MythX

Analysis 67ad076c-cf9a-444a-aa4f-a1f9ce04ec10 MythX

Started	Wed Apr 14 2021 02:16:30 GMT+0000 (Coordinated Universal Time)
Finished	Wed Apr 14 2021 02:33:17 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Remythx
Main Source File	MasterChef.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	26	44

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

```
552 * thereby removing any functionality that is only available to the owner.
553 */
554 function renounceOwnership() public virtual onlyOwner {
555 emit OwnershipTransferred(_owner, address(0));
556 _owner = address(0);
557 }
558
559 /**
```

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

```
561 * Can only be called by the current owner.
562 */
563 function transferOwnership(address newOwner) public virtual onlyOwner {
564 require(newOwner != address(0), "Ownable: new owner is the zero address");
565 emit OwnershipTransferred(_owner, newOwner);
566 _owner = newOwner;
567 }
568 }
569
```

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

```
639 * name.
640 */
641 function symbol() public override view returns (string memory) {
642 return _symbol;
643 }
644
645 /**
```

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

```
646 * @dev Returns the number of decimals used to get its user representation.
647 */
648 function decimals() public override view returns (uint8) {
649 return _decimals;
650 }
651
652 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

```
653 * @dev See {BEP20-totalSupply}.
654 */
655 function totalSupply() public override view returns (uint256) {
656 return _totalSupply;
657 }
658
659 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

```
672 * - the caller must have a balance of at least `amount`.
673 */
674 function transfer(address recipient, uint256 amount) public override returns (bool) {
675 _transfer(_msgSender(), recipient, amount);
676 return true;
677 }
678
679 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

```
680 * @dev See {BEP20-allowance}.
681 */
682 function allowance(address owner, address spender) public override view returns (uint256) {
683 return _allowances[owner][spender];
684 }
685
686 /**
```

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

```
691 * - `spender` cannot be the zero address.
692 */
693 function approve(address spender, uint256 amount) public override returns (bool) {
694 _approve(_msgSender(), spender, amount);
695 return true;
696 }
697
698 /**
```

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

```
708 * `amount`.
709 */
710 function transferFrom (address sender, address recipient, uint256 amount) public override returns (bool) {
711 _transfer(sender, recipient, amount);
712 _approve(
713 sender,
714 _msgSender(),
715 _allowances[sender][_msgSender()].sub(amount, 'BEP20: transfer amount exceeds allowance')
716 );
717 return true;
718 }
719
720 /**
```

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

```
730 * - `spender` cannot be the zero address.
731 */
732 function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
733 _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
734 return true;
735 }
736
737 /**
```

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

```
749 * `subtractedValue`.
750 */
751 function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
752 _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, 'BEP20: decreased allowance below zero'));
753 return true;
754 }
755
756 /**
```

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

```
762 * - `msg.sender` must be the token owner
763 */
764 function mint(uint256 amount) public onlyOwner returns (bool) {
765 _mint(_msgSender(), amount);
766 return true;
767 }
768
769 /**
```

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

```
925 contract BallToken is BEP20('Ball Token', 'BALL') {
926 /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
927 function mint(address _to, uint256 _amount) public onlyOwner {
928 _mint(_to, _amount);
929 _moveDelegates(address(0), _delegates[_to], _amount);
930 }
931
932 // 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

```
1285
1286 // Add a new lp to the pool. Can only be called by the owner.
1287 function add(
1288 uint256 _allocPoint,
1289 IBEP20 _lpToken,
1290 uint16 _depositFeeBP,
1291 bool _withUpdate
1292 ) public onlyOwner nonDuplicated(_lpToken) {
1293 require(_depositFeeBP <= MAXIMUM_DEPOSIT_FEE_BP, "add: invalid deposit fee basis points");
1294 if (_withUpdate) {
1295 massUpdatePools();
1296 }
1297 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1298 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1299 poolExistence[_lpToken] = true;
1300 poolInfo.push(
1301 PoolInfo({
1302 lpToken: _lpToken,
1303 allocPoint: _allocPoint,
1304 lastRewardBlock: lastRewardBlock,
1305 accBallPerShare: 0,
1306 depositFeeBP: _depositFeeBP
1307 })
1308 );
1309 poolIdForLpAddress[_lpToken] = poolInfo.length - 1;
1310 }
1311
1312 // Update the given pool's BALL 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

```
1311
1312 // Update the given pool's BALL allocation point and deposit fee. Can only be called by the owner.
1313 function set(
1314 uint256 _pid,
1315 uint256 _allocPoint,
1316 uint16 _depositFeeBP,
1317 bool _withUpdate
1318 ) public onlyOwner {
1319 require(_depositFeeBP <= MAXIMUM_DEPOSIT_FEE_BP, "set: invalid deposit fee basis points");
1320 if (_withUpdate) {
1321 massUpdatePools();
1322 }
1323 totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
1324 _allocPoint
1325 );
1326 poolInfo[_pid].allocPoint = _allocPoint;
1327 poolInfo[_pid].depositFeeBP = _depositFeeBP;
1328 }
1329
1330 // 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

```
1394
1395 // Deposit LP tokens to MasterChef for BALL allocation.
1396 function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
1397 PoolInfo storage pool = poolInfo[_pid];
1398 UserInfo storage user = userInfo[_pid][msg.sender];
1399 updatePool(_pid);
1400 if (user.amount > 0) {
1401 uint256 pending =
1402 user.amount.mul(pool.accBallPerShare).div(1e12).sub(
1403 user.rewardDebt
1404 );
1405 if (pending > 0) {
1406 safeBallTransfer(msg.sender, pending);
1407 }
1408 }
1409 if (_amount > 0) {
1410 pool.lpToken.safeTransferFrom(
1411 address(msg.sender),
1412 address(this),
1413 _amount
1414 );
1415 if (pool.depositFeeBP > 0) {
1416 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1417 user.amount = user.amount.add(_amount).sub(depositFee);
1418 pool.lpToken.safeTransfer(feeAddress, depositFee);
1419 } else {
1420 user.amount = user.amount.add(_amount);
1421 }
1422 }
1423 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1424 emit Deposit(msg.sender, _pid, _amount);
1425 }
1426
1427 // Deposit LP tokens to MasterChef for BALL allocation with referral.
```

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

```
1426
1427 // Deposit LP tokens to MasterChef for BALL allocation with referral.
1428 function deposit(uint256 _pid, uint256 _amount, address _referrer) public nonReentrant {
1429 require(_referrer == address(_referrer),"deposit: Invalid referrer address");
1430 PoolInfo storage pool = poolInfo[_pid];
1431 UserInfo storage user = userInfo[_pid][msg.sender];
1432 updatePool(_pid);
1433 if (user.amount > 0) {
1434 uint256 pending =
1435 user.amount.mul(pool.accBallPerShare).div(1e12).sub(
1436 user.rewardDebt
1437 );
1438 if (pending > 0) {
1439 safeBallTransfer(msg.sender, pending);
1440 payReferralCommission(msg.sender, pending);
1441 }
1442 }
1443 if (_amount > 0) {
1444 setReferral(msg.sender, _referrer);
1445 pool.lpToken.safeTransferFrom(
1446 address(msg.sender),
1447 address(this),
1448 _amount
1449 );
1450 if (pool.depositFeeBP > 0) {
1451 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1452 user.amount = user.amount.add(_amount).sub(depositFee);
1453 pool.lpToken.safeTransfer(feeAddress, depositFee);
1454 } else {
1455 user.amount = user.amount.add(_amount);
1456 }
1457 }
1458 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1459 emit Deposit(msg.sender, _pid, _amount);
1460 }
1461
1462 // 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

```
1461
1462 // Withdraw LP tokens from MasterChef.
1463 function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
1464 PoolInfo storage pool = poolInfo[_pid];
1465 UserInfo storage user = userInfo[_pid][msg.sender];
1466 require(user.amount >= _amount, "withdraw: not good");
1467 updatePool(_pid);
1468 uint256 pending =
1469 user.amount.mul(pool.accBallPerShare).div(1e12).sub(
1470 user.rewardDebt
1471 );
1472 if (pending > 0) {
1473 safeBallTransfer(msg.sender, pending);
1474 payReferralCommission(msg.sender, pending);
1475 }
1476 if (_amount > 0) {
1477 user.amount = user.amount.sub(_amount);
1478 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1479 }
1480 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1481 emit Withdraw(msg.sender, _pid, _amount);
1482 }
1483
1484 // 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

```
1483
1484 // Withdraw without caring about rewards. EMERGENCY ONLY.
1485 function emergencyWithdraw(uint256 _pid) public nonReentrant {
1486 PoolInfo storage pool = poolInfo[_pid];
1487 UserInfo storage user = userInfo[_pid][msg.sender];
1488 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1489 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1490 user.amount = 0;
1491 user.rewardDebt = 0;
1492 }
1493
1494 // Safe ball transfer function, just in case if rounding error causes pool to not have enough BALLs.
```

MEDIUM

Function could be marked as external.

SWC-000

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

```
1505
1506  // Update dev address by the previous dev.
1507  function setDevAddress(address _devaddr) public {
1508  require(_devaddr != address(0), "dev: invalid address");
1509  require(msg.sender == devAddr, "dev: wut?");
1510  devAddr = _devaddr;
1511  emit SetDevAddress(msg.sender, _devaddr);
1512  }
1513
1514  // Update fee address by the previous fee address.
```

MEDIUM

Function could be marked as external.

SWC-000

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

```
1513
1514  // Update fee address by the previous fee address.
1515  function setFeeAddress(address _feeAddress) public {
1516  require(_feeAddress != address(0), "setFeeAddress: invalid address");
1517  require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
1518  feeAddress = _feeAddress;
1519  emit SetFeeAddress(msg.sender, _feeAddress);
1520  }
1521
1522  // Reduce emission rate by 3% every 14,400 blocks ~ 12hours till the emission rate is 0.5 BALL.
```

MEDIUM

Function could be marked as external.

SWC-000

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

```
1522 // Reduce emission rate by 3% every 14,400 blocks ~ 12hours till the emission rate is 0.5 BALL.
1523 // This function can be called publicly.
1524 function updateEmissionRate() public {
1525 require(block.number > startBlock, "updateEmissionRate: Can only be called after mining starts");
1526 require(ballPerBlock > MINIMUM_EMISSION_RATE, "updateEmissionRate: Emission rate has reached the minimum threshold");
1527
1528 uint256 currentIndex = block.number.sub(startBlock).div(EMISSION_REDUCTION_PERIOD_BLOCKS);
1529 if (currentIndex <= lastReductionPeriodIndex) {
1530 return;
1531 }
1532
1533 uint256 newEmissionRate = ballPerBlock;
1534 for (uint256 index = lastReductionPeriodIndex; index < currentIndex; ++index) {
1535 newEmissionRate = newEmissionRate.mul(1e4 - EMISSION_REDUCTION_RATE_PER_PERIOD).div(1e4);
1536 }
1537
1538 newEmissionRate = newEmissionRate < MINIMUM_EMISSION_RATE ? MINIMUM_EMISSION_RATE : newEmissionRate;
1539 if (newEmissionRate >= ballPerBlock) {
1540 return;
1541 }
1542
1543 massUpdatePools();
1544 lastReductionPeriodIndex = currentIndex;
1545 uint256 previousEmissionRate = ballPerBlock;
1546 ballPerBlock = newEmissionRate;
1547 emit EmissionRateUpdated(msg.sender, previousEmissionRate, newEmissionRate);
1548 }
1549
1550 // Set Referral Address for a user
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "updateReferralBonusBp" 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

```
1575 // Initially set to 2%, this this the ability to increase or decrease the Bonus percentage based on
1576 // community voting and feedback.
1577 function updateReferralBonusBp(uint256 _newRefBonusBp) public onlyOwner {
1578 require(_newRefBonusBp <= MAXIMUM_REFERRAL_BP, "updateRefBonusPercent: invalid referral bonus basis points");
1579 require(_newRefBonusBp != refBonusBP, "updateRefBonusPercent: same bonus bp set");
1580 uint256 previousRefBonusBP = refBonusBP;
1581 refBonusBP = _newRefBonusBp;
1582 emit ReferralBonusBpChanged(previousRefBonusBP, _newRefBonusBp);
1583 }
1584
1585 }
```

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

```
365
366 // solhint-disable-next-line avoid-low-level-calls
367 (bool success, bytes memory returndata) = target.call{ value: value }(data);
368 return _verifyCallResult(success, returndata, errorMessage);
369 }
```

MEDIUM

Loop over unbounded data structure.

SWC-128

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

MasterChef.sol

Locations

```
1364 function massUpdatePools() public {
1365 uint256 length = poolInfo.length;
1366 for (uint256 pid = 0; pid < length; ++pid) {
1367 updatePool(pid);
1368 }
```

MEDIUM

Loop over unbounded data structure.

SWC-128

Gas consumption in function "updateEmissionRate" 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

MasterChef.sol

Locations

```
1532
1533 uint256 newEmissionRate = ballPerBlock;
1534 for (uint256 index = lastReductionPeriodIndex; index < currentIndex; ++index) {
1535 newEmissionRate = newEmissionRate.mul(1e4 - EMISSION_REDUCTION_RATE_PER_PERIOD).div(1e4);
1536 }
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""=0.6.0<0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

MasterChef.sol

Locations

```
1   // SPDX-License-Identifier: MIT
2
3   pragma solidity >=0.6.0 <0.8.0;
4
5   /**
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""=0.6.0<0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

MasterChef.sol

Locations

```
502 }
503
504 pragma solidity >=0.6.0 <0.8.0;
505
506 /**
```

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

```
1413 _amount
1414 );
1415 if (pool.depositFeeBP > 0) {
1416 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1417 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

```
1414 );
1415 if (pool.depositFeeBP > 0) {
1416 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1417 user.amount = user.amount.add(_amount).sub(depositFee);
1418 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

```
1415 if (pool.depositFeeBP > 0) {
1416 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1417 user.amount = user.amount.add(_amount).sub(depositFee);
1418 pool.lpToken.safeTransfer(feeAddress, depositFee);
1419 } else {
```

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

```
1415 if (pool.depositFeeBP > 0) {
1416 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1417 user.amount = user.amount.add(_amount).sub(depositFee);
1418 pool.lpToken.safeTransfer(feeAddress, depositFee);
1419 } 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

```
1416 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1417 user.amount = user.amount.add(_amount).sub(depositFee);
1418 pool.lpToken.safeTransfer(feeAddress, depositFee);
1419 } else {
1420 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

```
1416 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1417 user.amount = user.amount.add(_amount).sub(depositFee);
1418 pool.lpToken.safeTransfer(feeAddress, depositFee);
1419 } else {
1420 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

```
361 */
362 function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage) internal returns (bytes memory) {
363 require(address(this).balance >= value, "Address: insufficient balance for call");
364 require(isContract(target), "Address: call to non-contract");
365
```

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

```
1421 }
1422 }
1423 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1424 emit Deposit(msg.sender, _pid, _amount);
1425 }
```

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

```
1421 }
1422 }
1423 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1424 emit Deposit(msg.sender, _pid, _amount);
1425 }
```

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

```
1421 }
1422 }
1423 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1424 emit Deposit(msg.sender, _pid, _amount);
1425 }
```

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

```
1487 UserInfo storage user = userInfo[_pid][msg.sender];
1488 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1489 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1490 user.amount = 0;
1491 user.rewardDebt = 0;
```

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

```
1488 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1489 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1490 user.amount = 0;
1491 user.rewardDebt = 0;
1492 }
```

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

```
1489 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1490 user.amount = 0;
1491 user.rewardDebt = 0;
1492 }
1493
```

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

```
1418 pool.lpToken.safeTransfer(feeAddress, depositFee);
1419 } else {
1420 user.amount = user.amount.add(_amount);
1421 }
1422 }
```

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

```
1418 pool.lpToken.safeTransfer(feeAddress, depositFee);
1419 } else {
1420 user.amount = user.amount.add(_amount);
1421 }
1422 }
```

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

```
1448 _amount
1449 );
1450 if (pool.depositFeeBP > 0) {
1451 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1452 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

```
1449 );
1450 if (pool.depositFeeBP > 0) {
1451 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1452 user.amount = user.amount.add(_amount).sub(depositFee);
1453 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

```
1450 if (pool.depositFeeBP > 0) {
1451 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1452 user.amount = user.amount.add(_amount).sub(depositFee);
1453 pool.lpToken.safeTransfer(feeAddress, depositFee);
1454 } 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

```
1451 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1452 user.amount = user.amount.add(_amount).sub(depositFee);
1453 pool.lpToken.safeTransfer(feeAddress, depositFee);
1454 } else {
1455 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

```
1451 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1452 user.amount = user.amount.add(_amount).sub(depositFee);
1453 pool.lpToken.safeTransfer(feeAddress, depositFee);
1454 } else {
1455 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

```
1456 }
1457 }
1458 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1459 emit Deposit(msg.sender, _pid, _amount);
1460 }
```

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

```
1456 }
1457 }
1458 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1459 emit Deposit(msg.sender, _pid, _amount);
1460 }
```

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

```
1450 if (pool.depositFeeBP > 0) {
1451 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1452 user.amount = user.amount.add(_amount).sub(depositFee);
1453 pool.lpToken.safeTransfer(feeAddress, depositFee);
1454 } else {
```

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

```
1456 }
1457 }
1458 user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
1459 emit Deposit(msg.sender, _pid, _amount);
1460 }
```

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

```
918 // By storing the original value once again, a refund is triggered (see
919 // https://eips.ethereum.org/EIPS/eip-2200)
920 _status = _NOT_ENTERED;
921 }
922 }
```

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

```
1453 pool.lpToken.safeTransfer(feeAddress, depositFee);
1454 } else {
1455 user.amount = user.amount.add(_amount);
1456 }
1457 }
```

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

```
1453 pool.lpToken.safeTransfer(feeAddress, depositFee);
1454 } else {
1455 user.amount = user.amount.add(_amount);
1456 }
1457 }
```

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

```
1064 returns (uint256)
1065 {
1066 require(blockNumber < block.number, "BALL::getPriorVotes: not yet determined");
1067
1068 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

```
1137 internal
1138 {
1139 uint32 blockNumber = safe32(block.number, "BALL::_writeCheckpoint: block number exceeds 32 bits");
1140
1141 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

```
1295 massUpdatePools();
1296 }
1297 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1298 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1299 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

MasterChef.sol

Locations

```
1295 massUpdatePools();
1296 }
1297 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1298 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1299 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

MasterChef.sol

Locations

```
1347 uint256 accBallPerShare = pool.accBallPerShare;
1348 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1349 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1350 uint256 multiplier =
1351 getMultiplier(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

MasterChef.sol

Locations

```
1349 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1350 uint256 multiplier =
1351 getMultiplier(pool.lastRewardBlock, block.number);
1352 uint256 ballReward =
1353 multiplier.mul(ballPerBlock).mul(pool.allocPoint).div(
```

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

```
1372 function updatePool(uint256 _pid) public {
1373 PoolInfo storage pool = poolInfo[_pid];
1374 if (block.number <= pool.lastRewardBlock) {
1375 return;
1376 }
```

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

```
1377 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1378 if (lpSupply == 0 || pool.allocPoint == 0) {
1379 pool.lastRewardBlock = block.number;
1380 return;
1381 }
```

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

```
1380 return;
1381 }
1382 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1383 uint256 ballReward =
1384 multiplier.mul(ballPerBlock).mul(pool.allocPoint).div(
```

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

```
1390 ballReward.mul(1e12).div(lpSupply)
1391 );
1392 pool.lastRewardBlock = block.number;
1393 }
1394
```

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

```
1523 // This function can be called publicly.
1524 function updateEmissionRate() public {
1525 require(block.number > startBlock, "updateEmissionRate: Can only be called after mining starts");
1526 require(ballPerBlock > MINIMUM_EMISSION_RATE, "updateEmissionRate: Emission rate has reached the minimum threshold");
1527
```

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

```
1526 require(ballPerBlock > MINIMUM_EMISSION_RATE, "updateEmissionRate: Emission rate has reached the minimum threshold");
1527
1528 uint256 currentIndex = block.number.sub(startBlock).div(EMISSION_REDUCTION_PERIOD_BLOCKS);
1529 if (currentIndex <= lastReductionPeriodIndex) {
1530 return;
```

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

```
1375 return;
1376 }
1377 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1378 if (lpSupply == 0 || pool.allocPoint == 0) {
1379 pool.lastRewardBlock = block.number;
```

Source file

MasterChef.sol

Locations

```
1169 //
1170 // Have fun reading it. Hopefully it's bug-free. God bless.
1171 contract MasterChef is Ownable, ReentrancyGuard {
1172 using SafeMath for uint256;
1173 using SafeBEP20 for IBEP20;
1174
1175 // Info of each user.
1176 struct UserInfo {
1177 uint256 amount; // How many LP tokens the user has provided.
1178 uint256 rewardDebt; // Reward debt. See explanation below.
1179 //
1180 // We do some fancy math here. Basically, any point in time, the amount of BALLs
1181 // entitled to a user but is pending to be distributed is:
1182 //
1183 // pending reward = (user.amount * pool.accBallPerShare) - user.rewardDebt
1184 //
1185 // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
1186 // 1. The pool's `accBallPerShare` (and `lastRewardBlock`) gets updated.
1187 // 2. User receives the pending reward sent to his/her address.
1188 // 3. User's `amount` gets updated.
1189 // 4. User's `rewardDebt` gets updated.
1190 }
1191
1192 // Info of each pool.
1193 struct PoolInfo {
1194 IBEP20 lpToken; // Address of LP token contract.
1195 uint256 allocPoint; // How many allocation points assigned to this pool. BALLs to distribute per block.
1196 uint256 lastRewardBlock; // Last block number that BALLs distribution occurs.
1197 uint256 accBallPerShare; // Accumulated BALLs per share, times 1e12. See below.
1198 uint16 depositFeeBP; // Deposit fee in basis points
1199 }
1200
1201 // The BALL Token!
1202 BallToken public ball;
1203 // Dev address.
1204 address public devAddr;
1205 // BALL tokens created per block.
1206 uint256 public ballPerBlock;
1207 // Deposit Fee address
1208 address public feeAddress;
1209
1210 // Info of each pool.
1211 PoolInfo[] public poolInfo;
1212
```

```
// 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 BALL mining starts.
uint256 public startBlock;
// Referral Bonus in basis points. Initially set to 2%
uint256 public refBonusBP = 200;
// Max deposit fee: 10%.
uint16 public constant MAXIMUM_DEPOSIT_FEE_BP = 1000;
// Max referral commission rate: 20%.
uint16 public constant MAXIMUM_REFERRAL_BP = 2000;
// Referral Mapping
mapping(address => address) public referrers; // account_address -> referrer_address
mapping(address => uint256) public referredCount; // referrer_address -> num_of_referred
// Pool Exists Mapper
mapping(IBEP20 => bool) public poolExistence;
// Pool ID Tracker Mapper
mapping(IBEP20 => uint256) public poolIdForLpAddress;

// Initial emission rate: 1 BALL per block.
uint256 public constant INITIAL_EMISSION_RATE = 1 ether;
// Minimum emission rate: 0.5 BALL per block.
uint256 public constant MINIMUM_EMISSION_RATE = 500 finney;
// Reduce emission every 14,400 blocks ~ 12 hours.
uint256 public constant EMISSION_REDUCTION_PERIOD_BLOCKS = 14400;
// Emission reduction rate per period in basis points: 3%.
uint256 public constant EMISSION_REDUCTION_RATE_PER_PERIOD = 300;
// Last reduction period index
uint256 public lastReductionPeriodIndex = 0;

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 _devAddress);
event SetDevAddress(address indexed user, address indexed _feeAddress);
event Referral(address indexed _referrer, address indexed _user);
event ReferralPaid(address indexed _user, address indexed _userTo, uint256 _reward);
event ReferralBonusBpChanged(uint256 _oldBp, uint256 _newBp);
event EmissionRateUpdated(address indexed caller, uint256 previousAmount, uint256 newAmount);

constructor(
BallToken _ball,
address _devAddr,
address _feeAddress,
uint256 _startBlock
) public {
ball = _ball;
devAddr = _devAddr;
feeAddress = _feeAddress;
ballPerBlock = INITIAL_EMISSION_RATE;
startBlock = _startBlock;
}

// Get number of pools added.
function poolLength() external view returns (uint256) {
return poolInfo.length;
```

```
}

function getPoolIdForLpToken(IBEP20 _lpToken) external view returns (uint256) {
require(poolExistence[_lpToken] != false, "getPoolIdForLpToken: do not exist");
return poolIdForLpAddress[_lpToken];
}

// Modifier to check Duplicate pools
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 <= MAXIMUM_DEPOSIT_FEE_BP, "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,
accBallPerShare: 0,
depositFeeBP: _depositFeeBP
})
);
poolIdForLpAddress[_lpToken] = poolInfo.length - 1;
}

// Update the given pool's BALL 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 <= MAXIMUM_DEPOSIT_FEE_BP, "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
pure
```

```
returns (uint256)
{
return _to.sub(_from);
}

// View function to see pending BALLs on frontend.
function pendingBall(uint256 _pid, address _user)
external
view
returns (uint256)
{
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][_user];
uint256 accBallPerShare = pool.accBallPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier =
getMultiplier(pool.lastRewardBlock, block.number);
uint256 ballReward =
multiplier.mul(ballPerBlock).mul(pool.allocPoint).div(
totalAllocPoint
);
accBallPerShare = accBallPerShare.add(
ballReward.mul(1e12).div(lpSupply)
);
}
return user.amount.mul(accBallPerShare).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 ballReward =
multiplier.mul(ballPerBlock).mul(pool.allocPoint).div(
totalAllocPoint
);
ball.mint(devAddr, ballReward.div(10));
ball.mint(address(this), ballReward);
pool.accBallPerShare = pool.accBallPerShare.add(
ballReward.mul(1e12).div(lpSupply)
);
pool.lastRewardBlock = block.number;
}


```

```
// Deposit LP tokens to MasterChef for BALL 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.accBallPerShare).div(1e12).sub(
user.rewardDebt
);
if (pending > 0) {
safeBallTransfer(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);
user.amount = user.amount.add(_amount).sub(depositFee);
pool.lpToken.safeTransfer(feeAddress, depositFee);
} else {
user.amount = user.amount.add(_amount);
}
}
user.rewardDebt = user.amount.mul(pool.accBallPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}

// Deposit LP tokens to MasterChef for BALL allocation with referral.
function deposit(uint256 _pid, uint256 _amount, address _referrer) public nonReentrant {
require(_referrer == address(_referrer),"deposit: Invalid referrer address");
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
updatePool(_pid);
if (user.amount > 0) {
uint256 pending =
user.amount.mul(pool.accBallPerShare).div(1e12).sub(
user.rewardDebt
);
if (pending > 0) {
safeBallTransfer(msg.sender, pending);
payReferralCommission(msg.sender, pending);
}
}
if (_amount > 0) {
setReferral(msg.sender, _referrer);
pool.lpToken.safeTransferFrom(
address(msg.sender),
address(this),
_amount
);
if (pool.depositFeeBP > 0) {
uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
user.amount = user.amount.add(_amount).sub(depositFee);
pool.lpToken.safeTransfer(feeAddress, depositFee);
} else {
user.amount = user.amount.add(_amount);
```

```
}
}
user.rewardDebt = user.amount.mul(pool.accBallPerShare).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.accBallPerShare).div(1e12).sub(
user.rewardDebt
);
if (pending > 0) {
safeBallTransfer(msg.sender, pending);
payReferralCommission(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.accBallPerShare).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];
pool.lpToken.safeTransfer(address(msg.sender), user.amount);
emit EmergencyWithdraw(msg.sender, _pid, user.amount);
user.amount = 0;
user.rewardDebt = 0;
}

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

// Update dev address by the previous dev.
function setDevAddress(address _devaddr) public {
require(_devaddr != address(0), "dev: invalid address");
require(msg.sender == devAddr, "dev: wut?");
devAddr = _devaddr;
emit SetDevAddress(msg.sender, _devaddr);
}

// Update fee address by the previous fee address.
function setFeeAddress(address _feeAddress) public {
require(_feeAddress != address(0), "setFeeAddress: invalid address");

```

```
require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
feeAddress = _feeAddress;
emit SetFeeAddress(msg.sender, _feeAddress);
}

// Reduce emission rate by 3% every 14,400 blocks ~ 12hours till the emission rate is 0.5 BALL.
// This function can be called publicly.
function updateEmissionRate() public {
require(block.number > startBlock, "updateEmissionRate: Can only be called after mining starts");
require(ballPerBlock > MINIMUM_EMISSION_RATE, "updateEmissionRate: Emission rate has reached the minimum threshold");

uint256 currentIndex = block.number.sub(startBlock).div(EMISSION_REDUCTION_PERIOD_BLOCKS);
if (currentIndex <= lastReductionPeriodIndex) {
return;
}

uint256 newEmissionRate = ballPerBlock;
for (uint256 index = lastReductionPeriodIndex; index < currentIndex; ++index) {
newEmissionRate = newEmissionRate.mul(1e4 - EMISSION_REDUCTION_RATE_PER_PERIOD).div(1e4);
}

newEmissionRate = newEmissionRate < MINIMUM_EMISSION_RATE ? MINIMUM_EMISSION_RATE : newEmissionRate;
if (newEmissionRate >= ballPerBlock) {
return;
}

massUpdatePools();
lastReductionPeriodIndex = currentIndex;
uint256 previousEmissionRate = ballPerBlock;
ballPerBlock = newEmissionRate;
emit EmissionRateUpdated(msg.sender, previousEmissionRate, newEmissionRate);
}

// Set Referral Address for a user
function setReferral(address _user, address _referrer) internal {
if (_referrer == address(_referrer) && referrers[_user] == address(0) && _referrer != address(0) && _referrer != _user) {
referrers[_user] = _referrer;
referredCount[_referrer] += 1;
emit Referral(_user, _referrer);
}
}

// Get Referral Address for a Account
function getReferral(address _user) public view returns (address) {
return referrers[_user];
}

// Pay referral commission to the referrer who referred this user.
function payReferralCommission(address _user, uint256 _pending) internal {
address referrer = getReferral(_user);
if (referrer != address(0) && referrer != _user && refBonusBP > 0) {
uint256 refBonusEarned = _pending.mul(refBonusBP).div(10000);
ball.mint(referrer, refBonusEarned);
emit ReferralPaid(_user, referrer, refBonusEarned);
}
}

// Referral Bonus in basis points.
// Initially set to 2%, this this the ability to increase or decrease the Bonus percentage based on
// community voting and feedback.
function updateReferralBonusBp(uint256 _newRefBonusBp) public onlyOwner {

```

```
1583  require(_newRefBonusBp <= MAXIMUM_REFERRAL_BP, "updateRefBonusPercent: invalid referral bonus basis points");
1584  require(_newRefBonusBp != refBonusBP, "updateRefBonusPercent: same bonus bp set");
1585  uint256 previousRefBonusBP = refBonusBP;
      refBonusBP = _newRefBonusBp;
      emit ReferralBonusBpChanged(previousRefBonusBP, _newRefBonusBp);
      }


      }
```

LOW

SWC-128

Potentially unbounded data structure passed to builtin.

Gas consumption in function "delegateBySig" in contract "BallToken" depends on the size of data structures that may grow unboundedly. Specifically the "1-st" argument to builtin "keccak256" may be able to grow unboundedly causing the builtin to consume more gas than 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

MasterChef.sol

Locations

```
1008  abi.encode(
1009  DOMAIN_TYPEHASH,
1010  keccak256(bytes(name())),
1011  getChainId(),
1012  address(this)
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "getPriorVotes" in contract "BallToken" 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

MasterChef.sol

Locations

```
1083  uint32 lower = 0;
1084  uint32 upper = nCheckpoints - 1;
1085  while (upper > lower) {
1086  uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
1087  Checkpoint memory cp = checkpoints[account][center];
```