Bird Money

Smart Contract Audit Report

IMMUNE BYTES

Audits

Bird.Money

April 07, 2021

Introduction **3**
Abou BIrd.Money 3
About ImmuneBytes 3
Documentation Details **3**
Audit Process & Methodology **4**
Audit Details **4**
Audit Goals **5**
Security Level References **5**
High severity issues 6
Medium severity issues 11
Low severity issues 13
Recommendations **16**
Automated Audit **17**
Slither 17
Concluding Remarks **18**
Disclaimer **18**

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.

Introduction

1. Abou BIrd.Money

Bird Money is combining bleeding-edge technologies to deliver new DeFi products to all members of our flock.
The mission is to harness the power of data to create a financial ecosystem tailored to you. We're using our off-chain oracle analytics to provide protocols with trust data that's available on-chain.

Visit to know more about https://bird.money/

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

Bird Money team has provided documentation for the purpose of conducting the audit. The documents are:

1. https://docs.bird.money/

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.

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: Bird Money
- Languages: Solidity(Smart contract), Javascript(Unit Testing)
- Github commit hash for audit: dfd2502b73c8f54c9081682c3eb070fcd3c00629

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.

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.

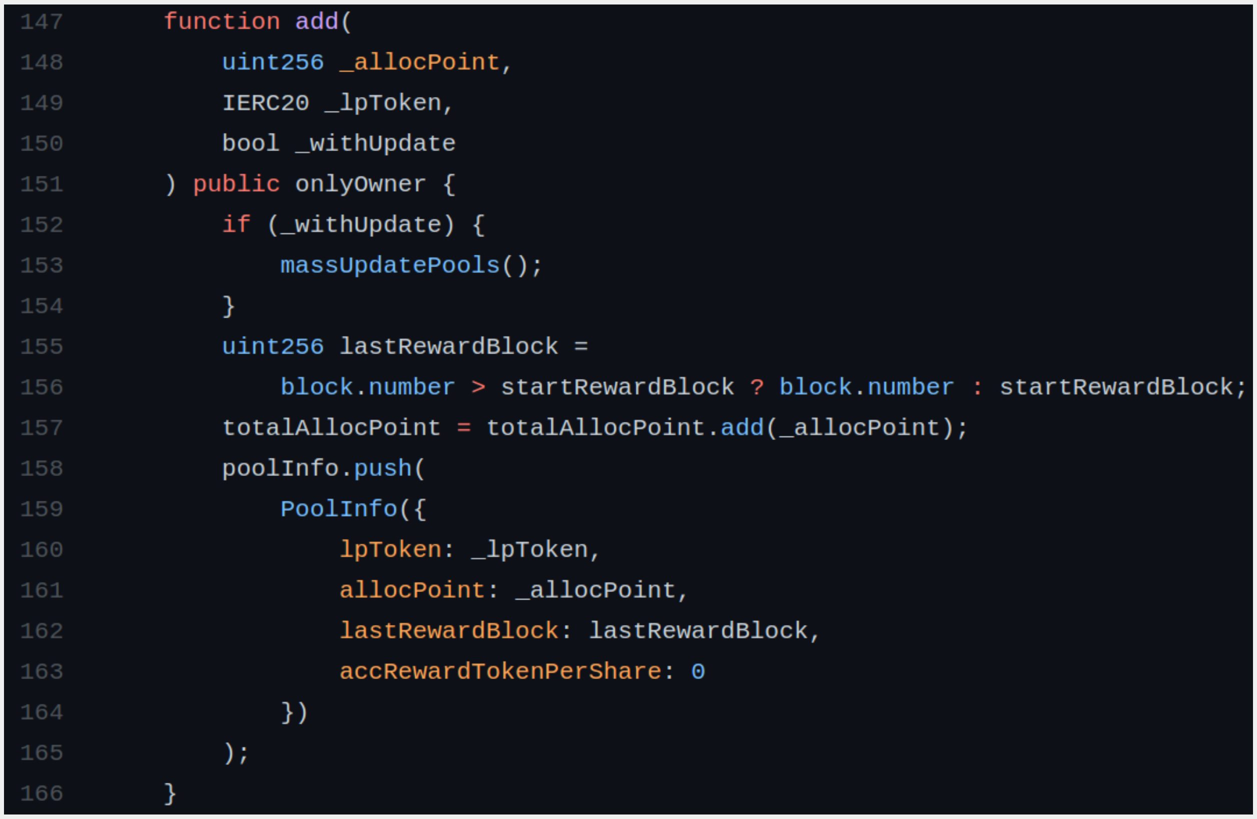
Issues	High	Medium	Low
Open	5	3	5
Closed	-	-	-

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.

High severity issues

1. **Similar LP Token Address can be added more than once.**
   Line no - 147
   Explanation:
   As per the current architecture of the MasterChef contract, the LP Tokens added in the pool of this contract should not be repeated. Since the address of an LP Token plays an imperative role in the calculation of rewards as well as keeping track of specific LP supply in the pool, 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 147 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.



```
147        function add(
148            uint256 _allocPoint,
149            IERC20 _lpToken,
150            bool _withUpdate
151        ) public onlyOwner {
152            if (_withUpdate) {
153                massUpdatePools();
154            }
155            uint256 lastRewardBlock =
156                block.number > startRewardBlock ? block.number : startRewardBlock;
157            totalAllocPoint = totalAllocPoint.add(_allocPoint);
158            poolInfo.push(
159                PoolInfo({
160                    lpToken: _lpToken,
161                    allocPoint: _allocPoint,
162                    lastRewardBlock: lastRewardBlock,
163                    accRewardTokenPerShare: 0
164                })
165            );
166        }
```

   Recommendation:
   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.

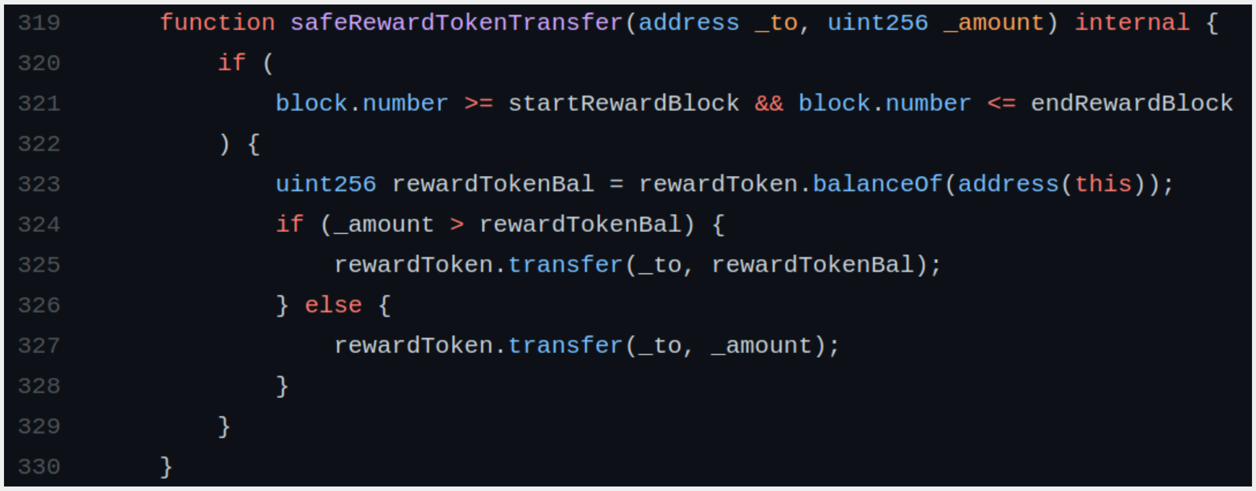
This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The require statement should be designed to check whether or not the passed lpToken address is already available in the contract. Moreover, the add() function should only execute if a new lpToken address is passed, thus eliminating any chances of repeating a similar lpToken in more than one pool.

2. **safeRewardTokenTransfer function does not execute adequately if Reward Token Balance in the contract is less than the amount of reward tokens to be transferred to a particular user**
Line no - 319
Explanation:
The safeRewardTokenTransfer 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 324).

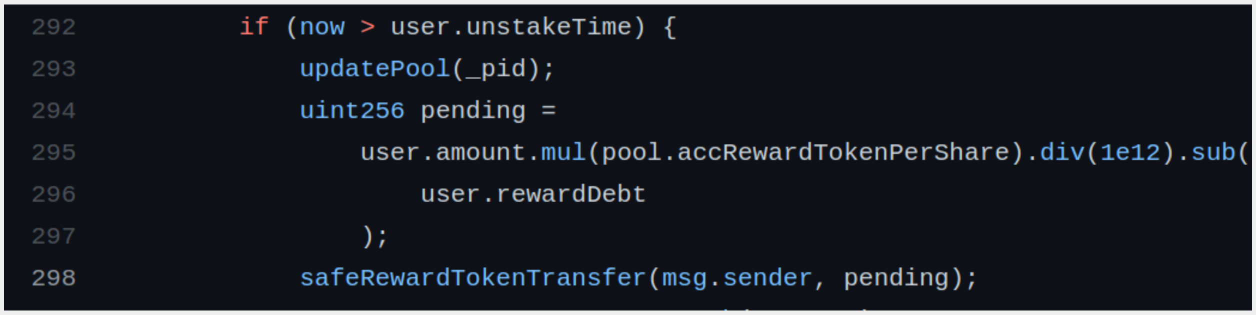
If the MasterChef contract has less reward token balance than the amount to be transferred, the user gets only the remaining reward tokens in the contract and not the actual amount that was supposed to be transferred(Line 325).

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


```
319     function safeRewardTokenTransfer(address _to, uint256 _amount) internal {
320         if (
321             block.number >= startRewardBlock && block.number <= endRewardBlock
322         ) {
323             uint256 rewardTokenBal = rewardToken.balanceOf(address(this));
324             if (_amount > rewardTokenBal) {
325                 rewardToken.transfer(_to, rewardTokenBal);
326             } else {
327                 rewardToken.transfer(_to, _amount);
328             }
329         }
330     }
```

For instance, if the user is supposed to receive 1000 reward tokens while calling the withdraw function after the complete lock-up period is over.
The withdraw function will call the safeRewardTokenTransfer function(Line 298) and pass the user's address and the reward token amount of 1000 tokens in the pending variable.

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.



```
        if (now > user.unstakeTime) {
            updatePool(_pid);
            uint256 pending =
                user.amount.mul(pool.accRewardTokenPerShare).div(1e12).sub(
                    user.rewardDebt
                );
            safeRewardTokenTransfer(msg.sender, pending);
```

However, if the MasterChef contract has only 800 reward tokens then the if condition at Line 324 will be executed and the user will only receive 800 reward tokens instead of 1000 reward tokens.

Now, because of the fact that the safeRewardTokenTransfer 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.

Is this scenario Intended?
Recommendation:
If the above-mentioned scenario was not considered while developing the safeRewardTokenTransfer, then the function must be updated in such a way that the user gets the actual amount of reward tokens whenever the safeRewardTokenTrasnfer function is called.

3. **Contract State Variables are being updated after External Calls. Leads to a Potential Reentrancy Scenario**
Line no - 278-282, 313-316
Explanation:
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.

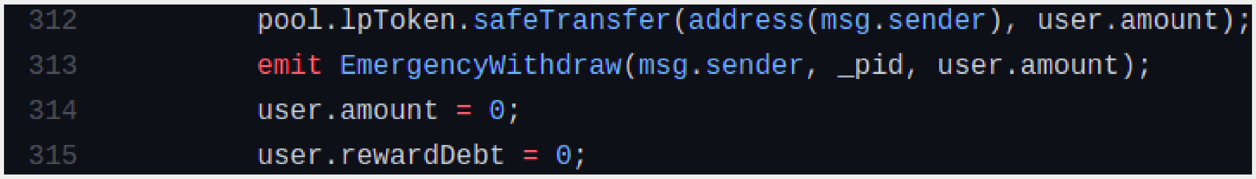
An external call within a function technically shifts the control flow of the contract to another contract for a particular period of time. Therefore, as per the Solidity Guidelines, any modification of the state variables in the base contract must be performed before executing the external call.
Updating 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

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.

- deposit() function at Line 278-282
- emergencyWithdraw() function at Line 314-316

For instance, the emergencyWithdraw function makes an external call to the lpToken contract(Line 312) to transfer the deposited amount of tokens(user.amount) back to the user. However the User struct(state variable in the contract) is updated after the external. This is not considered a secure practice while developing smart contracts in Solidity.



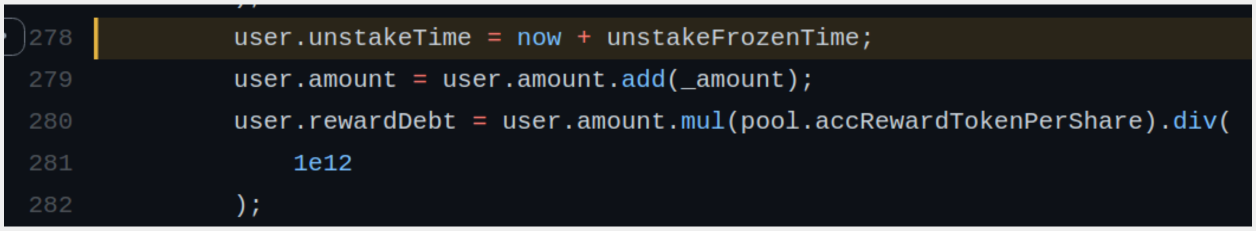
```
312            pool.lpToken.safeTransfer(address(msg.sender), user.amount);
313            emit EmergencyWithdraw(msg.sender, _pid, user.amount);
314            user.amount = 0;
315            user.rewardDebt = 0;
```

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

4. **Unstake Time increases 72 Hours with every new Deposit**
Line no - 278
Explanation:
The total lock period for every deposit has been assigned to be 72 hours(Line no - 73). It indicates that the user will be able withdraw his/her lp tokens as well as the rewards, after a total duration of 72 hours.

However, as per the current design of the deposit function, the unstake time for a particular user keeps increasing whenever any new deposit is made.

For instance, if a user deposits 1000 LP tokens initially, the unstake time for this user will be 72 hours, i.e., (current time + 72 hours) and the user will be able to withdraw his 1000 LP tokens only after this time period is over.
But if the user once again deposits 500 LP tokens before the unstake time is over, then the unstake time for this user is again updated to 72 hours more for his entire deposit amount. The user will be able to unstake his entire deposit amount of 1500 LP tokens only after the complete unstake period of 72 hours ends.



```
278          user.unstakeTime = now + unstakeFrozenTime;
279          user.amount = user.amount.add(_amount);
280          user.rewardDebt = user.amount.mul(pool.accRewardTokenPerShare).div(
281              1e12
282          );
```

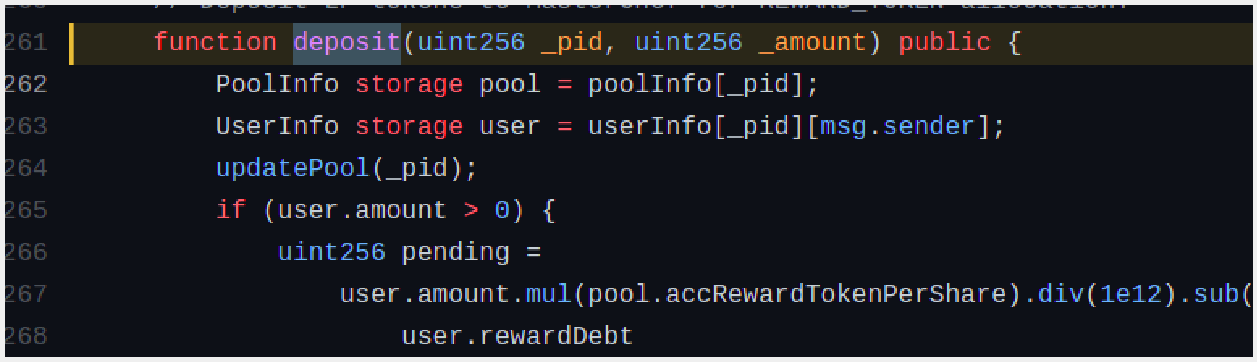
Recommendation:

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.

Is the above-mentioned behavior intended?
If the above-mentioned issue was not considered during the design of the deposit function, then the function should be modified in a way that the unstake period of each deposit is treated separately so that the users can get a clear idea of the withdrawal time for each of their deposits.

However, if this behaviour is intended, the community should be informed about this deposit & unstake time update functionality beforehand to eliminate any confusion later.

5. **User is capable of depositing ZERO amount of LP Tokens**
   Line no - 261-284
   Explanation:
   As per the current implementation of the deposit function, a user is capable of depositing ZERO amount of Lp Tokens as well.
   Since the user deposit amount plays a significant role in the reward calculation as well as withdrawal procedure, the input validation for the deposit amount must be implemented in the function.
   Moreover, allowing users to deposit ZERO amount of tokens doesn't represent a better function design for the deposit function.



```
261     function deposit(uint256 _pid, uint256 _amount) public {
262         PoolInfo storage pool = poolInfo[_pid];
263         UserInfo storage user = userInfo[_pid][msg.sender];
264         updatePool(_pid);
265         if (user.amount > 0) {
266             uint256 pending =
267                 user.amount.mul(pool.accRewardTokenPerShare).div(1e12).sub(
268                     user.rewardDebt
```

   Recommendation:
   The deposit function should include a require statement to validate the _amount argument passed by the user. The require statement must ensure that this argument is either greater than ZERO or a Minimum Deposit Threshold that can be set by the owner of the contract

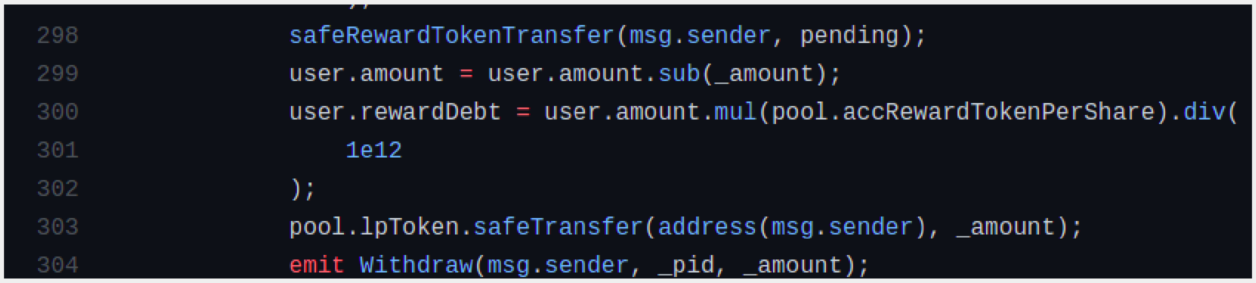
This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Medium severity issues

1. **Violation of Check_Effects_Interaction Pattern in the Withdraw function**
   Line no - 292-305

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

   In case of the withdraw function, however, the Check Effects Interaction Pattern is not followed effectively.



```
298                 safeRewardTokenTransfer(msg.sender, pending);
299                 user.amount = user.amount.sub(_amount);
300                 user.rewardDebt = user.amount.mul(pool.accRewardTokenPerShare).div(
301                     1e12
302                 );
303                 pool.lpToken.safeTransfer(address(msg.sender), _amount);
304                 emit Withdraw(msg.sender, _pid, _amount);
```

   The withdraw function makes 2 external calls within the function. The first external call is made to the reward token contract(Line 298). However, the user state is updated after the external call at Line 299-301.

   The second external call is made to the lpToken(Line 303) but the Withdrawn event is emitted after this external call at Line 304.
   Recommendation:
   Check Effects Interaction Pattern must be followed while implementing external calls in a function.

2. **updatePool and massUpdatePools functions have been assigned a Public visibility**
   Line no - 229-234, 237-258
   Explanation:
   The updatePool and massUpdatePools functions include imperative 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.

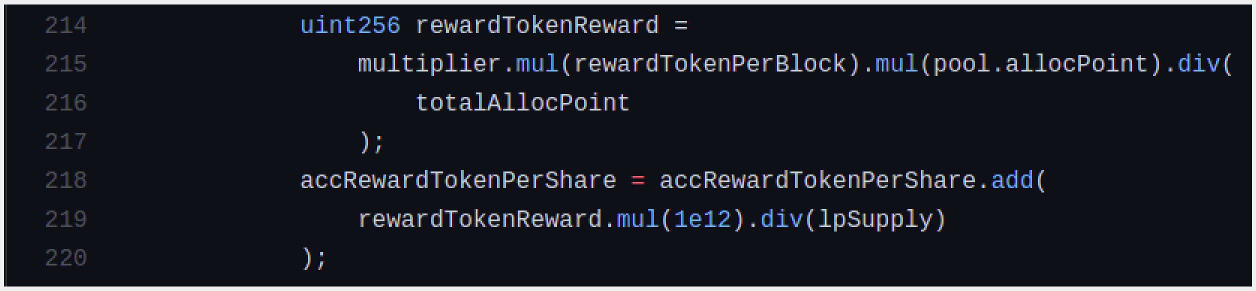
This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

However, instead of an internal visibility, these functions have been assigned a public visibility.
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.

3. **Multiplication is being performed on the result of Division**
Line no - Line 214-219, 248-256
Explanation:
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:

- 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            );
```

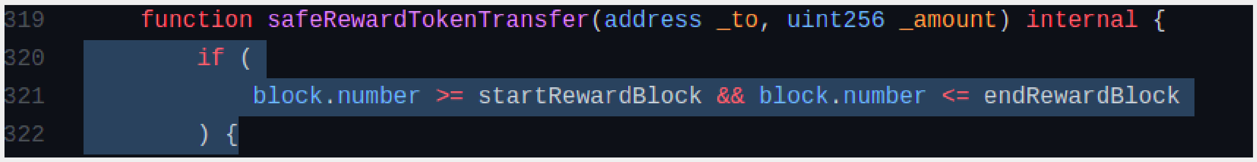
Recommendation:
Solidity doesn't encourage arithmetic operations that involve division before multiplication. Therefore the above-mentioned function should be checked once and redesigned if they do not lead to expected results.

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.

Low severity issues

1. **safeRewardTokenTransfer function should include require statement instead of IF-Else Statement**
   Line no: 320-322
   Explanation
   The safeRewardTokenTransfer function includes an if statement at the very beginning of the function to check whether or not the block.number lies in the valid range of reward blocks.
   The function body is only executed if this IF statement holds true.

   In order to check for such validations in a function, require statements are more preferable and effective solidity. While it helps in gas optimizations it also enhances the readability of the code.



```
319     function safeRewardTokenTransfer(address _to, uint256 _amount) internal {
320         if (
321             block.number >= startRewardBlock && block.number <= endRewardBlock
322         ) {
```

   Recommendation
   Use require statement instead of IF statement in the above-mentioned function line.
   For instance,
   require(block.number >= startRewardBlock && block.number <= endRewardBlock,”Error MSG: Block Number doesn’t lie in Valid Range”);

2. **External Visibility should be preferred**
   Explanation:
   Those functions that are never called throughout the contract should be marked as external visibility instead of public visibility.
   This will effectively result in Gas Optimization as well.

   Therefore, the following function must be marked as external within the contract:
   - setRewardToken
   - setAll
   - setUnstakeFrozenTime
   - setRewardTokenPerBlock
   - setStartRewardBlock
   - serEndRewardBlock
   - setBonusEndBlock
   - add

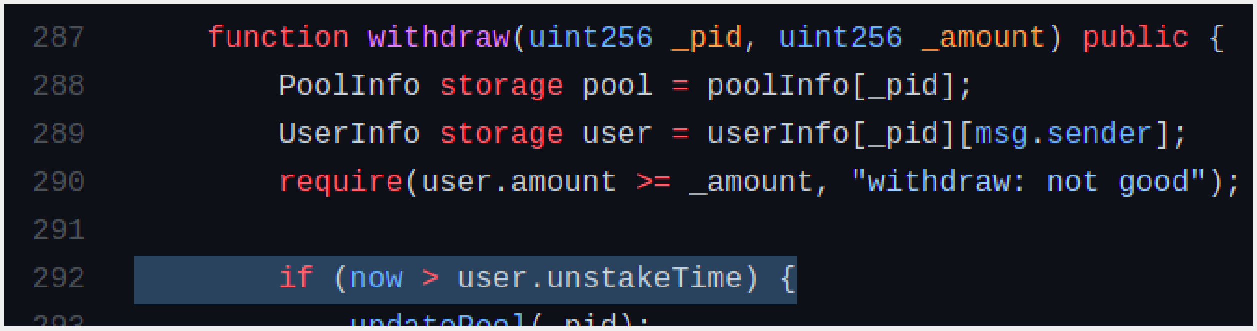
This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

- set
- deposit
- withdraw
- emergencyWithdraw
- setMigrator
- depositRewardTokens
- withdrawRewardTokens

3. **withdraw function should include require statement instead of IF-Else Statement**
   Line no: 292
   Explanation:
   As per the current architecture of the contract, the withdraw function should only enter its function body if 2 imperative conditions are met:
   - User Deposit amount is more than ZERO
   - Current Time is more than the unstake time

   Since these are strict conditions and must be fulfilled before a user can withdraw his/her LP Tokens and reward, require statements will be a more effective approach to check such conditions.

   While the amount condition is already being checked using the require statement, the unstake time condition should also be checked with a require statement instead of a IF-Else statement.



```
287     function withdraw(uint256 _pid, uint256 _amount) public {
288         PoolInfo storage pool = poolInfo[_pid];
289         UserInfo storage user = userInfo[_pid][msg.sender];
290         require(user.amount >= _amount, "withdraw: not good");
291
292         if (now > user.unstakeTime) {
293             updatePool(_pid);
```

   Recommendation;
   Use require statement instead of IF statement in the above-mentioned function line.
   For instance,
   j`1wrequire(now> user.unstakeTime,"Error MSG: Unstake Time Not Reached Yet");

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.

4. **Return Value of an External Call is never used Effectively**

   Line no - 325, 327, 345, 349
   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 MasterChef contract never uses these return values throughout the contract.
   Recommendation:
   Effective use of all the return values from external calls must be ensured within the contract.

5. **No Events emitted after imperative State Variable modification**
   Line no -118,122,129,133,137

   Description:
   Functions that update an imperative arithmetic state variable contract should emit an event after the updation.
   The following functions modify some crucial arithmetic parameters like rewardTokenPerblock, startRewardBlock, endRewardBlock etc in the Masterchef contract but don't emit any event after that:
   - setUnstakeFrozenTime
   - setRewardTokenPerBlock
   - setStartRewardBlock
   - setEndRewardBlock
   - setEndRewardBlock
   - setBonusEndBlock

   Since there is no event emitted on updating these variables, it might be difficult to track it off-chain.

   Recommendation:
   An event should be fired after changing crucial arithmetic state variables.

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.

Recommendations

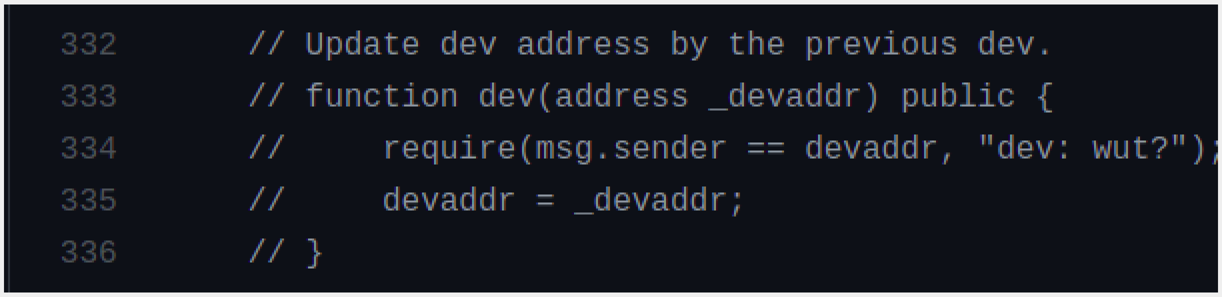
1. **NatSpec Annotations must be included**

   Description:
   Smart contract does not include the NatSpec annotations adequately.

   Recommendation:
   Cover by NatSpec all Contract methods.

2. Commented codes must be wiped-out before deployment
   Explanation
   The Masterchef contract includes quite a few commented codes regarding a devAddress state variable(Line 51, 85).

   This badly affects the readability of the code.



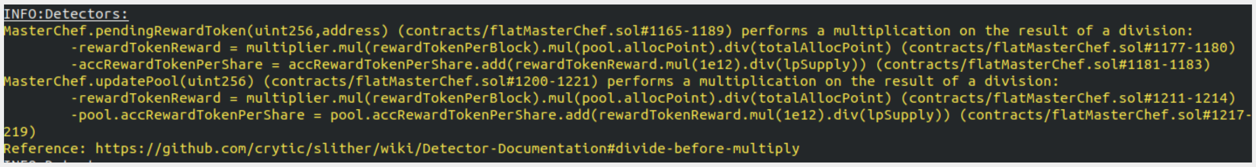
```
    // Update dev address by the previous dev.
    // function dev(address _devaddr) public {
    //     require(msg.sender == devaddr, "dev: wut?");
    //     devaddr = _devaddr;
    // }
```

   Recommendation:
   If these instances of code are not required in the current version of the contract, then the commented codes must be removed before deployment.

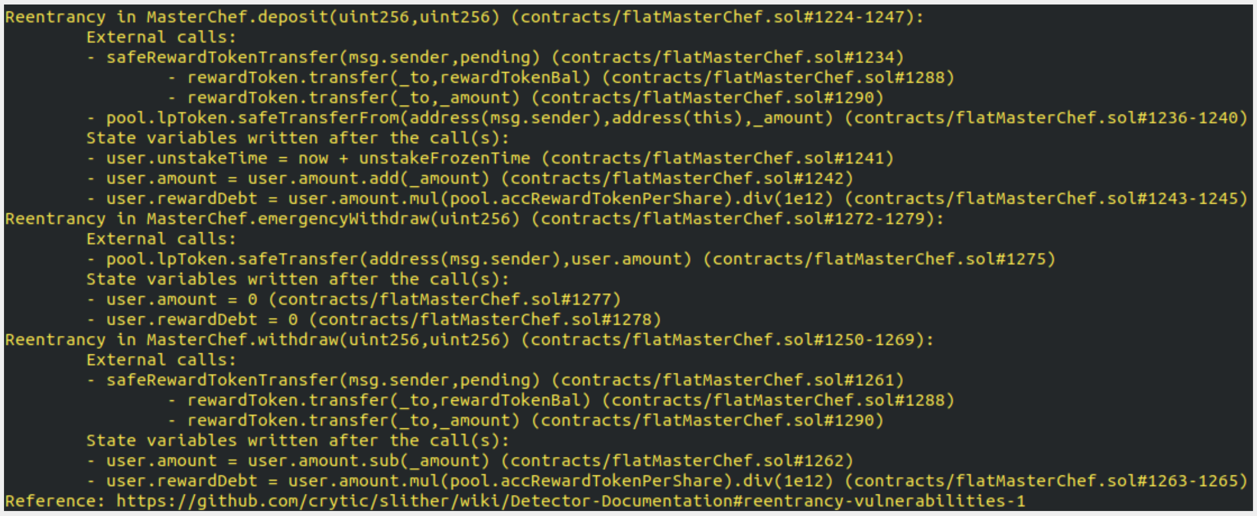
This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Automated Audit

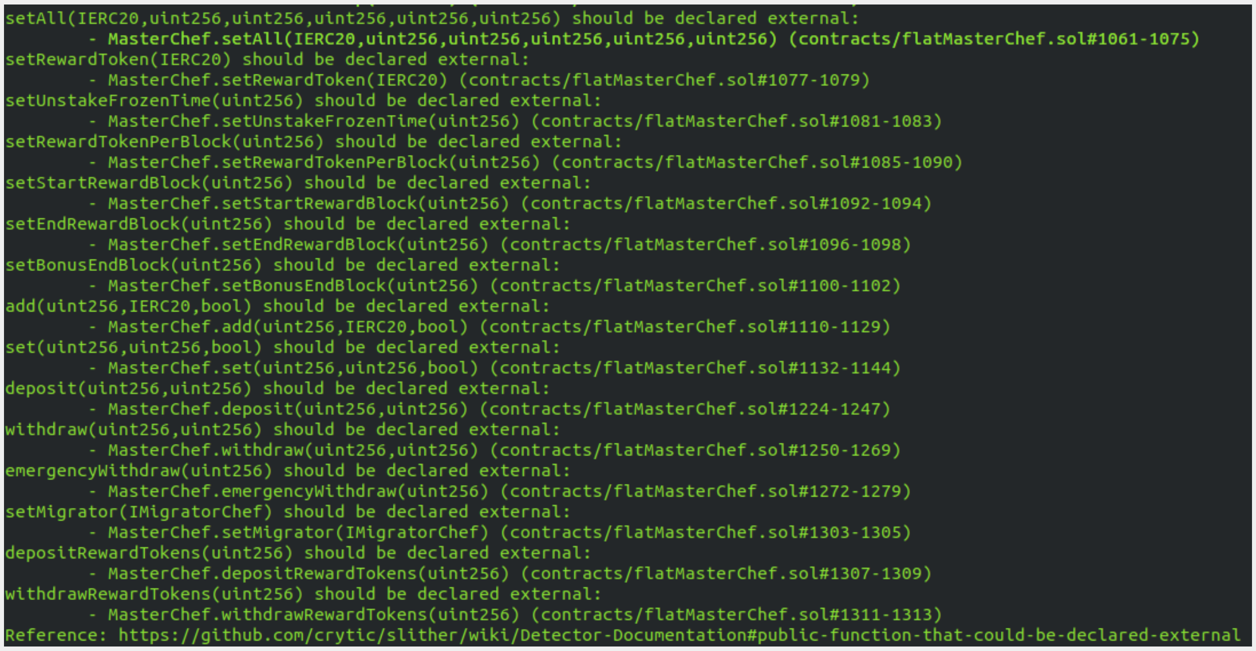
Slither



```
INFO:Detectors:
MasterChef.pendingRewardToken(uint256,address) (contracts/flatMasterChef.sol#1165-1189) performs a multiplication on the result of a division:
        -rewardTokenReward = multiplier.mul(rewardTokenPerBlock).mul(pool.allocPoint).div(totalAllocPoint) (contracts/flatMasterChef.sol#1177-1180)
        -accRewardTokenPerShare = accRewardTokenPerShare.add(rewardTokenReward.mul(1e12).div(lpSupply)) (contracts/flatMasterChef.sol#1181-1183)
MasterChef.updatePool(uint256) (contracts/flatMasterChef.sol#1200-1221) performs a multiplication on the result of a division:
        -rewardTokenReward = multiplier.mul(rewardTokenPerBlock).mul(pool.allocPoint).div(totalAllocPoint) (contracts/flatMasterChef.sol#1211-1214)
        -pool.accRewardTokenPerShare = pool.accRewardTokenPerShare.add(rewardTokenReward.mul(1e12).div(lpSupply)) (contracts/flatMasterChef.sol#1217-
219)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
```



```
Reentrancy in MasterChef.deposit(uint256,uint256) (contracts/flatMasterChef.sol#1224-1247):
        External calls:
        - safeRewardTokenTransfer(msg.sender,pending) (contracts/flatMasterChef.sol#1234)
                - rewardToken.transfer(_to,rewardTokenBal) (contracts/flatMasterChef.sol#1288)
                - rewardToken.transfer(_to,_amount) (contracts/flatMasterChef.sol#1290)
        - pool.lpToken.safeTransferFrom(address(msg.sender),address(this),_amount) (contracts/flatMasterChef.sol#1236-1240)
        State variables written after the call(s):
        - user.unstakeTime = now + unstakeFrozenTime (contracts/flatMasterChef.sol#1241)
        - user.amount = user.amount.add(_amount) (contracts/flatMasterChef.sol#1242)
        - user.rewardDebt = user.amount.mul(pool.accRewardTokenPerShare).div(1e12) (contracts/flatMasterChef.sol#1243-1245)
Reentrancy in MasterChef.emergencyWithdraw(uint256) (contracts/flatMasterChef.sol#1272-1279):
        External calls:
        - pool.lpToken.safeTransfer(address(msg.sender),user.amount) (contracts/flatMasterChef.sol#1275)
        State variables written after the call(s):
        - user.amount = 0 (contracts/flatMasterChef.sol#1277)
        - user.rewardDebt = 0 (contracts/flatMasterChef.sol#1278)
Reentrancy in MasterChef.withdraw(uint256,uint256) (contracts/flatMasterChef.sol#1250-1269):
        External calls:
        - safeRewardTokenTransfer(msg.sender,pending) (contracts/flatMasterChef.sol#1261)
                - rewardToken.transfer(_to,rewardTokenBal) (contracts/flatMasterChef.sol#1288)
                - rewardToken.transfer(_to,_amount) (contracts/flatMasterChef.sol#1290)
        State variables written after the call(s):
        - user.amount = user.amount.sub(_amount) (contracts/flatMasterChef.sol#1262)
        - user.rewardDebt = user.amount.mul(pool.accRewardTokenPerShare).div(1e12) (contracts/flatMasterChef.sol#1263-1265)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```



```
setAll(IERC20,uint256,uint256,uint256,uint256,uint256) should be declared external:
        - MasterChef.setAll(IERC20,uint256,uint256,uint256,uint256,uint256) (contracts/flatMasterChef.sol#1061-1075)
setRewardToken(IERC20) should be declared external:
        - MasterChef.setRewardToken(IERC20) (contracts/flatMasterChef.sol#1077-1079)
setUnstakeFrozenTime(uint256) should be declared external:
        - MasterChef.setUnstakeFrozenTime(uint256) (contracts/flatMasterChef.sol#1081-1083)
setRewardTokenPerBlock(uint256) should be declared external:
        - MasterChef.setRewardTokenPerBlock(uint256) (contracts/flatMasterChef.sol#1085-1090)
setStartRewardBlock(uint256) should be declared external:
        - MasterChef.setStartRewardBlock(uint256) (contracts/flatMasterChef.sol#1092-1094)
setEndRewardBlock(uint256) should be declared external:
        - MasterChef.setEndRewardBlock(uint256) (contracts/flatMasterChef.sol#1096-1098)
setBonusEndBlock(uint256) should be declared external:
        - MasterChef.setBonusEndBlock(uint256) (contracts/flatMasterChef.sol#1100-1102)
add(uint256,IERC20,bool) should be declared external:
        - MasterChef.add(uint256,IERC20,bool) (contracts/flatMasterChef.sol#1110-1129)
set(uint256,uint256,bool) should be declared external:
        - MasterChef.set(uint256,uint256,bool) (contracts/flatMasterChef.sol#1132-1144)
deposit(uint256,uint256) should be declared external:
        - MasterChef.deposit(uint256,uint256) (contracts/flatMasterChef.sol#1224-1247)
withdraw(uint256,uint256) should be declared external:
        - MasterChef.withdraw(uint256,uint256) (contracts/flatMasterChef.sol#1250-1269)
emergencyWithdraw(uint256) should be declared external:
        - MasterChef.emergencyWithdraw(uint256) (contracts/flatMasterChef.sol#1272-1279)
setMigrator(IMigratorChef) should be declared external:
        - MasterChef.setMigrator(IMigratorChef) (contracts/flatMasterChef.sol#1303-1305)
depositRewardTokens(uint256) should be declared external:
        - MasterChef.depositRewardTokens(uint256) (contracts/flatMasterChef.sol#1307-1309)
withdrawRewardTokens(uint256) should be declared external:
        - MasterChef.withdrawRewardTokens(uint256) (contracts/flatMasterChef.sol#1311-1313)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
```

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.

Concluding Remarks

While conducting the audits of Bird Money smart contract, 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 Bird Money 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 Bird Money 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.

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.