Missile Farm
Smart Contract Audit Report

May 12, 2021

Introduction

1. About Missile Farm

MissileFarm is a yield farming aggregator running on Binance Smart Chain.

Missile Farm Vaults is a yield optimizer platform focused on providing DeFi users with auto-compounded yields at empirical optimal intervals.

Known as dynamic harvesting, Missile Farm compounds staked assets in our vaults automatically by using algorithmically profitable strategies at optimal compounding intervals. This custom optimizing function was built in-house to enable only the best yields for Missile Farm users.

Visit https://www.missile.farm/ to know more.

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 15+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up 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 Missile Farm team has provided documentation for the purpose of conducting the audit.
The documents are:

1. https://docs.missile.farm/

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: Missile Farm
- Contract Audited: FarmChef.sol & MSIToken1.sol
- Languages: Solidity(Smart contract), Javascript(Unit Testing)
- Github commit hash for audit: c29c216daeec42081187a5e998c50545bdac971a
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

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 References

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

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	3	5	3
Closed	-	-	-

Contract Name: FarmChef

High Severity Issues

1. **Imperative Contract functions are accessible to everyone instead of Only OWNER.**
   Line no - 1426 to 1438, 1441 to 1447

   Description:
   The Masterchef contract includes a few functions that include crucial functionality, like adding a new **_lpToken** in the Pool, which should be accessible to only the owner of the contract.

   However, no such **require statements** or **modifiers** were found in those functions.
   The following functions should be accessible to only the owner but are marked **PUBLIC** instead:

   - **add function** at Line 1426
   - **set function** at Line 1441

```
1425      // Add a new lp to the pool. Can only be called by the owner.
1426      function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public {
1427          if (_withUpdate) {
1428              massUpdatePools();
1429          }
1430          uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1431          totalAllocPoint = totalAllocPoint.add(_allocPoint);
1432          poolInfo.push(PoolInfo({
1433              lpToken: _lpToken,
1434              allocPoint: _allocPoint,
1435              lastRewardBlock: lastRewardBlock,
1436              accSushiPerShare: 0
1437          }));
1438      }
```

   Recommendation:
   The **onlyOwner** modifier should be attached to the above-mentioned functions.

2. **safeSushiTransfer function does not execute adequately if Sushi Token Balance in the contract is less than the amount of tokens to be transferred**
   Line no - 1557

   Description:
   The **safeSushiTransfer** is designed in a way that it first checks whether or not the MasterChef contract has more reward token balance than the amount of tokens to be transferred to the user(Line 1558).

If the MasterChef contract has less reward token balance than the amount to be transferred, the user gets only the remaining reward tokens in the contract and not the actual amount that was supposed to be transferred(Line 325).

However, the major issue in this function is that if the above-mentioned condition is met and the user only gets the remaining sushi tokens in MasterChef contract instead of the actual sushi tokens that should be transferred, then the remaining amount of tokens that the user didn't receive yet is never stored throughout the function.

```
1556    // Safe sushi transfer function, just in case if rounding error cau
1557    function safeSushiTransfer(address _to, uint256 _amount) internal {
1558        uint256 sushiBal = sushi.balanceOf(address(this));
1559        if (_amount > sushiBal) {
1560            sushi.transfer(_to, sushiBal);
1561        } else {
1562            sushi.transfer(_to, _amount);
1563        }
1564    }
```

For instance, if the user is supposed to receive **1000** pending tokens while calling the **withdraw function**.
The withdraw function will call the **safeSushiTransfer** function(Line 1528) and passes the user's address and the pending token amount of 1000 tokens.

```
1527            if(pending > 0) {
1528                safeSushiTransfer(msg.sender, pending);
1529            }
```

However, if the MasterChef contract has only 800 reward tokens then the **if condition** at Line 1559 will be executed and the user will only receive **800 reward tokens** instead of **1000 reward tokens**.
Now, because of the fact that the **safeSushiTransfer** function doesn't store this information that the user still owes 200 reward tokens will lead to an unexpected scenario where the users receive less reward token than expected.

Was this scenario Intended or Considered during the development of this function?

Recommendation:
If the above-mentioned scenario was not considered, then the function must be updated in such a way that the user gets the actual amount of reward tokens whenever the safeSushiTransfer function is called.

3. **add function allows adding similar LP Token Address more than once.**
   Line no - 1426

   Description:
   As per the current architecture of the MasterChef contract, the **LP Tokens** added in the pool of this contract should not be repeated as the address of an **LP Token** plays an imperative role in the calculation of rewards the presence of a similar LP Token address more than once will break some of the core functionalities of the contract.

   However, the **add()** function at Line 1426 allows storing a similar LP Token Address more than once. This will lead to an unexpected scenario where different pools will have a similar LP token address.

```
1425    // Add a new lp to the pool. Can only be called by the owner.
1426    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public {
1427        if (_withUpdate) {
1428            massUpdatePools();
1429        }
1430        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1431        totalAllocPoint = totalAllocPoint.add(_allocPoint);
1432        poolInfo.push(PoolInfo({
1433            lpToken: _lpToken,
1434            allocPoint: _allocPoint,
1435            lastRewardBlock: lastRewardBlock,
1436            accSushiPerShare: 0
1437        }));
1438    }
```

   Is this SCENARIO INTENDED?

   Recommendation:
   If the above mentioned scenario is not intended, the argument **_lpToken** (LP token address) passed in the **add()** function must be checked at the very beginning of the function with a **require statement** to ensure no other **LP Token** with a similar address have been passed before.

Medium Severity Issues

1. **Multiplication is being performed on the result of Division**
   Line no - 1471-1472

   Description:
   During the automated testing of the MasterChef contract, it was found that some of the functions in the contract 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:

   - **pendingSushi**

```
1463    uint256 sushiReward = multiplier.mul(sushiPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1464    accSushiPerShare = accSushiPerShare.add(sushiReward.mul(1e12).div(lpSupply));
1465
```

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

2. **Contract State Variables are being updated after External Calls. Violation of Check_Effects_Interaction Pattern**
   Line no - 1548 to 1555, 1513 to 1516

   Description:
   The MasterChef contract includes quite a few functions that update 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 any state variables in the base contract must be performed before executing the external call.
   Updating state variables after an external call might lead to a potential re-entrancy scenario.

   The following functions in the contract updates the state variables after making an external call, at the line numbers specified below, and violates the Check-Effects Interaction Pattern:

   - **deposit() function** at Line **1513**

- **emergencyWithdraw** at Line **1548**

```
1547    // Withdraw without caring about rewards. EMERGENCY ONLY.
1548    function emergencyWithdraw(uint256 _pid) public {
1549        PoolInfo storage pool = poolInfo[_pid];
1550        UserInfo storage user = userInfo[_pid][msg.sender];
1551        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1552        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1553        user.amount = 0;
1554        user.rewardDebt = 0;
1555    }
```

Recommendation:
Modification of any State Variables must be performed before making an external call.

3. **updatePool and massUpdatePools functions have been assigned a Public visibility**
   Line no - 1478, 1486

   Description:
   The **updatePool** and **massUpdatePools** functions include significant functionalities as they deal with updating the reward variables of a given pool.
   These functions are called within the contract by some crucial functions like **add(), deposit, withdraw** etc.
   However, instead of an **internal visibility**, these functions have been assigned a public visibility.

```
1477    // Update reward variables for all pools. Be care
1478    function massUpdatePools() public {
1479        uint256 length = poolInfo.length;
1480        for (uint256 pid = 0; pid < length; ++pid) {
1481            updatePool(pid);
1482        }
1483    }
```

   Since **public** visibility will make the **updatePool** & **massUpdatePools** function accessible to everyone, it would have been a more effective and secure approach to mark these functions as **internal**.

   Recommendation:
   If both of these functions are only to be called from within the contract, their visibility specifier should be changed from **PUBLIC** to **INTERNAL**.

Low Severity Issues

1. **External Visibility should be preferred**

   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 function must be marked as **external** within the contract:
   - **mint**
   - **add**
   - **set**
   - **deposit**
   - **withdraw**
   - **emergencyWithdraw**
   - **dev**

   Recommendation:
   If the PUBLIC visibility of these functions is not intended, the visibility keyword must be modified to EXTERNAL.

2. **Absence of Zero Address Validation in dev function**
   Line no- 1568 to 1572

   Description:
   The **dev** function in the contract updates the devAddress state variable which is a crucial variable in the contract.
   However, during the automated testing of the contact it was found that no Zero Address Validation is implemented for the **_devaddr** argument passed to this function.

```
1568    function dev(address _devaddr) public {
1569        require(msg.sender == devaddr, "dev: wut?");
1570        devaddr = _devaddr;
1571    }
```

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

Recommendations

1. **Coding Style Issues in the Contract**

   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 MasterChef contract had quite a few code style issues.

```
Parameter MasterChef.add(uint256,IERC20,bool)._allocPoint (Contracts/FarmChef.sol#1419) is not in mixedCase
Parameter MasterChef.add(uint256,IERC20,bool)._lpToken (Contracts/FarmChef.sol#1419) is not in mixedCase
Parameter MasterChef.add(uint256,IERC20,bool)._withUpdate (Contracts/FarmChef.sol#1419) is not in mixedCase
Parameter MasterChef.set(uint256,uint256,bool)._pid (Contracts/FarmChef.sol#1434) is not in mixedCase
Parameter MasterChef.set(uint256,uint256,bool)._allocPoint (Contracts/FarmChef.sol#1434) is not in mixedCase
Parameter MasterChef.set(uint256,uint256,bool)._withUpdate (Contracts/FarmChef.sol#1434) is not in mixedCase
Parameter MasterChef.getMultiplier(uint256,uint256)._from (Contracts/FarmChef.sol#1443) is not in mixedCase
Parameter MasterChef.getMultiplier(uint256,uint256)._to (Contracts/FarmChef.sol#1443) is not in mixedCase
Parameter MasterChef.pendingSushi(uint256,address)._pid (Contracts/FarmChef.sol#1456) is not in mixedCase
Parameter MasterChef.pendingSushi(uint256,address)._user (Contracts/FarmChef.sol#1456) is not in mixedCase
```

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

2. **NatSpec Annotations must be included**

   Description:
   The smart contracts do not include the NatSpec annotations adequately.

   Recommendation:
   Cover by NatSpec all Contract methods.

Automated Test Results

```
MasterChef (FarmChef.sol#1357-1573) contract sets array length with a user-controlled value:
        - poolInfo.push(PoolInfo(_lpToken,_allocPoint,lastRewardBlock,0)) (FarmChef.sol#1434-1439)
```

```
BaseFallback.constructor(address,bytes)._logic (FarmChef.sol#921) lacks a zero-check on :
                - (success) = _logic.delegatecall(_data) (FarmChef.sol#925)
FarmChef.upgradeToAndCall(address,bytes).newImplementation (FarmChef.sol#1076) lacks a zero-check on :
                - (success) = newImplementation.delegatecall(data) (FarmChef.sol#1078)
MasterChef.constructor(SushiToken,address,uint256,uint256,uint256)._devaddr (FarmChef.sol#1411) lacks a zero-check on :
                - devaddr = _devaddr (FarmChef.sol#1417)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
```

```
mint(address,uint256) should be declared external:
        - SushiToken.mint(address,uint256) (FarmChef.sol#1119-1122)
add(uint256,IERC20,bool) should be declared external:
        - MasterChef.add(uint256,IERC20,bool) (FarmChef.sol#1428-1440)
set(uint256,uint256,bool) should be declared external:
        - MasterChef.set(uint256,uint256,bool) (FarmChef.sol#1443-1449)
deposit(uint256,uint256) should be declared external:
        - MasterChef.deposit(uint256,uint256) (FarmChef.sol#1504-1520)
withdraw(uint256,uint256) should be declared external:
        - MasterChef.withdraw(uint256,uint256) (FarmChef.sol#1523-1546)
emergencyWithdraw(uint256) should be declared external:
        - MasterChef.emergencyWithdraw(uint256) (FarmChef.sol#1549-1556)
dev(address) should be declared external:
        - MasterChef.dev(address) (FarmChef.sol#1569-1572)
```

```
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#7) is too complex
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#30) is too complex
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#96) is too complex
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#172) is too complex
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#330) is too complex
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#631) is too complex
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#669) is too complex
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#1113) is too complex
Pragma version>=0.6.0<0.9.0 (FarmChef.sol#1355) is too complex
solc-0.7.6 is not recommended for deployment
```

Contract Name: MSI Token1

Medium Severity Issues

1. **Require Statement could be preferred instead of IF Statement**
   Line no - 928 to 946

   Description:
   The **_moveDelegates** function ensures that that the function is executed only if:
   - The **srcRep** address and the **dstRep** address should be different, and
   - The amount passed should be greater than **Zero**.

   According to the current function design, if any of these conditions is not fulfilled, the function will not be executed.

```
928     function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal
929         if (srcRep != dstRep && amount > 0) {
930             if (srcRep != address(0)) {
931                 // decrease old representative
932                 uint32 srcRepNum = numCheckpoints[srcRep];
933                 uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1
934                 uint256 srcRepNew = srcRepOld.sub(amount);
935                 _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
936             }
937
```

   Therefore, keeping in mind that the function strictly checks the arguments passed and stops the function execution if the above-mentioned conditions are not met, it would be more effective to use a **require statement** instead of **IF-Else Statement**.
   It's not only considered a better practice to use **require statements** for input validation but it also improves the code readability and helps in gas optimization.

   Recommendation:
   It is recommended to validate the user inputs using **require statements** for the above-mentioned function.

2. **Strict Equality is being used in the IF statement**
   Line no: 958

   Description:
   During the automated testing of the contract, it was found that the **_writeCheckpoint** function includes a STRICT EQUALITY check in the if statement.

```
948     function _writeCheckpoint(
949         address delegatee,
950         uint32 nCheckpoints,
951         uint256 oldVotes,
952         uint256 newVotes
953     )
954         internal
955     {
956         uint32 blockNumber = safe32(block.number, "FarmToken::_writeCheckpoint: block number exceed
957
958         if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber)
959             checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
```

Is this Intended?

Recommendation:
It is not considered a better practice in Solidity to implement a Strict Equality check in the **If** or **require** statements.
If the above-mentioned logic is not intended, then the **if statement** should be modified.

Low Severity Issues

1. **User ETHER Units instead of Large Digits**
   Line no: 739, 746

   Description:
   During the automated testing of the contracts it was found that the **MSI Token** contract includes large digits in the contract.

```
742     function mintOnceForFarm(address _address) external onlyOwner {
743         require(!doOnce, "only can mint once");
744         require(_address != address(0), "!_address");
745         doOnce = true;
746         _mint(_address, 1600000 * 1e18);
747     }
```

   The globally available Ether Units can be used instead of multiplying amounts to 1e18 while minting a particular token amount. This will enhance the readability of the contract code.

   Solidity provides some globally available units like **ether** which symbolizes **10^18**.
   For instance,
   "_mint(msg.sender, 400000 * 1e18);" can simply be written as
   "_mint(msg.sender, 400000 ether);"

Recommendations

1. **NatSpec Annotations must be included**

   Description:
   Smart Contract does not include the NatSpec Annotations adequately.

   Recommendations:
   Cover by NatSpec all Contract methods.

Automated Test Results

```
MsiToken._writeCheckpoint(address,uint32,uint256,uint256) (MSIToken1.sol#948-966) uses a dangerous strict equality:
        - nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber (MSIToken1.sol#958)
```

```
transfer(address,uint256) should be declared external:
        - ERC20.transfer(address,uint256) (MSIToken1.sol#536-539)
allowance(address,address) should be declared external:
        - ERC20.allowance(address,address) (MSIToken1.sol#544-546)
approve(address,uint256) should be declared external:
        - ERC20.approve(address,uint256) (MSIToken1.sol#555-558)
transferFrom(address,address,uint256) should be declared external:
        - ERC20.transferFrom(address,address,uint256) (MSIToken1.sol#573-577)
increaseAllowance(address,uint256) should be declared external:
        - ERC20.increaseAllowance(address,uint256) (MSIToken1.sol#591-594)
decreaseAllowance(address,uint256) should be declared external:
        - ERC20.decreaseAllowance(address,uint256) (MSIToken1.sol#610-613)
initialize() should be declared external:
        - MsiToken.initialize() (MSIToken1.sol#734-740)
```

```
MsiToken.initialize() (MSIToken1.sol#734-740) uses literals with too many digits:
        - _mint(msg.sender,400000 * 1e18) (MSIToken1.sol#739)
MsiToken.mintOnceForFarm(address) (MSIToken1.sol#742-747) uses literals with too many digits:
        - _mint(_address,1600000 * 1e18) (MSIToken1.sol#746)
```

Concluding Remarks

While conducting the audits of Missile Farm smart contracts - FarmChef.sol & MSIToken1.sol, it was observed that the contracts contain High, Medium, and Low severity issues, along with several areas of recommendations.

Our auditors suggest that High, Medium, Low severity issues should be resolved by the Missile Farm developers. Resolving the areas of recommendations are up to the team's discretion. 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 Missile Farm platform or its product nor this audit is an 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 Pvt Ltd.