4RX

MasterChef.sol

Smart Contract Audit
Final Report

June 1, 2021

Introduction

1. About 4RX

The purpose of the 4RX project is to build one token that reflects a wide range of crypto assets with a high potential growing curve.

4RX is the very first base price index utility token, used as an indicator that indicates the market volatility in real-time with 15-20% of total all tokens market cap valued as $360B as of April 1st 2021.

4RX indices 65 crypto assets, mostly defi projects with high potential curve for the next coming years. This enables the 4RX community to passively stake a long list of coins with one token and save a tremendous volume of money on trading and gas fees.

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 4RX team has provided documentation for the purpose of conducting the audit. The documents are:

1. 4RX Auditing.docx

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: 4RX
- Contract Name(s): *MasterChef*
- Languages: Solidity(Smart contract)
- Github commit hash for audit: **f7d395e86028056ba5e88ee50ddbd933a1a0779d**
- Github commit hash for final audit: **57193fa3acb212cdb827e24513687deabe2bf7af**
- GitHub Link: https://github.com/FourRX/4rx/blob/master/contracts/Farm/MasterChef.sol
- 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	-	-	-
Closed	-	4	5

Medium Severity Issues

1. **safeFRXTransfer function does not execute adequately if FRX Token Balance in the contract is less than the amount of tokens to be transferred**
   Line no - 295 to 302

   Description:
   The **safeFRXTransfer** is designed in a way that it first checks whether or not the MasterChef contract has more FRX token balance than the amount of tokens to be transferred to the user(Line 296).

   If the MasterChef contract has less reward token balance than the amount to be transferred, the user gets only the remaining FRX tokens in the contract and not the actual amount that was supposed to be transferred(Line 298).

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

```
294     // Safe AUTO transfer function, just in case if rounding error cau
295     function safeFRXTransfer(address _to, uint256 _FRXAmt) internal {
296         uint256 FRXBal = IERC20(FRX).balanceOf(address(this));
297         if (_FRXAmt > FRXBal) {
298             IERC20(FRX).transfer(_to, FRXBal);
299         } else {
300             IERC20(FRX).transfer(_to, _FRXAmt);
301         }
302     }
```

   For instance, if the user is supposed to receive **1000** FRX tokens while calling the **withdraw or deposit function**.
   The deposit function will call the **safeFRXTransfer** function(Line 207) and pass the user's address and the pending token amount of 1000 tokens.

```
205
206             if (pending > 0) {
207                 safeFRXTransfer(msg.sender, pending);
208             }
209         }
```

However, if the MasterChef contract has only 800 reward tokens then the **if condition** at Line 297 will be executed and the user will only receive **800 FRX** tokens instead of **1000 FRX** tokens.
Now, because of the fact that the **safeFRXTransfer** function doesn't store this information, the user still owes 200 FRX 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 safeFRXTransfer function is called. Otherwise, an adequate amount of FRX tokens should always be maintained in the contract.

Acknowledged(May 28th 2021): 4RX team has acknowledged the issue. This is an intentional functionality by the dev team.

2. **Multiplication is being performed on the result of Division**
Line no - 137, 180,182, 192

Description:
During the automated testing of the MasterChef.sol 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:

- **pendingFRX** at 137
- **updatePool** at 180-182
- **syncUser** at 192

```
180         uint256 FRXReward = multiplier.mul(FRXPerBlock).
181         mul(pool.allocPoint).div(totalAllocPoint);
182
183         FRXToken(FRX).mint(owner(), FRXReward.mul(ownerFRXReward).div(10000));
```

Automated Test Results on Updated Contract:

```
FrxFarm.pendingFRX(uint256,address) (FixedFlat_Masterchef.sol#1377-1388) performs a multiplication on the result of a division:
        -FRXReward = multiplier.mul(FRXPerBlock).mul(pool.allocPoint).div(totalAllocPoint) (FixedFlat_Masterchef.sol#1384)
        -accFRXPerShare = accFRXPerShare.add(FRXReward.mul(1e8).div(sharesTotal)) (FixedFlat_Masterchef.sol#1385)
FrxFarm.updatePool(uint256) (FixedFlat_Masterchef.sol#1413-1434) performs a multiplication on the result of a division:
        -FRXReward = multiplier.mul(FRXPerBlock).mul(pool.allocPoint).div(totalAllocPoint) (FixedFlat_Masterchef.sol#1427)
        -pool.accFRXPerShare = pool.accFRXPerShare.add(FRXReward.mul(1e8).div(sharesTotal)) (FixedFlat_Masterchef.sol#1429)
FrxFarm.updatePool(uint256) (FixedFlat_Masterchef.sol#1413-1434) performs a multiplication on the result of a division:
        -FRXReward = multiplier.mul(FRXPerBlock).mul(pool.allocPoint).div(totalAllocPoint) (FixedFlat_Masterchef.sol#1427)
        -FRXToken(FRX).mint(owner(),FRXReward.mul(ownerFRXReward).div(10000)) (FixedFlat_Masterchef.sol#1432)
FrxFarm.syncUser(address,uint256) (FixedFlat_Masterchef.sol#1436-1441) performs a multiplication on the result of a division:
        -user.shares = user.shares.add(pool.distributionDebt.sub(user.distributionDebt).mul(user.shares.div(1e8))) (FixedFlat_M
```

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.

Amended (June 1st 2021): Issue was fixed by 4RX team and is no longer present in the code.

3. **Contract State Variables are being updated after External Calls. Violation of Check_Effects_Interaction Pattern**
   Line no - 90-101, 198-223, 288-291, 228-268, 182-186

   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.

   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 violates the Check-Effects-Interaction Pattern.

   The following functions in the contract updates the state variables after making an external call at the line numbers specified below:
   - **add() function** at Line 90
   - **deposit() function** at Line 198,211, etc
   - **emergencyWithdraw** at Line 288-291
   - **updatePool** at Line 182-183
   - **withdraw**

```
287
288         pool.want.safeTransfer(address(msg.sender), amount);
289         emit EmergencyWithdraw(msg.sender, _pid, amount);
290         user.shares = 0;
291         user.rewardDebt = 0;
```

Recommendation:
Modification of any State Variables must be performed before making an external call. Check-Effects Interaction Pattern should be followed.

Amended (May 28th 2021): Issue was fixed by 4RX team and is no longer present in the code.

4. **updatePool and massUpdatePools functions have been assigned a Public visibility**
Line no - 166, 158

Description:
The **updatePool** and **massUpdatePools** functions include imperative functionalities as they deal with updating the reward blocks 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.

```
157     // Update reward variables for all pools. Be careful of gas spending!
158     function massUpdatePools() public {
159         uint256 length = poolInfo.length;
160         for (uint256 pid = 0; pid < length; ++pid) {
161             updatePool(pid);
162         }
163     }
```

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**.

Acknowledged(May 28th 2021): 4RX team has acknowledged the issue. This is an intentional functionality by the dev team.

Low Severity Issues

1. **getMultiplier function could be marked as INTERNAL**
   Line no - 122

   Description:
   Functions that are only supposed to be called from within the contract should be marked as **Internal**.
   In the Masterchef contract, the **getMultiplier** has been assigned a Public visibility while it is being called within other functions of the contract. Moreover, it accepts **block.number** as arguments.

```
121     // Return reward multiplier over the given _from to _to block.
122     function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
123         if (IERC20(FRX).totalSupply() >= FRXMaxSupply) {
124             return 0;
125         }
126         return _to.sub(_from);
127     }
```

   Recommendation:
   If the **PUBLIC** visibility of the getMultiplier function is not intended, the function should be marked as **INTERNAL**.

   Amended (May 28th 2021): Issue was fixed by 4RX team and is no longer present in the code.

2. **Return Value of an External Call is never used Effectively**
   Line no - 286, 298, 300

   Description:
   The external calls made in the above-mentioned lines return a boolean as well as **uint256** values.
   These boolean return values can be used in the function as a check to ensure that the further execution of the function is only allowed if the external is successfully made.
   However, the MasterChef contract does not use these return values effectively in some instances of the contract.

```
295     function safeFRXTransfer(address _to, uint256 _FRXAmt) internal {
296         uint256 FRXBal = IERC20(FRX).balanceOf(address(this));
297         if (_FRXAmt > FRXBal) {
298             IERC20(FRX).transfer(_to, FRXBal);
299         } else {
300             IERC20(FRX).transfer(_to, _FRXAmt);
301         }
302     }
```

Recommendation:
Effective use of all the return values from external calls must be ensured within the contract.

Acknowledged(May 28th 2021): 4RX team has acknowledged the issue. This is an intentional functionality by the dev team.

3. **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:

- **add**
- **set**
- **withdrawAll**
- **emergencyWithdraw**
- **inCaseTokensGetStuck**

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

Acknowledged(May 28th 2021): 4RX team has acknowledged the issue. This is an intentional functionality by the dev team.

4. **Constant declaration should be preferred**
   Line no- 53 to 63

Description:
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.

- **FRXMaxSupply**
- **FRXPerBlock**
- **distributionBP**
- **ownerFRXReward**
- **startBlock**

Amended (May 28th 2021): Issue was fixed by 4RX team and is no longer present in the code.

5. **Too many Digits used**
Line no - 57

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

```
56      // Frx total supply: 200 mil = 200000000e18
57      uint256 public FRXMaxSupply = 200000000e18;
58      // Frxs per block: (1e18 - owner 10%)
```

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

Note:
So initially the contract had this:
uint256 public FRXMaxSupply = 200000000e18;
Which could have been written as 200000000 ether.
Therefore, this issue showed up in the automated testing.

Now they have updated this line to this
uint256 public constant FRXMaxSupply = 200000000e8;

So the Too Many Digits issue doesn't hold any significance now

Informational

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 FrxFarm contract had quite a few code style issues.

```
Parameter FrxFarm.pendingFRX(uint256,address)._pid (Flat_MasterChef.sol#1403) is not in mixedCase
Parameter FrxFarm.pendingFRX(uint256,address)._user (Flat_MasterChef.sol#1403) is not in mixedCase
Parameter FrxFarm.stakedWantTokens(uint256,address)._pid (Flat_MasterChef.sol#1417) is not in mixedCase
Parameter FrxFarm.stakedWantTokens(uint256,address)._user (Flat_MasterChef.sol#1417) is not in mixedCase
Parameter FrxFarm.updatePool(uint256)._pid (Flat_MasterChef.sol#1439) is not in mixedCase
Parameter FrxFarm.syncUser(address,uint256)._user (Flat_MasterChef.sol#1462) is not in mixedCase
Parameter FrxFarm.syncUser(address,uint256)._pid (Flat_MasterChef.sol#1462) is not in mixedCase
Parameter FrxFarm.deposit(uint256,uint256)._pid (Flat_MasterChef.sol#1470) is not in mixedCase
Parameter FrxFarm.deposit(uint256,uint256)._wantAmt (Flat_MasterChef.sol#1470) is not in mixedCase
Parameter FrxFarm.withdraw(uint256,uint256)._pid (Flat_MasterChef.sol#1500) is not in mixedCase
Parameter FrxFarm.withdraw(uint256,uint256)._wantAmt (Flat_MasterChef.sol#1500) is not in mixedCase
Parameter FrxFarm.withdrawAll(uint256)._pid (Flat_MasterChef.sol#1545) is not in mixedCase
Parameter FrxFarm.emergencyWithdraw(uint256)._pid (Flat_MasterChef.sol#1550) 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.

Masterchef.sol

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	ERC20 Token Standards	Pass
		Compiler Version Security	Pass
		Visibility Specifiers	Pass
		Gas Consumption	Pass
		SafeMath Features	Pass
		Fallback Usage	Pass
		tx.origin Usage	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		Overriding Variables	Pass
2	Function Call Audit	Authorization of Function Call	Pass
		Low-level Function (call/delegate call) Security	Pass
		Returned Value Security	Pass
		selfdestruct Function Security	Pass
3	Business Security	Access Control of Owner	Pass
		Business Logics	Pass
		Business Implementations	Pass
4	Integer Overflow/underflow	-	Pass
5	Reentrancy	-	Pass

This 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.

6	Exceptional Reachable state	-	Pass
7	Transaction-Ordering Dependence	-	Pass
8	Block Properties Dependence	-	Pass
9	Pseudo-random Number Generator (PRNG)	-	Pass
10	DoS (Denial of Service)	-	Pass
11	Token Vesting Implementation	-	N/A
12	Fake Deposit	-	Pass
13	Event security	-	Pass

Business Logic:

No.	Categories	Subitems	Results
1	**Token Supply** **Initial - 100,000** **Max Supply - 200M**	-	Pass
2	**Token Distribution**	-	Pass
3	**Reward Distribution**	-	Pass
4	**Fees**	-	Pass
5	**Pairs**	-	Pass
7	**Rug Pull Testing**	-	Pass
8	**Security Holes**	-	Pass
9	**Pairs**	FRX – Stand alone with no pair as Staking FRX-4RX, FRX-BNB , FRX-CAKE, FRX-BUSD, FRX-GRT, FRX-1INCH, FRX-BTCB, FRX-ONT, FRX-XVS	Pass

Automated Test Results

```
FrxFarm.pendingFRX(uint256,address) (Flat_MasterChef.sol#1403-1414) performs a multiplication on the result of a division:
        -FRXReward = multiplier.mul(FRXPerBlock).mul(pool.allocPoint).div(totalAllocPoint) (Flat_MasterChef.sol#1410)
        -accFRXPerShare = accFRXPerShare.add(FRXReward.mul(1e12).div(sharesTotal)) (Flat_MasterChef.sol#1411)
FrxFarm.updatePool(uint256) (Flat_MasterChef.sol#1439-1460) performs a multiplication on the result of a division:
        -FRXReward = multiplier.mul(FRXPerBlock).mul(pool.allocPoint).div(totalAllocPoint) (Flat_MasterChef.sol#1453)
        -FRXToken(FRX).mint(owner(),FRXReward.mul(ownerFRXReward).div(10000)) (Flat_MasterChef.sol#1455)
FrxFarm.updatePool(uint256) (Flat_MasterChef.sol#1439-1460) performs a multiplication on the result of a division:
        -FRXReward = multiplier.mul(FRXPerBlock).mul(pool.allocPoint).div(totalAllocPoint) (Flat_MasterChef.sol#1453)
        -pool.accFRXPerShare = pool.accFRXPerShare.add(FRXReward.mul(1e12).div(sharesTotal)) (Flat_MasterChef.sol#1458)
FrxFarm.syncUser(address,uint256) (Flat_MasterChef.sol#1462-1467) performs a multiplication on the result of a division:
```

```
Reentrancy in FrxFarm.add(uint256,IERC20,bool,address) (Flat_MasterChef.sol#1356-1377):
        External calls:
        - massUpdatePools() (Flat_MasterChef.sol#1363)
                - FRXToken(FRX).mint(owner(),FRXReward.mul(ownerFRXReward).div(10000)) (Flat_MasterChef.sol#1455)
                - FRXToken(FRX).mint(address(this),FRXReward) (Flat_MasterChef.sol#1456)
        State variables written after the call(s):
        - poolInfo.push(PoolInfo(_want,_allocPoint,lastRewardBlock,0,0,_strat)) (Flat_MasterChef.sol#1367-1376)
        - totalAllocPoint = totalAllocPoint.add(_allocPoint) (Flat_MasterChef.sol#1366)
Reentrancy in FrxFarm.deposit(uint256,uint256) (Flat_MasterChef.sol#1470-1497):
        External calls:
        - updatePool(_pid) (Flat_MasterChef.sol#1471)
                - FRXToken(FRX).mint(owner(),FRXReward.mul(ownerFRXReward).div(10000)) (Flat_MasterChef.sol#1455)
                - FRXToken(FRX).mint(address(this),FRXReward) (Flat_MasterChef.sol#1456)
        - safeFRXTransfer(msg.sender,pending) (Flat_MasterChef.sol#1480)
                - IERC20(FRX).transfer(_to,FRXBal) (Flat_MasterChef.sol#1571)
                - IERC20(FRX).transfer(_to,_FRXAmt) (Flat_MasterChef.sol#1573)
        - pool.want.safeTransferFrom(address(msg.sender),address(this),_wantAmt) (Flat_MasterChef.sol#1484)
        State variables written after the call(s):
```

```
Reentrancy in FrxFarm.withdraw(uint256,uint256) (Flat_MasterChef.sol#1500-1543):
        External calls:
        - updatePool(_pid) (Flat_MasterChef.sol#1501)
                - FRXToken(FRX).mint(owner(),FRXReward.mul(ownerFRXReward).div(10000)) (Flat_MasterChef.sol#1455)
                - FRXToken(FRX).mint(address(this),FRXReward) (Flat_MasterChef.sol#1456)
        - safeFRXTransfer(msg.sender,pending) (Flat_MasterChef.sol#1516)
                - IERC20(FRX).transfer(_to,FRXBal) (Flat_MasterChef.sol#1571)
                - IERC20(FRX).transfer(_to,_FRXAmt) (Flat_MasterChef.sol#1573)
        - sharesRemoved = IStrategy(poolInfo[_pid].strat).withdraw(msg.sender,_wantAmt) (Flat_MasterChef.sol#1526)
```

```
Parameter FrxFarm.add(uint256,IERC20,bool,address)._want (Flat_MasterChef.sol#1358) is not in mixedCase
Parameter FrxFarm.add(uint256,IERC20,bool,address)._withUpdate (Flat_MasterChef.sol#1359) is not in mixedCase
Parameter FrxFarm.add(uint256,IERC20,bool,address)._strat (Flat_MasterChef.sol#1360) is not in mixedCase
Parameter FrxFarm.set(uint256,uint256,bool)._pid (Flat_MasterChef.sol#1381) is not in mixedCase
Parameter FrxFarm.set(uint256,uint256,bool)._allocPoint (Flat_MasterChef.sol#1382) is not in mixedCase
Parameter FrxFarm.set(uint256,uint256,bool)._withUpdate (Flat_MasterChef.sol#1383) is not in mixedCase
Parameter FrxFarm.getMultiplier(uint256,uint256)._from (Flat_MasterChef.sol#1395) is not in mixedCase
Parameter FrxFarm.getMultiplier(uint256,uint256)._to (Flat_MasterChef.sol#1395) is not in mixedCase
Parameter FrxFarm.pendingFRX(uint256,address)._pid (Flat_MasterChef.sol#1403) is not in mixedCase
Parameter FrxFarm.pendingFRX(uint256,address)._user (Flat_MasterChef.sol#1403) is not in mixedCase
Parameter FrxFarm.stakedWantTokens(uint256,address)._pid (Flat_MasterChef.sol#1417) is not in mixedCase
Parameter FrxFarm.stakedWantTokens(uint256,address)._user (Flat_MasterChef.sol#1417) is not in mixedCase
```

Concluding Remarks

While conducting the audits of 4RX smart contract - MasterChef.sol, it was observed that the contracts contain Medium, and Low severity issues, along with a few areas of recommendations.

Our auditors suggest that Medium and Low severity issues should be resolved by the 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 4RX platform or its product neither 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 Pvt Ltd.