SOMISH

Blockchain Labs

Smart Contract Audit Report

Stabl.co

January 15th, 2020

Index

Summary

Audit Report prepared by Somish Blockchain Labs for Stabl.co smart contract.

Process and Delivery

Two (2) independent experts performed an unbiased and isolated audit of the code below. The debrief took place on December 27th, 2019, and the final results are presented here.

Audited Files

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

- **EscrowExchange.sol**.
- **White.sol**.

Client Documents

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

- Readme.md - File Outlining Important details of Stabl.co.

Notes

Audit was performed on commit a6f2383896197c6f39bbbd990c3eb6ee72628bdb and pragma version 0.4.24.

Intended Behavior

1. Owner Will add or list tokens with AddNewToken method.
2. Owner will update token status for sale before user make trade with updateStatusForSale method.
3. Now user will make trade by calling tradeToken method.(depositTokenID, withdrawTokenId, TokenName, isSeller, contractID.)

Note: isSeller and contractId id optional means there is no use except adding them into orderbook.
Also before calling this method user has to approve funds from erc20 method of his deposit token approve method.

4. Owner will withdraw tokens by calling withdrawToken method by adding orderID.
5. Owner can update token status for End sale with updateStatusForSale method.
6. Owner can editToken price.

7. Owner can refund Token to depositor by adding order id.
8. Owner can setMainAddress where all the fee will go.
9. Owner can setFeeRate.
10. Owner can transfer his ownership.

Issues Found

Critical

1. refund() function can be called multiple times for the same order ID by Admin.

There is no state variable that determines if an order is refunded or not. For example, let us assume an order is created and a user has deposited X tokens. Now, if the contract has 3X amount of same tokens and the admin calls the function **refund**() multiple times, it will result in the user getting a refund 3 times.

Recommendation

Consider using state variable to determine if an order is refunded or not. The function **refund**() must check the value of this variable and should update the variable once refund is called.

Amended (Jan 15th 2020): State variable added by Stabl.co team commit 2b5829e878c349dab75a73de48e2db27249ec3b8. Would recommend state change for **isRefunded** flag to be done before external function call **transfer().**

2. withdraw() function can be executed even after the refund process.

The function **withdraw**() only checks for **isDeposited** and **isWithrawn** variable for an order. Similar to issue 1, there is no state variable that determines if an order has been refunded or not. For example, if a user has initiated a trade and and has been refunded the same, the function **withdraw**() can still be called, resulting in user receiving tokens for an order which has already been refunded.

Recommendation

Same to recommendation for issue 1, a check can be added to ensure that order is not refunded.

Amended (Jan 15th 2020): Check added by Stabl.co team in commit 2b5829e878c349dab75a73de48e2db27249ec3b8. Would recommend state change for **isWithrawn** flag to be done before external function call **transfer().**

3. Denial of service

The contract should hold liquidity of all the tokens listed in order to run system smoothly. There are few cases when user's funds can get stuck if contract doesn't have liquidity. For example, assume contract has 0 liquidity and user Alice deposited 100 tokens of type1 and requested 50 tokens of type 2. Now, contract will have 100 tokens of type1 and 0 tokens of type2. As contract doesn't have 50 tokens of type2, owner will not be able to fulfil user's requirements. Next, assume Bob deposits 200 tokens of type3 and requests 100 tokens of type1. The contract now has 100 tokens of type1 and 200 tokens of type3, Now since contract can fulfil the requirements of Bob, owner sends 100 tokens of type1 to Bob, contract will have only 200 tokens of type3. In such a scenario, Alice will not be able to withdraw as contract do not have tokens of type2 and Alice will not be able to ask for a refund either as contract no longer have tokens of type1.

Recommendation
If someone is yet to withdraw/refund, consider preventing user's deposited balance to be transferred to another user.

4. Overflowing/underflowing of Integers in calculation of withdrawal amount.

Function **depositToken**() does not use **SafeMath** for all operations, which can result in an overflow/underflow of integer values. Enforce usage of **SafeMath** in every function. Line numbers: EscrowExchange.sol 255

Recommendation
Consider using **SafeMath** library by **OpenZeppelin**.

Amended (Jan 15th 2020): Issue was fixed by Stabl.co team and is no longer present in commit 2b5829e878c349dab75a73de48e2db27249ec3b8.

Major

1. Ownership renouncing can result in funds/tokens getting stuck with escrow contract.

As most functions in the Escrow contract are protected with **onlyOwner**, the function **renounceOwnership**() can result in deposited tokens being stuck forever.

Recommendation

Consider adding a check to ensure there is no fund available in escrow contract before renouncing ownership.

Amended (Jan 15th 2020): Issue was fixed by Stabl.co team and is no longer present in commit 2b5829e878c349dab75a73de48e2db27249ec3b8.

2. The owner has too much privileges.

Most of the functions are protected by **onlyOwner** modifier which is not best suited for decentralized platforms.

Recommendation
Consider removing the onlyOwner modifier from refund() and withdrawToken() functions by adding other necessary conditions in require statements that results in similar functioning of smart contract as current implementation. For example, allow a user to refund/withdraw an order based on conditions. Also, consider using some governance like mechanism or at least a multi-sig wallet to update config parameters as it will ensure decentralization in system.

3. Decimals for ERC20 token should not be taken as input.

ERC20's decimal is being passed as input which is not best practice. Ideally, decimals should be fetched from ERC20's contract. Feeding wrong value in decimals would lead to calculation flaw and more/less tokens can be served to user.

Amended (Jan 15th 2020): Issue was fixed by Stabl.co team and is no longer present in commit 2b5829e878c349dab75a73de48e2db27249ec3b8.

4. feeRate can be changed anytime by admin.

Fees for token exchange is calculated at the time when **withdrawToken()** function is called. If feeRate is changed after depositing of tokens by the user, then fee calculations will be done according to the updated feeRate which is not the same as what user had agreed for while depositing. For example, assume **feeRate** is 3% and a user deposited tokens at the present feeRate. If later, the admin updates feeRate to 50% and calls **withdrawToken()** post the fee updation, the user who agreed on 3% fees will have to bear 50% fees.
Recommendation
Consider storing feesRate at time of deposit.

Amended (Jan 15th 2020): Fee rate for a transaction is stored at the time order is placed in commit 2b5829e878c349dab75a73de48e2db27249ec3b8. However, the contract may suffer front run attacks due to editable feeRate ie., Admin can front run

user's transaction with higher gas rate to set a high feeRate before the user's transaction is mined.

5. No input checking in few functions.

Input validation is missing in several functions. uint should be checked for non-zero and address variable should be validated for != address(0) while dealing with external data. It may otherwise result in invalid data feeding.
Function names:- addNewToken, setFeeRate, setMainAddress, tradeToken, editTokenPrice.

Recommendation
Consider validating input data.

Minor

1. Same token can be listed multiple times with different id

As new tokens are getting listed by assigning an incremented integer, there is no check to avoid repetition of token, resulting in same token being added multiple times.
Recommendation
Consider storing tokens based on token address, it will prevent repetition of tokens.

2. Use separate functions for updating allowances.

Follow the standard ERC20 pattern for increasing and decreasing allowances amount by using separate functions, which resolves the front-running attack.
Example for fronrun attack:
For instance, consider Bob has given allowance of 100 tokens to Alice, but after some time Bob decides to reduce allowance by 10 and calls approve function which sets allowance as 90 tokens. Now, if Alice front runs his transaction with high gas, he will be able to transfer 90 tokens from Bob's account, and later when Bob's approve transaction gets executed, then alice will still have allowance=9, instead of 0.

Recommendation
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.

3. No Refunded event emitted.

As per the best practices, every crucial state change should emit events. Consider raising event in **refund()** function of the EscrowExchange contract.

Amended (Jan 15th 2020): Issue was fixed by Stabl.co team and is no longer present in commit 2b5829e878c349dab75a73de48e2db27249ec3b8.

4. Improper validation checks in updateTokenStatusToEndSale() function.

isValidToken variable is used for checking validity of token. However, it will only ensure that a token has been listed. It will not ensure that whether a token is open for sale or not. For example, the time when token gets listed, its isValidToken will become true and will not be set false. It doesn't matter if token is openForSale or whatever it is, the condition isValidToken will be true for every scenario, even after the token has been set to SaleEnded.

Recommendation
Consider adding following check **require(SharesTokenInfo[tokenID].tokenStatus == TokenStatus.OpenForSale)**

5. SafeMath not used for tokenCount and orderCount variable.

For the above mentioned variables, solidity's increment operator is used at lines 182 and 286, which are at times susceptible to integer overflow.

6. Smart contract not compiling.

EscrowExchange smart contract is not compiling due to the use of non-standard comment tags :-
@General, @Main and @Token.
Please refer to the following link
https://solidity.readthedocs.io/en/v0.6.0/natspec-format.html

Amended (Jan 15th 2020): Issue was fixed by Stabl.co team and is no longer present in commit 2b5829e878c349dab75a73de48e2db27249ec3b8.

Notes

1. Upgrade code to latest Solidity version

Currently, the smart contract is written in Solidity version 0.4.24 which is now a known vulnerable version of Solidity. The contracts should be updated.

Recommendation
Refer to the Ethereum changelog for the most up to date Solidity version. Contract elements may need to be updated in order to work with the latest Solidity version. Ethereum changelog can be accessed on the following link:

https://github.com/ethereum/solidity/blob/develop/Changelog.md

2. Follow the Solidity style guide

- **Follow proper linting**
  Consider following the Solidity style guide. It is intended to provide coding/naming conventions for writing solidity code.

- **Code commenting**
  Consider adding comments to functions and variable declarations in the smart contracts as it gives clarity to the reader and reduces confusion for the developers as well. Comments are missing in a few functions.

Recommendation
You can find documentation for Solidity Style guide on the following link: https://solidity.readthedocs.io/en/v0.4.24/style-guide.html. Use solhint tool to check for linting errors.

3. Gas Optimizations

- **Using appropriate visibility to functions**
  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.

- **Re-use code**

updateTokenStatusForSale() and **updateTokenStatusToEndSale**() have almost the same code with the difference that in **updateTokenStatusForSale**(), tokenStatus is set to **OpenForSale** and in **updateTokenStatusToEndSale**(), tokenStatus is set to SaleEnded.

Recommendation
Consider reusing the code. It makes the code less complex and easier to understand.

- **Unnecessary validation checks**
  - In function **tradeToken()** the require statements in line 227 and 228 can be omitted as the next two require statements are adequate for those checks.
  - As **depositToken**() function is an internal function there is no need for validations. These checks are already performed in **tradeToken**() function.
- **Use of redundant variables**
  In struct SharesToken
  - **tokenId** can be omitted as it is already acting as the key for the mapping.
  - **isValidToken** can be omitted. If the value for a key exists, that itself proves the validity of that token.

  In struct OrderBook
  - **orderId** can be omitted as it is already acting as the key for the mapping.
  - **isDeposited** can be omitted, check can be done by if **depositAmount** is greater than zero.

4. Multiplication after division

Solidity operates only with integers. Thus, if division is done before multiplication, the rounding off errors can increase dramatically.
Recommendation
It is a better practice to use multiplication before divide. It will reduce rounding off errors.

5. Provide reasons for require

Consider adding messages for require or revert statements, it gives clarity to the reader and makes debugging easier.

6. Use of Events

Events should be emitted on any crucial state change in the smart contract

- Events should be emitted when feeRate and mainAddress are changed by admin in EcrowExchange contract and when ownership is transferred in White.sol contract.
- Single event for token status update can be used. Creation of separate events for different status of token can be avoided.
- Use indexed variables for emitting events. Indexed variables are useful in filtering events by a field.

7. Test coverage

Test coverage of smart contracts is low. Consider achieving 100% test coverage.

Amended (Jan 15th 2020):

- In function **updateOrderContractToEnd(),** transactions can get reverted due to hitting block Gas limit in case the **orderCount** variable becomes larger eventually. Additionally, there should be an exit mechanism from **for** loop when the **if** statement returns true in function **updateOrderContractToEnd(),** as execution of **for** loops are expensive in terms of gas.

Closing Summary

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

We recommend that critical, major and minor issues should be resolved. Resolving the areas of notes is up to Stabl.co'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 the Stabl.co'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