SMART CONTRACT AUDIT REPORT

For

CryptoMinerFund (Order # 22OCT2018A)

Prepared By: Yogesh Padsala

Prepared on: 23/10/2018

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:

- CryptoMinerFund.sol

It contains approx **179** 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 EtherAuthority 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

Delegatecall is not used in the contract thus it does not have this vulnerability. And visibility is mostly used properly. There are some places where it was not used, but that does not raise any problems as well.

3.4: Reentrancy / TheDAO hack

Use of “require” function (in token contract) 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 Fallback function manipulation without ether

The fallback function can be called without sending ether which calls requestPayDay() function. But the use of the good validations and logic, users can not withdraw any funds.

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 function is: requestPayDay().

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.

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

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

7.1: Compiler version should be fixed

Although, this is not a big issue, but the code has 'open' solidity compiler version.

```
pragma solidity ^0.4.25; // bad: compiles w 0.4.17 and above
```

```
pragma solidity 0.4.25; // 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.

https://ethereum.stackexchange.com/questions/50071/solidity-best-practices-which-compiler-version-should-i-use-advantages-dis

7.2 Unchecked Math

Safemath library is included, which is good. 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.

At line #66, please use:

```
walletDeposits[msg.sender]=walletDeposits[msg.sender].add( msg.value );
```

Same way at line #90, please use:

```
withdrawedAmounts[msg.sender]=withdrawedAmounts[msg.sender].add( payDay);
```

Please implement Safemath at those places.

7.3 Implicit visibility level

The default function visibility level in Solidity is public. Explicitly define function visibility to prevent confusion.

Please define visibility at these lines: #32, #29, #45, #39, #30, #41, #43, #44, #46, #48, #49, #50

7.4 Use of contract ether balance

At line number #126 and #149, there is use of contract's ether balance. Now, there is nothing wrong with it as that is not used as the main logic of the contract.

But just to aware that contract ether balance can be manipulated without calling any fallback function.

For example, sending some ether to the contract using self-destruct. In that case, users might slightly manipulate the phase percentage.

7.5 Code optimization

```
65      if(walletDeposits[msg.sender]>0){
66          walletDeposits[msg.sender] += msg.value;
67          walletTimer[msg.sender] = now;
68      }
69      else{
70          walletDeposits[msg.sender] = walletDeposits[msg.sender].add(msg.value);
71      }
72
73      walletTimer[msg.sender] = now;
```

The if…else condition at line number #65 is not needed because the walletDeposits mapping gets updated regardless walletDeposits[msg.sender] value is zero or not.

Also, the code at line #73, walletTimer[msg.sender] = now; runs regardless of if..else condition above it, so it is no need to put that same line at #67

This entire code can be simply replaced by:

```
walletDeposits[msg.sender] = walletDeposits[msg.sender].add(msg.value);

walletTimer[msg.sender] = now;
```

Developer should keep in mind that a single line of code increase gas cost to users and that may accumulate into huge sum in the future.

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 Ability to change _parojectMarketing address

It is good idea to have a function where admin or owner can change this address where all the referral bonus is going.

This is rather useful in any un-expected event where the key of this wallet address is stolen or lost.

Creating a modifier 'onlyOwner' also would do the trick!

8.3 Timestamp dependence awareness

This contract depends on the timestamp as places like #67 and #73. 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 Name of HourglassInterface Interface

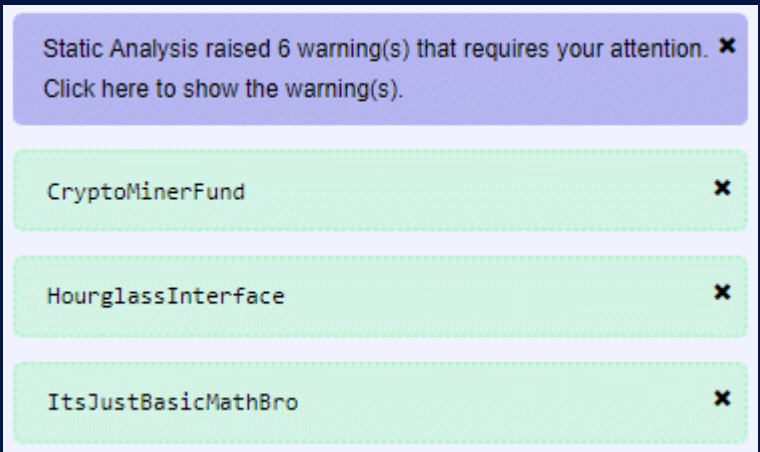
It is good for readability, that the name, HourglassInterface would be renamed to something meaningful as it reference to CMT Contract.

As like: CMTContractInterface

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 6 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.

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