MythX

REPORT 60F908730A20940019533407

Created	Thu Jul 22 2021 05:56:03 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	60f906b2a6e184dcafc6e947

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
b682a180-4da7-4613-9160-028e17311d16	Masterchef.sol	69

Analysis b682a180-4da7-4613-9160-028e17311d16 MythX

Started	Thu Jul 22 2021 05:56:07 GMT+0000 (Coordinated Universal Time)
Finished	Thu Jul 22 2021 06:41:46 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	Masterchef.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	30	39

ISSUES

MEDIUM Function could be marked as external.

SWC-000

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
Masterchef.sol
Locations

```
638 * thereby removing any functionality that is only available to the owner.
639 */
640 function renounceOwnership() public virtual onlyOwner {
641 emit OwnershipTransferred(_owner, address(0));
642 _owner = address(0);
643 }
644
645 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
647 * Can only be called by the current owner.
648 */
649 function transferOwnership(address newOwner) public virtual onlyOwner {
650 require(newOwner != address(0), "Ownable: new owner is the zero address");
651 emit OwnershipTransferred(_owner, newOwner);
652 _owner = newOwner;
653 }
654 }
655
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
785 * @dev Returns the token decimals.
786 */
787 function decimals() public override view returns (uint8) {
788 return _decimals;
789 }
790
791 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
792 * @dev Returns the token symbol.
793 */
794 function symbol() public override view returns (string memory) {
795 return _symbol;
796 }
797
798 /**
```

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

Masterchef.sol

Locations

```
799 * @dev See {BEP20-totalSupply}.
800 */
801 function totalSupply() public override view returns (uint256) {
802 return _totalSupply;
803 }
804
805 /**
```

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

Masterchef.sol

Locations

```
818 * - the caller must have a balance of at least `amount`.
819 */
820 function transfer(address recipient, uint256 amount) public override returns (bool) {
821 _transfer(_msgSender(), recipient, amount);
822 return true;
823 }
824
825 /**
```

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

Masterchef.sol

Locations

```
826 * @dev See {BEP20-allowance}.
827 */
828 function allowance(address owner, address spender) public override view returns (uint256) {
829 return _allowances[owner][spender];
830 }
831
832 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
837 * - `spender` cannot be the zero address.
838 */
839 function approve(address spender, uint256 amount) public override returns (bool) {
840 _approve(_msgSender(), spender, amount);
841 return true;
842 }
843
844 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
854 * `amount`.
855 */
856 function transferFrom(
857 address sender,
858 address recipient,
859 uint256 amount
860 ) public override returns (bool) {
861 _transfer(sender, recipient, amount);
862 _approve(
863 sender,
864 _msgSender(),
865 _allowances[sender][_msgSender()].sub(amount, 'BEP20: transfer amount exceeds allowance')
866 );
867 return true;
868 }
869
870 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
880 * - `spender` cannot be the zero address.
881 */
882 function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
883 _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
884 return true;
885 }
886
887 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
899 * `subtractedValue`.
900 */
901 function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
902 _approve(
903 _msgSender(),
904 spender,
905 _allowances[_msgSender()][spender].sub(subtractedValue, 'BEP20: decreased allowance below zero')
906 );
907 return true;
908 }
909
910 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
916 * - `msg.sender` must be the token owner
917 */
918 function mint(uint256 amount) public onlyOwner returns (bool) {
919 _mint(_msgSender(), amount);
920 return true;
921 }
922
923 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
1240
1241 /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
1242 function mint(address _to, uint256 _amount) public onlyOwner {
1243 _mint(_to, _amount);
1244 _moveDelegates(address(0), _delegates[_to], _amount);
1245 }
1246
1247 /// @dev overrides transfer function to meet tokenomics of BARON
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "updateTransferTaxRate" 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

Masterchef.sol

Locations

```
1351 * Can only be called by the current operator.
1352 */
1353 function updateTransferTaxRate(uint16 _transferTaxRate) public onlyOperator {
1354 require(_transferTaxRate <= MAXIMUM_TRANSFER_TAX_RATE, "BARON::updateTransferTaxRate: Transfer tax rate must not exceed the maximum rate.");
1355 emit TransferTaxRateUpdated(msg.sender, transferTaxRate, _transferTaxRate);
1356 transferTaxRate = _transferTaxRate;
1357 }
1358
1359 /**
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "updateBurnRate" 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

Masterchef.sol

Locations

```
1361 * Can only be called by the current operator.
1362 */
1363 function updateBurnRate(uint16 _burnRate) public onlyOperator {
1364 require(_burnRate <= 100, "BARON::updateBurnRate: Burn rate must not exceed the maximum rate.");
1365 emit BurnRateUpdated(msg.sender, burnRate, _burnRate);
1366 burnRate = _burnRate;
1367 }
1368
1369
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "updateMinAmountToLiquify" 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

Masterchef.sol

Locations

```
1372 * Can only be called by the current operator.
1373 */
1374 function updateMinAmountToLiquify(uint256 _minAmount) public onlyOperator {
1375 emit MinAmountToLiquifyUpdated(msg.sender, minAmountToLiquify, _minAmount);
1376 minAmountToLiquify = _minAmount;
1377 }
1378
1379 /**
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "updateSwapAndLiquifyEnabled" 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

Masterchef.sol

Locations

```
1381 * Can only be called by the current operator.
1382 */
1383 function updateSwapAndLiquifyEnabled(bool _enabled) public onlyOperator {
1384 emit SwapAndLiquifyEnabledUpdated(msg.sender, _enabled);
1385 swapAndLiquifyEnabled = _enabled;
1386 }
1387
1388 /**
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "updateBaronFarmRouter" 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

Masterchef.sol

Locations

```
1390 * Can only be called by the current operator.
1391 */
1392 function updateBaronFarmRouter(address _router) public onlyOperator {
1393 baronFarmRouter = IBaronfarmRouter02(_router);
1394 baronFarmPair = IBaronFactory(baronFarmRouter.factory()).getPair(address(this), baronFarmRouter.WETH());
1395 require(baronFarmPair != address(0), "BARON::updateBaronFarmRouter: Invalid pair address.");
1396 emit BaronFarmRouterUpdated(msg.sender, address(baronFarmRouter), baronFarmPair);
1397 }
1398
1399 /**
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "transferOperator" 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

Masterchef.sol

Locations

```
1408 * Can only be called by the current operator.
1409 */
1410 function transferOperator(address newOperator) public onlyOperator {
1411 require(newOperator != address(0), "BARON::transferOperator: new operator is the zero address");
1412 emit OperatorTransferred(_operator, newOperator);
1413 _operator = newOperator;
1414 }
1415
1416 // Copied and modified from YAM code:
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
1759 // Add a new lp to the pool. Can only be called by the owner.
1760 // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
1761 function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, uint256 _harvestInterval, bool _withUpdate) public onlyOwner {
1762 require(_depositFeeBP <= 10000, "add: invalid deposit fee basis points");
1763 require(_harvestInterval <= MAXIMUM_HARVEST_INTERVAL, "add: invalid harvest interval");
1764 if (_withUpdate) {
1765 massUpdatePools();
1766 }
1767 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1768 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1769 poolInfo.push(PoolInfo({
1770 lpToken: _lpToken,
1771 allocPoint: _allocPoint,
1772 lastRewardBlock: lastRewardBlock,
1773 accBaronPerShare: 0,
1774 depositFeeBP: _depositFeeBP,
1775 harvestInterval: _harvestInterval
1776 }));
1777 }
1778
1779 // Update the given pool's BARON allocation point and deposit fee. Can only be called by the owner.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
1778
1779 // Update the given pool's BARON allocation point and deposit fee. Can only be called by the owner.
1780 function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, uint256 _harvestInterval, bool _withUpdate) public onlyOwner {
1781 require(_depositFeeBP <= 10000, "set: invalid deposit fee basis points");
1782 require(_harvestInterval <= MAXIMUM_HARVEST_INTERVAL, "set: invalid harvest interval");
1783 if (_withUpdate) {
1784 massUpdatePools();
1785 }
1786 totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
1787 poolInfo[_pid].allocPoint = _allocPoint;
1788 poolInfo[_pid].depositFeeBP = _depositFeeBP;
1789 poolInfo[_pid].harvestInterval = _harvestInterval;
1790 }
1791
1792 // Return reward multiplier over the given _from to _to block.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
1844
1845  // Deposit LP tokens to MasterChef for BARON allocation.
1846  function deposit(uint256 _pid, uint256 _amount, address _referrer) public nonReentrant {
1847  PoolInfo storage pool = poolInfo[_pid];
1848  UserInfo storage user = userInfo[_pid][msg.sender];
1849  updatePool(_pid);
1850  if (_amount > 0 && address(baronReferral) != address(0) && _referrer != address(0) && _referrer != msg.sender) {
1851  baronReferral.recordReferral(msg.sender, _referrer);
1852  }
1853  payOrLockupPendingBaron(_pid);
1854  if (_amount > 0) {
1855  pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1856  if (address(pool.lpToken) == address(baron)) {
1857  uint256 transferTax = _amount.mul(baron.transferTaxRate()).div(10000);
1858  _amount = _amount.sub(transferTax);
1859  }
1860  if (pool.depositFeeBP > 0) {
1861  uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1862  pool.lpToken.safeTransfer(feeAddress, depositFee);
1863  user.amount = user.amount.add(_amount).sub(depositFee);
1864  } else {
1865  user.amount = user.amount.add(_amount);
1866  }
1867  }
1868  user.rewardDebt = user.amount.mul(pool.accBaronPerShare).div(1e12);
1869  emit Deposit(msg.sender, _pid, _amount);
1870  }
1871
1872  // Withdraw LP tokens from MasterChef.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
1871
1872 // Withdraw LP tokens from MasterChef.
1873 function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
1874 PoolInfo storage pool = poolInfo[_pid];
1875 UserInfo storage user = userInfo[_pid][msg.sender];
1876 require(user.amount >= _amount, "withdraw: not good");
1877 updatePool(_pid);
1878 payOrLockupPendingBaron(_pid);
1879 if (_amount > 0) {
1880 user.amount = user.amount.sub(_amount);
1881 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1882 }
1883 user.rewardDebt = user.amount.mul(pool.accBaronPerShare).div(1e12);
1884 emit Withdraw(msg.sender, _pid, _amount);
1885 }
1886
1887 // Withdraw without caring about rewards. EMERGENCY ONLY.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Masterchef.sol

Locations

```
1886
1887 // Withdraw without caring about rewards. EMERGENCY ONLY.
1888 function emergencyWithdraw(uint256 _pid) public nonReentrant {
1889 PoolInfo storage pool = poolInfo[_pid];
1890 UserInfo storage user = userInfo[_pid][msg.sender];
1891 uint256 amount = user.amount;
1892 user.amount = 0;
1893 user.rewardDebt = 0;
1894 user.rewardLockedUp = 0;
1895 user.nextHarvestUntil = 0;
1896 pool.lpToken.safeTransfer(address(msg.sender), amount);
1897 emit EmergencyWithdraw(msg.sender, _pid, amount);
1898 }
1899
1900 // Pay or lockup pending BARON.
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "setDevAddress" 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

Masterchef.sol

Locations

```
1939
1940 // Update dev address by the previous dev.
1941 function setDevAddress(address _devAddress) public {
1942 require(msg.sender == devAddress, "setDevAddress: FORBIDDEN");
1943 require(_devAddress != address(0), "setDevAddress: ZERO");
1944 devAddress = _devAddress;
1945 }
1946
1947 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

Masterchef.sol

Locations

```
1945 }
1946
1947 function setFeeAddress(address _feeAddress) public {
1948 require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
1949 require(_feeAddress != address(0), "setFeeAddress: ZERO");
1950 feeAddress = _feeAddress;
1951 }
1952
1953 // Pancake has to add hidden dummy pools in order 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

Masterchef.sol

Locations

```
1952
1953 // Pancake has to add hidden dummy pools in order to alter the emission, here we make it simple and transparent to all.
1954 function updateEmissionRate(uint256 _baronPerBlock) public onlyOwner {
1955 massUpdatePools();
1956 emit EmissionRateUpdated(msg.sender, baronPerBlock, _baronPerBlock);
1957 baronPerBlock = _baronPerBlock;
1958 }
1959
1960 // Update the BARON referral contract address by the owner
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "setBaronReferral" 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

Masterchef.sol

Locations

```
1959
1960 // Update the BARON referral contract address by the owner
1961 function setBaronReferral(IBaronReferral _baronReferral) public onlyOwner {
1962 baronReferral = _baronReferral;
1963 }
1964
1965 // Update referral commission rate by the owner
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "setReferralCommissionRate" 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

Masterchef.sol

Locations

```
1964
1965 // Update referral commission rate by the owner
1966 function setReferralCommissionRate(uint16 _referralCommissionRate) public onlyOwner {
1967 require(_referralCommissionRate <= MAXIMUM_REFERRAL_COMMISSION_RATE, "setReferralCommissionRate: invalid referral commission rate basis points");
1968 referralCommissionRate = _referralCommissionRate;
1969 }
1970
1971 // Pay referral commission to the referrer who referred this user.
```

MEDIUM

Multiple calls are executed in the same transaction.

SWC-113

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

Masterchef.sol

Locations

```
206
207 // solhint-disable-next-line avoid-low-level-calls
208 (bool success, bytes memory returndata) = target.call{ value: value }(data);
209 return _verifyCallResult(success, returndata, errorMessage);
210 }
```

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

Masterchef.sol

Locations

```
1854 if (_amount > 0) {
1855 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1856 if (address(pool.lpToken) == address(baron)) {
1857 uint256 transferTax = _amount.mul(baron.transferTaxRate()).div(10000);
1858 _amount = _amount.sub(transferTax);
```

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

Masterchef.sol

Locations

```
1854 if (_amount > 0) {
1855 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1856 if (address(pool.lpToken) == address(baron)) {
1857 uint256 transferTax = _amount.mul(baron.transferTaxRate()).div(10000);
1858 _amount = _amount.sub(transferTax);
```

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

Masterchef.sol

Locations

```
1858 _amount = _amount.sub(transferTax);
1859 }
1860 if (pool.depositFeeBP > 0) {
1861 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1862 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

Masterchef.sol

Locations

```
1863 | user.amount = user.amount.add(_amount).sub(depositFee);
1864 | } else {
1865 | user.amount = user.amount.add(_amount);
1866 | }
1867 | }
```

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

Masterchef.sol

Locations

```
1863 | user.amount = user.amount.add(_amount).sub(depositFee);
1864 | } else {
1865 | user.amount = user.amount.add(_amount);
1866 | }
1867 | }
```

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

Masterchef.sol

Locations

```
1866 | }
1867 | }
1868 | user.rewardDebt = user.amount.mul(pool.accBaronPerShare).div(1e12);
1869 | emit Deposit(msg.sender, _pid, _amount);
1870 | }
```

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

Masterchef.sol

Locations

```
1866 }
1867 }
1868 user.rewardDebt = user.amount.mul(pool.accBaronPerShare).div(1e12);
1869 emit Deposit(msg.sender, _pid, _amount);
1870 }
```

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

Masterchef.sol

Locations

```
1866 }
1867 }
1868 user.rewardDebt = user.amount.mul(pool.accBaronPerShare).div(1e12);
1869 emit Deposit(msg.sender, _pid, _amount);
1870 }
```

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

Masterchef.sol

Locations

```
1859 }
1860 if (pool.depositFeeBP > 0) {
1861 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1862 pool.lpToken.safeTransfer(feeAddress, depositFee);
1863 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

Masterchef.sol

Locations

```
1860 if (pool.depositFeeBP > 0) {
1861 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1862 pool.lpToken.safeTransfer(feeAddress, depositFee);
1863 user.amount = user.amount.add(_amount).sub(depositFee);
1864 } 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

Masterchef.sol

Locations

```
1860 if (pool.depositFeeBP > 0) {
1861 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1862 pool.lpToken.safeTransfer(feeAddress, depositFee);
1863 user.amount = user.amount.add(_amount).sub(depositFee);
1864 } 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

Masterchef.sol

Locations

```
202 */
203 function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage) internal returns (bytes memory) {
204 require(address(this).balance >= value, "Address: insufficient balance for call");
205 require(isContract(target), "Address: call to non-contract");
206
```

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

Masterchef.sol

Locations

```
1861 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1862 pool.lpToken.safeTransfer(feeAddress, depositFee);
1863 user.amount = user.amount.add(_amount).sub(depositFee);
1864 } else {
1865 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

Masterchef.sol

Locations

```
1861 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1862 pool.lpToken.safeTransfer(feeAddress, depositFee);
1863 user.amount = user.amount.add(_amount).sub(depositFee);
1864 } else {
1865 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

Masterchef.sol

Locations

```
1881 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1882 }
1883 user.rewardDebt = user.amount.mul(pool.accBaronPerShare).div(1e12);
1884 emit Withdraw(msg.sender, _pid, _amount);
1885 }
```

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

Masterchef.sol

Locations

```
1881 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1882 }
1883 user.rewardDebt = user.amount.mul(pool.accBaronPerShare).div(1e12);
1884 emit Withdraw(msg.sender, _pid, _amount);
1885 }
```

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

Masterchef.sol

Locations

```
1881 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1882 }
1883 user.rewardDebt = user.amount.mul(pool.accBaronPerShare).div(1e12);
1884 emit Withdraw(msg.sender, _pid, _amount);
1885 }
```

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

Masterchef.sol

Locations

```
710 // By storing the original value once again, a refund is triggered (see
711 // https://eips.ethereum.org/EIPS/eip-2200)
712 _status = _NOT_ENTERED;
713 }
714 }
```

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

Masterchef.sol

Locations

```
1900 // Pay or lockup pending BARON.
1901 function payOrLockupPendingBaron(uint256 _pid) internal {
1902 PoolInfo storage pool = poolInfo[_pid];
1903 UserInfo storage user = userInfo[_pid][msg.sender];
1904
```

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

Masterchef.sol

Locations

```
1903 UserInfo storage user = userInfo[_pid][msg.sender];
1904
1905 if (user.nextHarvestUntil == 0) {
1906 user.nextHarvestUntil = block.timestamp.add(pool.harvestInterval);
1907 }
```

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

Masterchef.sol

Locations

```
1904
1905 if (user.nextHarvestUntil == 0) {
1906 user.nextHarvestUntil = block.timestamp.add(pool.harvestInterval);
1907 }
1908
```

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

Masterchef.sol

Locations

```
1904
1905  if (user.nextHarvestUntil == 0) {
1906  user.nextHarvestUntil = block.timestamp.add(pool.harvestInterval);
1907  }
1908
```

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

Masterchef.sol

Locations

```
1907  }
1908
1909  uint256 pending = user.amount.mul(pool.accBaronPerShare).div(1e12).sub(user.rewardDebt);
1910  if (canHarvest(_pid, msg.sender)) {
1911  if (pending > 0 || user.rewardLockedUp > 0) {
```

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

Masterchef.sol

Locations

```
1907  }
1908
1909  uint256 pending = user.amount.mul(pool.accBaronPerShare).div(1e12).sub(user.rewardDebt);
1910  if (canHarvest(_pid, msg.sender)) {
1911  if (pending > 0 || user.rewardLockedUp > 0) {
```

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

Masterchef.sol

Locations

```
1907 }
1908
1909 uint256 pending = user.amount.mul(pool.accBaronPerShare).div(1e12).sub(user.rewardDebt);
1910 if (canHarvest(_pid, msg.sender)) {
1911 if (pending > 0 || user.rewardLockedUp > 0) {
```

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

Masterchef.sol

Locations

```
1813 function canHarvest(uint256 _pid, address _user) public view returns (bool) {
1814 UserInfo storage user = userInfo[_pid][_user];
1815 return block.timestamp >= user.nextHarvestUntil;
1816 }
1817
```

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

Masterchef.sol

Locations

```
1853 payOrLockupPendingBaron(_pid);
1854 if (_amount > 0) {
1855 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1856 if (address(pool.lpToken) == address(baron)) {
1857 uint256 transferTax = _amount.mul(baron.transferTaxRate()).div(10000);
```

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

Masterchef.sol

Locations

```
1909 uint256 pending = user.amount.mul(pool.accBaronPerShare).div(1e12).sub(user.rewardDebt);
1910 if (canHarvest(_pid, msg.sender)) {
1911 if (pending > 0 || user.rewardLockedUp > 0) {
1912 uint256 totalRewards = pending.add(user.rewardLockedUp);
1913
```

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

Masterchef.sol

Locations

```
1548 returns (uint256)
1549 {
1550 require(blockNumber < block.number, "BARON::getPriorVotes: not yet determined");
1551
1552 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

Masterchef.sol

Locations

```
1621 internal
1622 {
1623 uint32 blockNumber = safe32(block.number, "BARON::_writeCheckpoint: block number exceeds 32 bits");
1624
1625 if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
```

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

Masterchef.sol

Locations

```
1765 massUpdatePools();
1766 }
1767 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1768 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1769 poolInfo.push(PoolInfo({
```

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

Masterchef.sol

Locations

```
1765 massUpdatePools();
1766 }
1767 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1768 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1769 poolInfo.push(PoolInfo({
```

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

Masterchef.sol

Locations

```
1801 uint256 accBaronPerShare = pool.accBaronPerShare;
1802 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1803 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1804 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1805 uint256 baronReward = multiplier.mul(baronPerBlock).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

Masterchef.sol

Locations

```
1802 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1803 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1804 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1805 uint256 baronReward = multiplier.mul(baronPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1806 accBaronPerShare = accBaronPerShare.add(baronReward.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

Masterchef.sol

Locations

```
1827 function updatePool(uint256 _pid) public {
1828 PoolInfo storage pool = poolInfo[_pid];
1829 if (block.number <= pool.lastRewardBlock) {
1830 return;
1831 }
```

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

Masterchef.sol

Locations

```
1832 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1833 if (lpSupply == 0 || pool.allocPoint == 0) {
1834 pool.lastRewardBlock = block.number;
1835 return;
1836 }
```

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

Masterchef.sol

Locations

```
1835 return;
1836 }
1837 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1838 uint256 baronReward = multiplier.mul(baronPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1839 baron.mint(devAddress, baronReward.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

Masterchef.sol

Locations

```
1840 baron.mint(address(this), baronReward);
1841 pool.accBaronPerShare = pool.accBaronPerShare.add(baronReward.mul(1e12).div(lpSupply));
1842 pool.lastRewardBlock = block.number;
1843 }
1844
```

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

Masterchef.sol

Locations

```
1830 return;
1831 }
1832 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1833 if (lpSupply == 0 || pool.allocPoint == 0) {
1834 pool.lastRewardBlock = block.number;
```

Source file

Masterchef.sol

Locations

```
1669 //
1670 // Have fun reading it. Hopefully it's bug-free. God bless.
1671 contract MasterChef is Ownable, ReentrancyGuard {
1672 using SafeMath for uint256;
1673 using SafeBEP20 for IBEP20;
1674
1675 // Info of each user.
1676 struct UserInfo {
1677 uint256 amount; // How many LP tokens the user has provided.
1678 uint256 rewardDebt; // Reward debt. See explanation below.
1679 uint256 rewardLockedUp; // Reward locked up.
1680 uint256 nextHarvestUntil; // When can the user harvest again.
1681 //
1682 // We do some fancy math here. Basically, any point in time, the amount of BARON
1683 // entitled to a user but is pending to be distributed is:
1684 //
1685 // pending reward = (user.amount * pool.accBaronPerShare) - user.rewardDebt
1686 //
1687 // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
1688 // 1. The pool's `accBaronPerShare` (and `lastRewardBlock`) gets updated.
1689 // 2. User receives the pending reward sent to his/her address.
1690 // 3. User's `amount` gets updated.
1691 // 4. User's `rewardDebt` gets updated.
1692 }
1693
1694 // Info of each pool.
1695 struct PoolInfo {
1696 IBEP20 lpToken; // Address of LP token contract.
1697 uint256 allocPoint; // How many allocation points assigned to this pool. Baron to distribute per block.
1698 uint256 lastRewardBlock; // Last block number that Baron distribution occurs.
1699 uint256 accBaronPerShare; // Accumulated Baron per share, times 1e12. See below.
1700 uint16 depositFeeBP; // Deposit fee in basis points
1701 uint256 harvestInterval; // Harvest interval in seconds
1702 }
1703
1704 // The BARON TOKEN!
1705 BaronToken public baron;
1706 // Dev address.
1707 address public devAddress;
1708 // Deposit Fee address
1709 address public feeAddress;
1710 // Baron tokens created per block.
1711 uint256 public baronPerBlock;
1712 // Bonus muliplier for early baron makers.
1713 uint256 public constant BONUS_MULTIPLIER = 1;
```

```
// Max harvest interval: 14 days.
uint256 public constant MAXIMUM_HARVEST_INTERVAL = 14 days;

// 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 BARON mining starts.
uint256 public startBlock;
// Total locked up rewards
uint256 public totalLockedUpRewards;

// Baron referral contract address.
IBaronReferral public baronReferral;
// Referral commission rate in basis points.
uint16 public referralCommissionRate = 300;
// Max referral commission rate: 10%.
uint16 public constant MAXIMUM_REFERRAL_COMMISSION_RATE = 1000;

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 EmissionRateUpdated(address indexed caller, uint256 previousAmount, uint256 newAmount);
event ReferralCommissionPaid(address indexed user, address indexed referrer, uint256 commissionAmount);
event RewardLockedUp(address indexed user, uint256 indexed pid, uint256 amountLockedUp);

constructor(
BaronToken _baron,
uint256 _startBlock,
uint256 _baronPerBlock
) public {
baron = _baron;
startBlock = _startBlock;
baronPerBlock = _baronPerBlock;

devAddress = msg.sender;
feeAddress = msg.sender;
}

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

// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, uint256 _harvestInterval, bool _withUpdate) public onlyOwner {
require(_depositFeeBP <= 10000, "add: invalid deposit fee basis points");
require(_harvestInterval <= MAXIMUM_HARVEST_INTERVAL, "add: invalid harvest interval");
if (_withUpdate) {
massUpdatePools();
}
uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
totalAllocPoint = totalAllocPoint.add(_allocPoint);
poolInfo.push(PoolInfo({
lpToken: _lpToken,
allocPoint: _allocPoint,
lastRewardBlock: lastRewardBlock,
accBaronPerShare: 0,
depositFeeBP: _depositFeeBP,
harvestInterval: _harvestInterval
}));
```

```
}

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

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

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

// View function to see if user can harvest BECI.
function canHarvest(uint256 _pid, address _user) public view returns (bool) {
UserInfo storage user = userInfo[_pid][_user];
return block.timestamp >= user.nextHarvestUntil;
}

// 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 baronReward = multiplier.mul(baronPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
baron.mint(devAddress, baronReward.div(10));
```

```
baron.mint(address(this), baronReward);
pool.accBaronPerShare = pool.accBaronPerShare.add(baronReward.mul(1e12).div(lpSupply));
pool.lastRewardBlock = block.number;
}

// Deposit LP tokens to MasterChef for BARON allocation.
function deposit(uint256 _pid, uint256 _amount, address _referrer) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
updatePool(_pid);
if (_amount > 0 && address(baronReferral) != address(0) && _referrer != address(0) && _referrer != msg.sender) {
baronReferral.recordReferral(msg.sender, _referrer);
}
payOrLockupPendingBaron(_pid);
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if (address(pool.lpToken) == address(baron)) {
uint256 transferTax = _amount.mul(baron.transferTaxRate()).div(10000);
_amount = _amount.sub(transferTax);
}
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.accBaronPerShare).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);
payOrLockupPendingBaron(_pid);
if (_amount > 0) {
user.amount = user.amount.sub(_amount);
pool.lpToken.safeTransfer(address(msg.sender), _amount);
}
user.rewardDebt = user.amount.mul(pool.accBaronPerShare).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;
user.rewardLockedUp = 0;
user.nextHarvestUntil = 0;
pool.lpToken.safeTransfer(address(msg.sender), amount);
emit EmergencyWithdraw(msg.sender, _pid, amount);
}

// Pay or lockup pending BARON.
function payOrLockupPendingBaron(uint256 _pid) internal {
PoolInfo storage pool = poolInfo[_pid];
```

```
UserInfo storage user = userInfo[_pid][msg.sender];

if (user.nextHarvestUntil == 0) {
user.nextHarvestUntil = block.timestamp.add(pool.harvestInterval);
}

uint256 pending = user.amount.mul(pool.accBaronPerShare).div(1e12).sub(user.rewardDebt);
if (canHarvest(_pid, msg.sender)) {
if (pending > 0 || user.rewardLockedUp > 0) {
uint256 totalRewards = pending.add(user.rewardLockedUp);

// reset lockup
totalLockedUpRewards = totalLockedUpRewards.sub(user.rewardLockedUp);
user.rewardLockedUp = 0;
user.nextHarvestUntil = block.timestamp.add(pool.harvestInterval);

// send rewards
safeBaronTransfer(msg.sender, totalRewards);
payReferralCommission(msg.sender, totalRewards);
}
} else if (pending > 0) {
user.rewardLockedUp = user.rewardLockedUp.add(pending);
totalLockedUpRewards = totalLockedUpRewards.add(pending);
emit RewardLockedUp(msg.sender, _pid, pending);
}
}

// Safe BARON transfer function, just in case if rounding error causes pool to not have enough BARON.
function safeBaronTransfer(address _to, uint256 _amount) internal {
uint256 baronBal = baron.balanceOf(address(this));
if (_amount > baronBal) {
baron.transfer(_to, baronBal);
} else {
baron.transfer(_to, _amount);
}
}

// Update dev address by the previous dev.
function setDevAddress(address _devAddress) public {
require(msg.sender == devAddress, "setDevAddress: FORBIDDEN");
require(_devAddress != address(0), "setDevAddress: ZERO");
devAddress = _devAddress;
}

function setFeeAddress(address _feeAddress) public {
require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
require(_feeAddress != address(0), "setFeeAddress: ZERO");
feeAddress = _feeAddress;
}

// Pancake has to add hidden dummy pools in order to alter the emission, here we make it simple and transparent to all.
function updateEmissionRate(uint256 _baronPerBlock) public onlyOwner {
massUpdatePools();
emit EmissionRateUpdated(msg.sender, baronPerBlock, _baronPerBlock);
baronPerBlock = _baronPerBlock;
}

// Update the BARON referral contract address by the owner
function setBaronReferral(IBaronReferral _baronReferral) public onlyOwner {
baronReferral = _baronReferral;
}

// Update referral commission rate by the owner
```

```
function setReferralCommissionRate(uint16 _referralCommissionRate) public onlyOwner {
require(_referralCommissionRate <= MAXIMUM_REFERRAL_COMMISSION_RATE, "setReferralCommissionRate: invalid referral commission rate basis points");
referralCommissionRate = _referralCommissionRate;
}

// Pay referral commission to the referrer who referred this user.
function payReferralCommission(address _user, uint256 _pending) internal {
if (address(baronReferral) != address(0) && referralCommissionRate > 0) {
address referrer = baronReferral.getReferrer(_user);
uint256 commissionAmount = _pending.mul(referralCommissionRate).div(10000);

if (referrer != address(0) && commissionAmount > 0) {
baron.mint(referrer, commissionAmount);
baronReferral.recordReferralCommission(referrer, commissionAmount);
emit ReferralCommissionPaid(_user, referrer, commissionAmount);
}
}
}
}
```