SMART CONTRACT AUDIT REPORT

For

Banknet Token (Order #02 MAY2019)

Prepared By: Yogesh Padsala

Prepared on: 02/05/2019

audit@etherauthority.io

Prepared For: Banknet Token

https://banknetico.com

Table of Content

1. Disclaimer

2. Overview of the audit

3. Attacks made to the contract

4. Good things in smart contract

5. Critical vulnerabilities found in the contract

6. Medium vulnerabilities found in the contract

7. Low severity vulnerabilities found in the contract

8. Gas Optimization Discussion

9. Discussions and improvements

10. Summary of the audit

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:

- https://etherscan.io/address/0x18e46125233cb973cc04ae4f0a8f1ff63ed9541c#code

It contains **224** lines of Solidity code. All the functions and state variables are well commented, which raises readability.

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

This smart contract reflects correct data according to white paper found at:

https://banknetico.com/wp-content/uploads/2019/04/finalwhitepaper-1.pdf

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/23bcca626d2b4886a12e2f8c92f294a0

We checked those reports carefully and confirm that some of the warnings, either are just for information purpose or not very critical for our use case!

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	Visibility not explicitly declared	Not 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	N/A
	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: 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 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.

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 Accept ownership

It is often time overlooked by developers by just transferring ownership of address provided. We have seen incidents when ownership transferred to wrong address by mistake. So, to mitigate this human error, accept ownership function is a good thing.

4.2 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: transfer() function.

4.3 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 No unnecessary validations

```
function transfer(address to, uint tokens) public returns (bool success) {
    balances[msg.sender] = balances[msg.sender].sub(tokens);
    balances[to] = balances[to].add(tokens);
    emit Transfer(msg.sender, to, tokens);
    return true;
}
```

The SafeMath library checks for user token balance, as well as overflow. Hence, no need for extra validation conditions, which is good thing.

4.4 Unstuck token transfer

```
function transferAnyERC20Token(address tokenAddress, uint tokens) public
    return ERC20Interface(tokenAddress).transfer(owner, tokens);
}
```

The function, transferAnyERC20Token, allows owner to unstuck any tokens sent to this contract by mistake.

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

The contracts has lower solidity version than current one. This version gap is not too much and does not break anything, or generate any vulnerabilities.

However, it is good practice to deploy the contract having latest solidity version.

7.2: Explicit visibility declaration

Line number #107, #109, #110 does not have explicit visibility specified. Solidity takes "public" visibility by default, but it is good practice to specify visibility explicitly.

8. Gas Optimization Discussion

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

9. Discussions and improvements

9.1 No direct burn function

This contract does not have direct burn function. So, to burn any tokens, users have to send that to zero address (0x0).

9.2 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 While using SafeMath library

We **do not** recommend using SafeMath library for all arithmetic operations. It is good practice to use explicit checks where it is really needed, and to avoid extra checks where overflow/underflow is impossible, which is done here optimally!

9.4 Consider using upgradable contracts

It many times happens, where contract owner would need to upgrade the contract or to add any important feature in the contract.

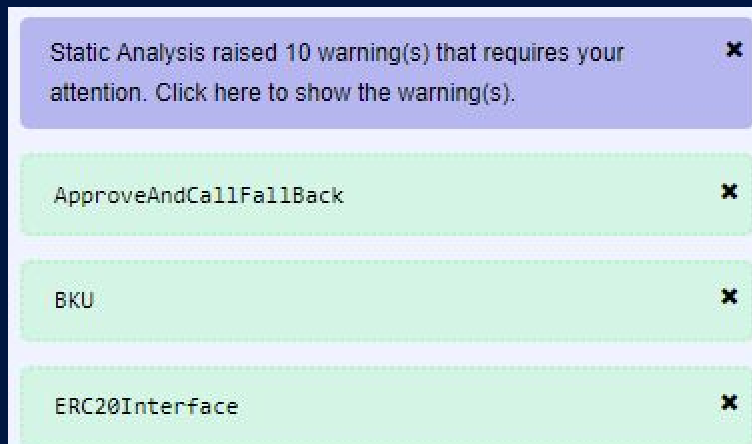
https://github.com/zeppelinos/labs/tree/master/upgradeability_using_unstructured_storage

On flip side, this pattern centralises the process removing immutability of the contract. So, consider this only if this pattern is okay with your business model.

10. Summary of the Audit

Overall, the code is simple and straightforward. apart from few improvements suggested above, rest is pretty good.

Compiler showed 10 warnings, as below:



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 the perfection.