SMART CONTRACT AUDIT REPORT

For

DlikeToken (Order #15MAR2019A)

Prepared By: Yogesh Padsala

Prepared on: 15/03/2019

Revised on: 16/03/2019

audit@etherauthority.io

Prepared For: DLike Group

https://dlike.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:

- DlikeToken.sol

It contains approx **378** lines of Solidity code. All the functions and state variables are well commented using the natspec documentation, which increases readability.

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

The audit was based on the solidity compiler 0.5.6+commit.b259423e with optimization enabled compiler in remix.ethereum.org

This audit procedure also included the scanning the code from other third party softwares to further identification of issues:

https://tool.smartdec.net/scan/46c4b3125df4473a8bd00d249dfa902c

As seen in above smartdec report, there are many warnings displayed. Now, we checked carefully about those and we confirm that many of those are not relevant to DLike use case and many are just for information.

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too 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	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	Passed
	“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 in order to make sure that the contract is secure and follows best practices.

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 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.

4.2 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 Good input validations

```
function _transfer(address _from, address _to, uint _value) internal {
    //checking conditions
    require(!safeguard);
    require (_to != address(0));                    // Prevent transfer to 0x0 ad
    require(!frozenAccount[_from]);                 // Check if sender is froze
    require(!frozenAccount[_to]);                   // Check if recipient is fr
    // overflow and undeflow checked by SafeMath Library
    balanceOf[_from] = balanceOf[_from].sub(_value);    // Subtract from the sender
    balanceOf[_to] = balanceOf[_to].add(_value);        // Add the same to the reci
```

This function checks all the possible data sets to be valid.

4.4 Declaring variables as constant

Line number #111, #112, #113 the variables are declared as constant, which is good as it saves some gas cost!

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.

=> No Low vulnerabilities found - Good job team!

8. Discussions and improvements

8.1 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

8.2 Over power user

This contract has only one owner, which does all the admin only functions. Now, it is good idea to have multiple admins and delegates roles and responsibilities to those. This is useful in any unintended events such as: death of owner or loss of private key of owner account.

8.3 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.

8.4 Maximum minting limit is not set

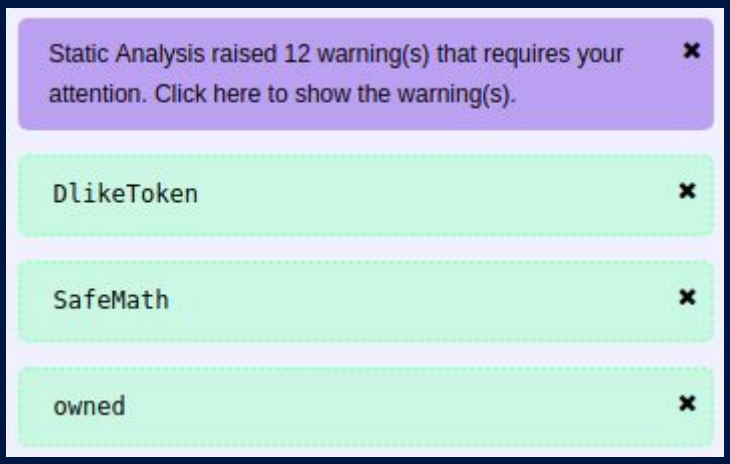
Owner must take responsibility as how many tokens should be minted. The more tokens generation would cause inflation in the system causing reduction of the token value in entire token ecosystem.

Edit: Dlike team has rectified this. And contract is no longer prone to this issue!

9. Summary of the Audit

Overall, the code is well commented and performs all the data validations.

Compiler also showed 12 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.