MythX

REPORT 60C6444F777A4E00187ECCD5

Created	Sun Jun 13 2021 17:45:51 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	60c63fab8bfa125f70f292e8

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
c2092122-eca6-4da8-b97c-259ebcf2885a	masterChef.sol	65

Analysis c2092122-eca6-4da8-b97c-259ebcf2885a MythX

Started	Sun Jun 13 2021 17:46:01 GMT+0000 (Coordinated Universal Time)
Finished	Sun Jun 13 2021 18:31:57 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	MasterChef.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	28	37

ISSUES

MEDIUM

SWC-000

Function could be marked as external.

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

Source file
masterChef.sol
Locations

```
622 * thereby removing any functionality that is only available to the owner.
623 */
624 function renounceOwnership() public onlyOwner {
625 emit OwnershipTransferred(_owner, address(0));
626 _owner = address(0);
627 }
628
629 /**
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file
masterChef.sol
Locations

```
631 * Can only be called by the current owner.
632 */
633 function transferOwnership(address newOwner) public onlyOwner {
634 _transferOwnership(newOwner);
635 }
636
637 /**
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

masterChef.sol

Locations

```
719 * @dev Returns the token decimals.
720 */
721 function decimals() public override view returns (uint8) {
722 return _decimals;
723 }
724
725 /**
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

masterChef.sol

Locations

```
726 * @dev Returns the token symbol.
727 */
728 function symbol() public override view returns (string memory) {
729 return _symbol;
730 }
731
732 /**
```

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

```
760 * @dev See {BEP20-allowance}.
761 */
762 function allowance(address owner, address spender) public override view returns (uint256) {
763 return _allowances[owner][spender];
764 }
765
766 /**
```

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

```
771 * - `spender` cannot be the zero address.
772 */
773 function approve(address spender, uint256 amount) public override returns (bool) {
774 _approve(_msgSender(), spender, amount);
775 return true;
776 }
777
778 /**
```

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

```
814 * - `spender` cannot be the zero address.
815 */
816 function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
817 _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
818 return true;
819 }
820
821 /**
```

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

```
833 * `subtractedValue`.
834 */
835 function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
836 _approve(
837 _msgSender(),
838 spender,
839 _allowances[_msgSender()][spender].sub(subtractedValue, 'BEP20: decreased allowance below zero')
840 );
841 return true;
842 }
843
844 /**
```

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

```
967 contract DLPHToken is BEP20('Dolphin Token', 'DLPH') {
968
969 function mint(address _to, uint256 _amount) public onlyOwner {
970 require(_amount != 0, "DLPH::mint: mint value should not be zero");
971 _mint(_to, _amount);
972 _moveDelegates(address(0), _delegates[_to], _amount);
973 }
974
975 function burn(uint256 value) public onlyOwner {
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "burn" 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

```
973 }
974
975 function burn(uint256 value) public onlyOwner {
976 require(value != 0, "DLPH::burn: burn value should not be zero");
977 uint totalSupply = totalSupply();
978 require(value <= totalSupply);
979
980 _burn(msg.sender, value);
981 }
982
983 function transfer(address _to, uint256 _value) public virtual override returns (bool success) {
```

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

```
981 }
982
983 function transfer(address _to, uint256 _value) public virtual override returns (bool success) {
984 require(_value != 0, "DLPH::transfer: transfer value should not be zero");
985 uint256 toBurn = _value / 100;
986
987 if(super.transfer(_to, _value - toBurn)) {
988 _burn(msg.sender, toBurn);
989 _moveDelegates(_delegates[msg.sender], _delegates[_to], _value - toBurn);
990 return true;
991 }
992 else
993 return false;
994 }
995
996 function transferFrom(address _from, address _to, uint256 _value) public override returns (bool success)
```

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

```
994 }
995
996 function transferFrom(address _from, address _to, uint256 _value) public override returns (bool success)
997 {
998 require(_value != 0, "DLPH::transfer: transfer value shouldnot be zero");
999 uint256 toBurn = _value / 100;
1000
1001 if(super.transferFrom(_from, _to, _value - toBurn)) {
1002 _burn(msg.sender, toBurn);
1003 _moveDelegates(_delegates[_from], _delegates[_to], _value - toBurn);
1004 return true;
1005 }
1006 else
1007 return false;
1008 }
1009
1010 // Copied and modified from YAM code:
```

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

```
1248 contract FoamBar is BEP20('FoamBar Token', 'FOAM') {
1249 /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
1250 function mint(address _to, uint256 _amount) public onlyOwner {
1251 _mint(_to, _amount);
1252 _moveDelegates(address(0), _delegates[_to], _amount);
1253 }
1254
1255 function burn(address _from ,uint256 _amount) public onlyOwner {
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "burn" 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

```
1253 }
1254
1255 function burn(address _from ,uint256 _amount) public onlyOwner {
1256 _burn(_from, _amount);
1257 _moveDelegates(_delegates[_from], address(0), _amount);
1258 }
1259
1260 // The Dolphin Token!
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "safeDLPHTransfer" 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

```
1269
1270 // Safe DLPH transfer function, just in case if rounding error causes pool to not have enough DLPHs.
1271 function safeDLPHTransfer(address _to, uint256 _amount) public onlyOwner {
1272 uint256 DLPHBal = DLPH.balanceOf(address(this));
1273 if (_amount > DLPHBal) {
1274 DLPH.transfer(_to, DLPHBal);
1275 } else {
1276 DLPH.transfer(_to, _amount);
1277 }
1278 }
1279
1280 // Copied and modified from YAM code:
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "updateMultiplier" 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

```
1622 }
1623
1624 function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
1625 BONUS_MULTIPLIER = multiplierNumber;
1626 }
1627
1628 function poolLength() external view returns (uint256) {
```

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

```
1632 // Add a new lp to the pool. Can only be called by the owner.
1633 // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
1634 function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {
1635 require(_depositFeeBP <= 10000, "add: invalid deposit fee basis points");
1636 if (_withUpdate) {
1637 massUpdatePools();
1638 }
1639 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1640 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1641 poolInfo.push(PoolInfo({
1642 lpToken: _lpToken,
1643 allocPoint: _allocPoint,
1644 lastRewardBlock: lastRewardBlock,
1645 accDLPHPerShare: 0,
1646 depositFeeBP: _depositFeeBP
1647 }));
1648 updateStakingPool();
1649 }
1650
1651 // Update the given pool's DLPH allocation point. 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

```
1650
1651 // Update the given pool's DLPH allocation point. Can only be called by the owner.
1652 function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {
1653 require(_depositFeeBP <= 10000, "set: invalid deposit fee basis points");
1654 if (_withUpdate) {
1655 massUpdatePools();
1656 }
1657 uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
1658 poolInfo[_pid].allocPoint = _allocPoint;
1659 poolInfo[_pid].depositFeeBP = _depositFeeBP;
1660 if (prevAllocPoint != _allocPoint) {
1661 totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
1662 updateStakingPool();
1663 }
1664 }
1665
1666 function updateStakingPool() internal {
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "getPerBlock" 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

```
1704 }
1705
1706 function getPerBlock() public view returns (uint256) {
1707 return DLPHPerBlock;
1708 }
1709
1710 // Update reward variables for all pools. Be careful of gas spending!
```

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

```
1748
1749  // Deposit LP tokens to MasterChef for DLPH allocation.
1750  function deposit(uint256 _pid, uint256 _amount) public {
1751
1752  require (_pid != 0, 'deposit DLPH by staking');
1753
1754  PoolInfo storage pool = poolInfo[_pid];
1755  UserInfo storage user = userInfo[_pid][msg.sender];
1756  address DLPHaddr = address(DLPH);
1757
1758  updatePool(_pid);
1759  if (user.amount > 0) {
1760  uint256 pending = user.amount.mul(pool.accDLPHPerShare).div(1e12).sub(user.rewardDebt);
1761  if(pending > 0) {
1762  safeDLPHTransfer(msg.sender, pending);
1763  }
1764  }
1765  if (_amount > 0) {
1766  pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1767  if(pool.depositFeeBP > 0){
1768  uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1769
1770  pool.lpToken.safeTransfer(treasuryaddr, depositFee.div(2));
1771  pool.lpToken.safeTransfer(address(this), depositFee.div(4));
1772  pool.lpToken.safeTransfer(DLPHaddr, depositFee.div(4).div(4).mul(3));
1773  pool.lpToken.safeTransfer(devaddr, depositFee.div(4).div(4));
1774
1775  user.amount = user.amount.add(_amount).sub(depositFee);
1776  }else{
1777  user.amount = user.amount.add(_amount);
1778  }
1779  }
1780  user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
1781  emit Deposit(msg.sender, _pid, _amount);
1782  }
1783
1784  // 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

```
1783
1784 // Withdraw LP tokens from MasterChef.
1785 function withdraw(uint256 _pid, uint256 _amount) public {
1786
1787 require (_pid != 0, 'withdraw DLPH by unstaking');
1788 PoolInfo storage pool = poolInfo[_pid];
1789 UserInfo storage user = userInfo[_pid][msg.sender];
1790 require(user.amount >= _amount, "withdraw: not good");
1791
1792 updatePool(_pid);
1793 uint256 pending = user.amount.mul(pool.accDLPHPerShare).div(1e12).sub(user.rewardDebt);
1794 if(pending > 0) {
1795 safeDLPHTransfer(msg.sender, pending);
1796 }
1797 if(_amount > 0) {
1798 user.amount = user.amount.sub(_amount);
1799 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1800 }
1801 user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
1802 emit Withdraw(msg.sender, _pid, _amount);
1803 }
1804
1805 // Stake Dolphin Tokens to MasterChef
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "enterStaking" 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

```
1804
1805 // Stake Dolphin Tokens to MasterChef
1806 function enterStaking(uint256 _amount) public {
1807 PoolInfo storage pool = poolInfo[0];
1808 UserInfo storage user = userInfo[0][msg.sender];
1809 updatePool(0);
1810 if (user.amount > 0) {
1811 uint256 pending = user.amount.mul(pool.accDLPHPerShare).div(1e12).sub(user.rewardDebt);
1812 if(pending > 0) {
1813 safeDLPHTransfer(msg.sender, pending);
1814 }
1815 }
1816 if(_amount > 0) {
1817 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1818 user.amount = user.amount.add(_amount);
1819 }
1820 user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
1821
1822 if (_amount > 0)
1823 foam.mint(msg.sender, _amount);
1824
1825 emit Deposit(msg.sender, 0, _amount);
1826 }
1827
1828 // Withdraw Dolphin Tokens from STAKING.
```

MEDIUM

Function could be marked as external.

SWC-000

The function definition of "leaveStaking" 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

```
1827
1828 // Withdraw Dolphin Tokens from STAKING.
1829 function leaveStaking(uint256 _amount) public {
1830 PoolInfo storage pool = poolInfo[0];
1831 UserInfo storage user = userInfo[0][msg.sender];
1832 require(user.amount >= _amount, "withdraw: not good");
1833 updatePool(0);
1834 uint256 pending = user.amount.mul(pool.accDLPHPerShare).div(1e12).sub(user.rewardDebt);
1835 if(pending > 0) {
1836 safeDLPHTransfer(msg.sender, pending);
1837 }
1838 if(_amount > 0) {
1839 user.amount = user.amount.sub(_amount);
1840 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1841 }
1842 user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
1843
1844 foam.burn(msg.sender, _amount);
1845 emit Withdraw(msg.sender, 0, _amount);
1846 }
1847
1848 // 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

```
1847
1848 // Withdraw without caring about rewards. EMERGENCY ONLY.
1849 function emergencyWithdraw(uint256 _pid) public {
1850 PoolInfo storage pool = poolInfo[_pid];
1851 UserInfo storage user = userInfo[_pid][msg.sender];
1852 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1853 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1854 user.amount = 0;
1855 user.rewardDebt = 0;
1856 }
1857
1858 // Safe DLPH transfer function, just in case if rounding error causes pool to not have enough DLPHs.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

masterChef.sol

Locations

```
1862
1863 // Update dev address by the previous dev.
1864 function dev(address _devaddr) public {
1865 require(msg.sender == devaddr, "dev: wut?");
1866 devaddr = _devaddr;
1867 }
1868 }
```

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

```
356
357 // solhint-disable-next-line avoid-low-level-calls
358 (bool success, bytes memory returndata) = target.call{value: weiValue}(data);
359 if (success) {
360 return returndata;
```

MEDIUM

Loop over unbounded data structure.

SWC-128

Gas consumption in function "updateStakingPool" 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

```
1667 uint256 length = poolInfo.length;
1668 uint256 points = 0;
1669 for (uint256 pid = 1; pid < length; ++pid) {
1670 points = points.add(poolInfo[pid].allocPoint);
1671 }
```

MEDIUM

SWC-128

Loop over unbounded data structure.

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

Source file

masterChef.sol

Locations

```
1711 function massUpdatePools() public {
1712 uint256 length = poolInfo.length;
1713 for (uint256 pid = 0; pid < length; ++pid) {
1714 updatePool(pid);
1715 }
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.4.0"". 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

```
3 // SPDX-License-Identifier: MIT
4
5 pragma solidity ^0.4.0;
6
7 /**
```

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

```
1851 UserInfo storage user = userInfo[_pid][msg.sender];
1852 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1853 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1854 user.amount = 0;
1855 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

```
1852 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1853 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1854 user.amount = 0;
1855 user.rewardDebt = 0;
1856 }
```

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

```
1853 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1854 user.amount = 0;
1855 user.rewardDebt = 0;
1856 }
```

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

```
1765 if (_amount > 0) {
1766 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1767 if(pool.depositFeeBP > 0){
1768 uint256 depositFee = _amount.mul(pool.depositFeeBP).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

```
1775 user.amount = user.amount.add(_amount).sub(depositFee);
1776 }else{
1777 user.amount = user.amount.add(_amount);
1778 }
1779 }
```

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

```
1775 user.amount = user.amount.add(_amount).sub(depositFee);
1776 }else{
1777 user.amount = user.amount.add(_amount);
1778 }
1779 }
```

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

```
1778 }
1779 }
1780 user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
1781 emit Deposit(msg.sender, _pid, _amount);
1782 }
```

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

```
1778 }
1779 }
1780 user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
1781 emit Deposit(msg.sender, _pid, _amount);
1782 }
```

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

```
1778 }
1779 }
1780 user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
1781 emit Deposit(msg.sender, _pid, _amount);
1782 }
```

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

```
1766 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1767 if(pool.depositFeeBP > 0){
1768 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1769
1770 pool.lpToken.safeTransfer(treasuryaddr, depositFee.div(2));
```

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

```
1768 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1769
1770 pool.lpToken.safeTransfer(treasuryaddr, depositFee.div(2));
1771 pool.lpToken.safeTransfer(address(this), depositFee.div(4));
1772 pool.lpToken.safeTransfer(DLPHaddr, depositFee.div(4).div(4).mul(3));
```

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

```
1768 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1769
1770 pool.lpToken.safeTransfer(treasuryaddr, depositFee.div(2));
1771 pool.lpToken.safeTransfer(address(this), depositFee.div(4));
1772 pool.lpToken.safeTransfer(DLPHaddr, depositFee.div(4).div(4).mul(3));
```

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

```
1769
1770 pool.lpToken.safeTransfer(treasuryaddr, depositFee.div(2));
1771 pool.lpToken.safeTransfer(address(this), depositFee.div(4));
1772 pool.lpToken.safeTransfer(DLPHaddr, depositFee.div(4).div(4).mul(3));
1773 pool.lpToken.safeTransfer(devaddr, depositFee.div(4).div(4));
```

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

```
1770 pool.lpToken.safeTransfer(treasuryaddr, depositFee.div(2));
1771 pool.lpToken.safeTransfer(address(this), depositFee.div(4));
1772 pool.lpToken.safeTransfer(DLPHaddr, depositFee.div(4).div(4).mul(3));
1773 pool.lpToken.safeTransfer(devaddr, depositFee.div(4).div(4));
```

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

```
1771 pool.lpToken.safeTransfer(address(this), depositFee.div(4));
1772 pool.lpToken.safeTransfer(DLPHaddr, depositFee.div(4).div(4).mul(3));
1773 pool.lpToken.safeTransfer(devaddr, depositFee.div(4).div(4));
1774
1775 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

```
1771 pool.lpToken.safeTransfer(address(this), depositFee.div(4));
1772 pool.lpToken.safeTransfer(DLPHaddr, depositFee.div(4).div(4).mul(3));
1773 pool.lpToken.safeTransfer(devaddr, depositFee.div(4).div(4));
1774
1775 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

```
1773 pool.lpToken.safeTransfer(devaddr, depositFee.div(4).div(4));
1774
1775 user.amount = user.amount.add(_amount).sub(depositFee);
1776 }else{
1777 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

```
1773  pool.lpToken.safeTransfer(devaddr, depositFee.div(4).div(4));
1774
1775  user.amount = user.amount.add(_amount).sub(depositFee);
1776  }else{
1777  user.amount = user.amount.add(_amount);
```

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

```
1141  returns (uint256)
1142  {
1143  require(blockNumber < block.number, "DLPH::getPriorVotes: not yet determined");
1144
1145  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

```
1214  internal
1215  {
1216  uint32 blockNumber = safe32(block.number, "DLPH::_writeCheckpoint: block number exceeds 32 bits");
1217
1218  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

```
1410 returns (uint256)
1411 {
1412 require(blockNumber < block.number, "DLPH::getPriorVotes: not yet determined");
1413
1414 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

```
1483 internal
1484 {
1485 uint32 blockNumber = safe32(block.number, "DLPH::_writeCheckpoint: block number exceeds 32 bits");
1486
1487 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

```
1637 massUpdatePools();
1638 }
1639 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1640 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1641 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

```
1637 massUpdatePools();
1638 }
1639 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1640 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1641 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

```
1696 DLPHPerBlock = 25*10**16;
1697
1698 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1699 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1700 uint256 DLPHReward = multiplier.mul(DLPHPerBlock).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

```
1697
1698 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1699 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1700 uint256 DLPHReward = multiplier.mul(DLPHPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1701 accDLPHPerShare = accDLPHPerShare.add(DLPHReward.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

```
1720 function updatePool(uint256 _pid) public {
1721 PoolInfo storage pool = poolInfo[_pid];
1722 if (block.number <= pool.lastRewardBlock) {
1723 return;
1724 }
```

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

```
1725 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1726 if (lpSupply == 0) {
1727 pool.lastRewardBlock = block.number;
1728 return;
1729 }
```

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

```
1736 DLPHPerBlock = 25*10**16;
1737
1738 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1739 uint256 DLPHReward = multiplier.mul(DLPHPerBlock).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

```
1744 }
1745 pool.accDLPHPerShare = pool.accDLPHPerShare.add(DLPHReward.mul(1e12).div(lpSupply));
1746 pool.lastRewardBlock = block.number;
1747 }
```

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

```
1687 UserInfo storage user = userInfo[_pid][_user];
1688 uint256 accDLPHPerShare = pool.accDLPHPerShare;
1689 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1690
1691 if(dayTimestamp < now && now < firstWeekTimestamp)
```

Source file

masterChef.sol

Locations

```
1526 //
1527 // Have fun reading it. Hopefully it's bug-free. God bless.
1528 contract MasterChef is Ownable {
1529 using SafeMath for uint256;
1530 using SafeBEP20 for IBEP20;
1531
1532 // Info of each user.
1533 struct UserInfo {
1534 uint256 amount; // How many LP tokens the user has provided.
1535 uint256 rewardDebt; // Reward debt. See explanation below.
1536 //
1537 // We do some fancy math here. Basically, any point in time, the amount of DLPHs
1538 // entitled to a user but is pending to be distributed is:
1539 //
1540 // pending reward = (user.amount * pool.accDLPHPerShare) - user.rewardDebt
1541 //
1542 // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
1543 // 1. The pool's `accDLPHPerShare` (and `lastRewardBlock`) gets updated.
1544 // 2. User receives the pending reward sent to his/her address.
1545 // 3. User's `amount` gets updated.
1546 // 4. User's `rewardDebt` gets updated.
1547 }
1548
1549 // Info of each pool.
1550 struct PoolInfo {
1551 IBEP20 lpToken; // Address of LP token contract.
1552 uint256 allocPoint; // How many allocation points assigned to this pool. DLPHs to distribute per block.
1553 uint256 lastRewardBlock; // Last block number that DLPHs distribution occurs.
1554 uint256 accDLPHPerShare; // Accumulated DLPHs per share, times 1e12. See below.
1555 uint16 depositFeeBP; // Deposit fee in basis points
1556 }
1557
1558 // The Dolphin Token!
1559 DLPHToken public DLPH;
1560 // The FOAM TOKEN!
1561 FoamBar public foam;
1562 // Dev address.
1563 address public devaddr;
1564 // Teasure address
1565 address public treasuryaddr;
1566 // Dolphin Tokens created per block.
1567 uint256 public DLPHPerBlock;
1568 // Bonus muliplier for early DLPH makers.
1569 uint256 public BONUS_MULTIPLIER;
1570
```

```
uint256 public dayTimestamp;
uint256 public firstWeekTimestamp;
uint256 public secondWeekTimestamp;
uint256 public thirdWeekTimestamp;

// 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 DLPH mining starts.
uint256 public startBlock;

event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);

constructor(
DLPHToken _DLPH,
FoamBar _foam,
address _devaddr,
address _treasuryaddr,
uint256 _DLPHPerBlock,
uint256 _startBlock,
uint256 _multiplier
) public {
DLPH = _DLPH;
foam = _foam;
devaddr = _devaddr;
treasuryaddr = _treasuryaddr;
DLPHPerBlock = _DLPHPerBlock;
startBlock = _startBlock;
BONUS_MULTIPLIER = _multiplier;

dayTimestamp = now + 1 days;
firstWeekTimestamp = now + 1 weeks;
secondWeekTimestamp = now + 2 weeks;
thirdWeekTimestamp = now + 3 weeks;

// staking pool
poolInfo.push(PoolInfo({
lpToken: _DLPH,
allocPoint: 1000,
lastRewardBlock: startBlock,
accDLPHPerShare: 0,
depositFeeBP: 0
}));

totalAllocPoint = 1000;

}

function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
BONUS_MULTIPLIER = multiplierNumber;
}

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, bool _withUpdate) public onlyOwner {
require(_depositFeeBP <= 10000, "add: invalid deposit fee basis points");
if (_withUpdate) {
massUpdatePools();
}
uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
totalAllocPoint = totalAllocPoint.add(_allocPoint);
poolInfo.push(PoolInfo({
lpToken: _lpToken,
allocPoint: _allocPoint,
lastRewardBlock: lastRewardBlock,
accDLPHPerShare: 0,
depositFeeBP: _depositFeeBP
}));
updateStakingPool();
}

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

function updateStakingPool() internal {
uint256 length = poolInfo.length;
uint256 points = 0;
for (uint256 pid = 1; pid < length; ++pid) {
points = points.add(poolInfo[pid].allocPoint);
}
if (points != 0) {
points = points.div(3);
totalAllocPoint = totalAllocPoint.sub(poolInfo[0].allocPoint).add(points);
poolInfo[0].allocPoint = points;
}
}

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

// View function to see pending DLPHs on frontend.
function pendingDLPH(uint256 _pid, address _user) external returns (uint256) {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][_user];
uint256 accDLPHPerShare = pool.accDLPHPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));

if(dayTimestamp < now && now < firstWeekTimestamp)
DLPHPerBlock = 1*10**18;
else if(firstWeekTimestamp < now && now < secondWeekTimestamp)
DLPHPerBlock = 5*10**17;
else if(now > secondWeekTimestamp)
DLPHPerBlock = 25*10**16;
```

```
1697
1698 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1699 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1700 uint256 DLPHReward = multiplier.mul(DLPHPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1701 accDLPHPerShare = accDLPHPerShare.add(DLPHReward.mul(1e12).div(lpSupply));
1702 }
1703 return user.amount.mul(accDLPHPerShare).div(1e12).sub(user.rewardDebt);
1704 }
1705
1706 function getPerBlock() public view returns (uint256) {
1707 return DLPHPerBlock;
1708 }
1709
1710 // Update reward variables for all pools. Be careful of gas spending!
1711 function massUpdatePools() public {
1712 uint256 length = poolInfo.length;
1713 for (uint256 pid = 0; pid < length; ++pid) {
1714 updatePool(pid);
1715 }
1716 }
1717
1718
1719 // Update reward variables of the given pool to be up-to-date.
1720 function updatePool(uint256 _pid) public {
1721 PoolInfo storage pool = poolInfo[_pid];
1722 if (block.number <= pool.lastRewardBlock) {
1723 return;
1724 }
1725 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1726 if (lpSupply == 0) {
1727 pool.lastRewardBlock = block.number;
1728 return;
1729 }
1730
1731 if(dayTimestamp < now && now < firstWeekTimestamp)
1732 DLPHPerBlock = 1*10**18;
1733 else if(firstWeekTimestamp < now && now < secondWeekTimestamp)
1734 DLPHPerBlock = 5*10**17;
1735 else if(now > secondWeekTimestamp)
1736 DLPHPerBlock = 25*10**16;
1737
1738 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1739 uint256 DLPHReward = multiplier.mul(DLPHPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1740
1741 if (DLPHReward > 0) {
1742 DLPH.mint(devaddr, DLPHReward.div(10));
1743 DLPH.mint(address(foam), DLPHReward);
1744 }
1745 pool.accDLPHPerShare = pool.accDLPHPerShare.add(DLPHReward.mul(1e12).div(lpSupply));
1746 pool.lastRewardBlock = block.number;
1747 }
1748
1749 // Deposit LP tokens to MasterChef for DLPH allocation.
1750 function deposit(uint256 _pid, uint256 _amount) public {
1751
1752 require (_pid != 0, 'deposit DLPH by staking');
1753
1754 PoolInfo storage pool = poolInfo[_pid];
1755 UserInfo storage user = userInfo[_pid][msg.sender];
1756 address DLPHaddr = address(DLPH);
1757
1758 updatePool(_pid);
1759 if (user.amount > 0) {
```

```
uint256 pending = user.amount.mul(pool.accDLPHPerShare).div(1e12).sub(user.rewardDebt);
if(pending > 0) {
safeDLPHTransfer(msg.sender, pending);
}
}
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if(pool.depositFeeBP > 0){
uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);

pool.lpToken.safeTransfer(treasuryaddr, depositFee.div(2));
pool.lpToken.safeTransfer(address(this), depositFee.div(4));
pool.lpToken.safeTransfer(DLPHaddr, depositFee.div(4).div(4).mul(3));
pool.lpToken.safeTransfer(devaddr, depositFee.div(4).div(4));

user.amount = user.amount.add(_amount).sub(depositFee);
}else{
user.amount = user.amount.add(_amount);
}
}
user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}

// Withdraw LP tokens from MasterChef.
function withdraw(uint256 _pid, uint256 _amount) public {

require (_pid != 0, 'withdraw DLPH by unstaking');
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.accDLPHPerShare).div(1e12).sub(user.rewardDebt);
if(pending > 0) {
safeDLPHTransfer(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.accDLPHPerShare).div(1e12);
emit Withdraw(msg.sender, _pid, _amount);
}

// Stake Dolphin Tokens to MasterChef
function enterStaking(uint256 _amount) public {
PoolInfo storage pool = poolInfo[0];
UserInfo storage user = userInfo[0][msg.sender];
updatePool(0);
if (user.amount > 0) {
uint256 pending = user.amount.mul(pool.accDLPHPerShare).div(1e12).sub(user.rewardDebt);
if(pending > 0) {
safeDLPHTransfer(msg.sender, pending);
}
}
if(_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
user.amount = user.amount.add(_amount);
}
user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);

if (_amount > 0)
```

```
1823  foam.mint(msg.sender, _amount);
1824
1825  emit Deposit(msg.sender, 0, _amount);
1826  }
1827
1828  // Withdraw Dolphin Tokens from STAKING.
1829  function leaveStaking(uint256 _amount) public {
1830  PoolInfo storage pool = poolInfo[0];
1831  UserInfo storage user = userInfo[0][msg.sender];
1832  require(user.amount >= _amount, "withdraw: not good");
1833  updatePool(0);
1834  uint256 pending = user.amount.mul(pool.accDLPHPerShare).div(1e12).sub(user.rewardDebt);
1835  if(pending > 0) {
1836  safeDLPHTransfer(msg.sender, pending);
1837  }
1838  if(_amount > 0) {
1839  user.amount = user.amount.sub(_amount);
1840  pool.lpToken.safeTransfer(address(msg.sender), _amount);
1841  }
1842  user.rewardDebt = user.amount.mul(pool.accDLPHPerShare).div(1e12);
1843
1844  foam.burn(msg.sender, _amount);
1845  emit Withdraw(msg.sender, 0, _amount);
1846  }
1847
1848  // Withdraw without caring about rewards. EMERGENCY ONLY.
1849  function emergencyWithdraw(uint256 _pid) public {
1850  PoolInfo storage pool = poolInfo[_pid];
1851  UserInfo storage user = userInfo[_pid][msg.sender];
1852  pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1853  emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1854  user.amount = 0;
1855  user.rewardDebt = 0;
1856  }
1857
1858  // Safe DLPH transfer function, just in case if rounding error causes pool to not have enough DLPHs.
1859  function safeDLPHTransfer(address _to, uint256 _amount) internal {
1860  foam.safeDLPHTransfer(_to, _amount);
1861  }
1862
1863  // Update dev address by the previous dev.
1864  function dev(address _devaddr) public {
1865  require(msg.sender == devaddr, "dev: wut?");
1866  devaddr = _devaddr;
1867  }
1868  }
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "sqrt" in contract "SafeMath" 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

```
181 z = y;
182 uint256 x = y / 2 + 1;
183 while (x < z) {
184 z = x;
185 x = (y / x + x) / 2;
```

LOW

SWC-128

Potentially unbounded data structure passed to builtin.

Gas consumption in function "delegateBySig" in contract "DLPHToken" 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

```
1085 abi.encode(
1086 DOMAIN_TYPEHASH,
1087 keccak256(bytes(name())),
1088 getChainId(),
1089 address(this)
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "getPriorVotes" in contract "DLPHToken" 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

```
1160 uint32 lower = 0;
1161 uint32 upper = nCheckpoints - 1;
1162 while (upper > lower) {
1163 uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
1164 Checkpoint memory cp = checkpoints[account][center];
```

LOW

SWC-128

Potentially unbounded data structure passed to builtin.

Gas consumption in function "delegateBySig" in contract "FoamBar" 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

```
1354 abi.encode(
1355 DOMAIN_TYPEHASH,
1356 keccak256(bytes(name())),
1357 getChainId(),
1358 address(this)
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "getPriorVotes" in contract "FoamBar" 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

```
1429 uint32 lower = 0;
1430 uint32 upper = nCheckpoints - 1;
1431 while (upper > lower) {
1432 uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
1433 Checkpoint memory cp = checkpoints[account][center];
```