseAllowance function is used (check https://github.com/ethereum/EIPs/issues/738 for reference). When Bob uses this function to decrease allowance by 10 tokens, even if Alice front-runs the transaction and transfers 90 tokens, the allowance limit post Bob's transaction will be set to zero.

2. ERC20 Transfers should be wrapped in require statements

The ERC20 standard defines that the **transfer()** function should return a boolean and although most implementations throw on failure, this behaviour is not required. Therefore it is recommended to wrap transfer calls in require.
Recommendation
Consider wrapping all ERC20 calls inside **require** statements.

3. Several functions do not disallow zero address as an input

The function **setStakeFarmingContract()** in **BREE.sol** allows variable **StakeFarmingContract** to be set as address(0). If this is done accidentally, or on intention, this will result in failing of **YIELD()** and **ClaimReward()** functions, disallowing generation of interest.

The function **mintTokens()** in Bree.sol mint token even for the address(0) . Which means there is no validation for the address so the token can mint any address.

The function **transferFrom()** in Bree.sol allows address(0) for receiver address.

Recommendation
It is recommended to add a check for beneficiary address to be non address(0).

Notes

1. No unit test cases

No test cases were provided with the smart contracts. We would recommend writing unit-test cases to achieve 100% line and branch coverage in order to avoid any unforeseen issues.

2. Follow Open Zeppelins implementation of ERC20 mintable and burnable tokens

Consider following the standard open-zeppelin's implementation of mintable and burnable tokens.

3. Multiplication after Division

In the calculation of pending reward and yield, the division operation was performed before doing all the multiplications. As there are no floating points in solidity the number gets rounded off when divided, which may lead to precision errors. So it is always recommended to perform division after multiplication.
Recommendation
Consider dividing with the days after performing all the multiplication operations.

4. Updating state variables after the external call

In most of the functions the state variables are updated after making external calls. It is always recommended to set the state variable before making external calls including transferring of assets.

- In the **YIELD()** function which is present in BREE_STAKE_FARM.sol contract, variables **totalYield**,**pendinGains**,**totalGained** and **lastClaimedDate** are updated after calling **transfer()** and **mintTokens().**

Line No: 1125,128,129,130

- In the **withdrawFarmedTokens()** function which is present in BREE_STAKE_FARM.sol contract, variables **pendinGains,activeDeposit,startTime** and **lastClaimedDate** are updated after calling **transfer().**
  Line No: 147,149,151,153
- In the **ClaimReward()** function which is present in BREE_STAKE_FARM.sol contract, variables **pendinGains, totalGained** and **lastClaimedDate** are updated after calling **mintTokens().**
  Line No: 233,235,237
- In the **ClaimStakedTokens()** function which is present in BREE_STAKE_FARM.sol contract, variables **pendingGains** and **activeDeposit are updated** after calling **transfer().**
  Line No: 207,209

Recommendation
Consider making external calls like transferring ether and token after setting the variables.

5. APY and Stake Reward percent cannot be in decimals

No extra decimal places were maintained for APY and Stake Reward percentages, which disallows setting the rates in decimal points in future.
Recommendation
Consider adding two decimal places for the APY and Stake reward percentages.

6. No indexed variables for events in BREE_STAKE_FARM contract

It would be difficult to access the logs of user's and token's added for stake/farm.
Recommendation
Consider making user and token addresses indexed variables in the event.

7. Owner address is set multiple times in token contract

Token contract is implementing **Ownable** implementation, so the deployer will be set as the owner in its constructor. But the owner parameter was again set to msg.sender in the token contracts constructor, which is not required.

8. Gas Optimizations

a. Most of the functions are declared public, even though they were intended to be called externally.
   Recommendation
   Consider using **internal** for the function which is callable only within the same

contract and using **external** for the function which is callable only from outside of contract. It will optimize the function call as well as gas required for deployment.

b. **pendingReward()** function was called multiple times inside the **ClaimReward()** function. Same was the case with **pendingYield()** in **YIELD()** function.
   Recommendation
   Consider creating a variable inside the function to store the **pendingReward() /pendingYield()** value and use that variable for further calculations in that function.

9. Follow the Solidity style guide

Consider following the Solidity style guide. It is intended to provide coding conventions for writing solidity code especially naming of functions.

Recommendation
You can find documentation for Solidity Style guide on the following link:
https://solidity.readthedocs.io/en/v0.5.9/style-guide.html. Use solhint tool to check for linting errors.

Closing Summary

Upon audit of the CoinBreederDAO's smart contract, it was observed that the contracts contain critical, major and minor issues, along with several areas of notes.

We recommend that the critical, major, minor issues should be resolved. Resolving for the areas of notes are up to CoinBreederDAO's discretion. The notes refer to improving the operations of the smart contract.

Disclaimer

Somish Blockchain Labs's audit is not a security warranty, investment advice, or an endorsement of CoinBreederDAO's platform or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Somish from legal and financial liability.

Somish Solutions Limited