Smart Contract Code Review

VERSION 1: *07/06/2021*

VERSION 2: *01/07/2021*

Audited by – AHR (OF-10) The_Swarm

Version: 2.0

THIS ASSESSMENT REPORT HAS BEEN TESTED, COMPLETED AND AUTHORISED ON 24/04/2019
APPROVED BY THE CEO | BLUESWARM LTD COMPANY NUMBER 11489054
BlueSwarm

GROUP

AiSecureMe BlueSwarm

Table of Content

Executive Summary

The purpose of this report

Smart Contract Code Review

This document aims to record the vulnerabilities found from a code review conducted by Blueswarm. The detected vulnerabilities are plotted against Best Practice Guidelines laid down by the community.

The Objective

Blueswarm to perform an industry best practice Vulnerability Assessment and Code Review and reports the findings from the following smart contracts only:

Crowdsale, EMIRouter, EMISwap, EMIVoting, EMIVesting,

Execution Strategy

Our execution strategy incorporates proven methodologies, extremely qualified personnel, and a highly responsive approach to managing deliverables and the utilization of proprietary software.

Methodology

The code audit was carried out using the specification of SWC (Smart Contract Weakness Classification) and CWE (Common Weakness Enumeration). The assessment was conducted using a combination of proprietary software and manual testing by highly skilled individuals.

Vulnerability Overview

Timeline and Audit Log

The Security Code Audit for the Smart Contract of EmiSwap project lasted 15 days from the 17th May 2021 to 3rd June 202. Where in total, 5 contracts : *Crowdsale, EMIRouter, EMISwap, EMIVoting, EMIVesting* have been analyzed

Vulnerabilities Detected

A total of 16 vulnerabilities were discovered and identified in the contracts.
The following and below mentioned charts show the respective severity classifications used, the breakdown and distribution of vulnerabilities.

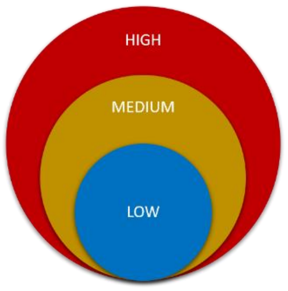


Fig 1. Vulnerability Classification

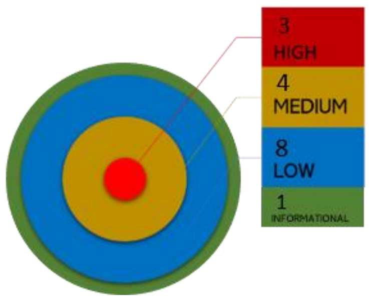


Fig 2. Vulnerability Breakdown in Numbers

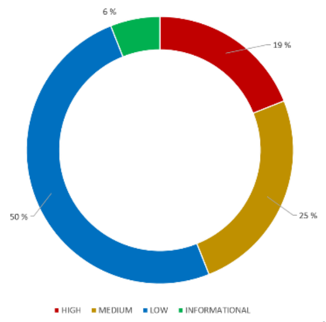


Fig 3. Vulnerability Distribution in %

HIGH RISK – There were a total of 2 High Severity Issues Found

Contract Files Affected - -

Crowdsale.sol - *EMIVesting.sol*

MEDIUM RISK - A total of 4 classified as medium risk vulnerabilities detected

Contract Files Affected – - *EMISwap.sol Crowdsale.sol, EMIRouter, EMIVesting, EMIVoting*

LOW RISK - A total of 4 classified as low risk vulnerabilities detected

Contract Files Affected: *Crowdsale.sol - EMIVesting.sol, EMIVoting.sol EMIRouter*,

Exploit Effort & Resource Classification

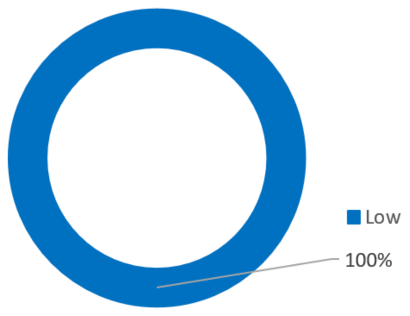
Rating	Definition of Risk Rating	Definition of Resource Requirement to Exploit	Definition of Effort to Exploit
HIGH	Deficiency creates a vulnerability that could result in loss of system control or override a desired function or give access to critical or sensitive information.	Recommendation either requires the purchase of hardware or, requires significant research and resources to exploit	To exploit the weakness requires a high level of expertise and advanced knowledge of smart contract design, and programming
MEDIUM	Deficiency creates an exposure to a larger, but limited loss of confidentiality or integrity, as the result of many user accounts being compromised, or restricted functions being accessed.	Recommendation may require the purchase of hardware or software and/or requires moderate, research and implementation activities to exploit	Requires medium level of effort. No tools are available but sample code or other similar exploits are known
LOW	Deficiency creates limited exposure to the compromise of user accounts or unauthorized access to data	Recommendation may require the purchase of minor hardware or software and/or requires minor research and implementation activities to exploit	Easy to exploit with known methods or tools with minimal modifications

Exploit Efforts & Resource Analysis

The following graphs below provide insight into the exploit efforts and resources needed in order to successfully complete or carry out exploitation mapped against the 16 vulnerabilities detected

Exploit Effort

Of the 16 Security issues currently identified, all vulnerabilities would require a low level of resources to exploit



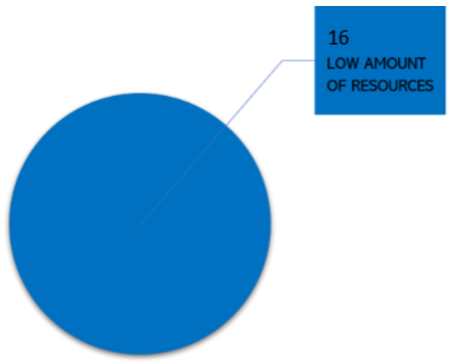
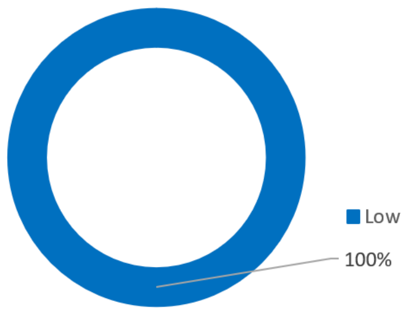


Fig 4. Exploit Effort Breakdown in %

Fig 5. Exploit Effort Breakdown in numbers

Exploit Resource Requirements

Of the 16 Security issues identified, all vulnerabilities that can be exploited require less resources to exploit.



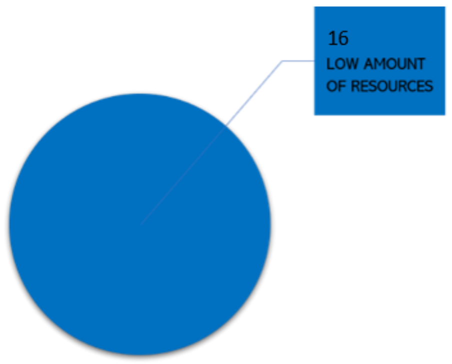


Fig 6. Resources need to exploit in %

Fig 7. Resources needed to exploit in numbers

Remediation Resource Requirements

Rating	Definition of Risk Rating	Definition of Resource Requirement to Remediate	Definition of Effort to Remediate
HIGH	Deficiency creates a vulnerability that could result in loss of system control or override a desired function or give access to critical or sensitive information.	Recommendation either requires the purchase of hardware or, requires significant changes to the code base or research and resources to remediate	To remediate the vulnerabilities requires a high level of expertise and advanced knowledge of smart contract design, and programming
MEDIUM	Deficiency creates an exposure to a larger, but limited loss of confidentiality or integrity, as the result of many user accounts being compromised, or restricted functions being accessed.	Recommendation may require the purchase of hardware or software and/or requires moderate changes to the codebase and/or research and implementation activities to remediate the vulnerability	Requires medium level of effort and changes to remediate.
LOW	Deficiency creates limited exposure to the compromise of user accounts or unauthorized access to data	Recommendation may require the purchase of minor hardware or software and/or requires minor changes in the codebase to remediate against the vulnerability	Easy to remediate with minimal modification or effort

Remediation Resource Requirements

Of the 16 Security issues identified, remediation efforts and resources required in all circumstances are considered Low. Therefore, minimal resources and programming efforts are required to implement satisfactory remediation.

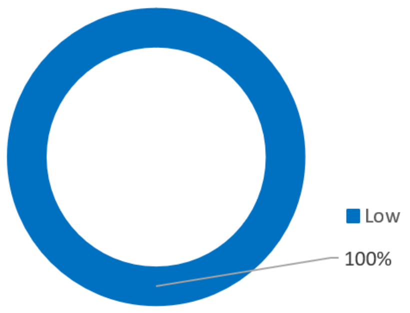


Fig 8. Resources need to remediate in %

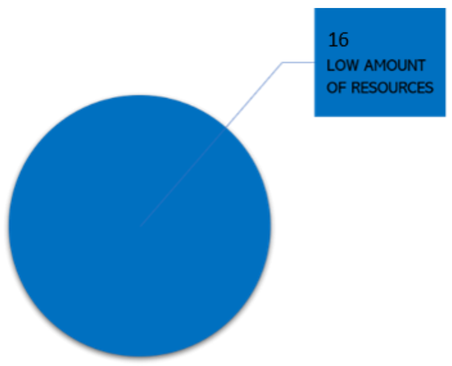


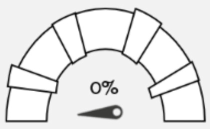
Fig 9. Resources needed to remediate in numbers

Finding Details

Severity

HIGH

Category: function does execute as expected for the First Coin Address Passed



False Positive Probability

True Positive Probability

List of Contracts Affected

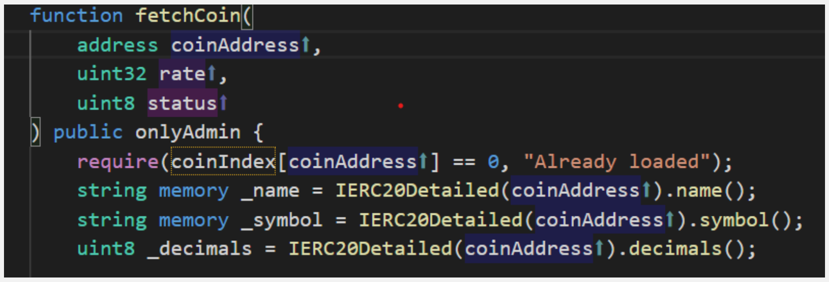
- *Crowdsale.sol*

STATUS
INTENTIONAL

Description

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.

Code Reference/s Lines: 161-181



```
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();
```

Remediation

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

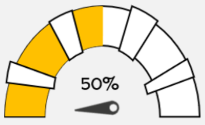
Severity

HIGH

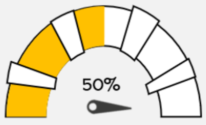
Category: The freeze functionalities are inaccessible from outside the contract

List of Contracts Affected

- *EmiVesting.sol*



False Positive Probability



True Positive Probability

STATUS CLOSED

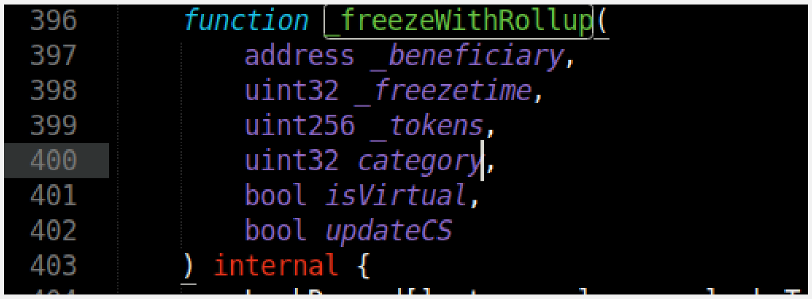
Description

The freeze functionalities are inaccessible from outside the contract
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.

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?

Code Reference/s Lines: 396-434



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

Remediation

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.

Severity

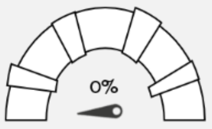
HIGH

Category: Lack of input validation

.

List of Contracts Affected

- *EmiVesting.sol*



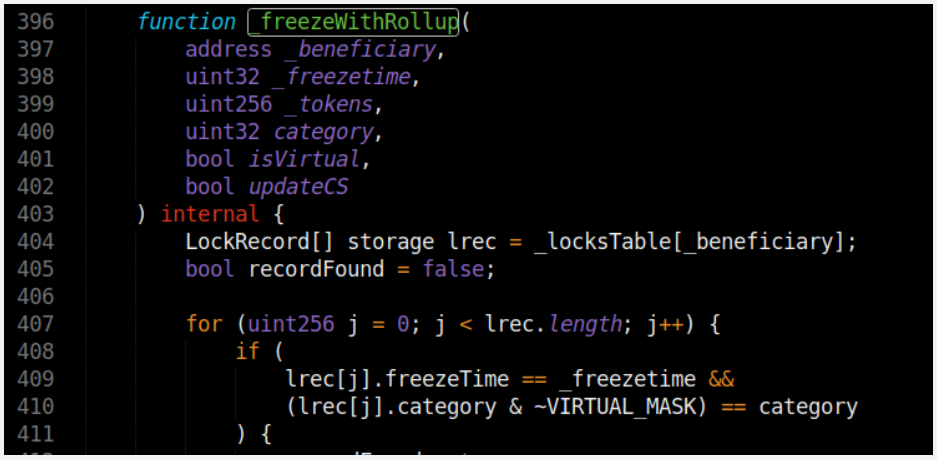
False Positive Probability

True Positive Probability

Description

The _freezeWithRollup function does not include any input validations on the arguments passed to it. This might lead to an unwanted scenario as it allows the caller of the function to freeze ZERO amount of tokens.

Code Reference/s Lines: 396



```
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              ) {
```

Remediation

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");

Severity

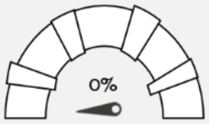
MEDIUM

Contract Name/s

Category: . Violation of Check-Effects

List of Contracts Affected

- *EmiSwap.sol*



False Positive Probability

True Positive Probability

STATUS
CLOSED

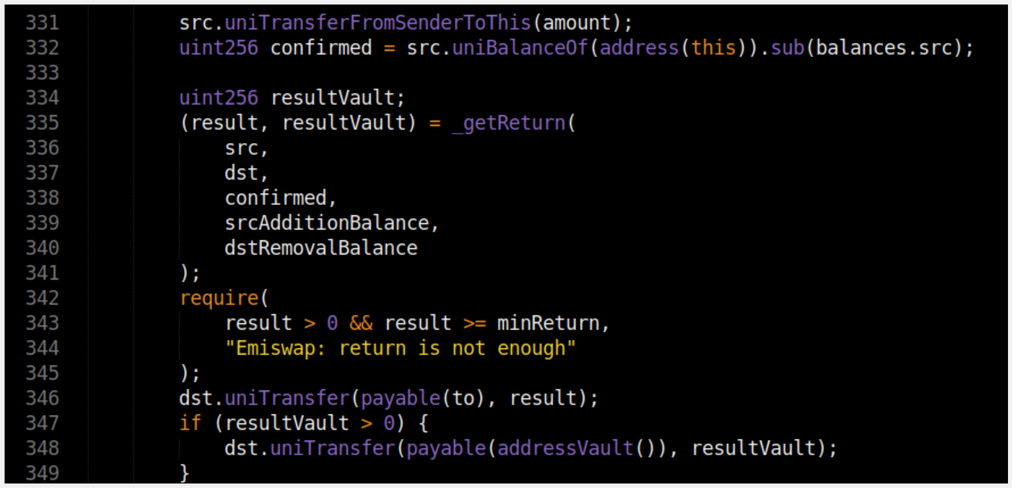
Description

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

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

Code Reference/s

Line 331, 346 and 348



```
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         }
```

Remediation

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

Severity

MEDIUM

Category: Loops are extremely Costly

STATUS CLOSED

Contract Name/s

List of Contracts Affected

- *Crowdsale.sol,, EMIRouter, EMIVesting, EMIVoting*

Description

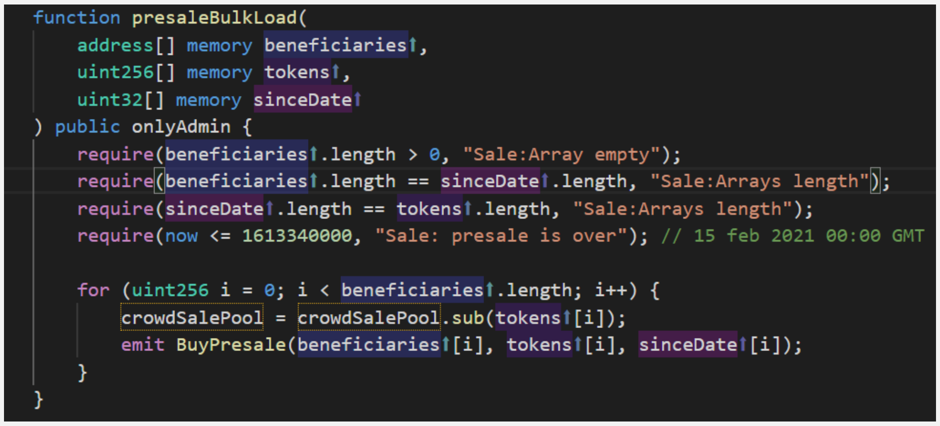
The Crowdsale contract has a for loops in the contract that include state variables like .length of a non-memory array, in the condition of the for loops.

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:

Code Reference/s

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

function revokeRole(bytes32 role, address account) public virtual {



```
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]);
    }
}
```

Remediation

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]);

}

Severity

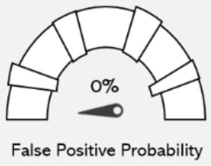
MEDIUM

Category: Multiplication is being performed on the result of Division

Contract Name/s

List of Contracts Affected

Crowdsale.sol, EMIVesting



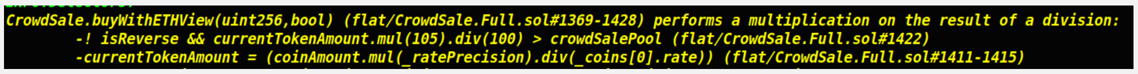
100%
True Positive Probability

Description

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:

Code Reference/s

a. Crowdsale Contract ● buyWithETHView at Line 533-573 b. EMIVesting Contract ● _burnLock at Line 359-362 ● _getBalance at Line 450-455● _getLock at Line 487-492



```
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)
```

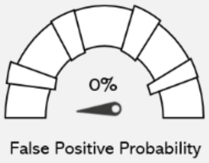
Remediation

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.

Severity

MEDIUM

Category: Violation of Check_Effects Interaction Pattern found in the contract



Contract Name/s

List of Contracts Affected

- *EmiVesting.sol*

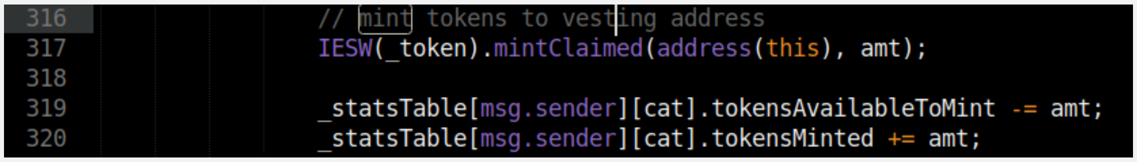
STATUS CLOSED BlueSwarm

Description

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:

Code Reference/s

_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;
```

Remediation

[Check Effects Interaction Pattern](#) must be followed while implementing external calls in a function.

Severity

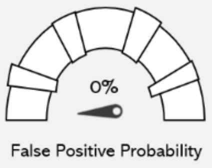
LOW

Category: State Variable initialized but. never used in the contract

Contract Name/s

List of Contracts Affected

- *CrowdSale.sol*



BlueSwarm STATUS INTENTIONAL

Description

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.

Code Reference/s

Line no - 56

adress internal defRef

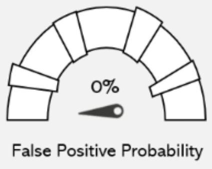
Remediation

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

Severity

LOW

Category: presaleBulkLoad function includes a Hardcoded date



100%
True Positive Probability

Contract Name/s

List of Contracts Affected

- *CrowdSale.sol*

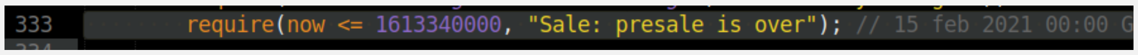
STATUS
INTENTIONAL

Description

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.

Code Reference/s

Line: 333



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

Remediation

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

Severity

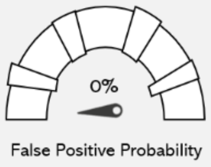
LOW

Category: Boolean Constant is being inadequately used in the buyWithETH function

Contract Name/s

List of Contracts Affected

CrowdSale.sol



100%
True Positive Probability

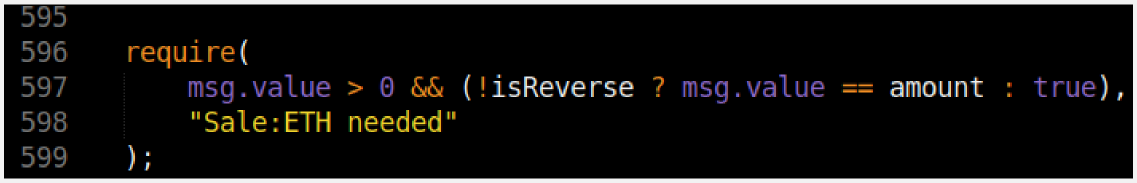
STATUS INTENTIONAL

Description

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.

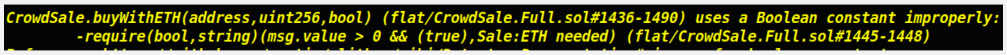
Code Reference/s

Line: - 596-599



```
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)
```

Remediation

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

Severity

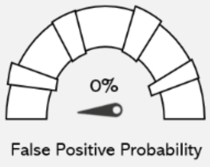
LOW

Category: too many digits used

Contract Name/s

List of Contracts Affected

EmiVesting.sol
EmiVoting.Sol



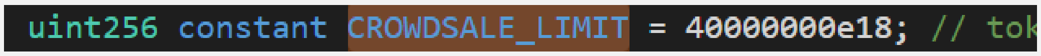
BlueSwarm STATUS CLOSED

Description

The 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:

Code Reference/s

a. EMIVesting ● CROWDSALE_LIMIT at Line 22



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

b. EMIVoting

quorumVotes() 21-23

proposalThreshold() 26-28

Remediation

Ether Suffix could be used to symbolize the 10^18 zeros

Severity

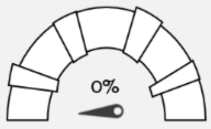
LOW

Category: State Variable never used

Contract Name/s

List of Contracts Affected

EmiVesting.sol



False Positive Probability

True Positive Probability

STATUS CLOSED

Description

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

Code Reference/s

Line: - 21

Remediation

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

Severity

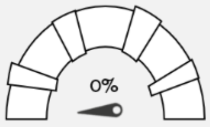
LOW

Category: Comparison to boolean Constant

Contract Name/s

List of Contracts Affected

EmiVoting.sol



False Positive Probability

True Positive Probability

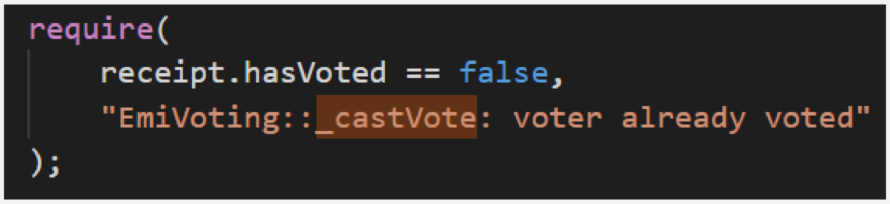
STATUS CLOSED

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.

Code Reference/s

Line: 427-430



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

Remediation

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

Severity

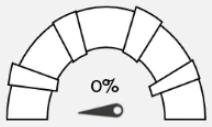
LOW

Category: External Visibility should be preferred

Contract Name/s

List of Contracts Affected

CrowdSale.sol, EmiRouter.sol EmiVoting.Sol



False Positive Probability

True Positive Probability

STATUS CONSIDERED

Description

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:

Code Reference/s

Crowdsale ● updateParams ● stopCrowdSale ● setPoolsize ● fetchCoin ● setStatusByID ● setRateByID ● coinCounter() ● coin() coinRate() ● coinData() ● presaleBulkLoad ● buy()

EMIRouter ● getPoolDataList() ● getReservesByPool() ● getReserves() ● getExpectedReturn() ● removeLiquidity() ● removeLiquidityETH()

EMIVoting ● propose ● queue() ● execute() ● cancel() ● getReceipt() ● castVote() ● castVoteBySig() ● __acceptAdmin() ● __abdicate() ● __queueSetTimelockPendingAdmin() ● __executeSetTimelockPendingAdmin()

Remediation

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

Severity

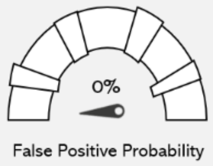
LOW

Category: Absence of zero address validation

Contract Name/s

List of Contracts Affected

EmiRouter.sol



100%
True Positive Probability

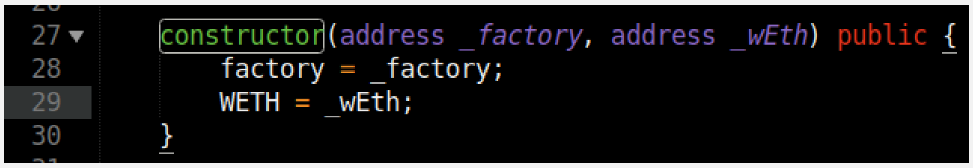
STATUS CONSIDERED

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.

Code Reference/s

Line: 27-30



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

Remediation

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

Severity

INFORMATIONAL

Contract Name/s

Category: Coding Style Issues in the contract

List of Contracts Affected

CrowdSale.sol, EmiVoting.sol

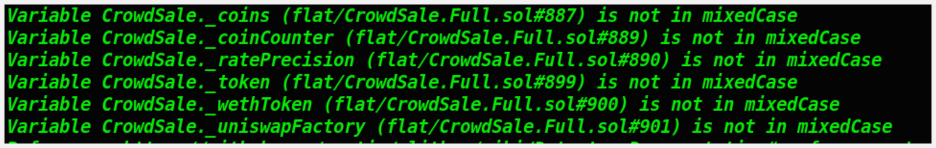
STATUS
NOT CONSIDERED

BlueSwarm

Description

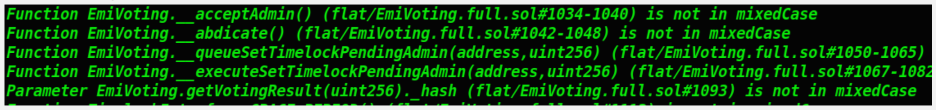
Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future. During the automated testing, it was found that the following contracts have quite a few code style issues:

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#1050-1065)
Function EmiVoting.__executeSetTimelockPendingAdmin(address,uint256) (flat/EmiVoting.full.sol#1067-1082
Parameter EmiVoting.getVotingResult(uint256)._hash (flat/EmiVoting.full.sol#1093) is not in mixedCase
```

Remediation

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

Open Cases

Issues	Open Issues	Closed Issues
Critical Severity		3
Medium Severity		4
Low Severity	2	6
Information	1	
Total Found	3	13

Conclusion

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 above. 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. Out of the vulnerabilities found, all vulnerabilities and issues identified were either corrected or had been adequately addressed through other controls.
either the contract in question have been deprecated or future low effect fixes will be modified in future upgrades.

DISCLAIMER [CLIENT: EMISWAP

V1: Original Report without remediation [ORIGINALTESTDATE]: 07/06/2021

V2: Remediation Report [REMEDIATIONTESTDATE]: 01/07/2021

BlueSwarm

BlueSwarm

This review is marked as V.2, which was conducted by Blueswarm's certified security engineers. We identified several security vulnerabilities and provided remediation advice to EmiSwap

After being notified by [CLIENT] that all vulnerabilities have been corrected, Blueswarm have performed a remediation test (V.2) on [REMEDIATIONTESTDATE] to confirm that all vulnerabilities and issues identified were either corrected or had been adequately addressed through other controls. While no application or system can be 100% secure, all of our security findings were corrected or addressed and it is our opinion that the contracts tested are reasonably well written from a security perspective and the applications and supporting systems are deployed, configured and implemented in a secure manner. IF NOT FULLY CORRECTED The review was conducted by Blueswarms's certified security engineers. We identified several security vulnerabilities and provided remediation advice to [CLIENT]. After being notified by [CLIENT] that these selected vulnerabilities had been corrected, Blueswarm performed a remediation test on [REMEDIATIONTESTDATE] and confirmed that these selected vulnerabilities were either corrected or had been adequately addressed through other controls. There were findings identified by Blueswarm that were not validated as corrected. Please contact [CLIENT] for further information regarding these findings and their resolution status. DISCLAIMER: Blueswarm conducted this testing on the smart contracts that existed as of [ORIGINALTESTDATE]. Information security threats are continually changing, with new vulnerabilities discovered on a daily basis, and no application can ever be 100% secure no matter how much security testing is conducted. This report is intended only to provide documentation that [CLIENT] has corrected all findings noted by Blueswarm as of [REMEDIATIONTESTDATE]. This report cannot and does not protect against personal or business loss as the result of use of the applications or systems described. Blueswarm offers no warranties, representations or legal certifications concerning the applications, code or systems it tests. All software includes defects: nothing in this document is intended to represent or warrant that security testing was complete and without error, nor does this document represent or warrant that the application tested is suitable to task, free of other defects than reported, fully DISCLAIMER - Compliant with any industry standards, or fully compatible with any operating system, hardware, or other application. By using this information you agree that Blueswarm shall be held harmless in any event