

REPORT 60646007DC5BD600119221E0

Created Wed Mar 31 2021 11:41:59 GMT+0000 (Coordinated Universal Time)

Number of analyses 1

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
347b4ab0-c2ef-472c-9f81-78d242aa6898	/contracts/masterchefv2.sol	25

Started	Wed Mar 31 2021 11:42:10 GMT+0000 (Coordinated Universal Time)
Finished	Wed Mar 31 2021 11:57:52 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	23

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

```
164 | function compound(uint256 _pid) public bonusCheck {
165 |     PoolInfo storage pool = poolInfo[_pid];
166 |     UserInfo storage user = userInfo[_pid][msg.sender];
167 |
168 |     if (user.amount > 0) {
```

LOW Read of persistent state following external call.

SWC-107

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

```
193 | uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
194 | pool.lpToken.safeTransfer(feeAddress, depositFee);
195 | user.amount = user.amount.add(_amount).sub(depositFee);
196 | } else {
197 | user.amount = user.amount.add(_amount);
```

LOW Read of persistent state following external call.

SWC-107

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

```
202 | }
203 |
204 | // Withdraw LP tokens from MasterChef.
205 | function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
206 | PoolInfo storage pool = poolInfo[_pid];
```

LOW Write to persistent state following external call.

SWC-107

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

```
200 | user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
201 | emit Deposit(msg.sender, _pid, _amount);
202 | }
203 |
204 | // Withdraw LP tokens from MasterChef.
205 | function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
206 | PoolInfo storage pool = poolInfo[_pid];
```

LOW Read of persistent state following external call.

SWC-107

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

```
203 |
204 | // Withdraw LP tokens from MasterChef.
205 | function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
206 |     PoolInfo storage pool = poolInfo[_pid];
207 |     UserInfo storage user = userInfo[_pid][msg.sender];
```

LOW Read of persistent state following external call.

SWC-107

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

```
203 |
204 | // Withdraw LP tokens from MasterChef.
205 | function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
206 |     PoolInfo storage pool = poolInfo[_pid];
207 |     UserInfo storage user = userInfo[_pid][msg.sender];
```

LOW Write to persistent state following external call.

SWC-107

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

```
203 |
204 | // Withdraw LP tokens from MasterChef.
205 | function withdraw(uint256 _pid, uint256 _amount) external nonReentrant bonusCheck {
206 |     PoolInfo storage pool = poolInfo[_pid];
207 |     UserInfo storage user = userInfo[_pid][msg.sender];
208 |     require(user.amount >= _amount, "withdraw: not good");
```

LOW Read of persistent state following external call.

SWC-107

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

```
221 |
222 | // Withdraw without caring about rewards. EMERGENCY ONLY.
223 | function emergencyWithdraw(uint256 _pid) external nonReentrant {
224 |     PoolInfo storage pool = poolInfo[_pid];
225 |     UserInfo storage user = userInfo[_pid][msg.sender];
```

LOW Read of persistent state following external call.

SWC-107

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

```
221 |
222 | // Withdraw without caring about rewards. EMERGENCY ONLY.
223 | function emergencyWithdraw(uint256 _pid) external nonReentrant {
224 |     PoolInfo storage pool = poolInfo[_pid];
225 |     UserInfo storage user = userInfo[_pid][msg.sender];
```

LOW Write to persistent state following external call.

SWC-107

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

```
221 |
222 | // Withdraw without caring about rewards. EMERGENCY ONLY.
223 | function emergencyWithdraw(uint256 _pid) external nonReentrant {
224 |     PoolInfo storage pool = poolInfo[_pid];
225 |     UserInfo storage user = userInfo[_pid][msg.sender];
226 |     uint256 amount = user.amount;
```

LOW Read of persistent state following external call.

SWC-107

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

```
195 | user.amount = user.amount.add(_amount).sub(depositFee);
196 | } else {
197 |   user.amount = user.amount.add(_amount);
198 | }
199 | }
```

LOW Read of persistent state following external call.

SWC-107

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

```
197 | user.amount = user.amount.add(_amount);
198 | }
199 |
200 | user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
201 | emit Deposit(msg.sender, _pid, _amount);
202 | }
```

LOW Read of persistent state following external call.

SWC-107

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

```
195 | user.amount = user.amount.add(_amount).sub(depositFee);
196 | } else {
197 |   user.amount = user.amount.add(_amount);
198 | }
199 | }
200 | user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
```

LOW Read of persistent state following external call.

SWC-107

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

```
198 | }
199 | }
200 | user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
201 | emit Deposit(msg.sender, _pid, _amount);
202 | }
```

LOW Write to persistent state following external call.

SWC-107

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

```
198 | }
199 | }
200 | user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
201 | emit Deposit(msg.sender, _pid, _amount);
202 | }
```

LOW Potential use of "block.number" as source of randomness.

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

/contracts/masterchefv2.sol

Locations

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

LOW Potential use of "block.number" as source of randomness.

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

/contracts/masterchefv2.sol

Locations

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

LOW Potential use of "block.number" as source of randomness.

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

/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 Potential use of "block.number" as source of randomness.

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

/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

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

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

/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

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

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

/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 | reward.mint(devaddr, rewardReward.div(10));
152 | reward.mint(address(this), rewardReward);
```

LOW

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

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

/contracts/masterchefv2.sol

Locations

```
150 | uint256 rewardReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
151 | reward.mint(devaddr, rewardReward.div(10));
152 | reward.mint(address(this), rewardReward);
153 | pool.accRewardPerShare = pool.accRewardPerShare.add(rewardReward.mul(1e12).div(lpSupply));
154 | pool.lastRewardBlock = block.number;
```

LOW

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

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

/contracts/masterchefv2.sol

Locations

```
160 | compound(pid);
161 | }
162 | 
163 | 
164 | function compound(uint256 _pid) public bonusCheck {
165 |     PoolInfo storage pool = poolInfo[_pid];
166 |     UserInfo storage user = userInfo[_pid][msg.sender];
```

LOW

Requirement violation.

SWC-123

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).

Source file

/contracts/masterchefv2.sol

Locations

```
147 | return;
148 | }
149 | uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
150 | uint256 rewardReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
151 | reward.mint(devaddr, rewardReward.div(10));
```