AntiHACK.me

ASIA'S TOP BUG BOUNTY PLATFORM

SMART CONTRACT AUDIT

PREPARED FOR

VISIONBANKER.IO [VBK TOKEN]

(Revised) REPORT DATE 10st June 2019

Table of Contents

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://github.com/visionbanker/smart-contract/blob/master/contracts/VBK.sol

It contains approximately 431 lines of Solidity code. All the functions and state variables are well commented using the natspec documentation, which increases the readability.

This audit performed verification of the details that exist in the following website: https://www.visionbanker.io

This audit also used automated scanning tools to identify any issues: https://tool.smartdec.net/scan/4e6b13c355494b919f25819b5c7973b8

Now, we carefully analyzed the warnings found by the above tool as not relevant to our use case or just for informational purposes.

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version is old	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	N/A
	High consumption 'for/while' loop	N/A
	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

Overall Audit Result: PASSED

3. Attacks Made to the Contract

In order to test the security of the contract, we tested several attacks in order to ensure that the contract is secure and that best practices are followed.

3.1 Over and underflows

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

3.2 Short address attacks

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, and thus this vulnerability does not exist in this contract. Visibility is also properly used.

3.4 Reentrancy/TheDAO hack

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

3.5 Forcing ether to a contract

The smart contract's balance was never used as a guard, therefore this vulnerability was mitigated.

3.6 Denial of Service (DOS)

There are no process consuming loops in the contract which could be used for DoS attacks. There are also no progressing states based on external calls, therefore this contract is not prone to DoS.

4. Good Things in Smart Contract

4.1 Checks-Effects-Interactions pattern

While transferring tokens, this contract does all the processes first, then transfers them. This is very good practice which prevents the possibility of malicious activity. For example: transfer() and approve() functions.

4.2 Use of Open Zeppelin standard

Open Zeppelin creates some of the best libraries and code templates in the industry. Their codes are community audited and time-tested. Therefore, it is an advantage that this project uses this as the standard code.

4.3 Functions input parameters passed

Due to the usage of OpenZeppelin, the code is well commented, which increases the readability.

4.4 Custom error message in require()

It is a good thing that custom error messages are used in rewuire() function, which makes debugging really easy

4.5 Well commented code

Thanks to OpenZeppelin, the code is well commented, which increases the readability

5. Critical Severity Vulnerabilities Found in the Contract

Critical issues that could heavily damage the integrity of the contract. Bugs that allow an attacker to steal ether is considered a critical vulnerability.

5.1 Inability to mint tokens, or no initial supply of the tokens [FIXED]

Although there is a standard_mint() function present in the ERC20 contract, there is no way to create any tokens. Because the function is internal, it can only be called within the contract. There are no other functions calling the contract to mint tokens.

If minting of new tokens are not required as the business login, then there should be an initial supply of tokens to be created.

So, either assign an initial supply of coins in the constructor function of the VBKToken contract, or create another minting function named internal_mint() to generate new tokens as per the business logic.

This issue is fixed by VisioBanker team

6.Medium Severity Vulnerabilities Found in the Contract

Medium severity vulnerabilities that could damage the contract, but with some form of limitation. Bugs that allow modification of random variables are considered a medium vulnerability.

- No Medium Vulnerabilities Found

7.Low Severity Vulnerabilities Found in the Contract

Low severity vulnerabilities do not damage the contract, but it would be better to resolve them in order to clean up the code.

7.1 Compiler version can be fixed [FIXED]

The contract has a lower solidity version

than the current one. The version gap is not big, neither does it generate any vulnerabilities or break any functions. However, it is good practice to deploy the contract with the latest solidity version, which is 0.5.9 at the time of this audit.

This issue is fixed by VisioBanker team

8. Discussions and Improvements

8.1 approve() of ERC20 Standard

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

8.2 While using SafeMath library

The SafeMath library is doing a great job in preventing overflow and underflow. However, it is recommended **NOT** to use it when overflow/underflow is impossible. This is because every unnecessary check contributes to gas cost.

8.3 Private Modifiers

The private modifiers are used in ERC20 contracts. Do note that private modifiers do not make its values private. Miners can access all of the information as everything is public in the Ethereum blockchain.

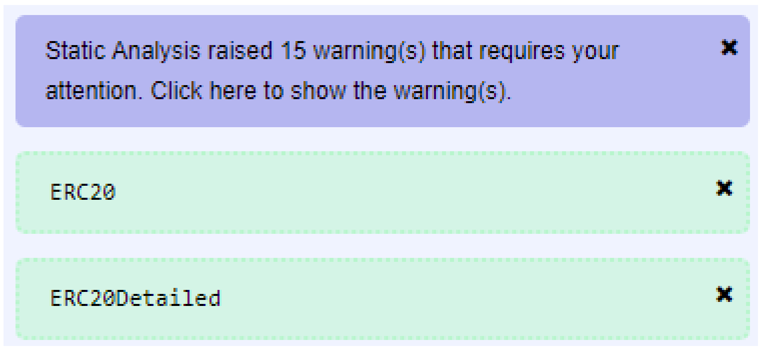
8.4 Absence of Safe-guard functionality

The option of “global halt” or “safeguard” functionality is missing in the code. Adding this feature will allow the owner to stop any token movements in event of any unexpected situations or attacks.

9. Summary of the Audit

Overall the code performs good data validations and performs all the ERC20 standard functions, using OpenZeppelin code templates.

The compiler also displayed 15 warnings:



```
Static Analysis raised 15 warning(s) that requires your attention. Click here to show the warning(s).

ERC20

ERC20Detailed
```

The warnings in the purple section were checked, and it was established that they appeared due to their static analysis, which includes gas estimations and such. Therefore, it is important to supply the correct gas values when calling various functions.

These warnings can be safely ignored as they would be taken care of when calling the smart contract functions.

Do also check the address and value of the token externally before sending it to the solidity code.

It is also encouraged to run a bug bounty program and let the community assist in further polishing the code.

- End of Report -

AntiHACK.me