Ether Authority

SMART CONTRACT AUDIT REPORT

For

Dapp Stats (Order #13SEP2019)

Prepared By: Yogesh Padsala

Prepared on: 13/09/2019

audit@etherauthority.io

Prepared For: Dapp Stats

https://www.dappstats.com

Table of Content

1. Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

2. Overview of the audit

The project has following smart contract code:

- https://rinkeby.etherscan.io/address/0x4c376c5e17f1ed1b6bf543587c62832f42668b86#code

It contains approx **230** lines of Solidity code. All the functions and state variables are **not** well commented using netscape style, but that does not raise any vulnerability. But it would increase readability.

The audit was performed by two senior solidity auditors at EtherAuthority. The team has extensive work experience in developing and auditing the smart contracts.

This audit procedure also included the use of automated software to further scan of the code to identify potential issues:

For example:

https://tool.smartdec.net/scan/814042953a55487dac4ef09cf2de4fc8

https://mythx.io tool provided as remix.ethereum.org plugin

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version is old	Not Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Not Passed
	Other programming issues	Passed
Code Specification	Visibility not explicitly declared	Moderated
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Passed

	Possibly High consumption ‘for/while’ loop	Not Passed
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	N/A
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: NOT PASSED

3. Attacks tested on the contract

In order to check for the security of the contract, we tested several attacks on the code. Some of those are as below:

3.1: Over and under flows

SafeMath library is used in the contract, which prevented the possibility of overflow and underflow attacks.

3.2: Short address attack

Although this contract **is not vulnerable** to this attack, it is highly recommended to call functions after checking the validity of the address from the outside client.

3.3: Visibility & Delegatecall

Delegatecall is not used in the contract thus it does not have this vulnerability. And visibility is also used properly except 2 places.

3.4: Reentrancy / TheDAO hack

Use of "require" function and Checks-Effects-Interactions pattern in this smart contract mitigated this vulnerability.

3.5: Forcing ether to a contract

Here, the Smart Contract's balance has never been used as guard, which mitigated this vulnerability

3.6: Denial Of Service (DoS)

There **is No** any process consuming loops in the contracts which can be used for DoS attacks. and thus this contract is not prone to DoS.

4. Good things in the smart contract

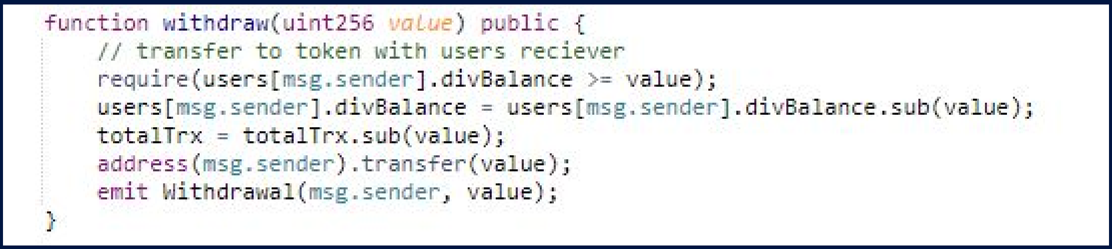
4.1 Checks-Effects-Interactions pattern

While transferring ether, this contract does all the process first and then transfers them. The same while doing other process too. This is very good practice which prevents malicious possibility. For example: withdraw() function.

4.2 Functions input parameters passed

The functions in this contract verifies the validity of the input parameters, and this validations cannot be by-passed in anyway.

4.3 Conditions validations



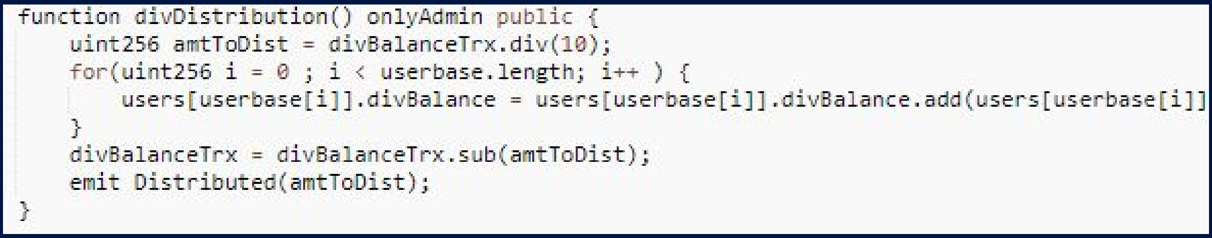
```
function withdraw(uint256 value) public {
    // transfer to token with users reciever
    require(users[msg.sender].divBalance >= value);
    users[msg.sender].divBalance = users[msg.sender].divBalance.sub(value);
    totalTrx = totalTrx.sub(value);
    address(msg.sender).transfer(value);
    emit Withdrawal(msg.sender, value);
}
```

The use of SafeMath library is a good programming flow.
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol

5. Critical vulnerabilities found in the contract

Critical issues that could damage heavily the integrity of the contract. Some bug that would allow attackers to steal ether is a critical issue.

5.1 Un-scalable for loops



```
function divDistribution() onlyAdmin public {
    uint256 amtToDist = divBalanceTrx.div(10);
    for(uint256 i = 0 ; i < userbase.length; i++ ) {
        users[userbase[i]].divBalance = users[userbase[i]].divBalance.add(users[userbase[i]]
    }
    divBalanceTrx = divBalanceTrx.sub(amtToDist);
    emit Distributed(amtToDist);
}
```

This type of active dividend distribution does not work when more users freeze their tokens. Because when users increases, then it will give timeout error or block gas limit reached error.

Resolution:

Implement passive dividend distribution. Basic programming flow can be:

=> we can think of a logic in which owner distributes dividend daily.. and it just added to a variable (dividend pool) and also updated other variables like time.

=> Then we create 'view' function for users to see how much dividend available to them and we calculate it dynamically from dividend variables and how many previous withdrawals user did and also share percentage of token frozen.

=> create a withdraw function for users to withdraw dividend if they have any positive amount. this will also update totalDividendWithdrawn variable so that users can not withdraw again and again.

6. Medium vulnerabilities found in the contract

Those vulnerabilities that could damage the contract but with some kind of limitations. Like a bug allowing people to modify a random variable.

=> No Medium severity vulnerabilities found

7. Low severity vulnerabilities found

Those do not damage the contract, but better to resolve and make code clean.

7.1: Compiler version can be fixed

The contract has lower solidity version than the current one. This version gap is quite high in contract and there were many improvements afterwards.

So, it is good practice to deploy the contract having latest solidity version. The solidity version at a time of audit is: 0.5.11

7.2: Sending ownership to incorrect address as human error

This does not happen always. But we have seen some cases (even our personal experiences) that we send ownership to incorrect address in rush or by mistake. and it will make the entire contract useless.

Solution is to implement logic where new owner has to accept ownership in order to take ownership transfer take place.

8. Very low severity vulnerabilities found

The presence of these things does not make any negative effect. But just to clean up the code.

8.1: No explicit visibility - AbleTokenSale contract

Visibility is not specified at line #93, #94. Please note that this is not a big issue as it takes default to “*public*”. But it's suggested to explicitly define visibility to avoid confusion.

9. Gas Optimization Discussion

9.1: Extra gas consumption in for loop

While using array.length in loop, it cost more gas than defining array length in a variable and then use it. Because it reads from the storage every loop iteration.

10. Discussions and improvements

10.1 Consider adding Safeguard function

In any unexpected events, owner of the contract can put safeguard (halt token movement). Once the problem is resolved, then the owner can lift the safeguard and everything comes back to normal.

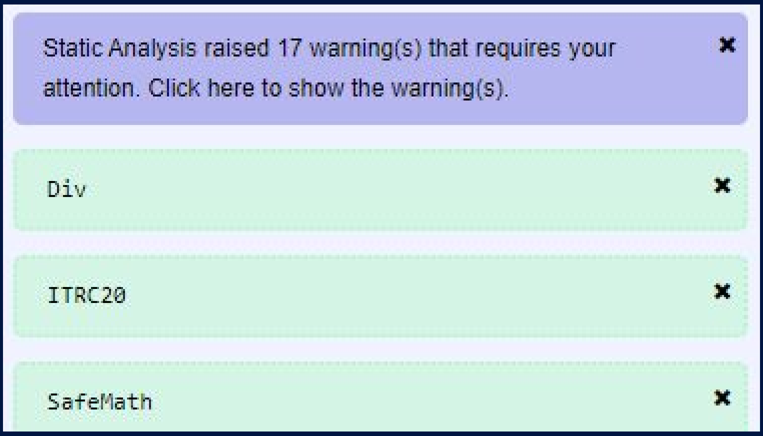
10.2 Double calls to users to freeze tokens

In current token freeze implementation, user have to approve before freezing tokens. so in GUI, user will have to approve two authentication alerts from metamask (if it is implemented that way).

If this freeze/unfreeze functions to be added in token contract (if that is still possible) then we can implement logic in which users do not have to call this approve or anything. so only one call freeze to token contract will do the things.. so no double calls to freeze tokens.

11. Summary of the Audit

Overall, the code is dividend distribution (active) and Compiler showed couple of warnings, as below:



Static Analysis raised 17 warning(s) that requires your attention. Click here to show the warning(s).

```
Div
ITRC20
SafeMath
```

Now, we checked that the warnings in purple division, are due to their static analysis, which includes like gas estimations and all. So, it is important to supply correct gas values while calling various functions.

Those warnings can be safely ignored as should be taken care while calling the smart contract functions.

Please try to check the address and value of token externally before sending to the solidity code.

It is also encouraged to run bug bounty program and let community help to further polish the code to perfection.