EtherAuthority

MODERN LIQUIDITY TOKEN SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer:	MODIC LTD (https://modic.fund)
Prepared on:	28/03/2021
Platform:	Binance Smart Chain
Language:	Solidity
Audit Type:	Intensive

audit@etherauthority.io

Table of contents

Project file

Name	Smart Contract Code Review and Security Analysis Report for Modern Liquidity Token (MLT)
Platform	Binance Smart Chain / Solidity
File 1	MLT.sol
File 1 MD5 hash	718199152918C2490F74F105143010EC
File 1 BscScan Contract URL	https://testnet.bscscan.com/address/0x80f4854ac56075e9aa7bb7456084f9a79083035c#code

Introduction

We were contracted by the MODIC team to perform the Security audit of the smart contracts code. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on 28/03/2021.

The Audit type was Intensive Audit. Which means this audit is concluded based on Standard audit scope, which is two security engineers performing audit procedure for 4 days. This document outlines all the findings as well as AS-IS overview of the smart contract codes.

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	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	Moderated
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Resolved
	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	Moderated
	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

Executive Summary

According to the **Intensive** audit assessment and after resolution of the issues identified, Customer`s solidity smart contract is **well secured**.

Insecure
Poor secured
Secure
Well-secured
You are here

We used various tools like MythX, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in AS-IS section and all issues can be found in the Audit overview section.

We found 2 critical, 2 medium and 1 low and some very low level issues.

Code Quality

We were given 1 smart contract file of MODIC Protocol. This smart contract also contains Smart contract inherits and Interfaces.

The abstract in the MODIC protocol is part of its logical algorithm. An abstract is a type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in the MODIC protocol.

The MODIC team has **not** provided scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Overall, code parts are **not** well commented. Commenting can provide rich documentation for functions, return variables and more. Ethereum Natural Language Specification Format (NatSpec) is recommended.

Documentation

We were given MODIC smart contracts code in the form of a file. The hash of that file and its BscScan testnet web links are mentioned in the above table.

As mentioned above, most code parts are not well commented. so anyone can not quickly understand the programming flow as well as complex code logic.

Another source of the reference was its official website modic.fund and blog: https://medium.com/modic/introduction-to-the-modic-ecosystem-462fead5df6f

Use of Dependencies

As per our observation, this smart contract interacts with other BEP20 token smart contracts, hence this is dependent on those token contracts.

Apart from that, MODIC smart contracts depend on an interconnected set of smart contracts.

AS-IS overview

Modern Liquidity Token protocol is a BEP20 compatible token smart contract running on Binance Smart Chain, with other features like staking, reward pool, etc. Following are the main components of core smart contracts.

MiningPool.sol

(1) Inherited contracts

(a) Ownable: ownership contract
(b) PoolRewardToken: BEP20 token functions

(2) Interfaces

(a) IERC20

(3) Structs

(a) Investor: Info about each investor user
(b) BlockInfo: Info about each blocks

(4) Events

(a) event Deposit(address indexed investor, uint256 indexed token, uint256 value);
(b) event Harvest(address indexed investor, uint256 value);
(c) event Withdraw(address indexed investor, uint256 indexed token, uint256 value);
(d) event FeesSpent(address indexed to, uint256 value);
(e) event StepChanged(uint8 newValue);
(f) event LockDurationChanged(uint256[14] values);
(g) event PoolRewardsChanged(uint256[14] values);
(h) event MinDepositsChanged(uint256[14] values);

(5) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	constructor	write	Passed	No Issue	Passed
2	setTokenAddress	write	Anyone can call this	This must be onlyOwner	Rectified
3	setBlockStep	write	Passed	No Issue	Passed
4	setLockDurations	write	Passed	No Issue	Passed
5	setMinDeposits	write	Passed	No Issue	Passed
6	setPoolRewards	write	Passed	No Issue	Passed
7	currentBlock	read	Passed	No Issue	Passed
8	getBlockTotalDeposits	read	Passed	No Issue	Passed
9	getPoolRewards	read	Passed	No Issue	Passed
10	getPointLength	read	Passed	No Issue	Passed
11	recordHistory	write	Passed	No Issue	Passed
12	recordHistoryNeeded	read	Passed	No Issue	Passed
13	_recordHistory	internal	Passed	No Issue	Passed
14	getRewardSum	write	Gas intensive	No Issue	Passed
15	_deposit	internal	Passed	No Issue	Passed
	deposit0	write	Passed	No Issue	Passed
	deposit1	write	Passed	No Issue	Passed
	deposit2	write	Passed	No Issue	Passed
	deposit3	write	Passed	No Issue	Passed
	deposit4	write	Passed	No Issue	Passed
	deposit5	write	Passed	No Issue	Passed
	deposit6	write	Passed	No Issue	Passed
	deposit7	write	Passed	No Issue	Passed
	deposit8	write	Passed	No Issue	Passed
	deposit9	write	Passed	No Issue	Passed
	deposit10	write	Passed	No Issue	Passed
	deposit11	write	Passed	No Issue	Passed
	deposit12	write	Passed	No Issue	Passed
	deposit13	write	Passed	No Issue	Passed
	canHarvest	read	Passed	No Issue	Passed
	harvestReward	write	Minting stops on max supply	Minting should be resumed on supply reduction	Rectified

	_withdraw	write	Passed	No Issue	Passed
	withdraw0	write	Passed	No Issue	Passed
	withdraw1	write	Passed	No Issue	Passed
	withdraw2	write	Passed	No Issue	Passed
	withdraw3	write	Passed	No Issue	Passed
	withdraw4	write	Passed	No Issue	Passed
	withdraw5	write	Passed	No Issue	Passed
	withdraw6	write	Passed	No Issue	Passed
	withdraw7	write	Passed	No Issue	Passed
	withdraw8	write	Passed	No Issue	Passed
	withdraw9	write	Passed	No Issue	Passed
	withdraw10	write	Passed	No Issue	Passed
	withdraw11	write	Passed	No Issue	Passed
	withdraw12	write	Passed	No Issue	Passed
	withdraw13	write	Passed	No Issue	Passed
	sendFeeFunds	write	Transfer event must be fired	Add Transfer Event	Rectified
	getInvestor	read	Passed	No Issue	Passed

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to tokens loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

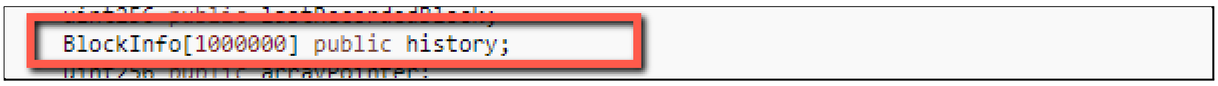
Audit Findings

Critical

(1) setTokenAddress function can be called by anyone. It should be onlyOwner function. If the plan was to add all those tokens before going to public launch, then it does not create a vulnerability, as these tokens can be set only once.

Resolution: The MODIC team confirmed that they are making it onlyOwner function.

(2) Fixed size array:



```
BlockInfo[1000000] public history;
```

This would make history for only 1 million elements. It would stop all the dependent functions after this million elements are filled.

Resolution: MODIC team has acknowledged and rectified this issue by removing this limit in revised code.

High

No high severity vulnerabilities were found.

Medium

(1) Token minting in the harvestReward function stops, when it reaches MAX_SUPPLY. If this is part of the plan, then this is not a vulnerability.

```
        if (totalSupply == MAX_SUPPLY) {
            recordHistory();
            miningFinished = true;
        }
```

Here, the issue arises in the case, when people/owner decide to burn tokens and which reduces the totalSupply. But this does not resume the minting again. So, once the minting stopped, then it's permanent regardless of the case of reduced supply due to burn.

Resolution: MODIC team acknowledged and resolved this issue.

(2) Missing Transfer event in sendFeeFunds

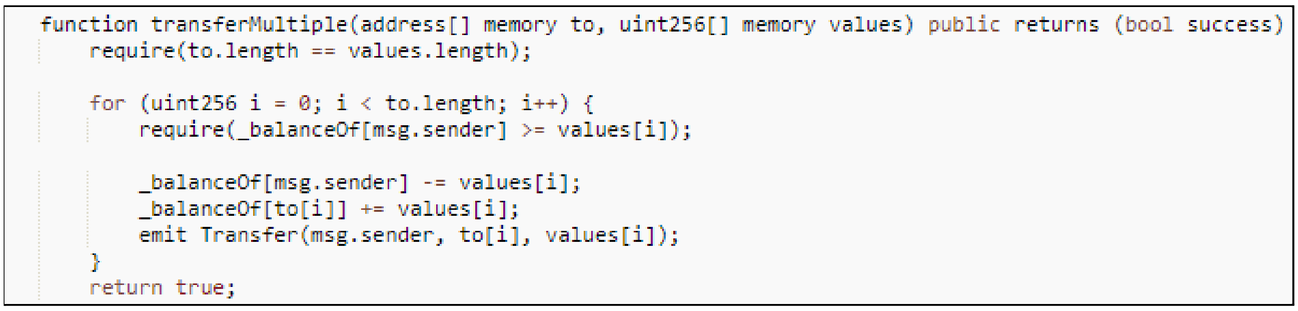
```
    function sendFeeFunds(address to, uint256 amount) public onlyOwner {
        require(feesBalance >= amount, "Insufficient funds");

        _balanceOf[to] += amount;
        feesBalance -= amount;
        emit FeesSpent(to, amount);
    }
```

Since token transfer is happening here. So, it is important to have a Transfer event. so that can be properly tracked in the explorers and client side.

Resolution: This issue is acknowledged.

Low

(1) Infinite loops possibility:



```
function transferMultiple(address[] memory to, uint256[] memory values) public returns (bool success)
    require(to.length == values.length);

    for (uint256 i = 0; i < to.length; i++) {
        require(_balanceOf[msg.sender] >= values[i]);

        _balanceOf[msg.sender] -= values[i];
        _balanceOf[to[i]] += values[i];
        emit Transfer(msg.sender, to[i], values[i]);
    }
    return true;
```

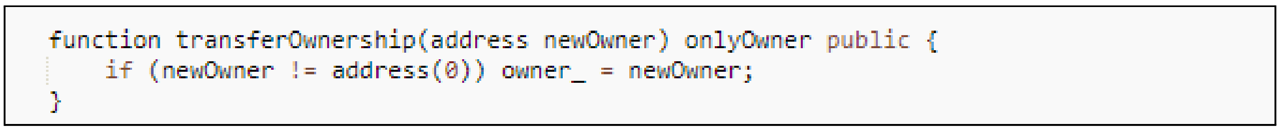
transferMultiple function in PoolRewardToken contract is having array.length directly used in the loop. It is recommended to put some kind of limits, so it does not go wild and create any scenario where it can hit the block gas limit.

Resolution: We got confirmation from the MODIC team that the array will be provided as limited length. And this will be taken care of from the client side.

Very Low / Best Practices

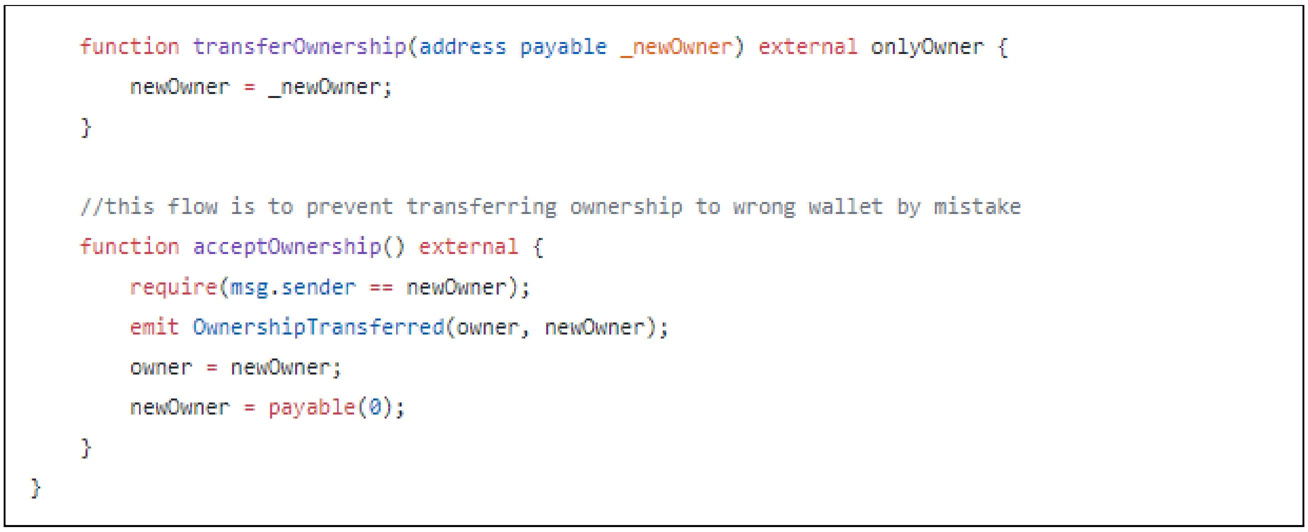
(1) Ownership transfer function:

Ownable.sol smart contract has active ownership transfer. This will be troublesome if the ownership was sent to an incorrect address by human error.



```
function transferOwnership(address newOwner) onlyOwner public {
    if (newOwner != address(0)) owner_ = newOwner;
}
```

so, it is a good practice to implement acceptOwnership style to prevent it. Code flow similar to below:



```
    function transferOwnership(address payable _newOwner) external onlyOwner {
        newOwner = _newOwner;
    }

    //this flow is to prevent transferring ownership to wrong wallet by mistake
    function acceptOwnership() external {
        require(msg.sender == newOwner);
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
        newOwner = payable(0);
    }
}
```

Resolution: MODIC team acknowledged this, as this should be taken care of from admin side.

(2) Use the latest solidity version while contract deployment to prevent any compiler version level bugs.
Resolution: This issue is acknowledged.

(3) combine all withdraw functions 0-13 into one and pass a param which identifies a particular array index. The same thing with all those deposit functions.
Resolution: This is acknowledged.

(4) Any functions which are not called internally, so it is better to set its visibility as external instead of public. It is more efficient and it saves gas.
https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices
Resolution: This is acknowledged.

Conclusion

We were given contract code. And we have used all possible tests based on given objects as files. We observed some issues, which we reported to the MODIC team. And we got confirmation from the MODIC team that they have resolved the issues identified. **So now it's good to go for production.**

Since possible test cases can be unlimited for such extensive smart contract protocol, so we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high level description of functionality was presented in the As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract, based on extensive audit procedure scope is "**Well Secured**".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Due to the fact that the total number of test cases are unlimited, so the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest to conduct a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

Email: audit@EtherAuthority.io