MythX

REPORT 5FFA8B140B940B0018248DB4

Created	Sun Jan 10 2021 05:05:24 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	block@chain.church

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
0b3b1090-0b06-47a5-9e56-4418617790a8	localhost/contracts/BonusRewards.sol	5

Analysis 0b3b1090-0b06-47a5-9e56-4418617790a8 MythX

Started	Sun Jan 10 2021 05:05:27 GMT+0000 (Coordinated Universal Time)
Finished	Sun Jan 10 2021 05:50:48 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	Localhost/Contracts/BonusRewards.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	5

ISSUES

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

localhost/contracts/BonusRewards.sol

Locations

```
1  // SPDX-License-Identifier: NONE
2
3  pragma solidity ^0.8.0;
4
5  import "./utils/Ownable.sol";
```

LOW

SWC-113

Multiple calls are executed in the same transaction.

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source file

localhost/contracts/BonusRewards.sol

Locations

```
255
256  if (_token == address(0)) { // token address(0) = ETH
257  payable(owner()).transfer(address(this).balance);
258  } else {
259  IERC20(_token).transfer(owner(), balance);
```

LOW

SWC-116

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

localhost/contracts/BonusRewards.sol

Locations

```
78  function updatePool(address _lpToken) public override {
79  Pool storage pool = pools[_lpToken];
80  if (block.timestamp <= pool.lastUpdatedAt) return;
81  uint256 lpTotal = IERC20(_lpToken).balanceOf(address(this));
82  if (lpTotal == 0) {
```

LOW

SWC-116

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

localhost/contracts/BonusRewards.sol

Locations

```
156 ) external override notPaused {
157 require(_isAuthorized(msg.sender, allowedTokenAuthorizers[_bonusTokenAddr]), "BonusRewards: not authorized caller");
158 require(_startTime >= block.timestamp, "BonusRewards: startTime in the past");
159
160 // make sure the pool is in the right state (exist with no active bonus at the moment) to add new bonus tokens
```

LOW

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file

localhost/contracts/BonusRewards.sol

Locations

```
286 function _safeTransfer(address _token, uint256 _amount) private returns (uint256 _transferred) {
287 IERC20 token = IERC20(_token);
288 uint256 balance = token.balanceOf(address(this));
289 if (balance > _amount) {
290 token.safeTransfer(msg.sender, _amount);
```

Source file

localhost/contracts/BonusRewards.sol

Locations

```
13 * @notice ETH is not allowed to be an bonus token, use wETH instead
14 */
15 contract BonusRewards is IBonusRewards, Ownable, ReentrancyGuard {
16 using SafeERC20 for IERC20;
17
18 uint256 private constant WEEK = 7 days;
19 uint256 private constant CAL_MULTIPLIER = 1e12; // help calculate rewards/bonus PerToken only. 1e12 will allow meaningful $1 deposit in a $1bn pool
20 bool public paused;
21 address[] private responders;
22 address[] private poolList;
23 // lpToken => Pool
24 mapping(address => Pool) private pools;
25 // lpToken => User address => User data
26 mapping(address => mapping(address => User)) private users;
27 // bonus token => [] allowed authorizers to add bonus tokens
28 mapping(address => address[]) private allowedTokenAuthorizers;
29 // bonusTokenAddr => 1, used to avoid collecting bonus token when not ready
30 mapping(address => uint8) private bonusTokenAddrMap;
31
32 modifier notPaused() {
33 require(!paused, "BonusRewards: paused");
34 _;
35 }
36
37 function getPoolList() external view override returns (address[] memory) {
38 return poolList;
39 }
40
41 function getPool(address _lpToken) external view override returns (Pool memory) {
42 return pools[_lpToken];
43 }
44
45 function viewRewards(address _lpToken, address _user) public view override returns (uint256[] memory) {
46 Pool memory pool = pools[_lpToken];
47 User memory user = users[_lpToken][_user];
48 uint256[] memory rewards = new uint256[](pool.bonuses.length);
49 if (user.amount <= 0) return rewards;
50
51 uint256 rewardsWriteoffsLen = user.rewardsWriteoffs.length;
52 for (uint256 i = 0; i < rewards.length; i ++) {
53 Bonus memory bonus = pool.bonuses[i];
54 if (bonus.startTime < block.timestamp && bonus.remBonus > 0) {
55 uint256 lpTotal = IERC20(_lpToken).balanceOf(address(this));
56 uint256 bonusForTime = _calRewardsForTime(bonus, pool.lastUpdatedAt);
57 uint256 bonusPerToken = bonus.accRewardsPerToken + bonusForTime / lpTotal;
```

```
uint256 rewardsWriteoff = rewardsWriteoffsLen <= i ? 0 : user.rewardsWriteoffs[i];
rewards[i] = user.amount * bonusPerToken / CAL_MULTIPLIER - rewardsWriteoff;
}
}
return rewards;
}

function getUser(address _lpToken, address _account) external view override returns (User memory, uint256[] memory) {
return (users[_lpToken][_account], viewRewards(_lpToken, _account));
}

function getAuthorizers(address _bonusTokenAddr) external view override returns (address[] memory) {
return allowedTokenAuthorizers[_bonusTokenAddr];
}

function getResponders() external view override returns (address[] memory) {
return responders;
}

/// @notice update pool's bonus per staked token till current block timestamp
function updatePool(address _lpToken) public override {
Pool storage pool = pools[_lpToken];
if (block.timestamp <= pool.lastUpdatedAt) return;
uint256 lpTotal = IERC20(_lpToken).balanceOf(address(this));
if (lpTotal == 0) {
pool.lastUpdatedAt = block.timestamp;
return;
}

for (uint256 i = 0; i < pool.bonuses.length; i ++) {
Bonus storage bonus = pool.bonuses[i];
if (pool.lastUpdatedAt < bonus.endTime && bonus.startTime < block.timestamp) {
uint256 bonusForTime = _calRewardsForTime(bonus, pool.lastUpdatedAt);
bonus.accRewardsPerToken = bonus.accRewardsPerToken + bonusForTime / lpTotal;
}
}
pool.lastUpdatedAt = block.timestamp;
}

function claimRewards(address _lpToken) public override {
User memory user = users[_lpToken][msg.sender];
if (user.amount == 0) return;

updatePool(_lpToken);
_claimRewards(_lpToken, user);
_updateUserWriteoffs(_lpToken);
}

function claimRewardsForPools(address[] calldata _lpTokens) external override {
for (uint256 i = 0; i < _lpTokens.length; i++) {
claimRewards(_lpTokens[i]);
}
}

function deposit(address _lpToken, uint256 _amount) external override nonReentrant notPaused {
require(pools[_lpToken].lastUpdatedAt > 0, "Blacksmith: pool does not exists");
require(IERC20(_lpToken).balanceOf(msg.sender) >= _amount, "Blacksmith: insufficient balance");

updatePool(_lpToken);
User storage user = users[_lpToken][msg.sender];
_claimRewards(_lpToken, user);
user.amount = user.amount + _amount;
_updateUserWriteoffs(_lpToken);
```

```
121
122 IERC20(_lpToken).safeTransferFrom(msg.sender, address(this), _amount);
123 emit Deposit(msg.sender, _lpToken, _amount);
124 }
125
126 /// @notice withdraw up to all user deposited
127 function withdraw(address _lpToken, uint256 _amount) external override nonReentrant notPaused {
128 updatePool(_lpToken);
129
130 User storage user = users[_lpToken][msg.sender];
131 _claimRewards(_lpToken, user);
132 uint256 amount = user.amount > _amount ? _amount : user.amount;
133 user.amount = user.amount - amount;
134 _updateUserWriteoffs(_lpToken);
135
136 _safeTransfer(_lpToken, amount);
137 emit Withdraw(msg.sender, _lpToken, amount);
138 }
139
140 /// @notice withdraw all without rewards
141 function emergencyWithdraw(address _lpToken) external override nonReentrant {
142 User storage user = users[_lpToken][msg.sender];
143 uint256 amount = user.amount;
144 user.amount = 0;
145 _safeTransfer(_lpToken, amount);
146 emit Withdraw(msg.sender, _lpToken, amount);
147 }
148
149 /// @notice called by authorizers only
150 function addBonus(
151 address _lpToken,
152 address _bonusTokenAddr,
153 uint256 _startTime,
154 uint256 _weeklyRewards,
155 uint256 _transferAmount
156 ) external override notPaused {
157 require(_isAuthorized(msg.sender, allowedTokenAuthorizers[_bonusTokenAddr]), "BonusRewards: not authorized caller");
158 require(_startTime >= block.timestamp, "BonusRewards: startTime in the past");
159
160 // make sure the pool is in the right state (exist with no active bonus at the moment) to add new bonus tokens
161 Pool memory pool = pools[_lpToken];
162 require(pool.lastUpdatedAt != 0, "BonusRewards: pool does not exist");
163 Bonus[] memory bonuses = pool.bonuses;
164 for (uint256 i = 0; i < bonuses.length; i++) {
165 if (bonuses[i].bonusTokenAddr == _bonusTokenAddr) {
166 // when there is alreay a bonus program with the same bonus token, make sure the program has ended properly
167 require(bonuses[i].endTime + WEEK < block.timestamp, "BonusRewards: last bonus period hasn't ended");
168 require(bonuses[i].remBonus == 0, "BonusRewards: last bonus not all claimed");
169 }
170 }
171
172 IERC20 bonusTokenAddr = IERC20(_bonusTokenAddr);
173 uint256 balanceBefore = bonusTokenAddr.balanceOf(address(this));
174 bonusTokenAddr.safeTransferFrom(msg.sender, address(this), _transferAmount);
175 uint256 received = bonusTokenAddr.balanceOf(address(this)) - balanceBefore;
176 // endTime is based on how much tokens transfered v.s. planned weekly rewards
177 uint256 endTime = (received / _weeklyRewards) * WEEK + _startTime;
178
179 pools[_lpToken].bonuses.push(Bonus({
180 bonusTokenAddr: _bonusTokenAddr,
181 startTime: _startTime,
182 endTime: endTime,
183 weeklyRewards: _weeklyRewards,
```

```
accRewardsPerToken: 0,
remBonus: received
}));
}

/// @notice extend the current bonus program, the program has to be active (endTime is in the future)
function extendBonus(
address _lpToken,
uint256 _poolBonusId,
address _bonusTokenAddr,
uint256 _transferAmount
) external override notPaused {
require(_isAuthorized(msg.sender, allowedTokenAuthorizers[_bonusTokenAddr]), "BonusRewards: not authorized caller");

Bonus memory bonus = pools[_lpToken].bonuses[_poolBonusId];
require(bonus.bonusTokenAddr == _bonusTokenAddr, "BonusRewards: bonus and id dont match");
require(bonus.endTime > block.timestamp, "BonusRewards: bonus program ended, please start a new one");

IERC20 bonusTokenAddr = IERC20(_bonusTokenAddr);
uint256 balanceBefore = bonusTokenAddr.balanceOf(address(this));
bonusTokenAddr.safeTransferFrom(msg.sender, address(this), _transferAmount);
uint256 received = bonusTokenAddr.balanceOf(address(this)) - balanceBefore;
// endTime is based on how much tokens transfered v.s. planned weekly rewards
uint256 endTime = (received / bonus.weeklyRewards) * WEEK + bonus.endTime;

pools[_lpToken].bonuses[_poolBonusId].endTime = endTime;
pools[_lpToken].bonuses[_poolBonusId].remBonus = bonus.remBonus + received;
}

/// @notice add pools and authorizers to add bonus tokens for pools, combine two calls into one. Only reason we add pools is when bonus tokens will be added
function addPoolsAndAllowBonus(
address[] calldata _lpTokens,
address[] calldata _bonusTokenAddrs,
address[] calldata _authorizers
) external override onlyOwner notPaused {
// add pools
for (uint256 i = 0; i < _lpTokens.length; i++) {
Pool memory pool = pools[_lpTokens[i]];
require(pool.lastUpdatedAt == 0, "BonusRewards: pool exists");
pools[_lpTokens[i]].lastUpdatedAt = block.timestamp;
poolList.push(_lpTokens[i]);
}

// add bonus tokens and their authorizers (who are allowed to add the token to pool)
for (uint256 i = 0; i < _bonusTokenAddrs.length; i++) {
allowedTokenAuthorizers[_bonusTokenAddrs[i]] = _authorizers;
bonusTokenAddrMap[_bonusTokenAddrs[i]] = 1;
}
}

/// @notice use start and end to avoid gas limit in one call
function updatePools(uint256 _start, uint256 _end) external override {
address[] memory poolListCopy = poolList;
for (uint256 i = _start; i < _end; i++) {
updatePool(poolListCopy[i]);
}
}

/// @notice collect bonus token dust to treasury
function collectDust(address _token, address _lpToken, uint256 _poolBonusId) external override onlyOwner {
require(pools[_token].lastUpdatedAt == 0, "BonusRewards: lpToken, not allowed");

uint256 balance = IERC20(_token).balanceOf(address(this));
```

```
if (bonusTokenAddrMap[_token] == 1) {
// bonus token
Bonus memory bonus = pools[_lpToken].bonuses[_poolBonusId];
require(bonus.bonusTokenAddr == _token, "BonusRewards: wrong pool");
require(bonus.endTime + WEEK < block.timestamp, "BonusRewards: not ready");
balance = bonus.remBonus;
pools[_lpToken].bonuses[_poolBonusId].remBonus = 0;
}

if (_token == address(0)) { // token address(0) = ETH
payable(owner()).transfer(address(this).balance);
} else {
IERC20(_token).transfer(owner(), balance);
}
}

function setResponders(address[] calldata _responders) external override onlyOwner {
responders = _responders;
}

function setPaused(bool _paused) external override {
require(_isAuthorized(msg.sender, responders), "BonusRewards: caller not responder");
paused = _paused;
}

function _updateUserWriteoffs(address _lpToken) private {
Bonus[] memory bonuses = pools[_lpToken].bonuses;
User storage user = users[_lpToken][msg.sender];
for (uint256 i = 0; i < bonuses.length; i++) {
// update writeoff to match current acc rewards per token
if (user.rewardsWriteoffs.length == i) {
user.rewardsWriteoffs.push(user.amount * bonuses[i].accRewardsPerToken / CAL_MULTIPLIER);
} else {
user.rewardsWriteoffs[i] = user.amount * bonuses[i].accRewardsPerToken / CAL_MULTIPLIER;
}
}
}

/// @notice tranfer upto what the contract has
function _safeTransfer(address _token, uint256 _amount) private returns (uint256 _transferred) {
IERC20 token = IERC20(_token);
uint256 balance = token.balanceOf(address(this));
if (balance > _amount) {
token.safeTransfer(msg.sender, _amount);
_transferred = _amount;
} else if (balance > 0) {
token.safeTransfer(msg.sender, balance);
_transferred = balance;
}
}

function _calRewardsForTime(Bonus memory _bonus, uint256 _lastUpdatedAt) internal view returns (uint256) {
if (_bonus.endTime <= _lastUpdatedAt) return 0;

uint256 calEndTime = block.timestamp > _bonus.endTime ? _bonus.endTime : block.timestamp;
uint256 calStartTime = _lastUpdatedAt > _bonus.startTime ? _lastUpdatedAt : _bonus.startTime;
uint256 timePassed = calEndTime - calStartTime;
return _bonus.weeklyRewards * CAL_MULTIPLIER * timePassed / WEEK;
}

function _claimRewards(address _lpToken, User memory _user) private {
// only claim if user has deposited before
if (_user.amount > 0) {
```

```
uint256 rewardsWriteoffsLen = _user.rewardsWriteoffs.length;
Bonus[] memory bonuses = pools[_lpToken].bonuses;
for (uint256 i = 0; i < bonuses.length; i++) {
uint256 rewardsWriteoff = rewardsWriteoffsLen == i ? 0 : _user.rewardsWriteoffs[i];
uint256 bonusSinceLastUpdate = _user.amount * bonuses[i].accRewardsPerToken / CAL_MULTIPLIER - rewardsWriteoff;
if (bonusSinceLastUpdate > 0) {
uint256 transferred = _safeTransfer(bonuses[i].bonusTokenAddr, bonusSinceLastUpdate); // transfer bonus tokens to user
pools[_lpToken].bonuses[i].remBonus = bonuses[i].remBonus - transferred;
}
}
}
}

// only owner or authorized users from list
function _isAuthorized(address _addr, address[] memory checkList) private view returns (bool) {
if (_addr == owner()) return true;

for (uint256 i = 0; i < checkList.length; i++) {
if (msg.sender == checkList[i]) {
return true;
}
}
return false;
}
}
```