MythX

REPORT 606C7D1E00B7D90018CAF67B

Created	Tue Apr 06 2021 15:24:14 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	takodefi@gmail.com

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
2f6e32ae-36d3-45e1-b773-8edfeeee430e	MasterChef_Flat.sol	60

Analysis 2f6e32ae-36d3-45e1-b773-8edfeeee430e MythX

Started	Tue Apr 06 2021 15:24:15 GMT+0000 (Coordinated Universal Time)
Finished	Tue Apr 06 2021 16:10:00 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	MasterChef_Flat.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	23	37

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_Flat.sol

Locations

```
574 * thereby removing any functionality that is only available to the owner.
575 */
576 function renounceOwnership() public virtual onlyOwner {
577 emit OwnershipTransferred(_owner, address(0));
578 _owner = address(0);
579 }
580
581 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
583 * Can only be called by the current owner.
584 */
585 function transferOwnership(address newOwner) public virtual onlyOwner {
586 require(newOwner != address(0), "Ownable: new owner is the zero address");
587 emit OwnershipTransferred(_owner, newOwner);
588 _owner = newOwner;
589 }
590 }
591
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
731 * name.
732 */
733 function symbol() public override view returns (string memory) {
734 return _symbol;
735 }
736
737 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
738 * @dev Returns the number of decimals used to get its user representation.
739 */
740 function decimals() public override view returns (uint8) {
741 return _decimals;
742 }
743
744 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
745 * @dev See {BEP20-totalSupply}.
746 */
747 function totalSupply() public override view returns (uint256) {
748 return _totalSupply;
749 }
750
751 /**
```

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_Flat.sol

Locations

```
764 * - the caller must have a balance of at least `amount`.
765 */
766 function transfer(address recipient, uint256 amount) public override returns (bool) {
767 _transfer(_msgSender(), recipient, amount);
768 return true;
769 }
770
771 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
772 * @dev See {BEP20-allowance}.
773 */
774 function allowance(address owner, address spender) public override view returns (uint256) {
775 return _allowances[owner][spender];
776 }
777
778 /**
```

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_Flat.sol

Locations

```
783 * - `spender` cannot be the zero address.
784 */
785 function approve(address spender, uint256 amount) public override returns (bool) {
786 _approve(_msgSender(), spender, amount);
787 return true;
788 }
789
790 /**
```

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_Flat.sol

Locations

```
800 * `amount`.
801 */
802 function transferFrom (address sender, address recipient, uint256 amount) public override returns (bool) {
803 _transfer(sender, recipient, amount);
804 _approve(
805 sender,
806 _msgSender(),
807 _allowances[sender][_msgSender()].sub(amount, 'BEP20: transfer amount exceeds allowance')
808 );
809 return true;
810 }
811
812 /**
```

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_Flat.sol

Locations

```
822 * - `spender` cannot be the zero address.
823 */
824 function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
825 _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
826 return true;
827 }
828
829 /**
```

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_Flat.sol

Locations

```
841 * `subtractedValue`.
842 */
843 function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
844 _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, 'BEP20: decreased allowance below zero'));
845 return true;
846 }
847
848 /**
```

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_Flat.sol

Locations

```
854 * - `msg.sender` must be the token owner
855 */
856 function mint(uint256 amount) public onlyOwner returns (bool) {
857 _mint(_msgSender(), amount);
858 return true;
859 }
860
861 /**
```

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_Flat.sol

Locations

```
959 contract TakoToken is BEP20('Tako Token', 'TAKO') {
960 /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
961 function mint(address _to, uint256 _amount) public onlyOwner {
962 _mint(_to, _amount);
963 _moveDelegates(address(0), _delegates[_to], _amount);
964 }
965
966 // Copied and modified from YAM code:
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
1294
1295 // Add a new lp to the pool. Can only be called by the owner.
1296 function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner nonDuplicated(_lpToken) {
1297 require(_depositFeeBP <= 10000, "add: invalid deposit fee basis points");
1298 if (_withUpdate) {
1299 massUpdatePools();
1300 }
1301 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1302 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1303 poolExistence[_lpToken] = true;
1304 poolInfo.push(PoolInfo({
1305 lpToken : _lpToken,
1306 allocPoint : _allocPoint,
1307 lastRewardBlock : lastRewardBlock,
1308 accTakoPerShare : 0,
1309 depositFeeBP : _depositFeeBP
1310 }));
1311 }
1312
1313 // Update the given pool's TAKO allocation point and deposit fee. Can only be called by the owner.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
1312
1313 // Update the given pool's TAKO allocation point and deposit fee. Can only be called by the owner.
1314 function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {
1315 require(_depositFeeBP <= 10000, "set: invalid deposit fee basis points");
1316 if (_withUpdate) {
1317 massUpdatePools();
1318 }
1319 totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
1320 poolInfo[_pid].allocPoint = _allocPoint;
1321 poolInfo[_pid].depositFeeBP = _depositFeeBP;
1322 }
1323
1324 // Return reward multiplier over the given _from to _to block.
```

MEDIUM Function could be marked as external.

SWC-000 The function definition of "deposit" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

MasterChef_Flat.sol

Locations

```
1370
1371 // Deposit LP tokens to MasterChef for TAKO allocation.
1372 function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
1373 PoolInfo storage pool = poolInfo[_pid];
1374 UserInfo storage user = userInfo[_pid][msg.sender];
1375 updatePool(_pid);
1376 if (user.amount > 0) {
1377 uint256 pending = user.amount.mul(pool.accTakoPerShare).div(1e12).sub(user.rewardDebt);
1378 if (pending > 0) {
1379 safeTakoTransfer(msg.sender, pending);
1380 }
1381 }
1382 if (_amount > 0) {
1383 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1384 if (pool.depositFeeBP > 0) {
1385 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1386 pool.lpToken.safeTransfer(feeAddress, depositFee);
1387 user.amount = user.amount.add(_amount).sub(depositFee);
1388 } else {
1389 user.amount = user.amount.add(_amount);
1390 }
1391 }
1392 user.rewardDebt = user.amount.mul(pool.accTakoPerShare).div(1e12);
1393 emit Deposit(msg.sender, _pid, _amount);
1394 }
1395
1396 // 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_Flat.sol

Locations

```
1395
1396 // Withdraw LP tokens from MasterChef.
1397 function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
1398 PoolInfo storage pool = poolInfo[_pid];
1399 UserInfo storage user = userInfo[_pid][msg.sender];
1400 require(user.amount >= _amount, "withdraw: not good");
1401 updatePool(_pid);
1402 uint256 pending = user.amount.mul(pool.accTakoPerShare).div(1e12).sub(user.rewardDebt);
1403 if (pending > 0) {
1404 safeTakoTransfer(msg.sender, pending);
1405 }
1406 if (_amount > 0) {
1407 user.amount = user.amount.sub(_amount);
1408 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1409 }
1410 user.rewardDebt = user.amount.mul(pool.accTakoPerShare).div(1e12);
1411 emit Withdraw(msg.sender, _pid, _amount);
1412 }
1413
1414 // 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_Flat.sol

Locations

```
1413
1414 // Withdraw without caring about rewards. EMERGENCY ONLY.
1415 function emergencyWithdraw(uint256 _pid) public nonReentrant {
1416 PoolInfo storage pool = poolInfo[_pid];
1417 UserInfo storage user = userInfo[_pid][msg.sender];
1418 uint256 amount = user.amount;
1419 user.amount = 0;
1420 user.rewardDebt = 0;
1421 pool.lpToken.safeTransfer(address(msg.sender), amount);
1422 emit EmergencyWithdraw(msg.sender, _pid, amount);
1423 }
1424
1425 // Safe tako transfer function, just in case if rounding error causes pool to not have enough TAKOs.
```

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_Flat.sol

Locations

```
1436
1437 // Update dev address by the previous dev.
1438 function dev(address _devaddr) public {
1439 require(msg.sender == devaddr, "dev: wut?");
1440 devaddr = _devaddr;
1441 emit SetDevAddress(msg.sender, _devaddr);
1442 }
1443
1444 function setFeeAddress(address _feeAddress) public {
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
1442 }
1443
1444 function setFeeAddress(address _feeAddress) public {
1445 require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
1446 feeAddress = _feeAddress;
1447 emit SetFeeAddress(msg.sender, _feeAddress);
1448 }
1449
1450 //Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef_Flat.sol

Locations

```
1449
1450 //Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
1451 function updateEmissionRate(uint256 _takoPerBlock) public onlyOwner {
1452 massUpdatePools();
1453 takoPerBlock = _takoPerBlock;
1454 emit UpdateEmissionRate(msg.sender, _takoPerBlock);
1455 }
1456 }
```

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_Flat.sol

Locations

```
374
375 // solhint-disable-next-line avoid-low-level-calls
376 (bool success, bytes memory returndata) = target.call{ value: value }(data);
377 return _verifyCallResult(success, returndata, errorMessage);
378 }
```

MEDIUM

Loop over unbounded data structure.

SWC-128

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

Source file

MasterChef_Flat.sol

Locations

```
1344 function massUpdatePools() public {
1345 uint256 length = poolInfo.length;
1346 for (uint256 pid = 0; pid < length; ++pid) {
1347 updatePool(pid);
1348 }
```

LOW

A floating pragma is set.

SWC-103

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_Flat.sol

Locations

```
3 // SPDX-License-Identifier: MIT
4
5 pragma solidity >=0.6.0 <0.8.0;
6
7 /**
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""=0.6.4"". 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_Flat.sol

Locations

```
163 // File: contracts\libs\IBEP20.sol
164
165 pragma solidity >=0.6.4;
166
167 interface IBEP20 {
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""=0.6.2<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_Flat.sol

Locations

```
258 // File: @openzeppelin\contracts\utils\Address.sol
259
260 pragma solidity >=0.6.2 <0.8.0;
261
262 /**
```

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_Flat.sol

Locations

```
424 // File: contracts\libs\SafeBEP20.sol
425
426 pragma solidity >=0.6.0 <0.8.0;
427
428
```

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_Flat.sol

Locations

```
499 // File: node_modules\@openzeppelin\contracts\GSN\Context.sol
500
501 pragma solidity >=0.6.0 <0.8.0;
502
503 /*
```

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_Flat.sol

Locations

```
524 // File: @openzeppelin\contracts\access\Ownable.sol
525
526 pragma solidity >=0.6.0 <0.8.0;
527
528 /**
```

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_Flat.sol

Locations

```
592 // File: @openzeppelin\contracts\utils\ReentrancyGuard.sol
593
594 pragma solidity >=0.6.0 <0.8.0;
595
596 /**
```

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_Flat.sol

Locations

```
655 // File: contracts\libs\BEP20.sol
656
657 pragma solidity ^0.4.0;
658
659
```

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_Flat.sol

Locations

```
1382 if (_amount > 0) {
1383 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1384 if (pool.depositFeeBP > 0) {
1385 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1386 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_Flat.sol

Locations

```
1383 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1384 if (pool.depositFeeBP > 0) {
1385 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1386 pool.lpToken.safeTransfer(feeAddress, depositFee);
1387 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_Flat.sol

Locations

```
1384 if (pool.depositFeeBP > 0) {
1385 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1386 pool.lpToken.safeTransfer(feeAddress, depositFee);
1387 user.amount = user.amount.add(_amount).sub(depositFee);
1388 } 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_Flat.sol

Locations

```
1384 if (pool.depositFeeBP > 0) {
1385 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1386 pool.lpToken.safeTransfer(feeAddress, depositFee);
1387 user.amount = user.amount.add(_amount).sub(depositFee);
1388 } 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_Flat.sol

Locations

```
370 */
371 function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage) internal returns (bytes memory) {
372 require(address(this).balance >= value, "Address: insufficient balance for call");
373 require(isContract(target), "Address: call to non-contract");
374
```

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_Flat.sol

Locations

```
1385 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1386 pool.lpToken.safeTransfer(feeAddress, depositFee);
1387 user.amount = user.amount.add(_amount).sub(depositFee);
1388 } else {
1389 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_Flat.sol

Locations

```
1385 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1386 pool.lpToken.safeTransfer(feeAddress, depositFee);
1387 user.amount = user.amount.add(_amount).sub(depositFee);
1388 } else {
1389 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_Flat.sol

Locations

```
1390 }
1391 }
1392 user.rewardDebt = user.amount.mul(pool.accTakoPerShare).div(1e12);
1393 emit Deposit(msg.sender, _pid, _amount);
1394 }
```

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_Flat.sol

Locations

```
1390 }
1391 }
1392 user.rewardDebt = user.amount.mul(pool.accTakoPerShare).div(1e12);
1393 emit Deposit(msg.sender, _pid, _amount);
1394 }
```

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_Flat.sol

Locations

```
1390 }
1391 }
1392 user.rewardDebt = user.amount.mul(pool.accTakoPerShare).div(1e12);
1393 emit Deposit(msg.sender, _pid, _amount);
1394 }
```

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_Flat.sol

Locations

```
1387 user.amount = user.amount.add(_amount).sub(depositFee);
1388 } else {
1389 user.amount = user.amount.add(_amount);
1390 }
1391 }
```

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_Flat.sol

Locations

```
1387 | user.amount = user.amount.add(_amount).sub(depositFee);
1388 | } else {
1389 | user.amount = user.amount.add(_amount);
1390 | }
1391 | }
```

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_Flat.sol

Locations

```
1408 | pool.lpToken.safeTransfer(address(msg.sender), _amount);
1409 | }
1410 | user.rewardDebt = user.amount.mul(pool.accTakoPerShare).div(1e12);
1411 | emit Withdraw(msg.sender, _pid, _amount);
1412 | }
```

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_Flat.sol

Locations

```
1408 | pool.lpToken.safeTransfer(address(msg.sender), _amount);
1409 | }
1410 | user.rewardDebt = user.amount.mul(pool.accTakoPerShare).div(1e12);
1411 | emit Withdraw(msg.sender, _pid, _amount);
1412 | }
```

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_Flat.sol

Locations

```
1408 | pool.lpToken.safeTransfer(address(msg.sender), _amount);
1409 | }
1410 | user.rewardDebt = user.amount.mul(pool.accTakoPerShare).div(1e12);
1411 | emit Withdraw(msg.sender, _pid, _amount);
1412 | }
```

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_Flat.sol

Locations

```
649 | // By storing the original value once again, a refund is triggered (see
650 | // https://eips.ethereum.org/EIPS/eip-2200)
651 | _status = _NOT_ENTERED;
652 | }
653 | }
```

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_Flat.sol

Locations

```
1098 | returns (uint256)
1099 | {
1100 | require(blockNumber < block.number, "TAKO::getPriorVotes: not yet determined");
1101 |
1102 | 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_Flat.sol

Locations

```
1171 internal
1172 {
1173 uint32 blockNumber = safe32(block.number, "TAKO::_writeCheckpoint: block number exceeds 32 bits");
1174
1175 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_Flat.sol

Locations

```
1299 massUpdatePools();
1300 }
1301 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1302 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1303 poolExistence[_lpToken] = true;
```

LOW

SWC-120

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

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

Source file

MasterChef_Flat.sol

Locations

```
1299 massUpdatePools();
1300 }
1301 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1302 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1303 poolExistence[_lpToken] = true;
```

LOW

SWC-120

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

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

Source file

MasterChef_Flat.sol

Locations

```
1333 uint256 accTakoPerShare = pool.accTakoPerShare;
1334 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1335 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1336 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1337 uint256 takoReward = multiplier.mul(takoPerBlock).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_Flat.sol

Locations

```
1334 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1335 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1336 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1337 uint256 takoReward = multiplier.mul(takoPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1338 accTakoPerShare = accTakoPerShare.add(takoReward.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_Flat.sol

Locations

```
1352 function updatePool(uint256 _pid) public {
1353 PoolInfo storage pool = poolInfo[_pid];
1354 if (block.number <= pool.lastRewardBlock) {
1355 return;
1356 }
```

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_Flat.sol

Locations

```
1357 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1358 if (lpSupply == 0 || pool.allocPoint == 0) {
1359 pool.lastRewardBlock = block.number;
1360 return;
1361 }
```

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_Flat.sol

Locations

```
1360 return;
1361 }
1362 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1363 uint256 takoReward = multiplier.mul(takoPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1364 //8.3%~ to team funds
```

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_Flat.sol

Locations

```
1366 tako.mint(address(this), takoReward);
1367 pool.accTakoPerShare = pool.accTakoPerShare.add(takoReward.mul(1e12).div(lpSupply));
1368 pool.lastRewardBlock = block.number;
1369 }
1370
```

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_Flat.sol

Locations

```
1355 return;
1356 }
1357 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1358 if (lpSupply == 0 || pool.allocPoint == 0) {
1359 pool.lastRewardBlock = block.number;
```

Source file

MasterChef_Flat.sol

Locations

```
1212 //
1213 // Have fun reading it. Hopefully it's bug-free. Cthulhu bless.
1214 contract MasterChef is Ownable, ReentrancyGuard {
1215 using SafeMath for uint256;
1216 using SafeBEP20 for IBEP20;
1217
1218 // Info of each user.
1219 struct UserInfo {
1220 uint256 amount; // How many LP tokens the user has provided.
1221 uint256 rewardDebt; // Reward debt. See explanation below.
1222 //
1223 // We do some fancy math here. Basically, any point in time, the amount of TAKOs
1224 // entitled to a user but is pending to be distributed is:
1225 //
1226 // pending reward = (user.amount * pool.accTakoPerShare) - user.rewardDebt
1227 //
1228 // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
1229 // 1. The pool's `accTakoPerShare` (and `lastRewardBlock`) gets updated.
1230 // 2. User receives the pending reward sent to his/her address.
1231 // 3. User's `amount` gets updated.
1232 // 4. User's `rewardDebt` gets updated.
1233 }
1234
1235 // Info of each pool.
1236 struct PoolInfo {
1237 IBEP20 lpToken; // Address of LP token contract.
1238 uint256 allocPoint; // How many allocation points assigned to this pool. TAKOs to distribute per block.
1239 uint256 lastRewardBlock; // Last block number that TAKOs distribution occurs.
1240 uint256 accTakoPerShare; // Accumulated TAKOs per share, times 1e12. See below.
1241 uint16 depositFeeBP; // Deposit fee in basis points
1242 }
1243
1244 // The TAKO TOKEN!
1245 TakoToken public tako;
1246 // Dev address.
1247 address public devaddr;
1248 // TAKO tokens created per block.
1249 uint256 public takoPerBlock;
1250 // Bonus muliplier for early tako makers.
1251 uint256 public constant BONUS_MULTIPLIER = 1;
1252 // Deposit Fee address
1253 address public feeAddress;
1254
1255 // Info of each pool.
1256 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 TAKO mining starts.
uint256 public startBlock;

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

constructor(
TakoToken _tako,
address _devaddr,
address _feeAddress,
uint256 _takoPerBlock,
uint256 _startBlock
) public {
tako = _tako;
devaddr = _devaddr;
feeAddress = _feeAddress;
takoPerBlock = _takoPerBlock;
startBlock = _startBlock;
}

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

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

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

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

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

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

// View function to see pending TAKOs on frontend.
function pendingTako(uint256 _pid, address _user) external view returns (uint256) {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][_user];
uint256 accTakoPerShare = pool.accTakoPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 takoReward = multiplier.mul(takoPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
accTakoPerShare = accTakoPerShare.add(takoReward.mul(1e12).div(lpSupply));
}
return user.amount.mul(accTakoPerShare).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 takoReward = multiplier.mul(takoPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
//8.3%~ to team funds
tako.mint(devaddr, takoReward.div(12));
tako.mint(address(this), takoReward);
pool.accTakoPerShare = pool.accTakoPerShare.add(takoReward.mul(1e12).div(lpSupply));
pool.lastRewardBlock = block.number;
}

// Deposit LP tokens to MasterChef for TAKO allocation.
function deposit(uint256 _pid, uint256 _amount) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
updatePool(_pid);
if (user.amount > 0) {
uint256 pending = user.amount.mul(pool.accTakoPerShare).div(1e12).sub(user.rewardDebt);
if (pending > 0) {
safeTakoTransfer(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.accTakoPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}

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

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

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

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

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

```
1446  feeAddress = _feeAddress;
1447  emit SetFeeAddress(msg.sender, _feeAddress);
1448  }
1449
1450  //Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
1451  function updateEmissionRate(uint256 _takoPerBlock) public onlyOwner {
1452  massUpdatePools();
1453  takoPerBlock = _takoPerBlock;
1454  emit UpdateEmissionRate(msg.sender, _takoPerBlock);
1455  }
1456  }
```

LOW

SWC-128

Potentially unbounded data structure passed to builtin.

Gas consumption in function "delegateBySig" in contract "TakoToken" 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_Flat.sol

Locations

```
1042  abi.encode(
1043  DOMAIN_TYPEHASH,
1044  keccak256(bytes(name())),
1045  getChainId(),
1046  address(this)
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "getPriorVotes" in contract "TakoToken" 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_Flat.sol

Locations

```
1117  uint32 lower = 0;
1118  uint32 upper = nCheckpoints - 1;
1119  while (upper > lower) {
1120  uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
1121  Checkpoint memory cp = checkpoints[account][center];
```