FourRX Finance Smart Contract Preliminary Audit Report

Project Synopsis

Project Name	**FOUR RX**
Platform	Ethereum, Solidity
Github Repo	https://github.com/FourRX/4rx/blob/master/contracts/FourRXFinance.sol
Deployed Contract	Not Deployed
Total Duration	3 Days
Timeline of Audit	**5th April 2021 to 8th April 2021**

Contract Details

Total Contract(s)	9
Name of Contract(s)	FourRXFinance,RewardsAndPenalties, Insurance, Pools, ReferralPool, SponsorPool, PercentageCalculator,InterestCalculator,SharedVariables
Language	Solidity
Commit Hash	**f7d395e86028056ba5e88ee50ddbd933a1a0779d**

Contract Vulnerabilities Synopsis

Issues	Open Issues	Closed Issues
Critical Severity	1	0
Medium Severity	4	0
Low Severity	6	0
Informational	3	0
Total Found	14	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.

A. Contract Name: FourRX Finance

Medium Severity Issues

A.1 State Variables Updated After External Call. Violation of Check_Effects_Interaction Pattern

Explanation:

As per the Check_Effects_Interaction Pattern in Solidity, external calls should be made at the very end of the function.

Event emission as well as any state variable modification must be done before the external call is made.

However, during the automated testing, it was found that some of the functions in the FourRXFinance contract violate this ***Check-Effects-Interaction*** pattern as they update state variables after making the external call.

Such functions are mentioned below along with the specific line number of the external calls:

- ***deposit at Line 102***
- ***withdraw at Line 185***
- ***withdrawDevFee at Line 246***

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.2 Multiplication is being performed on the result of Division

Line no - 214-219, 248-256

***Explanation*:**

During the automated testing of the BirdFarm.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:

- ***pendingRewardToken*** *at 214-219*

- ***updatePool*** *at 248-256*

```
214             uint256 rewardTokenReward =
215                 multiplier.mul(rewardTokenPerBlock).mul(pool.allocPoint).div(
216                     totalAllocPoint
217                 );
218             accRewardTokenPerShare = accRewardTokenPerShare.add(
219                 rewardTokenReward.mul(1e12).div(lpSupply)
220             );
```

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.

Low Severity Issues

A.3 Return Value of an External Call is never used Effectively

Line no -185, 215, 246

Explanation:
The external calls made in the above-mentioned lines do return a boolean value that indicates whether or not the external call made was successful.
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 FourRXFinance contract never uses these return values throughout the contract.

```
214
215         fourRXToken.transfer(user.wallet, availableAmount);
216
```

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

A.4 updateDevAddress function does not include Zero Address Validation

Line no: 250-253

Explanation:

The **updateDevAddress** function initializes one of the most imperative state variables, i.e., ***devAddress*** in the FourRXFinance contract.

```
250    function updateDevAddress(address newDevAddress) external {
251        require(msg.sender == devAddress);
252        devAddress = newDevAddress;
253    }
```

However, during the automated testing of the contract, it was found that the function doesn't implement any Zero Address Validation Check to ensure that no zero address is passed while calling this function.

Recommendation:

Since the **updateDevAddress** initializes a crucial address in the contract, it is quite important to implement zero address checks and ensure that only valid addresses are updated while calling this function.

A.5 Absence of Error messages in Require Statements

Description:

The **require** statements in the FourRXFinance contract do not include any error message.

While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

Recommendation:

Error Messages must be included in every **require** statement in the contract.

Informational

A.6 NatSpec Annotations must be included

Description:

The smart contracts do not include the NatSpec annotations adequately.

Recommendation:

Cover by NatSpec all Contract methods.

B. Contract Names: Pool, ReferralPool, SponsorPool, PercentageCalculator, InterestCalculator

High Severity Issues

B.1 calcPercentage allows passing ZERO as its second argument

Contract Name - PercentageCalculator

Line no - 13

Description:

The ***calcPercentage*** function includes a require statement at Line 13 that allows the **basisPoints** arguments to be **equal to ZERO**.

```
12      function _calcPercentage(uint amount, uint basisPoints) internal pure returns (uint) {
13          require(basisPoints >= 0);
14          return amount.mul(basisPoints).div(PERCENT_MULTIPLIER);
15      }
```

Since ***basisPoints*** is an imperative argument and is being used in the calculation of percentage value in multiple instances in the contract, this **require** statement can lead to unwanted scenarios.

During the automated testing of the contract, a similar warning was found as well:

```
PercentageCalculator._calcPercentage(uint256,uint256) (Pool_FLAT.sol#653-656) contains a tautology or contradiction:
        - require(bool)(basisPoints >= 0) (Pool_FLAT.sol#654)
```

Recommendation:

If the above-mentioned function design is not intended, **require statement** of the **calcPercentage** should be updated as follows:

require(basisPoints > 0, “Basis Point cannot be ZERO”)

Medium Severity Issues

B.2 Function design of *_addSponsorPoolRecord* does not match to *_addRefPoolRecord*.

Contract: ReferralPool, SponsorPool

Line no - 11-17, 9-11

The function *_addRefPoolRecord*, in the ReferralPool contract, includes a **IF_ELSE** mechanism to check whether or not the provided information is already present in the list(using SortedLinkedList.isInList function). It only uses the **addNode** function if

the information is not present already. Otherwise, it uses the **updateNode** function to store the data.

```
11      function _addRefPoolRecord(address user, uint amount, uint8 stakeId) public {
12          if (!SortedLinkedList.isInList(refPoolUsers, user, stakeId)) {
13              SortedLinkedList.addNode(refPoolUsers, user, amount, stakeId);
14          } else {
15              SortedLinkedList.updateNode(refPoolUsers, user, amount, stakeId);
16          }
17      }
```

However, no such IF_ELSE mechanism was found in the _addSponsorPoolRecord function in the ***SponsorPool*** contract despite the fact that both the functions perform almost similar tasks of adding records to their respective pools.

```
 9      function _addSponsorPoolRecord(address user, uint amount, uint8 stakeId) interna
10          SortedLinkedList.addNode(sponsorPoolUsers, user, amount, stakeId);
11      }
```

IS THIS INTENDED?

Recommendation:

If the above mentioned scenario is not intended, then the _addSponsorPoolRecord function in SponsorPool contract should also include IF_ELSE mechanism and use updateNode function to store already available information.

Low Severity Issues

B.3 Absence of Error messages in Require Statements

Line no: 383

Contract Name: *InterestCalculator*

Description:

The **require** statement in the InterestCalculator contract does not include any error message.

```
382      function _getInterestTillDays(uint _day) internal pure
383          require(_day <= MAX_DAYS);
384
385          return _initCumulativeInterestForDays()[_day];
386      }
```

While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

Recommendation:

Error Messages must be included in every **require** statement in the contract.

Informational

B.4 Coding Style Issues in the Contract

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 ReferralPool contract had quite a few code style issues.

```
Function ReferralPool._addRefPoolRecord(address,uint256,uint8) (Pool_FLAT.sol#881-887) is not in mixedCase
Function ReferralPool._cleanRefPoolUsers() (Pool_FLAT.sol#889-892) 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.

B.5 NatSpec Annotations must be included

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

Recommendation:
Cover by NatSpec all Contract methods.

C. Contract Name - RewardsAndPenalties, SharedVariables

Medium Severity:

C.1 Multiplication is being performed on the result of Division

***Explanation*:**
During the automated testing of the RewardsAndPenalties.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:

- ***_calcContractBonus*** *at 52-53*
- ***_calcHoldRewards*** *at 63-64*

```
50
51      function _calcContractBonus(Stake memory stake) internal view returns (uint) {
52          uint contractBonusPercent = fourRXToken.balanceOf(address(this)).
53          div(10**fourRXTokenDecimals).mul(CONTRACT_BONUS_PER_UNIT_BP).div(CONTRACT_BONUS_UNIT);
54
```

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.

Low Severity:

C.2 Constant declaration should be preferred

Line no: 15
Contract: *SharedVariables*
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.

- ***fourRXTokenDecimals***

C.3 Contract includes Hardcoded Addresses

Line no - 17
Contract: *SharedVariables*
Description:
Keeping in mind the immutable nature of smart contracts, it is not considered a better practise to hardcode any address in the contract before deployment.
Most importantly, when that particular address is involved in token transfers.

```
16
17      address public devAddress = 0x64B8cb4C04Ba902010856d913B4e5DF940748Bf2;
18
```

Recommendation:
Instead of including hardcoded addresses in the contract, initialize those addresses within the constructors at the time of deployment.

Automated Test Results

```
FourRXFinance.deposit(uint256,address,uint8).stake (Flat_Finance.sol#1211) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
FourRXFinance.withdraw(uint256) (Flat_Finance.sol#1259-1303) ignores return value by fourRXToken.transfer(user.wallet,availableAmount) (Flat_Finance.sol#1288)
FourRXFinance.exitProgram(uint256) (Flat_Finance.sol#1305-1327) ignores return value by fourRXToken.transfer(user.wallet,availableAmount) (Flat_Finance.sol#1318)
FourRXFinance.withdrawDevFee(address,uint256) (Flat_Finance.sol#1346-1351) ignores return value by fourRXToken.transfer(withdrawingAddress,amount) (Flat_Finance.
```

```
Pragma version^0.6.12 (Flat_Finance.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
solc-0.6.12 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Function ReferralPool._addRefPoolRecord(address,uint256,uint8) (Flat_Finance.sol#884-890) is not in mixedCase
Function ReferralPool._cleanRefPoolUsers() (Flat_Finance.sol#892-895) is not in mixedCase
Parameter FourRXFinance.balanceOf(address,uint256)._userAddress (Flat_Finance.sol#1252) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

```
Pools.drawPool() (Flat_Finance.sol#927-952) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > poolDrewAt + 86400 (Flat_Finance.sol#928)
RewardsAndPenalties._calcHoldRewards(SharedVariables.Stake) (Flat_Finance.sol#1055-1064) uses timestamp for comparisons
        Dangerous comparisons:
        - holdBonusPercent > MAX_HOLD_BONUS_BP (Flat_Finance.sol#1059)
RewardsAndPenalties._calcRewards(SharedVariables.Stake) (Flat_Finance.sol#1076-1090) uses timestamp for comparisons
        Dangerous comparisons:
        - _calcBasisPoints(stake.deposit,rewards) >= REWARD_THRESHOLD_BP (Flat_Finance.sol#1079)
        - rewards > maxRewards (Flat_Finance.sol#1085)
RewardsAndPenalties._calcPenalty(SharedVariables.Stake,uint256) (Flat_Finance.sol#1092-1100) uses timestamp for comparisons
        Dangerous comparisons:
        - basisPoints >= REWARD_THRESHOLD_BP (Flat_Finance.sol#1095)
Insurance._checkForBaseInsuranceTrigger() (Flat_Finance.sol#1112-1118) uses timestamp for comparisons
        Dangerous comparisons:
        - fourRXToken.balanceOf(address(this)) <= _calcPercentage(maxContractBalance,BASE_INSURANCE_FOR_BP) (Flat_Finance.sol#1113)
Insurance._getInsuredAvailableAmount(SharedVariables.Stake,uint256) (Flat_Finance.sol#1120-1138) uses timestamp for comparisons
        Dangerous comparisons:
```

```
PercentageCalculator._calcPercentage(uint256,uint256) (Flat_Finance.sol#654-657) contains a tautology or contradiction:
        - require(bool)(basisPoints >= 0) (Flat_Finance.sol#655)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#tautology-or-contradiction
```