EmiSwap Smart Contract Preliminary Audit Report

Project Synopsis

Project Name	**EmiSwap**
Platform	Ethereum, Solidity
Github Repo	https://github.com/EMISWAP-COM/emiswap/tree/master/contracts
Deployed Contract	Not Deployed
Total Duration	15 Days
Timeline of Audit	**17th May 2021 to 3rd June 2021**

Contract Details

Total Contract(s)	5
Name of Contract(s)	Crowdsale, EMIRouter, EMISwap, EMIVoting, EMIVesting
Language	Solidity
Commit Hash	`ab5221643be9d79b335573bc1a271fb4e43f6b60`

Contract Vulnerabilities Synopsis

Issues	Open Issues	Closed Issues
Critical Severity	3	0
Medium Severity	4	0
Low Severity	8	0
Information	1	0
Total Found	16	0

Detailed Results

The contract has gone through several stages of the audit procedure that includes structural analysis, automated testing, manual code review etc.

All the issues have been explained and discussed in detail below. Along with the explanation of the issue found during the audit, the recommended way to overcome the issue or improve the code quality has also been mentioned.

High Severity Issues

A.1 "fetchCoin" function does execute as expected for the First Coin Address Passed.

Contract - Crowdsale.sol

SWC Reference : SWC 123 – Requirement Violation

CWE Reference : CWE 345 - Insufficient Verification of Data Authenticity

Line no - 161-181

Explanation:

As per the current design of the fetchCoin function, it doesn't allow a particular coin address to be stored twice in the contract.

In order to ensure this, it includes a **require** statement where the ***coinIndex*** mapping is queried by passing in the address of the coin. It assumes that for an address, that has not yet been added to the ***coinIndex*** mapping will return 0 as it is the default value for uint16.

However, it doesn't take into consideration the fact that the very *first coin address passed in this function is being assigned a ZERO index as well.*

This leads to an unexpected scenario where the first coin address that was passed in this function can be passed once again as it passes the required statement at line 166. Thus,leading to a situation where the same coin address can be added more than once.

```
function fetchCoin(
    address coinAddress,
    uint32 rate,
    uint8 status
) public onlyAdmin {
    require(coinIndex[coinAddress] == 0, "Already loaded");
    string memory _name = IERC20Detailed(coinAddress).name();
    string memory _symbol = IERC20Detailed(coinAddress).symbol();
    uint8 _decimals = IERC20Detailed(coinAddress).decimals();
```

Recommendation:

The reason behind this scenario is the fact that the state variable ***coinCounter*** is being updated at the very end of the fetchCoin function.

Keeping in mind the desired functionality of the **fetchCoin** function and especially the **require** statement in this function, the ***coinCounter*** state variable should be incremented before initializing the struct.

Following this approach will assign an index value of ONE for the very first coin address that is passed to this function. Thus, eliminating any chances of storing the same coin address twice.

A.2 _freezeWithRollup functionalities of EMIVesting contracts can be called with ZERO token amount. Lack of Input Validation.

Contract - EMIVesting.sol

SWC Reference : SWC 123 – Requirement Violation

CWE Reference : CWE 345 - Insufficient Verification of Data Authenticity

Line no - 396

Explanation:

The _freezeWithRollup function does not include any input validations on the arguments passed to it.

```
396     function _freezeWithRollup(
397         address _beneficiary,
398         uint32 _freezetime,
399         uint256 _tokens,
400         uint32 category,
401         bool isVirtual,
402         bool updateCS
403     ) internal {
404         LockRecord[] storage lrec = _locksTable[_beneficiary];
405         bool recordFound = false;
406 
407         for (uint256 j = 0; j < lrec.length; j++) {
408             if (
409                 lrec[j].freezeTime == _freezetime &&
410                 (lrec[j].category & ~VIRTUAL_MASK) == category
411             ) {
```

This might lead to an unwanted scenario as it allows the caller of the function to freeze ZERO amount of tokens.

Recommendation:

The function must include input validations on imperative arguments. This ensures that no invalid arguments are being passed while calling the function.
For instance,
require(_tokens > 0, "Token amount must be greater than ZERO");

A.3 The *freeze* functionalities are inaccessible from outside the contract.

Contract - EMIVesting.sol
SWC Reference : SWC 100 – Function Visibility
CWE Reference : CWE 710 - Improper Adherence to Coding Standards

Line no 396-434

Explanation:
As per the current design of the contract, the ***_freeze function,*** that actually initializes the **LockRecord** struct with the imperative vesting details of a particular address, has been assigned an internal visibility as it is supposed to be called via ***_freezeWithRollup*** function.

However, during the manual code review of the EMIVesting contract, it was found that the ***_freezeWithRollup*** function has also been assigned an **internal** visibility.

```
396     function _freezeWithRollup(
397         address _beneficiary,
398         uint32 _freezetime,
399         uint256 _tokens,
400         uint32 category,
401         bool isVirtual,
402         bool updateCS
403     ) internal {
```

This makes it only accessible from within the contract despite the fact that this function(*_freezeWithRollup*) is not being called anywhere within the EMIVesting contract.
Is this INTENDED?

Recommendation:

If the above mentioned scenario is not intended, the visibility keywords should be reassigned to the functions accordingly to make the function accessible from outside the contract.

Medium Severity Issues

A.4 State Variables updated after External Calls . Violation of Check-Effects-Interaction Pattern

SWC Reference : SWC 107 – Reentrancy

CWE Reference : CWE 841 -Improper Enforcement of Behavioral Workflow

Contract - EMISwap.sol

Line no - 303-381

The **EMISwap** contract includes the **swap function** that updates some of the very imperative state variables of the contract after the external calls are being made.

An external call within a function technically shifts the control flow of the contract to another contract for a particular period of time. Therefore, as per the Solidity Guidelines, any modification of the state variables in the base contract must be performed before executing the external call.
Although the function has been assigned the ***nonReentrant*** modifier, the approach used, in this function, for making an external call violates the **Check Effects Interaction Pattern**.

```
331         src.uniTransferFromSenderToThis(amount);
332         uint256 confirmed = src.uniBalanceOf(address(this)).sub(balances.src);
333
334         uint256 resultVault;
335         (result, resultVault) = _getReturn(
336             src,
337             dst,
338             confirmed,
339             srcAdditionBalance,
340             dstRemovalBalance
341         );
342         require(
343             result > 0 && result >= minReturn,
344             "Emiswap: return is not enough"
345         );
346         dst.uniTransfer(payable(to), result);
347         if (resultVault > 0) {
348             dst.uniTransfer(payable(addressVault()), resultVault);
349         }
```

The following functions in the contract updates the state variables after making an external call at the lines mentioned below:

- ***swap() function** at Line **331, 346 and 348***

Recommendation:
Modification of any State Variables must be performed before making an external call. **Check Effects Interaction Pattern** must be followed while implementing external calls in a function.

A.5 Loops are extremely Costly

SWC Reference : SWC 128 – DoS With Block Gas Limit
CWE Reference : CWE 400 - Uncontrolled Resource Consumption

Contract - Crowdsale.sol, EMIRouter, EMIVesting, EMIVoting

Explanation:

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

```
function presaleBulkLoad(
    address[] memory beneficiaries↑,
    uint256[] memory tokens↑,
    uint32[] memory sinceDate↑
) public onlyAdmin {
    require(beneficiaries↑.length > 0, "Sale:Array empty");
    require(beneficiaries↑.length == sinceDate↑.length, "Sale:Arrays length");
    require(sinceDate↑.length == tokens↑.length, "Sale:Arrays length");
    require(now <= 1613340000, "Sale: presale is over"); // 15 feb 2021 00:00 GMT

    for (uint256 i = 0; i < beneficiaries↑.length; i++) {
        crowdSalePool = crowdSalePool.sub(tokens↑[i]);
        emit BuyPresale(beneficiaries↑[i], tokens↑[i], sinceDate↑[i]);
    }
}
```

As a result, these state variables consume a lot more extra gas for every iteration of the for loop.

The following function includes such loops at the mentioned lines of the contracts mentioned below:

a. **Crowdsale Contract**
   - *presaleBulkLoad at Line 335*

b. **EMIRouter Contract**
   - *getPoolDataList at Line 48*

c. **EMIVesting Contract**
    - *getNextUnlock() at Line 143*
    - *claim() at Line 273*
    - *mint() at Line 310*
    - *_freezeWithRollup at Line 407*
    - *_getBalance at Line 445*

d. **EMIVoting**
    - *queue() at Line 247*
    - *execute() at Line 283*
    - *cancel() at Line 309*

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,

```
local_variable = beneficiaries.length;
 for (uint256 i = 0; i < local_variable; i++) {
            crowdSalePool = crowdSalePool.sub(tokens[i]);
            emit BuyPresale(beneficiaries[i], tokens[i],
sinceDate[i]);
 }
```

A.6 Multiplication is being performed on the result of Division

Contract - Crowdsale.sol, EMIVesting
SWC Reference : SWC 101 – Integer Overflow and Underflow
CWE Reference : CWE 682 - Incorrect Calculation

Line no - 533-573

Explanation:
During the automated testing of the contracts, it was found that some of the functions in the contracts are performing multiplication on the result of a Division.
Integer Divisions in Solidity might truncate. Moreover, this performing division before multiplication might lead to loss of precision.

The following functions involve division before multiplication in the mentioned lines of the respective contracts:

a. **Crowdsale Contract**
    - *buyWithETHView* at Line 533-573

```
CrowdSale.buyWithETHView(uint256,bool) (flat/CrowdSale.Full.sol#1369-1428) performs a multiplication on the result of a division:
        -! isReverse && currentTokenAmount.mul(105).div(100) > crowdSalePool (flat/CrowdSale.Full.sol#1422)
        -currentTokenAmount = (coinAmount.mul(_ratePrecision).div(_coins[0].rate)) (flat/CrowdSale.Full.sol#1411-1415)
```

b. **EMIVesting Contract**
    - *_burnLock at Line 359-362*
    - *_getBalance at Line 450-455*
    - *_getLock at Line 487-492*

Recommendation:

Solidity doesn't encourage arithmetic operations that involve division before multiplication. Therefore the above-mentioned functions should be checked once and redesigned if they do not lead to expected results.

A.7 Violation of Check_Effects_Interaction Pattern found in the contract

SWC Reference : SWC 107 – Reentrancy

CWE Reference : CWE 841 -Improper Enforcement of Behavioral Workflow

Contract - EMIVesting.sol

Explanation:

As per the Check_Effects_Interaction Pattern in Solidity, external calls should be made at the very end of the function and event emission, as well as any state variable modification, must be done before the external call is made.

The following functions, however, violate the Check-Effects Interaction pattern:

- **_burnLock** at Line 363-365
- **mint()** at Line 317-320

```
316                 // mint tokens to vesting address
317                 IESW(_token).mintClaimed(address(this), amt);
318
319                 _statsTable[msg.sender][cat].tokensAvailableToMint -= amt;
320                 _statsTable[msg.sender][cat].tokensMinted += amt;
```

Recommendation:
[Check Effects Interaction Pattern] must be followed while implementing external calls in a function.

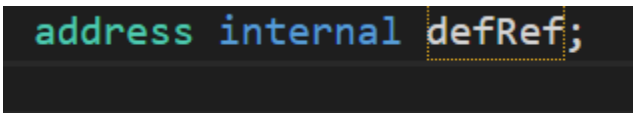
Low Severity Issues

A.8 State Variable initialized but never used in the contract.

SWC Reference : [SWC 131 – Presence of unused variables]
CWE Reference : [CWE 1164 - Irrelevant Code]

Contract - Crowdsale.sol
Line no - 56

Explanation:
The Crowdsale contract includes a state variable **defRef,** with an internal visibility, that is being initialized but never used throughout the gas.



Recommendation:
State variables should either be used effectively in the contract or removed to reduce gas usage.

A.9 presaleBulkLoad function includes a Hardcoded date

Contract - Crowdsale.sol
CWE Reference : [CWE 665 - Improper Initialization]

Line no - 333

Explanation:
Keeping in mind the immutable nature of smart contracts, it is not considered a better practise to hardcode any address or imperative uint in the contract before deployment.

```
333         require(now <= 1613340000, "Sale: presale is over"); // 15 feb 2021 00:00 G
```

Recommendation:

It is recommended to use a state variable for storing such an integer and initialize them in the constructor.

A.10 Boolean Constant is being inadequately used in the *buyWithETH* function

Contract - Crowdsale.sol

SWC Reference : SWC 135 – Code With No Effects

CWE Reference : CWE 1164 - Irrelevant Code

Line no - 596-599

Explanation:

During the automated testing of the crowdsale contract, it was found that the **buyWithEth** function includes a **require statement** that doesn't implement proper usage of boolean constant.

```
595
596     require(
597         msg.value > 0 && (!isReverse ? msg.value == amount : true),
598         "Sale:ETH needed"
599     );
```

Automated Test Result:

```
CrowdSale.buyWithETH(address,uint256,bool) (flat/CrowdSale.Full.sol#1436-1490) uses a Boolean constant improperly:
        -require(bool,string)(msg.value > 0 && (true),Sale:ETH needed) (flat/CrowdSale.Full.sol#1445-1448)
```

Recommendation:

It is recommended to modify the **require** statement and implement the boolean constant usage correctly.

A.11 Too many Digits used

Contract - EMIVesting.sol, EMIVoting.sol

SWC Reference : SWC 135 – Code With No Effects

CWE Reference : CWE 1164 - Irrelevant Code

Line no - 22

Description:
The above-mentioned lines have a large number of digits that makes it difficult to review and reduce the readability of the code.

The following State Variables/Functions of respective contracts mentioned below include large digits:

a. EMIVesting

- **CROWDSALE_LIMIT** at Line 22

```
uint256 constant CROWDSALE_LIMIT = 40000000e18; // tokens
```

b. EMIVoting

- *quorumVotes() 21-23*
- *proposalThreshold() 26-28*

Recommendation:
Ether Suffix couldshutt be used to symbolize the 10^18 zeros.

A.12 State Variable Never Used

SWC Reference : SWC 131 – Presence of unused variables
CWE Reference : CWE 1164 - Irrelevant Code

Contract - EMIVesting
Line no - 21

Description:
The EMIVesting contract includes a state variable, i.e., **WEEK.** However, the state variable is never used throughout the contract.

Recommendation:
State variables should either be used effectively or removed from the contract.

A.13 Comparison to boolean Constant

SWC Reference : SWC 135 – Code With No Effects
CWE Reference : CWE 1164 - Irrelevant Code

Contract - EMIVoting
Line no: 427-430

Description:

Boolean constants can directly be used in conditional statements or require statements.
Therefore, it's not considered a better practice to explicitly use **TRUE or FALSE** in the **require** statements.

```
require(
    receipt.hasVoted == false,
    "EmiVoting::_castVote: voter already voted"
);
```

Recommendation:
The equality to boolean constants must be removed from the above-mentioned line.

A.14 External Visibility should be preferred

SWC Reference : SWC 100 – Function Visibility
CWE Reference : CWE 710 - Improper Adherence to Coding Standards

Contract - Crowdsale.sol, EMIRouter, EMIVoting
Explanation:

Those 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 functions of respective contracts mentioned below should must be marked as **external** within the contract:

a. **Crowdsale Contract**
- *updateParams*
- *stopCrowdSale*
- *setPoolsize*
- *fetchCoin*
- *setStatusByID*
- *setRateByID*
- *coinCounter()*
- *coin()*
- *coinRate()*
- *coinData()*
- *presaleBulkLoad*

- *buy()*

b. EMIRouter

- *getPoolDataList()*
- *getReservesByPool()*
- *getReserves()*
- *getExpectedReturn()*
- *removeLiquidity()*
- *removeLiquidityETH()*

c. EMIVoting

- *propose*
- *queue()*
- *execute()*
- *cancel()*
- *getReceipt()*
- *castVote()*
- *castVoteBySig()*
- *__acceptAdmin()*
- *__abdicate()*
- *__queueSetTimelockPendingAdmin()*
- *__executeSetTimelockPendingAdmin()*

Recommendation:

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

A.15 Absence of Zero Address Validation

Contract - EMIRouter

SWC Reference : SWC 123 – Requirement Violation

CWE Reference : CWE 345 - Insufficient Verification of Data Authenticity

Line no- 27-30

Description:

The **EMIRouter** contract initializes some imperative state variables in the constructor.

However, during the automated testing of the contact it was found that no Zero Address Validation is implemented to ensure that no invalid argument is passed while initializing the state variables.

```
27 ▾    constructor(address _factory, address _wEth) public {
28          factory = _factory;
29          WETH = _wEth;
30      }
```

Recommendation:
A **require** statement should be included in such functions to ensure no zero address is passed in the arguments.

Informational

A.16 Coding Style Issues in the Contract

CWE Reference : CWE 710 - Improper Adherence to Coding Standards

Explanation:
Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future.
During the automated testing, it was found that the following contracts have quite a few code style issues:

a. Crowdsale

```
Variable CrowdSale._coins (flat/CrowdSale.Full.sol#887) is not in mixedCase
Variable CrowdSale._coinCounter (flat/CrowdSale.Full.sol#889) is not in mixedCase
Variable CrowdSale._ratePrecision (flat/CrowdSale.Full.sol#890) is not in mixedCase
Variable CrowdSale._token (flat/CrowdSale.Full.sol#899) is not in mixedCase
Variable CrowdSale._wethToken (flat/CrowdSale.Full.sol#900) is not in mixedCase
Variable CrowdSale._uniswapFactory (flat/CrowdSale.Full.sol#901) is not in mixedCase
```

b. EMIVoting

```
Function EmiVoting.__acceptAdmin() (flat/EmiVoting.full.sol#1034-1040) is not in mixedCase
Function EmiVoting.__abdicate() (flat/EmiVoting.full.sol#1042-1048) is not in mixedCase
Function EmiVoting.__queueSetTimelockPendingAdmin(address,uint256) (flat/EmiVoting.full.sol#1050-1
Function EmiVoting.__executeSetTimelockPendingAdmin(address,uint256) (flat/EmiVoting.full.sol#1067
Parameter EmiVoting.getVotingResult(uint256)._hash (flat/EmiVoting.full.sol#1093) is not in mixedCa
```

Recommendation:
Therefore, it is highly recommended to fix the issues like naming convention, indentation, and code layout issues in a smart contract.