SMART CONTRACT AUDIT REPORT

For

Crypto Miner Token (Order # FO711C99)

Prepared By: Yogesh Padsala

Prepared on: 04/10/2018

audit@etherauthority.io

Prepared For: CryptoCloud solutions

https://minertoken.cloud

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 file:

- CryptoMinerToken.sol

It contains approx **343** lines of Solidity code. All the functions and state variables are **not** well commented using the natspec documentation. However, that does not raise any vulnerability. It just increases the readability.

The audit was performed by Yogesh Padsala, from Ether Authority Limited. Yogesh has extensive work experience of developing and auditing the smart contracts.

The audit was based on the solidity compiler 0.4.25+commit.59dbf8f1 with optimization enabled compiler in remix.ethereum.org

This audit was also performed the verification of the details exit in the main website: https://minertoken.cloud

3. Attacks tested on the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

3.1: Over and under flows

This contract **does** check for overflows and underflows by using OpenZeppelin's SafeMath to mitigate this attack, and all the functions have strong validations, which prevented this attack.

3.2: Short address attack

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

3.3: Visibility & Delegatecall

No such issues found in this smart contract and visibility also properly addressed.

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 process consuming loops in the contracts which can be used for DoS attacks. Also, there is no progressing state based on external calls, 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 the ether. This is very good practice which prevents reentrancy possibility. The functions are: exit() and withdraw().

4.2 Declaring the variable as constant

If the state variables are not supposed to be changed, then it is good practice to declare them as constant. It saves less gas compared to the variables which are not declared as constant.

4.3 Minimum data stored in the contract

This contract stores very minimum amount of data in the smart contract, which is really good thing as that minimize the gas cost to users of the contract down the road.

4.4 Good validations

This contract processes loop with good validations as well as functions are having good require conditions.

4.4 Good things in the code

- transfer function

```
133     function transfer(address _toAddress, uint256 _amountOfTokens) onlyBagholder
134         address _customerAddress = msg.sender;
135         require(_amountOfTokens <= tokenBalanceLedger_[_customerAddress]);
136         if (myDividends(true) > 0) {
137             withdraw();
138         }
139         uint256 _tokenFee = SafeMath.div(SafeMath.mul(_amountOfTokens, transferF
```

This function validates amount of tokens, gives ether if dividend is available, and also does all the checking first before transfering the tokens.

- reinvest function

```
86    function reinvest() onlyStronghands public {
87        uint256 _dividends = myDividends(false);
88        address _customerAddress = msg.sender;
89        payoutsTo_[_customerAddress] +=  (int256) (_dividends * magnitude);
90        _dividends += referralBalance_[_customerAddress];
91        referralBalance_[_customerAddress] = 0;
92        uint256 _tokens = purchaseTokens(_dividends, 0x0);
93        emit onReinvestment(_customerAddress, _dividends, _tokens);
```

This function checks for the dividends. And it processes token transfet after doing all other process in the end, which is a good thing.

- purchaseTokens function

```
222    function purchaseTokens(uint256 _incomingEthereum, address _referredBy) inte
223        address _customerAddress = msg.sender;
224        uint256 _undividedDividends = SafeMath.div(SafeMath.mul(_incomingEthereu
225        uint256 _referralBonus = SafeMath.div(SafeMath.mul(_undividedDividends,
226        uint256 _dividends = SafeMath.sub(_undividedDividends, _referralBonus);
227        uint256 _taxedEthereum = SafeMath.sub(_incomingEthereum, _undividedDivid
```

All the variables are created first and then all the validations are being done. Main thing is that the payout is done to user after all the validations which is a good thing.

- sell function

```
113    function sell(uint256 _amountOfTokens) onlyBagholders public {
114        address _customerAddress = msg.sender;
115        require(_amountOfTokens <= tokenBalanceLedger_[_customerAddress]);
116        uint256 _tokens = _amountOfTokens;
117        uint256 _ethereum = tokensToEthereum_(_tokens);
118        uint256 _dividends = SafeMath.div(SafeMath.mul(_ethereum, exitFee_), 100
119        uint256 _taxedEthereum = SafeMath.sub(_ethereum, _dividends);
```

This functions does a great validations before selling tokens.

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.

=> No critical vulnerabilities found

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 vulnerabilities found

7. Low severity vulnerabilities found

7.1: Deprecated element

At line #157, using contract member "balance" inherited from the address type is deprecated in new solidity version. Convert the contract to "address" type to access the member. Please use as like this:

```
return  address(this).balance;
```

7.2: Compiler version should be fixed

Although, this is not a big issue, but source files indicate the versions of the compiler they can be compiled with.

```
pragma solidity ^0.4.17; // bad: compiles w 0.4.17 and above
```

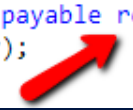
```
pragma solidity 0.4.17; // good : compiles w 0.4.17 only
```

It is recommended to follow the second example, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

7.3 No return statement in buy function

There is no return value for a function whose signature only denotes the type of the return value

```
78    function buy(address _referredBy) public payable returns (uint256) {
79        purchaseTokens(msg.value, _referredBy);
80    }
```



If you don't need the return value of the function, do not specify returns in function signature.

7.4 Costly loop possibility

The function sqrt() implements 'while' loop. In ideal condition that does not raise any problem as there will not be many iterations. But still there is possibility where loop can go out of control and which max out the block's gas limit making the contract stuck. Please try to put logic to restrict more potential iterations.

7.5 Unchecked Math

Safemath library is included, which is good thing. But at some place, it is not used.

This is not a big issue, as validations are done well. But it is good practice to use it at all the mathematical calculations. Following lines does not have safemath used.

#248, #179, #149, #255, #270-276, #311, #248, #249, #263, #125, #150, #197, #148, #106, #295-298, #128, #89, #124, #243, #231, #185, #171, #70, #90, #107

Please implement Safemath at those places.

8. Discussions and improvements

8.1 Putting higher degree of control

It is good idea to put ability for owner to put safeguard in the code. So, let's say for example, there would be any un-intended event occurred in the future, then owner can put a safeguard and which prevents all the process from happening until the issue is resolved.

This can be easily achieved by declaring a variable for that, which can be used in all the functions. Admin can make this variable true or false. Another way is to create modifier for that and use it in every function.

8.2 Declaring custom function for require () validation

It is good idea to create a custom function for the require condition, and make an Event to fire in case of failed "required" validation. It helps client to better understand why any possible error occurred.

```
bool internal dorequireRevert; // <=== IMPORTANT DEBUG/REVERT SWITCH
                                    // false => keep going but emit RequireFailed
                                    // true  => do the revert
function dorequire(bool testresult, string message) internal {
        if (!testresult) {
                emit RequireFailed(message);
                if (dorequireRevert) {
                        require(false, message);
                }
        }
}
dorequire (1 != 0, "one is not equal zero!");
```

8.3 Timestamp dependence awareness

This contract depends on the timestamp as places like #130 and #257. There is nothing wrong in that but please be aware that the timestamp of the block can be slightly manipulated by the miner.

8.4 Style guide violation

In Solidity, function and event names usually start with a lower- and uppercase letter respectively:

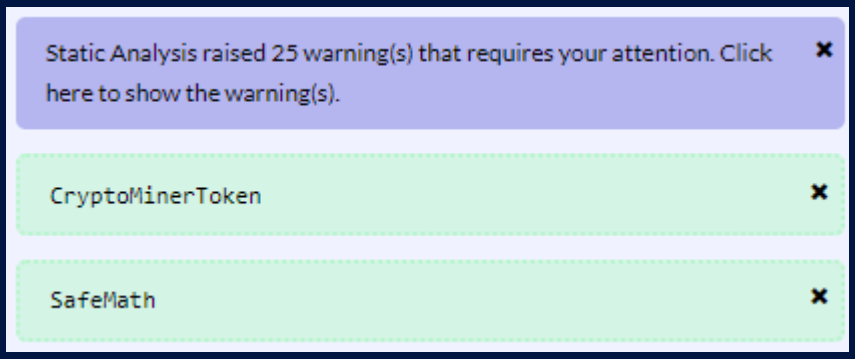
```
function foo();    // good
event LogFoo();  // good
```

Violating the style guide decreases readability and may lead to confusion. Thus, please follow the style guide at these lines: #27, #36, #44, #50

9. Summary of the Audit

Overall the code performs good data validations as well as meets the calculations according to the information presented in the website: https://minertoken.cloud

The compiler also displayed 25 warnings:



Now, we checked those warnings are due to their static analysis, which includes like gas errors 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.

There are many places in the contract where many variables are marked as “Private”. One thing to understand is that, making variables private, does not make a them invisible. Miners have access to all contracts’ code and data. Developers must account for the lack of privacy in Ethereum

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