MythX

REPORT 605B45A2DD3EBE0012F5C354

Created	Wed Mar 24 2021 13:58:58 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	contact@comos.finance

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
6f45354d-40df-4e90-a01e-ba03787555e8	MasterChef.sol	53

Started	Wed Mar 24 2021 13:59:02 GMT+0000 (Coordinated Universal Time)
Finished	Wed Mar 24 2021 14:14:47 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Remythx
Main Source File	MasterChef.Sol

DETECTED VULNERABILITIES

(HIGH	(MEDIUM	(LOW
0	23	30

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

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

```
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

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

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

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

```
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   /**
```

```
563   function transferOwnership(address newOwner) public virtual onlyOwner {
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

MasterChef.sol

Locations

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

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

MasterChef.sol

Locations

```
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

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

```
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   /**
```

```
655   function totalSupply() public override view returns (uint256) {
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

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

SWC-000

Function could be marked as external.

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

Source file

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

SWC-000

Function could be marked as external.

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

Source file

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

SWC-000

Function could be marked as external.

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

Source file

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

SWC-000

Function could be marked as external.

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

Source file

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

SWC-000

Function could be marked as external.

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

Source file

MasterChef.sol

Locations

```
869   address public constant BURN_ADDRESS = 0x000000000000000000000000000000000000dEaD;
870   /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
871   function mint(address _to, uint256 _amount) public onlyOwner {
872   _mint(_to, _amount);
873   _moveDelegates(address(0), _delegates[_to], _amount);
874   }
875
876   /// @dev overrides transfer function to meet tokenomics of COMOS
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

MasterChef.sol

Locations

```
1278
1279    // Deposit LP tokens to MasterChef for EGG allocation.
1280    function deposit(uint256 _pid, uint256 _amount) public {
1281    PoolInfo storage pool = poolInfo[_pid];
1282    UserInfo storage user = userInfo[_pid][msg.sender];
1283    updatePool(_pid);
1284    if (user.amount > 0) {
1285    uint256 pending = user.amount.mul(pool.accEggPerShare).div(1e12).sub(user.rewardDebt);
1286    if(pending > 0) {
1287    safeEggTransfer(msg.sender, pending);
1288    }
1289    }
1290    if(_amount > 0) {
1291    pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1292    if(pool.depositFeeBP > 0){
1293    uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1294    pool.lpToken.safeTransfer(feeAddress, depositFee);
1295    user.amount = user.amount.add(_amount).sub(depositFee);
1296    }else{
1297    user.amount = user.amount.add(_amount);
1298    }
1299    }
1300    user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
1301    emit Deposit(msg.sender, _pid, _amount);
1302    }
1303
1304    // 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

```
1303
1304    // Withdraw LP tokens from MasterChef.
1305    function withdraw(uint256 _pid, uint256 _amount) public {
1306    PoolInfo storage pool = poolInfo[_pid];
1307    UserInfo storage user = userInfo[_pid][msg.sender];
1308    require(user.amount >= _amount, "withdraw: not good");
1309    updatePool(_pid);
1310    uint256 pending = user.amount.mul(pool.accEggPerShare).div(1e12).sub(user.rewardDebt);
1311    if(pending > 0) {
1312    safeEggTransfer(msg.sender, pending);
1313    }
1314    if(_amount > 0) {
1315    user.amount = user.amount.sub(_amount);
1316    pool.lpToken.safeTransfer(address(msg.sender), _amount);
1317    }
1318    user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
1319    emit Withdraw(msg.sender, _pid, _amount);
1320    }
1321
1322    // 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

```
1321
1322    // Withdraw without caring about rewards. EMERGENCY ONLY.
1323    function emergencyWithdraw(uint256 _pid) public {
1324    PoolInfo storage pool = poolInfo[_pid];
1325    UserInfo storage user = userInfo[_pid][msg.sender];
1326    uint256 amount = user.amount;
1327    user.amount = 0;
1328    user.rewardDebt = 0;
1329    pool.lpToken.safeTransfer(address(msg.sender), amount);
1330    emit EmergencyWithdraw(msg.sender, _pid, amount);
1331    }
1332
1333    // Safe comos transfer function, just in case if rounding error causes pool to not have enough EGGs.
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

`MasterChef.sol`

Locations

```
1342
1343    // Update dev address by the previous dev.
1344    function dev(address _devaddr) public {
1345    require(msg.sender == devaddr, "dev: wut?");
1346    devaddr = _devaddr;
1347    }
1348
1349    function setFeeAddress(address _feeAddress) public{
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

`MasterChef.sol`

Locations

```
1347    }
1348
1349    function setFeeAddress(address _feeAddress) public{
1350    require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
1351    feeAddress = _feeAddress;
1352    }
1353
1354    //Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
```

MEDIUM

SWC-000

Function could be marked as external.

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

Source file

`MasterChef.sol`

Locations

```
1353
1354    //Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
1355    function updateEmissionRate(uint256 _comosPerBlock) public onlyOwner {
1356    massUpdatePools();
1357    comosPerBlock = _comosPerBlock;
1358    }
1359    }
```

MEDIUM

SWC-113

Multiple calls are executed in the same transaction.

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

Source file

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

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

```
1253    function massUpdatePools() public {
1254    uint256 length = poolInfo.length;
1255    for (uint256 pid = 0; pid < length; ++pid) {
1256    updatePool(pid);
1257    }
```

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

```
1290  if(_amount > 0) {
1291  pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1292  if(pool.depositFeeBP > 0){
1293  uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1294  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

```
1295  user.amount = user.amount.add(_amount).sub(depositFee);
1296  }else{
1297  user.amount = user.amount.add(_amount);
1298  }
1299  }
```

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

```
1298    }
1299    }
1300    user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
1301    emit Deposit(msg.sender, _pid, _amount);
1302    }
```

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

```
1295    user.amount = user.amount.add(_amount).sub(depositFee);
1296    }else{
1297    user.amount = user.amount.add(_amount);
1298    }
1299    }
```

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

```
1298    }
1299    }
1300    user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
1301    emit Deposit(msg.sender, _pid, _amount);
1302    }
```

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

```
1298        }
1299        }
1300        user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
1301        emit Deposit(msg.sender, _pid, _amount);
1302        }
```

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

```
1291        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1292        if(pool.depositFeeBP > 0){
1293        uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1294        pool.lpToken.safeTransfer(feeAddress, depositFee);
1295        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

```
1292        if(pool.depositFeeBP > 0){
1293        uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1294        pool.lpToken.safeTransfer(feeAddress, depositFee);
1295        user.amount = user.amount.add(_amount).sub(depositFee);
1296        }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

```
1292   if(pool.depositFeeBP > 0){
1293   uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1294   pool.lpToken.safeTransfer(feeAddress, depositFee);
1295   user.amount = user.amount.add(_amount).sub(depositFee);
1296   }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

```
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

```
1293   uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1294   pool.lpToken.safeTransfer(feeAddress, depositFee);
1295   user.amount = user.amount.add(_amount).sub(depositFee);
1296   }else{
1297   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

```
1293    uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1294    pool.lpToken.safeTransfer(feeAddress, depositFee);
1295    user.amount = user.amount.add(_amount).sub(depositFee);
1296    }else{
1297    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

```
1316    pool.lpToken.safeTransfer(address(msg.sender), _amount);
1317    }
1318    user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
1319    emit Withdraw(msg.sender, _pid, _amount);
1320    }
```

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

```
1316    pool.lpToken.safeTransfer(address(msg.sender), _amount);
1317    }
1318    user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
1319    emit Withdraw(msg.sender, _pid, _amount);
1320    }
```

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

```
1316    pool.lpToken.safeTransfer(address(msg.sender), _amount);
1317    }
1318    user.rewardDebt = user.amount.mul(pool.accEggPerShare).div(1e12);
1319    emit Withdraw(msg.sender, _pid, _amount);
1320    }
```

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

```
1024    returns (uint256)
1025    {
1026    require(blockNumber < block.number, "COMOS::getPriorVotes: not yet determined");
1027
1028    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

```
1097    internal
1098    {
1099    uint32 blockNumber = safe32(block.number, "TOKEN::_writeCheckpoint: block number exceeds 32 bits");
1100
1101    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

```
1209    massUpdatePools();
1210    }
1211    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1212    totalAllocPoint = totalAllocPoint.add(_allocPoint);
1213    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

```
1209    massUpdatePools();
1210    }
1211    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1212    totalAllocPoint = totalAllocPoint.add(_allocPoint);
1213    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

```
1242    uint256 accEggPerShare = pool.accEggPerShare;
1243    uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1244    if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1245    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1246    uint256 comosReward = multiplier.mul(comosPerBlock).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

```
1243   uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1244   if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1245   uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1246   uint256 comosReward = multiplier.mul(comosPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1247   accEggPerShare = accEggPerShare.add(comosReward.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

```
1261   function updatePool(uint256 _pid) public {
1262   PoolInfo storage pool = poolInfo[_pid];
1263   if (block.number <= pool.lastRewardBlock) {
1264   return;
1265   }
```

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

```
1266   uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1267   if (lpSupply == 0 || pool.allocPoint == 0) {
1268   pool.lastRewardBlock = block.number;
1269   return;
1270   }
```

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

```
1269    return;
1270    }
1271    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1272    uint256 comosReward = multiplier.mul(comosPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1273    comos.mint(devaddr, comosReward.div(10));
```

LOW

SWC-120

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

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

Source file

MasterChef.sol

Locations

```
1274    comos.mint(address(this), comosReward);
1275    pool.accEggPerShare = pool.accEggPerShare.add(comosReward.mul(1e12).div(lpSupply));
1276    pool.lastRewardBlock = block.number;
1277    }
1278
```

Source file

MasterChef.sol

Locations

```
1264   return;
1265   }
1266   uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1267   if (lpSupply == 0 || pool.allocPoint == 0) {
1268   pool.lastRewardBlock = block.number;
```

Source file

MasterChef.sol

Locations

```
1130   //
1131   // Have fun reading it. Hopefully it's bug-free. God bless.
1132   contract MasterChef is Ownable {
1133   using SafeMath for uint256;
1134   using SafeBEP20 for IBEP20;
1135
1136   // Info of each user.
1137   struct UserInfo {
1138   uint256 amount; // How many LP tokens the user has provided.
1139   uint256 rewardDebt; // Reward debt. See explanation below.
1140   //
1141   // We do some fancy math here. Basically, any point in time, the amount of EGGs
1142   // entitled to a user but is pending to be distributed is:
1143   //
1144   // pending reward = (user.amount * pool.accEggPerShare) - user.rewardDebt
1145   //
1146   // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
1147   // 1. The pool's `accEggPerShare` (and `lastRewardBlock`) gets updated.
1148   // 2. User receives the pending reward sent to his/her address.
1149   // 3. User's `amount` gets updated.
1150   // 4. User's `rewardDebt` gets updated.
1151   }
1152
1153   // Info of each pool.
1154   struct PoolInfo {
1155   IBEP20 lpToken; // Address of LP token contract.
1156   uint256 allocPoint; // How many allocation points assigned to this pool. EGGs to distribute per block.
1157   uint256 lastRewardBlock; // Last block number that EGGs distribution occurs.
1158   uint256 accEggPerShare; // Accumulated EGGs per share, times 1e12. See below.
1159   uint16 depositFeeBP; // Deposit fee in basis points
1160   }
1161
1162   // The COMOS TOKEN!
1163   ComosToken public comos;
1164   // Dev address.
1165   address public devaddr;
1166   // EGG tokens created per block.
1167   uint256 public comosPerBlock;
1168   // Bonus muliplier for early comos makers.
1169   uint256 public constant BONUS_MULTIPLIER = 1;
1170   // Deposit Fee address
1171   address public feeAddress;
1172
1173   // Info of each pool.
1174   PoolInfo[] public poolInfo;
```

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

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

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

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

constructor(
    ComosToken _comos,
    address _devaddr,
    address _feeAddress,
    uint256 _comosPerBlock,
    uint256 _startBlock
) public {
    comos = _comos;
    devaddr = _devaddr;
    feeAddress = _feeAddress;
    comosPerBlock = _comosPerBlock;
    startBlock = _startBlock;
}

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,
        accEggPerShare: 0,
        depositFeeBP: _depositFeeBP
    }));
}

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

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

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

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

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

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

```solidity
        emit Deposit(msg.sender, _pid, _amount);
    }


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


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


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


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


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


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

LOW

SWC-128

Potentially unbounded data structure passed to builtin.

Gas consumption in function "delegateBySig" in contract "ComosToken" 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

```
968    abi.encode(
969    DOMAIN_TYPEHASH,
970    keccak256(bytes(name())),
971    getChainId(),
972    address(this)
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "getPriorVotes" in contract "ComosToken" 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

```
1043    uint32 lower = 0;
1044    uint32 upper = nCheckpoints - 1;
1045    while (upper > lower) {
1046    uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
1047    Checkpoint memory cp = checkpoints[account][center];
```