MythX

REPORT 60B0B4A7BC0D4400198902F6

Created	Fri May 28 2021 09:15:19 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	60a9cb528bfa1219abf290cb

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
8bcacd37-94ab-4fe9-8702-5e4988e06a7a	MasterChef.sol	58

Analysis 8bcacd37-94ab-4fe9-8702-5e4988e06a7a MythX

Started	Fri May 28 2021 09:15:24 GMT+0000 (Coordinated Universal Time)
Finished	Fri May 28 2021 10:00:45 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	MasterChef.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	23	35

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

```
648 * thereby removing any functionality that is only available to the owner.
649 */
650 function renounceOwnership() public virtual onlyOwner {
651 emit OwnershipTransferred(_owner, address(0));
652 _owner = address(0);
653 }
654
655 /**
```

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

```
657 * Can only be called by the current owner.
658 */
659 function transferOwnership(address newOwner) public virtual onlyOwner {
660 require(newOwner != address(0), "Ownable: new owner is the zero address");
661 emit OwnershipTransferred(_owner, newOwner);
662 _owner = newOwner;
663 }
664 }
665
```

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

```
738 * name.
739 */
740 function symbol() public override view returns (string memory) {
741 return _symbol;
742 }
743
744 /**
```

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

```
745 * @dev Returns the number of decimals used to get its user representation.
746 */
747 function decimals() public override view returns (uint8) {
748 return _decimals;
749 }
750
751 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
752 * @dev See {BEP20-totalSupply}.
753 */
754 function totalSupply() public override view returns (uint256) {
755 return _totalSupply;
756 }
757
758 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
771 * - the caller must have a balance of at least `amount`.
772 */
773 function transfer(address recipient, uint256 amount) public override returns (bool) {
774 _transfer(_msgSender(), recipient, amount);
775 return true;
776 }
777
778 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
779 * @dev See {BEP20-allowance}.
780 */
781 function allowance(address owner, address spender) public override view returns (uint256) {
782 return _allowances[owner][spender];
783 }
784
785 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
790 * - `spender` cannot be the zero address.
791 */
792 function approve(address spender, uint256 amount) public override returns (bool) {
793 _approve(_msgSender(), spender, amount);
794 return true;
795 }
796
797 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
807 * `amount`.
808 */
809 function transferFrom (address sender, address recipient, uint256 amount) public override returns (bool) {
810 _transfer(sender, recipient, amount);
811 _approve(
812 sender,
813 _msgSender(),
814 _allowances[sender][_msgSender()].sub(amount, 'BEP20: transfer amount exceeds allowance')
815 );
816 return true;
817 }
818
819 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
829 * - `spender` cannot be the zero address.
830 */
831 function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
832 _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
833 return true;
834 }
835
836 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
848 * `subtractedValue`.
849 */
850 function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
851 _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, 'BEP20: decreased allowance below zero'));
852 return true;
853 }
854
855 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
861 * - `msg.sender` must be the token owner
862 */
863 function mint(uint256 amount) public onlyOwner returns (bool) {
864 _mint(_msgSender(), amount);
865 return true;
866 }
867
868 /**
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
965 contract GragasToken is BEP20('GragasFinance', 'GRAGAS') {
966 /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
967 function mint(address _to, uint256 _amount) public onlyOwner {
968 _mint(_to, _amount);
969 _moveDelegates(address(0), _delegates[_to], _amount);
970 }
971
972 // Copied and modified from YAM code:
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
1285 // Add a new lp to the pool. Can only be called by the owner.
1286 // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
1287 function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {
1288 require(_depositFeeBP <= 10000, "add: invalid deposit fee basis points");
1289 if (_withUpdate) {
1290 massUpdatePools();
1291 }
1292 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1293 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1294 poolInfo.push(PoolInfo({
1295 lpToken: _lpToken,
1296 allocPoint: _allocPoint,
1297 lastRewardBlock: lastRewardBlock,
1298 accGragasPerShare: 0,
1299 depositFeeBP: _depositFeeBP
1300 }));
1301 }
1302
1303 // Update the given pool's GRAGAS allocation point and deposit fee. Can only be called by the owner.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
1302
1303 // Update the given pool's GRAGAS allocation point and deposit fee. Can only be called by the owner.
1304 function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {
1305 require(_depositFeeBP <= 10000, "set: invalid deposit fee basis points");
1306 if (_withUpdate) {
1307 massUpdatePools();
1308 }
1309 totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
1310 poolInfo[_pid].allocPoint = _allocPoint;
1311 poolInfo[_pid].depositFeeBP = _depositFeeBP;
1312 }
1313
1314 // Return reward multiplier over the given _from to _to block.
```

MEDIUM Function could be marked as external.

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

Source file

MasterChef.sol

Locations

```
1359
1360 // Deposit LP tokens to MasterChef for GRAGAS allocation.
1361 function deposit(uint256 _pid, uint256 _amount) public {
1362 PoolInfo storage pool = poolInfo[_pid];
1363 UserInfo storage user = userInfo[_pid][msg.sender];
1364 updatePool(_pid);
1365 if (user.amount > 0) {
1366 uint256 pending = user.amount.mul(pool.accGragasPerShare).div(1e12).sub(user.rewardDebt);
1367 if(pending > 0) {
1368 safeGragasTransfer(msg.sender, pending);
1369 }
1370 }
1371 if(_amount > 0) {
1372 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1373 if(pool.depositFeeBP > 0){
1374 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1375 pool.lpToken.safeTransfer(feeAddress, depositFee);
1376 user.amount = user.amount.add(_amount).sub(depositFee);
1377 }else{
1378 user.amount = user.amount.add(_amount);
1379 }
1380 }
1381 user.rewardDebt = user.amount.mul(pool.accGragasPerShare).div(1e12);
1382 emit Deposit(msg.sender, _pid, _amount);
1383 }
1384
1385 // 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

```
1384
1385 // Withdraw LP tokens from MasterChef.
1386 function withdraw(uint256 _pid, uint256 _amount) public {
1387 PoolInfo storage pool = poolInfo[_pid];
1388 UserInfo storage user = userInfo[_pid][msg.sender];
1389 require(user.amount >= _amount, "withdraw: not good");
1390 updatePool(_pid);
1391 uint256 pending = user.amount.mul(pool.accGragasPerShare).div(1e12).sub(user.rewardDebt);
1392 if(pending > 0) {
1393 safeGragasTransfer(msg.sender, pending);
1394 }
1395 if(_amount > 0) {
1396 user.amount = user.amount.sub(_amount);
1397 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1398 }
1399 user.rewardDebt = user.amount.mul(pool.accGragasPerShare).div(1e12);
1400 emit Withdraw(msg.sender, _pid, _amount);
1401 }
1402
1403 // 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

```
1402
1403 // Withdraw without caring about rewards. EMERGENCY ONLY.
1404 function emergencyWithdraw(uint256 _pid) public {
1405 PoolInfo storage pool = poolInfo[_pid];
1406 UserInfo storage user = userInfo[_pid][msg.sender];
1407 uint256 amount = user.amount;
1408 user.amount = 0;
1409 user.rewardDebt = 0;
1410 pool.lpToken.safeTransfer(address(msg.sender), amount);
1411 emit EmergencyWithdraw(msg.sender, _pid, amount);
1412 }
1413
1414 // Safe gragas transfer function, just in case if rounding error causes pool to not have enough GRAGASs.
```

MEDIUM

Function could be marked as external.

SWC-000

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

Source file

MasterChef.sol

Locations

```
1423
1424 // Update dev address by the previous dev.
1425 function dev(address _devaddr) public {
1426 require(msg.sender == devaddr, "dev: wut?");
1427 devaddr = _devaddr;
1428 }
1429
1430 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.sol

Locations

```
1428 }
1429
1430 function setFeeAddress(address _feeAddress) public{
1431 require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
1432 feeAddress = _feeAddress;
1433 }
1434
1435 //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.sol

Locations

```
1434
1435 //Pancake has to add hidden dummy pools inorder to alter the emission, here we make it simple and transparent to all.
1436 function updateEmissionRate(uint256 _gragasPerBlock) public onlyOwner {
1437 massUpdatePools();
1438 gragasPerBlock = _gragasPerBlock;
1439 }
1440 }
```

MEDIUM

Multiple calls are executed in the same transaction.

SWC-113

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

Source file

MasterChef.sol

Locations

```
428
429 // solhint-disable-next-line avoid-low-level-calls
430 (bool success, bytes memory returndata) = target.call{ value: value }(data);
431 return _verifyCallResult(success, returndata, errorMessage);
432 }
```

MEDIUM

Loop over unbounded data structure.

SWC-128

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

Source file

MasterChef.sol

Locations

```
1334 function massUpdatePools() public {
1335 uint256 length = poolInfo.length;
1336 for (uint256 pid = 0; pid < length; ++pid) {
1337 updatePool(pid);
1338 }
```

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.sol

Locations

```
5 // File: contracts\libs\SafeMath.sol
6
7 pragma solidity >=0.6.0 <0.8.0;
8
9 /**
```

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.sol

Locations

```
217 }
218
219 pragma solidity >=0.6.4;
220
221 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.sol

Locations

```
312 // File: contracts\libs\Address.sol
313
314 pragma solidity >=0.6.2 <0.8.0;
315
316 /**
```

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 // File: contracts\libs\SafeBEP20.sol
503
504 pragma solidity >=0.6.0 <0.8.0;
505
506 /**
```

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

```
574 | // File: contracts\libs\Context.sol
575 |
576 | pragma solidity >=0.6.0 <0.8.0;
577 |
578 | /*
```

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

```
599 | // File: contracts\libs\Ownable.sol
600 |
601 | pragma solidity >=0.6.0 <0.8.0;
602 | /**
603 | * @dev Contract module which provides a basic access control mechanism, where
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""=0.4.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

MasterChef.sol

Locations

```
666 | // File: contracts\libs\BEP20.sol
667 |
668 | pragma solidity >=0.4.0;
669 |
670 | /**
```

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

```
1371 if(_amount > 0) {
1372 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1373 if(pool.depositFeeBP > 0){
1374 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1375 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

```
1376 user.amount = user.amount.add(_amount).sub(depositFee);
1377 }else{
1378 user.amount = user.amount.add(_amount);
1379 }
1380 }
```

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

```
1376 user.amount = user.amount.add(_amount).sub(depositFee);
1377 }else{
1378 user.amount = user.amount.add(_amount);
1379 }
1380 }
```

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

```
1379 }
1380 }
1381 user.rewardDebt = user.amount.mul(pool.accGragasPerShare).div(1e12);
1382 emit Deposit(msg.sender, _pid, _amount);
1383 }
```

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

```
1379 }
1380 }
1381 user.rewardDebt = user.amount.mul(pool.accGragasPerShare).div(1e12);
1382 emit Deposit(msg.sender, _pid, _amount);
1383 }
```

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

```
1379 }
1380 }
1381 user.rewardDebt = user.amount.mul(pool.accGragasPerShare).div(1e12);
1382 emit Deposit(msg.sender, _pid, _amount);
1383 }
```

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

```
1372 | pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
1373 | if(pool.depositFeeBP > 0){
1374 | uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1375 | pool.lpToken.safeTransfer(feeAddress, depositFee);
1376 | 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

```
1373 | if(pool.depositFeeBP > 0){
1374 | uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1375 | pool.lpToken.safeTransfer(feeAddress, depositFee);
1376 | user.amount = user.amount.add(_amount).sub(depositFee);
1377 | }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

```
1373 | if(pool.depositFeeBP > 0){
1374 | uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1375 | pool.lpToken.safeTransfer(feeAddress, depositFee);
1376 | user.amount = user.amount.add(_amount).sub(depositFee);
1377 | }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

```
424 */
425 function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage) internal returns (bytes memory) {
426 require(address(this).balance >= value, "Address: insufficient balance for call");
427 require(isContract(target), "Address: call to non-contract");
428
```

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

```
1374 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1375 pool.lpToken.safeTransfer(feeAddress, depositFee);
1376 user.amount = user.amount.add(_amount).sub(depositFee);
1377 }else{
1378 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

```
1374 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
1375 pool.lpToken.safeTransfer(feeAddress, depositFee);
1376 user.amount = user.amount.add(_amount).sub(depositFee);
1377 }else{
1378 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

```
1397 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1398 }
1399 user.rewardDebt = user.amount.mul(pool.accGragasPerShare).div(1e12);
1400 emit Withdraw(msg.sender, _pid, _amount);
1401 }
```

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

```
1397 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1398 }
1399 user.rewardDebt = user.amount.mul(pool.accGragasPerShare).div(1e12);
1400 emit Withdraw(msg.sender, _pid, _amount);
1401 }
```

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

```
1397 pool.lpToken.safeTransfer(address(msg.sender), _amount);
1398 }
1399 user.rewardDebt = user.amount.mul(pool.accGragasPerShare).div(1e12);
1400 emit Withdraw(msg.sender, _pid, _amount);
1401 }
```

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

```
1104 returns (uint256)
1105 {
1106 require(blockNumber < block.number, "GRAGAS::getPriorVotes: not yet determined");
1107
1108 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

```
1177 internal
1178 {
1179 uint32 blockNumber = safe32(block.number, "GRAGAS::_writeCheckpoint: block number exceeds 32 bits");
1180
1181 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

```
1290 massUpdatePools();
1291 }
1292 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1293 totalAllocPoint = totalAllocPoint.add(_allocPoint);
1294 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

```
1290  massUpdatePools();
1291  }
1292  uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
1293  totalAllocPoint = totalAllocPoint.add(_allocPoint);
1294  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

```
1323  uint256 accGragasPerShare = pool.accGragasPerShare;
1324  uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1325  if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1326  uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1327  uint256 gragasReward = multiplier.mul(gragasPerBlock).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

```
1324  uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1325  if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1326  uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1327  uint256 gragasReward = multiplier.mul(gragasPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1328  accGragasPerShare = accGragasPerShare.add(gragasReward.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

```
1342 function updatePool(uint256 _pid) public {
1343 PoolInfo storage pool = poolInfo[_pid];
1344 if (block.number <= pool.lastRewardBlock) {
1345 return;
1346 }
```

LOW

SWC-120

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

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

Source file

MasterChef.sol

Locations

```
1347 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1348 if (lpSupply == 0 || pool.allocPoint == 0) {
1349 pool.lastRewardBlock = block.number;
1350 return;
1351 }
```

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

```
1350 return;
1351 }
1352 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1353 uint256 gragasReward = multiplier.mul(gragasPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
1354 gragas.mint(devaddr, gragasReward.div(10));
```

LOW

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

SWC-120

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

```
1355 gragas.mint(address(this), gragasReward);
1356 pool.accGragasPerShare = pool.accGragasPerShare.add(gragasReward.mul(1e12).div(lpSupply));
1357 pool.lastRewardBlock = block.number;
1358 }
1359
```

LOW

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file

MasterChef.sol

Locations

```
1345 return;
1346 }
1347 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1348 if (lpSupply == 0 || pool.allocPoint == 0) {
1349 pool.lastRewardBlock = block.number;
```

Source file

MasterChef.sol

Locations

```
1211 //
1212 // Have fun reading it. Hopefully it's bug-free. God bless.
1213 contract MasterChef is Ownable {
1214 using SafeMath for uint256;
1215 using SafeBEP20 for IBEP20;
1216
1217 // Info of each user.
1218 struct UserInfo {
1219 uint256 amount; // How many LP tokens the user has provided.
1220 uint256 rewardDebt; // Reward debt. See explanation below.
1221 //
1222 // We do some fancy math here. Basically, any point in time, the amount of GRAGASs
1223 // entitled to a user but is pending to be distributed is:
1224 //
1225 // pending reward = (user.amount * pool.accGragasPerShare) - user.rewardDebt
1226 //
1227 // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
1228 // 1. The pool's `accGragasPerShare` (and `lastRewardBlock`) gets updated.
1229 // 2. User receives the pending reward sent to his/her address.
1230 // 3. User's `amount` gets updated.
1231 // 4. User's `rewardDebt` gets updated.
1232 }
1233
1234 // Info of each pool.
1235 struct PoolInfo {
1236 IBEP20 lpToken; // Address of LP token contract.
1237 uint256 allocPoint; // How many allocation points assigned to this pool. GRAGASs to distribute per block.
1238 uint256 lastRewardBlock; // Last block number that GRAGASs distribution occurs.
1239 uint256 accGragasPerShare; // Accumulated GRAGASs per share, times 1e12. See below.
1240 uint16 depositFeeBP; // Deposit fee in basis points
1241 }
1242
1243 // The GRAGAS TOKEN!
1244 GragasToken public gragas;
1245 // Dev address.
1246 address public devaddr;
1247 // GRAGAS tokens created per block.
1248 uint256 public gragasPerBlock;
1249 // Bonus muliplier for early gragas makers.
1250 uint256 public constant BONUS_MULTIPLIER = 1;
1251 // Deposit Fee address
1252 address public feeAddress;
1253
1254 // Info of each pool.
1255 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 GRAGAS 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(
GragasToken _gragas,
address _devaddr,
address _feeAddress,
uint256 _gragasPerBlock,
uint256 _startBlock
) public {
gragas = _gragas;
devaddr = _devaddr;
feeAddress = _feeAddress;
gragasPerBlock = _gragasPerBlock;
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,
accGragasPerShare: 0,
depositFeeBP: _depositFeeBP
}));
}

// Update the given pool's GRAGAS 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 GRAGASs on frontend.
function pendingGragas(uint256 _pid, address _user) external view returns (uint256) {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][_user];
uint256 accGragasPerShare = pool.accGragasPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 gragasReward = multiplier.mul(gragasPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
accGragasPerShare = accGragasPerShare.add(gragasReward.mul(1e12).div(lpSupply));
}
return user.amount.mul(accGragasPerShare).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 gragasReward = multiplier.mul(gragasPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
gragas.mint(devaddr, gragasReward.div(10));
gragas.mint(address(this), gragasReward);
pool.accGragasPerShare = pool.accGragasPerShare.add(gragasReward.mul(1e12).div(lpSupply));
pool.lastRewardBlock = block.number;
}

// Deposit LP tokens to MasterChef for GRAGAS 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.accGragasPerShare).div(1e12).sub(user.rewardDebt);
if(pending > 0) {
safeGragasTransfer(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.accGragasPerShare).div(1e12);
```

```
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.accGragasPerShare).div(1e12).sub(user.rewardDebt);
if(pending > 0) {
safeGragasTransfer(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.accGragasPerShare).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 gragas transfer function, just in case if rounding error causes pool to not have enough GRAGASs.
function safeGragasTransfer(address _to, uint256 _amount) internal {
uint256 gragasBal = gragas.balanceOf(address(this));
if (_amount > gragasBal) {
gragas.transfer(_to, gragasBal);
} else {
gragas.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 _gragasPerBlock) public onlyOwner {
massUpdatePools();
gragasPerBlock = _gragasPerBlock;
}
}
```

LOW

SWC-128

Potentially unbounded data structure passed to builtin.

Gas consumption in function "delegateBySig" in contract "GragasToken" 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

```
1048 abi.encode(
1049 DOMAIN_TYPEHASH,
1050 keccak256(bytes(name())),
1051 getChainId(),
1052 address(this)
```

LOW

SWC-128

Loop over unbounded data structure.

Gas consumption in function "getPriorVotes" in contract "GragasToken" 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

```
1123 uint32 lower = 0;
1124 uint32 upper = nCheckpoints - 1;
1125 while (upper > lower) {
1126 uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
1127 Checkpoint memory cp = checkpoints[account][center];
```