LiquidMarketplace

Smart Contract Audit Report

January 25, 2022

IMMUNE BYTES

Introduction 3
- About LiquidMarketplace 3
- About ImmuneBytes 3

Documentation Details 3

Audit Process & Methodology 4

Audit Details 4

Audit Goals 5

Security Level Reference 5

Contract Name: LiquidMarketVoting 6
- High Severity Issues 6
- Medium severity issues 7
- Low severity issues 8
- Recommendations/Informational 10

Contract Name: LiquidMarket 12
- High Severity Issues 12
- Medium severity issues 12
- Low severity issues 12
- Recommendations/Informational 14

Automated Audit Result 16

Unit Test 17

Concluding Remarks 18

Disclaimer 18

Introduction

1. About LiquidMarketplace

Liquid Marketplace began with our passion for collectible items. As the market matured, we saw a tremendous opportunity for growth. We also saw a divide. Growing prices meant die-hard fans and collectors were often priced out. As collectors first, we wanted to change that.

They want to make high-valued collectibles accessible to anyone interested in building their collection. To create a level playing field where those who truly appreciate these unique items can own something legendary.

Visit https://www.liquidmarketplace.io/ to know more about it.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 125+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-ups with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The LiquidMarketplace team has provided the following doc for the purpose of audit:

1. https://github.com/likezninjaz/liquid-market-smart-contract/blob/master/README.md

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: LiquidMarketplace
- Token Name: LiquidMarket.sol, LiquidMarketVoting.sol
- GitHub Address: https://github.com/likezninjaz/liquid-market-smart-contract
- Commit Hash for Initial Audit: 5cb29ee9d97e68a3784b19db52ae364c635f2e31
- Languages: Solidity(Smart contract), Typescript (Unit Testing)
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck, echinda

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level Reference

Every issue in this report were assigned a severity level from the following:

Admin/Owner Privileges can be misused either intentionally or unintentionally.

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	2	-	9
Closed	-	-	-

Contract Name: LiquidMarketVoting

High Severity Issues

1. **startNewVote does not reset all crucial voting state variables adequately. Breaks the contract's expected behavior**
   Line no: 53-63

   Explanation:
   The **startNewVote** function in the contract plays a significant role in starting a new voting series as it resets all the previous vote's data to either their default value or to new values.

```
53    function startNewVote(uint256 period, string memory votingText, uint quorumForPass) public onlyOwner {
54      require(_tokenAddress != address(0), "Token contract does not exist");
55      require(quorumForPass <= 100, 'Quorum for pass should be less than 100');
56
57      delete votersReceived;
58      _votesYes = 0;
59      _votingPeriod = period;
60      _votingText = votingText;
61      _quorumForPass = quorumForPass;
62      _votingStart = block.timestamp;
63    }
64
```

   However, as per the current design of the function, it was found that the function doesn't reset all the state variables adequately.

   The **startNewVote** function fails to reset the **_votesNo** state variable of the contract before starting a new voting series.

   This leads to an extremely undesirable situation where every new voting proposal will also contain the negative votes for the previous voting series. This will ultimately lead to a scenario where either most of the proposals or all proposals shall fail to pass due to the high value of negative votes and low value of positive votes.

   Recommendation:
   It is recommended to reset all imperative voting state variables in the contract adequately before starting out a new voting series.

2. **Inadequate function design might lead to costly transactions and hit Block Gas Limit**
   Line no: 79, 94

   Explanation:
   As per the current design of the **voteYes()** & **voteNo()** function, in order to check that no user votes for a single proposal twice, a for loop with an if statement is included.

```
79  function voteYes() public {
80    require(_votingStart > 0, 'Voting is not started yet');
81    require(_votingStart + _votingPeriod > block.timestamp, 'Voting is over');
82
83    for (uint i = 0; i < votersReceived.length; i++) {
84      require(votersReceived[i] != msg.sender, 'Voter already voted');
85    }
86
```

   However, it is clearly not taken into consideration that as the numbers of voters increase this for loop will eventually have to iterate over the entire list of voters to ensure this statement.

   This doesn't represent an adequate function design. While this is extremely costly, it could also hit a gas limit and cause the transaction to fail.

   Recommendation:
   The function could be redesigned to avoid the above-mentioned scenarios and enhance its performance.

Medium severity issues

No issues were found.

Low severity issues

1. **Loops are extremely costly**
   Line no - 69, 77

   Explanation:
   The **LiquidMarketVoting** contract has some **for loops** in the contract that include state variables like .length of a non-memory array, in the condition of the for loops.

   As a result, these state variables consume a lot more extra gas for every iteration of the for a loop.
   The following function includes such loops at the mentioned lines:
   - **addressCastVote at Line 66**

   Recommendation:
   It's quite effective to use a local variable instead of a state variable like .length in a loop. This will be a significant step in optimizing gas usage.
   For instance,

```
function addressCastVote(address _address) public view returns (bool) {
    uint256 local_variable = votersReceived.length;
   for (uint i = 0; i < local_variable; i++) {
     if (votersReceived[i] == _address) {
       return true;
     }
   }
   return false;
 }
```

2. **A constant declaration should be preferred**
   Line no- 13, 23, 25, 27

   Explanation:
   State variables that are not supposed to change throughout the contract should be declared as **constant**.

   Recommendation:
   The following state variables need to be declared as constant unless the current contract design is intended.

- **_votingPeriod**
- **_votingStart**
- **_votingText**
- **_quorumForPass**

3. **External visibility should be preferred**

   Explanation:
   Functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
   This will effectively result in Gas Optimization as well.

   Therefore, the following function must be marked as external within the contract:
   - setTokenAddress()
   - getTokenAddress()
   - getVotingPeriod()
   - getVotingStart()
   - getQuorumForPass()
   - startNewVote()
   - isVotingInProgress()
   - addressCastVote()
   - voteYes()
   - voteNo()
   - getVotesYes()
   - getVotingText()
   - getVotesNo()
   - getVoteResult()

   Recommendation:
   If the **PUBLIC** visibility of the above-mentioned functions is not intended, then the **EXTERNAL** Visibility keyword should be preferred.

4. **Absence of Event Emission after imperative State variable Update**
   Line no - 33, 53, 80, 95

   Explanation:
   Functions that update an imperative arithmetic state variable contract should emit an event after the update.

As per the current architecture of the contract, the following functions update crucial arithmetic state variables but don't emit any event on its modification:

- setTokenAddress()
- startNewVote()
- voteYes()
- voteNo()

The absence of event emission for important state variables update also makes it difficult to track them off-chain as well.

Recommendation:
As per the best practices in smart contract development, an event should be fired after changing crucial arithmetic state variables.

5. **Absence of Zero Address validation**
   Line no: 34

   Explanation:
   The setTokenAddress() function updates an imperative address in the contract. However, during the automated testing of the contact, it was found that no Zero Address Validation is implemented within the function.

   Recommendation:
   A require statement should be included in such functions to ensure no invalid addresses are passed in the arguments.

Recommendations/Informational

1. **Unlocked Pragma statements found in the contracts**
   Line no: 3

   Explanation:
   During the code review, it was found that the contracts included unlocked pragma solidity version statements.

   It's not considered a better practice in Smart contract development to do so as it might lead to accidental deployment to a version with unfixed bugs.

   Recommendation:
   It's always recommended to lock pragma statements to a specific version while writing contracts.

2. Coding Style Issues

It is recommended keeping lines under the PEP 8 recommendation to a maximum of 79 (or 99) characters helps readers easily parse the code.
More info here.

Contract Name: LiquidMarket

High Severity Issues

No issues were found.

Medium severity issues

No issues were found.

Low severity issues

1. **Redundant IF Statements found in Functions**
   Line no- 73,79, 102, 108

   Explanation:
   During the manual code review, it was found that the transfer and transferFrom functions in the contract include redundant if statements.

```
71          uint fee = 0;
72
73          if (_msgSender() != owner()) {
74              fee = getFee(amount);
75          }
76
77          _transfer(_msgSender(), recipient, amount - fee);
78
79          if (_msgSender() != owner()) {
80              _transfer(_msgSender(), owner(), fee);
81          }
82
```

   A single if statement could be used to ensure that if the caller is not the owner, the correct amount of the transaction fee is transferred to the owner. This will enhance the code readability and reduce space and gas consumption.

   Recommendation:
   Function design could be improved for the above-mentioned functions.

2. **Absence of Zero Address validation**
   Line no: 50

   Explanation:
   The setVotingAddress() function updates an imperative address in the contract. However, during the automated testing of the contact, it was found that no Zero Address Validation is implemented within the function.

   Recommendation:
   A require statement should be included in such functions to ensure no invalid addresses are passed in the arguments.

3. **Constant declaration should be preferred**
   Status: Not Considered

   Line no- 13, 23, 25, 27
   Explanation:
   State variables that are not supposed to change throughout the contract should be declared as constant.

   Recommendation:
   The following state variables need to be declared as constant unless the current contract design is intended.
   - **_name**
   - **_symbol**
   - **DECIMALS**
   - **TRANSACTION_FEE**

4. **External visibility should be preferred**

   Explanation:
   Functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
   This will effectively result in Gas Optimization as well.

   Therefore, the following function must be marked as external within the contract:
   - name()
   - symbol()

- decimals()
- setVotingAddress()
- totalSupply()
- balanceOf()
- transfer()
- allowance()
- approve()
- transferFrom()
- increaseAllowance()
- burn()
- decreaseAllowance()

Recommendation:
If the **PUBLIC** visibility of the above-mentioned functions is not intended, then the **EXTERNAL** Visibility keyword should be preferred.

Recommendations/Informational

1. **The owner's Withdrawal and Fund Split functions weren't found.**

Explanation:
As per the SPECIFICATION.md file, the contract was supposed to have a withdrawal function for the owner, which also split the withdrawable amount into 1.5% and 0.1% as well as transferred both portions into different accounts.

```
ERC20: development, documentation, unit tests and test of token
  Based on ERC20 standard
  Ownable
  Fixed supply
  No lending
  No interests
  1.6% transactions fee.
  When the owner call the withdrawal method the money will be split between accounts (split 1.5% and
  0.1%)
```

However, no such functions were found in the **LiquidMarket** contract during the audit.

Recommendation:
If the above-mentioned function is a part of the contract architecture, it should be included in it.

2. **Unlocked Pragma statements found in the contracts**
   Line no: 3

   Explanation:
   During the code review, it was found that the contracts included unlocked pragma solidity version statements.

   It's not considered a better practice in Smart contract development to do so as it might lead to accidental deployment to a version with unfixed bugs.

   Recommendation:
   It's always recommended to lock pragma statements to a specific version while writing contracts.

3. **Coding Style Issues**

   It is recommended keeping lines under the PEP 8 recommendation to a maximum of 79 (or 99) characters helps readers easily parse the code.
   More info here.

Automated Audit Result

1. LiquidMarketVoting.sol

```
Compiled with solc
Number of lines: 552 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 6 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 16
Number of informational issues: 17
Number of low issues: 11
Number of medium issues: 1
Number of high issues: 0

ERCs: ERC20

+--------------------+-------------+-------+--------------------+--------------+--------------------+
|        Name        | # functions | ERCS  |     ERC20 info     | Complex code |      Features      |
+--------------------+-------------+-------+--------------------+--------------+--------------------+
| LiquidMarketVoting |      22     |       |                    |      No      | Tokens interaction |
|   IERC20Burnable   |      7      | ERC20 |     No Minting     |      No      |                    |
|                    |             |       | Approve Race Cond. |              |                    |
|                    |             |       |                    |              |                    |
|      SafeMath      |      13     |       |                    |      No      |                    |
+--------------------+-------------+-------+--------------------+--------------+--------------------+
contracts/LiquidMarketVoting.sol analyzed (6 contracts)
```

2. LiquidMarket.sol

```
Compiled with solc
Number of lines: 744 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 8 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 29
Number of informational issues: 17
Number of low issues: 12
Number of medium issues: 1
Number of high issues: 0

ERCs: ERC20, ERC165

+--------------------+-------------+--------+--------------------+--------------+--------------------+
|        Name        | # functions |  ERCS  |     ERC20 info     | Complex code |      Features      |
+--------------------+-------------+--------+--------------------+--------------+--------------------+
|    LiquidMarket    |      34     | ERC20  |     No Minting     |      No      |                    |
|                    |             |        | Approve Race Cond. |              |                    |
|                    |             |        |                    |              |                    |
| LiquidMarketVoting |      22     |        |                    |      No      | Tokens interaction |
|      IERC165       |      1      | ERC165 |                    |      No      |                    |
|      SafeMath      |      13     |        |                    |      No      |                    |
+--------------------+-------------+--------+--------------------+--------------+--------------------+
contracts/LiquidMarket.sol analyzed (8 contracts)
```

Unit Test

```
LiquidMarket contract
  ✓ get name
  ✓ get symbol
  ✓ get decimals
  ✓ get total supply
  ✓ transfer with address(0) for voting (62ms)
  ✓ transfer (79ms)
  ✓ can not transfer if voted (101ms)
  ✓ transfer from (107ms)
  ✓ transfer from with address(0) for voting (104ms)
  ✓ burn
  ✓ can not transfer from if voted (58ms)
  ✓ allowance (53ms)

LiquidMarketVoting contract
  ✓ get token address
  ✓ get voting period
  ✓ get voting text
  ✓ get quorum for pass
  ✓ get voting start
  ✓ does not show the voting results before the vote is over (57ms)
  ✓ vote yes one time (46ms)
  ✓ vote no one time (42ms)
  ✓ vote twice (44ms)
  ✓ vote when voting finished
  ✓ vote when voting not started
  ✓ get voting result if passed (75ms)
  ✓ get voting result if failed (no one voted)
  ✓ get voting result if failed (75ms)
  ✓ get voting result if not passed (95ms)
```

Concluding Remarks

While conducting the audits of the LiquidMarketplace smart contracts, it was observed that the contracts contain High and Low severity issues.

Our auditors suggest that High and Low severity issues should be resolved by the developers. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the LiquidMarketplace platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes