MythX

# REPORT 604AAD0F0518C00018E452BD

| | |
|---|---|
| Created | Thu Mar 11 2021 23:51:43 GMT+0000 (Coordinated Universal Time) |
| Number of analyses | 1 |
| User | team@hyruleswap.com |

## REPORT SUMMARY

| Analyses ID | Main source file | Detected vulnerabilities |
|---|---|---|
| 70f085ad-d861-431f-b503-114e34e6886b | contracts/MasterChef.sol | 50 |

# Analysis 70f085ad-d861-431f-b503-114e34e6886b

MythX

| | |
|---|---|
| Started | Thu Mar 11 2021 23:51:51 GMT+0000 (Coordinated Universal Time) |
| Finished | Fri Mar 12 2021 00:37:33 GMT+0000 (Coordinated Universal Time) |
| Mode | Deep |
| Client Tool | Mythx-Cli-0.6.22 |
| Main Source File | Contracts/MasterChef.Sol |

## DETECTED VULNERABILITIES

| HIGH | MEDIUM | LOW |
|---|---|---|
| 0 | 23 | 27 |

## ISSUES

MEDIUM

SWC-000

### Function could be marked as external.

The function definition of "add" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

contracts/MasterChef.sol

Locations

```

// Add a new lp to the pool. Can only be called by the owner.
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner nonDuplicated(_lpToken) {
require(_depositFeeBP <= 10000, "add: invalid deposit fee basis points");
if (_withUpdate) {
massUpdatePools();
}
uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
totalAllocPoint = totalAllocPoint.add(_allocPoint);
poolExistence[_lpToken] = true;
poolInfo.push(PoolInfo({
lpToken : _lpToken,
allocPoint : _allocPoint,
lastRewardBlock : lastRewardBlock,
accEggPerShare : 0,
depositFeeBP : _depositFeeBP
}));
}

// Update the given pool's EGG allocation point and deposit fee. Can only be called by the owner.
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "set" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/MasterChef.sol`

Locations

```

// Update the given pool's EGG allocation point and deposit fee. Can only be called by the owner.
function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {
require(_depositFeeBP <= 10000, "set: invalid deposit fee basis points");
if (_withUpdate) {
massUpdatePools();
}
totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
poolInfo[_pid].allocPoint = _allocPoint;
poolInfo[_pid].depositFeeBP = _depositFeeBP;
}

// Return reward multiplier over the given _from to _to block.
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "deposit" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/MasterChef.sol`

Locations

```

// Deposit LP tokens to MasterChef for EGG allocation.
function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
updatePool(_pid);
if (user.amount > 0) {
uint256 pending = user.amount.mul(pool.accEggPerShare).div(1e12).sub(user.rewardDebt);
if (pending > 0) {
safeEggTransfer(msg.sender, pending);
}
}
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if (pool.depositFeeBP > 0) {
uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
pool.lpToken.safeTransfer(feeAddress, depositFee);
user.amount = user.amount.add(_amount).sub(depositFee);
} else {
user.amount = user.amount.add(_amount);
}
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}

// Withdraw LP tokens from MasterChef.
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "withdraw" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/MasterChef.sol`

Locations

```

// Withdraw LP tokens from MasterChef.
function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
require(user.amount >= _amount, "withdraw: not good");
updatePool(_pid);
uint256 pending = user.amount.mul(pool.accEggPerShare).div(1e12).sub(user.rewardDebt);
if (pending > 0) {
safeEggTransfer(msg.sender, pending);
}
if (_amount > 0) {
user.amount = user.amount.sub(_amount);
pool.lpToken.safeTransfer(address(msg.sender), _amount);
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Withdraw(msg.sender, _pid, _amount);
}

// Withdraw without caring about rewards. EMERGENCY ONLY.
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "emergencyWithdraw" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/MasterChef.sol`

Locations

```

// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
uint256 amount = user.amount;
user.amount = 0;
user.rewardDebt = 0;
pool.lpToken.safeTransfer(address(msg.sender), amount);
emit EmergencyWithdraw(msg.sender, _pid, amount);
}

// Safe rupee transfer function, just in case if rounding error causes pool to not have enough EGGs.
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "dev" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/MasterChef.sol`

Locations

```

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

function setFeeAddress(address _feeAddress) public {
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "setFeeAddress" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/MasterChef.sol`

Locations

```
}

function setFeeAddress(address _feeAddress) public {
require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
feeAddress = _feeAddress;
emit SetFeeAddress(msg.sender, _feeAddress);
}

//Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "updateEmissionRate" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/MasterChef.sol`

Locations

```

//Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
function updateEmissionRate(uint256 _rupeePerBlock) public onlyOwner {
massUpdatePools();
rupeePerBlock = _rupeePerBlock;
emit UpdateEmissionRate(msg.sender, _rupeePerBlock);
}
}
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "mint" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

contracts/RupeeToken.sol

Locations

```
contract RupeeToken is BEP20('Rupee Token', 'RUPEE') {
/// @dev Creates `_amount` token to `_to`. Must only be called by the owner (Link).
function mint(address _to, uint256 _amount) public onlyOwner {
_mint(_to, _amount);
_moveDelegates(address(0), _delegates[_to], _amount);
}

// Copied and modified from YAM code:
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "symbol" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

contracts/libs/BEP20.sol

Locations

```
* name.
*/
function symbol() public override view returns (string memory) {
return _symbol;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "decimals" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

contracts/libs/BEP20.sol

Locations

```
* @dev Returns the number of decimals used to get its user representation.
*/
function decimals() public override view returns (uint8) {
return _decimals;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "totalSupply" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/libs/BEP20.sol`

Locations

```
* @dev See {BEP20-totalSupply}.
*/
function totalSupply() public override view returns (uint256) {
return _totalSupply;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "transfer" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/libs/BEP20.sol`

Locations

```
* - the caller must have a balance of at least `amount`.
*/
function transfer(address recipient, uint256 amount) public override returns (bool) {
_transfer(_msgSender(), recipient, amount);
return true;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "allowance" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/libs/BEP20.sol`

Locations

```
* @dev See {BEP20-allowance}.
*/
function allowance(address owner, address spender) public override view returns (uint256) {
return _allowances[owner][spender];
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "approve" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/libs/BEP20.sol`

Locations

```
* - `spender` cannot be the zero address.
*/
function approve(address spender, uint256 amount) public override returns (bool) {
_approve(_msgSender(), spender, amount);
return true;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "transferFrom" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/libs/BEP20.sol`

Locations

```
* `amount`.
*/
function transferFrom (address sender, address recipient, uint256 amount) public override returns (bool) {
_transfer(sender, recipient, amount);
_approve(
sender,
_msgSender(),
_allowances[sender][_msgSender()].sub(amount, 'BEP20: transfer amount exceeds allowance')
);
return true;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "increaseAllowance" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/libs/BEP20.sol`

Locations

```
* - `spender` cannot be the zero address.
*/
function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
_approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
return true;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "decreaseAllowance" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/libs/BEP20.sol`

Locations

```
* `subtractedValue`.
*/
function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
_approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, 'BEP20: decreased allowance below zero'));
return true;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "mint" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`contracts/libs/BEP20.sol`

Locations

```
* - `msg.sender` must be the token owner
*/
function mint(uint256 amount) public onlyOwner returns (bool) {
_mint(_msgSender(), amount);
return true;
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "renounceOwnership" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`node_modules/@openzeppelin/contracts/access/Ownable.sol`

Locations

```
* thereby removing any functionality that is only available to the owner.
*/
function renounceOwnership() public virtual onlyOwner {
emit OwnershipTransferred(_owner, address(0));
_owner = address(0);
}

/**
```

MEDIUM

SWC-000

## Function could be marked as external.

The function definition of "transferOwnership" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`node_modules/@openzeppelin/contracts/access/Ownable.sol`

Locations

```
* Can only be called by the current owner.
*/
function transferOwnership(address newOwner) public virtual onlyOwner {
require(newOwner != address(0), "Ownable: new owner is the zero address");
emit OwnershipTransferred(_owner, newOwner);
_owner = newOwner;
}
}
```

MEDIUM

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

`node_modules/@openzeppelin/contracts/utils/Address.sol`

Locations

```

// solhint-disable-next-line avoid-low-level-calls
(bool success, bytes memory returndata) = target.call{ value: value }(data);
return _verifyCallResult(success, returndata, errorMessage);
}
```

MEDIUM

SWC-128

## Loop over unbounded data structure.

Gas consumption in function "massUpdatePools" in contract "MasterChef" depends on the size of data structures or values that may grow unboundedly. If the data structure grows too large, the gas required to execute the code will exceed the block gas limit, effectively causing a denial-of-service condition. Consider that an attacker might attempt to cause this condition on purpose.

Source file

`contracts/MasterChef.sol`

Locations

```
function massUpdatePools() public {
uint256 length = poolInfo.length;
for (uint256 pid = 0; pid < length; ++pid) {
updatePool(pid);
}
```

LOW

SWC-107

## Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if (pool.depositFeeBP > 0) {
uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
pool.lpToken.safeTransfer(feeAddress, depositFee);
```

LOW

SWC-107

## Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
user.amount = user.amount.add(_amount).sub(depositFee);
} else {
user.amount = user.amount.add(_amount);
}
}
```

LOW

SWC-107

## Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
user.amount = user.amount.add(_amount).sub(depositFee);
} else {
user.amount = user.amount.add(_amount);
}
}
```

LOW

SWC-107

## Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
}
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}
```

LOW

SWC-107

## Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
}
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}
```

LOW

SWC-107

## Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
}
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}
```

LOW

SWC-107

### Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
186 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
187 if (pool.depositFeeBP > 0) {
188 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
189 pool.lpToken.safeTransfer(feeAddress, depositFee);
190 user.amount = user.amount.add(_amount).sub(depositFee);
```

LOW

SWC-107

### Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
187 if (pool.depositFeeBP > 0) {
188 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
189 pool.lpToken.safeTransfer(feeAddress, depositFee);
190 user.amount = user.amount.add(_amount).sub(depositFee);
191 } else {
```

LOW

SWC-107

### Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
187 if (pool.depositFeeBP > 0) {
188 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
189 pool.lpToken.safeTransfer(feeAddress, depositFee);
190 user.amount = user.amount.add(_amount).sub(depositFee);
191 } else {
```

LOW

SWC-107

## Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`node_modules/@openzeppelin/contracts/utils/Address.sol`

Locations

```
113 */
114 function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage) internal returns (bytes memory) {
115 require(address(this).balance >= value, "Address: insufficient balance for call");
116 require(isContract(target), "Address: call to non-contract");
```

LOW

SWC-107

## Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
188 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
189 pool.lpToken.safeTransfer(feeAddress, depositFee);
190 user.amount = user.amount.add(_amount).sub(depositFee);
191 } else {
192 user.amount = user.amount.add(_amount);
```

LOW

SWC-107

## Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
188 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
189 pool.lpToken.safeTransfer(feeAddress, depositFee);
190 user.amount = user.amount.add(_amount).sub(depositFee);
191 } else {
192 user.amount = user.amount.add(_amount);
```

LOW

SWC-107

### Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
pool.lpToken.safeTransfer(address(msg.sender), _amount);
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Withdraw(msg.sender, _pid, _amount);
}
```

LOW

SWC-107

### Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
pool.lpToken.safeTransfer(address(msg.sender), _amount);
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Withdraw(msg.sender, _pid, _amount);
}
```

LOW

SWC-107

### Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`contracts/MasterChef.sol`

Locations

```
pool.lpToken.safeTransfer(address(msg.sender), _amount);
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Withdraw(msg.sender, _pid, _amount);
}
```

LOW

SWC-107

### Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

`node_modules/@openzeppelin/contracts/utils/ReentrancyGuard.sol`

Locations

```
58  // By storing the original value once again, a refund is triggered (see
59  // https://eips.ethereum.org/EIPS/eip-2200)
60  _status = _NOT_ENTERED;
61  }
62  }
```

LOW

SWC-120

### Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

`contracts/MasterChef.sol`

Locations

```
103  massUpdatePools();
104  }
105  uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
106  totalAllocPoint = totalAllocPoint.add(_allocPoint);
107  poolExistence[_lpToken] = true;
```

LOW

SWC-120

### Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

`contracts/MasterChef.sol`

Locations

```
103  massUpdatePools();
104  }
105  uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
106  totalAllocPoint = totalAllocPoint.add(_allocPoint);
107  poolExistence[_lpToken] = true;
```

LOW

SWC-120

## Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

`contracts/MasterChef.sol`

Locations

```
uint256 accEggPerShare = pool.accEggPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 rupeeReward = multiplier.mul(rupeePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
```

LOW

SWC-120

## Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

`contracts/MasterChef.sol`

Locations

```
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 rupeeReward = multiplier.mul(rupeePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
accEggPerShare = accEggPerShare.add(rupeeReward.mul(1e12).div(lpSupply));
```

LOW

SWC-120

## Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

`contracts/MasterChef.sol`

Locations

```
function updatePool(uint256 _pid) public {
PoolInfo storage pool = poolInfo[_pid];
if (block.number <= pool.lastRewardBlock) {
return;
}
```

LOW

SWC-120

## Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

contracts/MasterChef.sol

Locations

```
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (lpSupply == 0 || pool.allocPoint == 0) {
pool.lastRewardBlock = block.number;
return;
}
```

LOW

SWC-120

## Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

contracts/MasterChef.sol

Locations

```
return;
}
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 rupeeReward = multiplier.mul(rupeePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
rupee.mint(devaddr, rupeeReward.div(10));
```

LOW

SWC-120

## Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

contracts/MasterChef.sol

Locations

```
rupee.mint(address(this), rupeeReward);
pool.accEggPerShare = pool.accEggPerShare.add(rupeeReward.mul(1e12).div(lpSupply));
pool.lastRewardBlock = block.number;
}
```

LOW

SWC-120

### Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

`contracts/RupeeToken.sol`

Locations

```
returns (uint256)

{

require(blockNumber < block.number, "TOKEN::getPriorVotes: not yet determined");


uint32 nCheckpoints = numCheckpoints[account];
```

LOW

SWC-120

### Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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

`contracts/RupeeToken.sol`

Locations

```
internal

{

uint32 blockNumber = safe32(block.number, "TOKEN::_writeCheckpoint: block number exceeds 32 bits");


if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
```

LOW

SWC-123

## 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).

Source file

`contracts/MasterChef.sol`

Locations

```
return;
}
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (lpSupply == 0 || pool.allocPoint == 0) {
pool.lastRewardBlock = block.number;
```

Source file

`contracts/MasterChef.sol`

Locations

```
//
// Have fun reading it. Hopefully it's bug-free. God bless.
contract MasterChef is Ownable, ReentrancyGuard {
using SafeMath for uint256;
using SafeBEP20 for IBEP20;

// Info of each user.
struct UserInfo {
uint256 amount; // How many LP tokens the user has provided.
uint256 rewardDebt; // Reward debt. See explanation below.
//
// We do some fancy math here. Basically, any point in time, the amount of EGGs
// entitled to a user but is pending to be distributed is:
//
// pending reward = (user.amount * pool.accEggPerShare) - user.rewardDebt
//
// Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
// 1. The pool's `accEggPerShare` (and `lastRewardBlock`) gets updated.
// 2. User receives the pending reward sent to his/her address.
// 3. User's `amount` gets updated.
// 4. User's `rewardDebt` gets updated.
}

// Info of each pool.
struct PoolInfo {
IBEP20 lpToken; // Address of LP token contract.
uint256 allocPoint; // How many allocation points assigned to this pool. EGGs to distribute per block.
uint256 lastRewardBlock; // Last block number that EGGs distribution occurs.
uint256 accEggPerShare; // Accumulated EGGs per share, times 1e12. See below.
uint16 depositFeeBP; // Deposit fee in basis points
}

// The RUPEE TOKEN!
RupeeToken public rupee;
// Dev address.
address public devaddr;
// RUPEE tokens created per block.
uint256 public rupeePerBlock;
// Bonus muliplier for early rupee makers.
uint256 public constant BONUS_MULTIPLIER = 1;
// Deposit Fee address
address public feeAddress;

// Info of each pool.
PoolInfo[] public poolInfo;
```

```
// Info of each user that stakes LP tokens.
mapping(uint256 => mapping(address => UserInfo)) public userInfo;
// Total allocation points. Must be the sum of all allocation points in all pools.
uint256 public totalAllocPoint = 0;
// The block number when EGG mining starts.
uint256 public startBlock;

event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);
event SetFeeAddress(address indexed user, address indexed newAddress);
event SetDevAddress(address indexed user, address indexed newAddress);
event UpdateEmissionRate(address indexed user, uint256 goosePerBlock);

constructor(
RupeeToken _rupee,
address _devaddr,
address _feeAddress,
uint256 _rupeePerBlock,
uint256 _startBlock
) public {
rupee = _rupee;
devaddr = _devaddr;
feeAddress = _feeAddress;
rupeePerBlock = _rupeePerBlock;
startBlock = _startBlock;
}

function poolLength() external view returns (uint256) {
return poolInfo.length;
}

mapping(IBEP20 => bool) public poolExistence;
modifier nonDuplicated(IBEP20 _lpToken) {
require(poolExistence[_lpToken] == false, "nonDuplicated: duplicated");
_;
}

// Add a new lp to the pool. Can only be called by the owner.
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner nonDuplicated(_lpToken) {
require(_depositFeeBP <= 10000, "add: invalid deposit fee basis points");
if (_withUpdate) {
massUpdatePools();
}
uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
totalAllocPoint = totalAllocPoint.add(_allocPoint);
poolExistence[_lpToken] = true;
poolInfo.push(PoolInfo({
lpToken : _lpToken,
allocPoint : _allocPoint,
lastRewardBlock : lastRewardBlock,
accEggPerShare : 0,
depositFeeBP : _depositFeeBP
}));
}

// Update the given pool's EGG allocation point and deposit fee. Can only be called by the owner.
function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {
require(_depositFeeBP <= 10000, "set: invalid deposit fee basis points");
if (_withUpdate) {
massUpdatePools();
}
totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
```

```
poolInfo[_pid].allocPoint = _allocPoint;
poolInfo[_pid].depositFeeBP = _depositFeeBP;
}

// Return reward multiplier over the given _from to _to block.
function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
return _to.sub(_from).mul(BONUS_MULTIPLIER);
}

// View function to see pending EGGs on frontend.
function pendingEgg(uint256 _pid, address _user) external view returns (uint256) {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][_user];
uint256 accEggPerShare = pool.accEggPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 rupeeReward = multiplier.mul(rupeePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
accEggPerShare = accEggPerShare.add(rupeeReward.mul(1e12).div(lpSupply));
}
return user.amount.mul(accEggPerShare).div(1e12).sub(user.rewardDebt);
}

// Update reward variables for all pools. Be careful of gas spending!
function massUpdatePools() public {
uint256 length = poolInfo.length;
for (uint256 pid = 0; pid < length; ++pid) {
updatePool(pid);
}
}

// Update reward variables of the given pool to be up-to-date.
function updatePool(uint256 _pid) public {
PoolInfo storage pool = poolInfo[_pid];
if (block.number <= pool.lastRewardBlock) {
return;
}
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (lpSupply == 0 || pool.allocPoint == 0) {
pool.lastRewardBlock = block.number;
return;
}
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 rupeeReward = multiplier.mul(rupeePerBlock).mul(pool.allocPoint).div(totalAllocPoint);
rupee.mint(devaddr, rupeeReward.div(10));
rupee.mint(address(this), rupeeReward);
pool.accEggPerShare = pool.accEggPerShare.add(rupeeReward.mul(1e12).div(lpSupply));
pool.lastRewardBlock = block.number;
}

// Deposit LP tokens to MasterChef for EGG allocation.
function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
updatePool(_pid);
if (user.amount > 0) {
uint256 pending = user.amount.mul(pool.accEggPerShare).div(1e12).sub(user.rewardDebt);
if (pending > 0) {
safeEggTransfer(msg.sender, pending);
}
}
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if (pool.depositFeeBP > 0) {
uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
pool.lpToken.safeTransfer(feeAddress, depositFee);
user.amount = user.amount.add(_amount).sub(depositFee);
} else {
user.amount = user.amount.add(_amount);
}
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}

// Withdraw LP tokens from MasterChef.
function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
require(user.amount >= _amount, "withdraw: not good");
updatePool(_pid);
uint256 pending = user.amount.mul(pool.accEggPerShare).div(1e12).sub(user.rewardDebt);
if (pending > 0) {
safeEggTransfer(msg.sender, pending);
}
if (_amount > 0) {
user.amount = user.amount.sub(_amount);
pool.lpToken.safeTransfer(address(msg.sender), _amount);
}
user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
emit Withdraw(msg.sender, _pid, _amount);
}

// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
uint256 amount = user.amount;
user.amount = 0;
user.rewardDebt = 0;
pool.lpToken.safeTransfer(address(msg.sender), amount);
emit EmergencyWithdraw(msg.sender, _pid, amount);
}

// Safe rupee transfer function, just in case if rounding error causes pool to not have enough EGGs.
function safeEggTransfer(address _to, uint256 _amount) internal {
uint256 rupeeBal = rupee.balanceOf(address(this));
bool transferSuccess = false;
if (_amount > rupeeBal) {
transferSuccess = rupee.transfer(_to, rupeeBal);
} else {
transferSuccess = rupee.transfer(_to, _amount);
}
require(transferSuccess, "safeRupeeTransfer: transfer failed");
}

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

function setFeeAddress(address _feeAddress) public {
require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
feeAddress = _feeAddress;
```

```
emit SetFeeAddress(msg.sender, _feeAddress);
}

//Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
function updateEmissionRate(uint256 _rupeePerBlock) public onlyOwner {
massUpdatePools();
rupeePerBlock = _rupeePerBlock;
emit UpdateEmissionRate(msg.sender, _rupeePerBlock);
}
}
```