SMART CONTRACT AUDIT REPORT

For

Bitherplatform (Order # 08FAB2019A)

Prepared By: Yogesh Padsala

Prepared on: 08/02/2019

audit@etherauthority.io

Prepared For: Bitherplatform

https://bitherplatform.io

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:

- NNNToken.sol
- MMMToken.sol
- MMMCrowdsale.sol
- PaymentDistributor.sol

It contains approx **1640** lines of Solidity code (including all the imported code from the OpenZeppelin). All the functions and state variables are well commented using the natspec documentation, which increased the readability: https://github.com/ethereum/wiki/wiki/Ethereum-Natural-Specification-Format

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 0.5.3+commit.10d17f24 with optimization enabled compiler in remix.ethereum.org.

This audit was also performed verification of the details exist in whitepaper: https://bitherplatform.io/docs/Whitepaper.pdf

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	Passed
	Other programming issues	Passed
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	Evil mint/burn	Passed
	The maximum limit for mintage not set	Passed
	“Fake Charge” Attack	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed
Auto Fuzzing		Passed

Overall Audit Result: PASSED

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 also used properly at most 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 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 Declaring variables as constant

The value of variables from line number #552 to #582 in MMMCrowdsale.sol, is not expected to change. Thus it is good thing to declare them as constant, which helps reduce the gas cost.

4.2 Use of ReentrancyGuard

```
function buyTokens(address beneficiary) public nonReentrant payable {
    uint256 weiAmount = msg.value;
    _preValidatePurchase(beneficiary, weiAmount);
```

This is good feature to prevent re-entrancy attack.

4.3 Use of Openzeppelin code

This code is audited by community and time tested and used by many companies as standard to create smart contract. Thus, this code is seen as virtually “security issues free”.

4.4 Checks-Effects-Interactions pattern

While transferring tokens, 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: buyTokens() function in MMMCrowdsale.sol.

4.4 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.

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 - Good job team!

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 - Good job again!

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

Firstly, we have newer version of solidity compiler (0.5.3). So, better to use latest one.

Secondly, it is recommended to remove caret (^) symbol in solidity pragma declaration. The reason why Openzeppelin team keeps it because that will allow other compiler versions also can use their code. But in production, it is recommended to remove it.

Although there is no problem once the code is deployed, but this may cause issues in case company wish to re-use the code in the future, as there may be many things would have changed in future versions of solidity at that time.

https://ethereum.stackexchange.com/questions/45231/pragma-solidity-0-4-11-vulnarability

8. Gas Optimization Discussion

=> Contract is **most optimum** for the gas cost. There is no gas expensive loops, or logical unnecessary processes.

9. Discussions and improvements

9.1 approve() of ERC20 Standard

To prevent attack vectors regarding approve() like the one described here: https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit , clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. THOUGH the contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

9.3 Custom error message in require() function

It is good idea to specify a custom error message in require function, which can be useful in GUI and error debugging down the road.

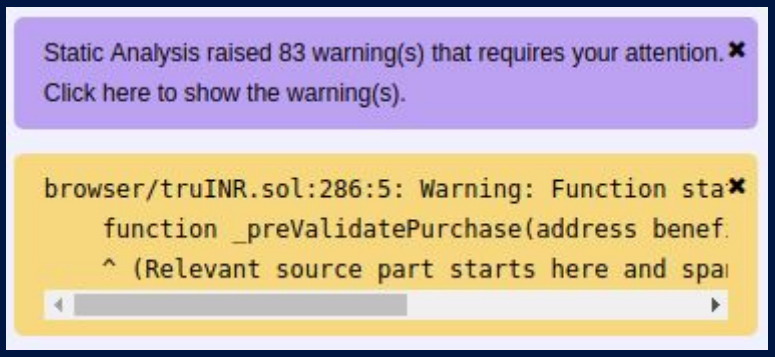
9.4 While using SafeMath library

The SafeMath library is doing the great job to prevent overflow and underflow. However, it is recommended **NOT** to use it when overflow/underflow is impossible. Because please keep in mind that every unnecessary checks contribute to increased gas cost!

10. Summary of the Audit

Overall the code performs good data validations as well as meets the correctness of data according to the information presented in the whitepaper: https://bitherplatform.io/docs/Whitepaper.pdf

The compiler also displayed 83 warnings for MMMCrowedsale contract:



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.

But the warnings in orange color can be resolved. Although that does not raise any vulnerabilities, but better to resolve them and make code warnings free.

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.