MythX

REPORT 60670BB7DD06F7001141452E

Created Fri Apr 02 2021 12:19:03 GMT+0000 (Coordinated Universal Time)

Number of analyses 1

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
83e468f7-e9b3-4363-b63b-358251c753f3	/contracts/masterchefv2.sol	22

Analysis 83e468f7-e9b3-4363-b63b-358251c753f3 MythX

Started	Fri Apr 02 2021 12:19:08 GMT+0000 (Coordinated Universal Time)
Finished	Fri Apr 02 2021 12:34:54 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Mythx-Vscode-Extension
Main Source File	/Contracts/Masterchefv2.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	2	20

ISSUES

MEDIUM

Loop over unbounded data structure.

SWC-128

Gas consumption in function "massUpdatePools" in contract "MasterChefV2" 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

/contracts/masterchefv2.sol

Locations

```
138 // Update reward variables of the given pool to be up-to-date.
139 function updatePool(uint256 _pid) public {
140 PoolInfo storage pool = poolInfo[_pid];
141 if (block.number <= pool.lastRewardBlock) {
142 return;
```

MEDIUM

Loop over unbounded data structure.

SWC-128

Gas consumption in function "compoundAll" in contract "MasterChefV2" 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

/contracts/masterchefv2.sol

Locations

```
170 }
171
172 function _compound(uint256 _pid) internal bonusCheck {
173 PoolInfo storage pool = poolInfo[_pid];
174 UserInfo storage user = userInfo[_pid][msg.sender];
```

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

/contracts/masterchefv2.sol

Locations

```
204 uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
205 pool.lpToken.safeTransfer(feeAddress, depositFee);
206 user.amount = user.amount.add(_amount).sub(depositFee);
207 } else {
208 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

/contracts/masterchefv2.sol

Locations

```
213 }
214
215 // Withdraw LP tokens from MasterChef.
216 function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
217 PoolInfo storage pool = poolInfo[_pid];
```

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

/contracts/masterchefv2.sol

Locations

```
211 user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
212 emit Deposit(msg.sender, _pid, _amount);
213 }
214
215 // Withdraw LP tokens from MasterChef.
216 function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
217 PoolInfo storage pool = poolInfo[_pid];
```

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

/contracts/masterchefv2.sol

Locations

```
214
215 // Withdraw LP tokens from MasterChef.
216 function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
217 PoolInfo storage pool = poolInfo[_pid];
218 UserInfo storage user = userInfo[_pid][msg.sender];
```

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

/contracts/masterchefv2.sol

Locations

```
214
215 // Withdraw LP tokens from MasterChef.
216 function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
217 PoolInfo storage pool = poolInfo[_pid];
218 UserInfo storage user = userInfo[_pid][msg.sender];
```

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

/contracts/masterchefv2.sol

Locations

```
214
215 // Withdraw LP tokens from MasterChef.
216 function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
217 PoolInfo storage pool = poolInfo[_pid];
218 UserInfo storage user = userInfo[_pid][msg.sender];
219 require(user.amount >= _amount, "withdraw: not good");
```

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

/contracts/masterchefv2.sol

Locations

```
206 | user.amount = user.amount.add(_amount).sub(depositFee);
207 | } else {
208 | user.amount = user.amount.add(_amount);
209 | }
210 | }
```

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

/contracts/masterchefv2.sol

Locations

```
208 | user.amount = user.amount.add(_amount);
209 | }
210 | }
211 | user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
212 | emit Deposit(msg.sender, _pid, _amount);
213 | }
```

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

/contracts/masterchefv2.sol

Locations

```
206 | user.amount = user.amount.add(_amount).sub(depositFee);
207 | } else {
208 | user.amount = user.amount.add(_amount);
209 | }
210 | }
211 | user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
```

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

/contracts/masterchefv2.sol

Locations

```
209 }
210 }
211 user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
212 emit Deposit(msg.sender, _pid, _amount);
213 }
```

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

/contracts/masterchefv2.sol

Locations

```
209 }
210 }
211 user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
212 emit Deposit(msg.sender, _pid, _amount);
213 }
```

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

/contracts/masterchefv2.sol

Locations

```
90 poolExistence[_lpToken] = true;
91 poolInfo.push(PoolInfo({
92 lpToken : _lpToken,
93 allocPoint : _allocPoint,
94 lastRewardBlock : lastRewardBlock,
```

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

/contracts/masterchefv2.sol

Locations

```
91  poolInfo.push(PoolInfo({
92  lpToken : _lpToken,
93  allocPoint : _allocPoint,
94  lastRewardBlock : lastRewardBlock,
95  accRewardPerShare : 0,
```

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

/contracts/masterchefv2.sol

Locations

```
122  if (block.number > pool.lastRewardBlock && lpSupply != 0) {
123  uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
124  uint256 rewardReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
125  accRewardPerShare = accRewardPerShare.add(rewardReward.mul(1e12).div(lpSupply));
126  }
```

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

/contracts/masterchefv2.sol

Locations

```
123  uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
124  uint256 rewardReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
125  accRewardPerShare = accRewardPerShare.add(rewardReward.mul(1e12).div(lpSupply));
126  }
127  return user.amount.mul(accRewardPerShare).div(1e12).sub(user.rewardDebt);
```

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

/contracts/masterchefv2.sol

Locations

```
144 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
145 if (lpSupply == 0 || pool.allocPoint == 0) {
146 pool.lastRewardBlock = block.number;
147 return;
148 }
```

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

/contracts/masterchefv2.sol

Locations

```
148 }
149 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
150 uint256 rewardReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
151 if (rewardReward > 0) {
152 reward.mint(devaddr, rewardReward.div(10));
```

LOW

SWC-120

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

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

Source file

/contracts/masterchefv2.sol

Locations

```
150 uint256 rewardReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
151 if (rewardReward > 0) {
152 reward.mint(devaddr, rewardReward.div(10));
153 reward.mint(address(this), rewardReward);
154 pool.accRewardPerShare = pool.accRewardPerShare.add(rewardReward.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

/contracts/masterchefv2.sol

Locations

```
161 uint256 length = poolInfo.length;
162 for (uint256 pid = 0; pid < length; ++pid) {
163 _compound(pid);
164 }
165 }
```