MythX

Analysis bc510cbf-a39c-4953-8193-cc93a9c112b1 MythX

Started	Mon Sep 06 2021 05:59:04 GMT+0000 (Coordinated Universal Time)
Finished	Mon Sep 06 2021 06:44:37 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	Contracts/MasterChef.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	20

ISSUES

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/MasterChef.sol

Locations

```
293 UserInfo storage user = userInfo[_pid][msg.sender];
294 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
295 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
296 user.amount = 0;
297 user.rewardDebt = 0;
```

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/MasterChef.sol

Locations

```
294 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
295 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
296 user.amount = 0;
297 user.rewardDebt = 0;
298 }
```

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/MasterChef.sol

Locations

```
295 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
296 user.amount = 0;
297 user.rewardDebt = 0;
298 }
299
```

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/MasterChef.sol

Locations

```
220 if (_amount > 0) {
221 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
222 user.amount = user.amount.add(_amount);
223 }
224 user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);
```

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/MasterChef.sol

Locations

```
220 if (_amount > 0) {
221 pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
222 user.amount = user.amount.add(_amount);
223 }
224 user.rewardDebt = user.amount.mul(pool.accXttPerShare).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/MasterChef.sol

Locations

```
222 user.amount = user.amount.add(_amount);
223 }
224 user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);
225 emit Deposit(msg.sender, _pid, _amount);
226 }
```

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/MasterChef.sol

Locations

```
222 user.amount = user.amount.add(_amount);
223 }
224 user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);
225 emit Deposit(msg.sender, _pid, _amount);
226 }
```

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/MasterChef.sol

Locations

```
222 user.amount = user.amount.add(_amount);
223 }
224 user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);
225 emit Deposit(msg.sender, _pid, _amount);
226 }
```

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/MasterChef.sol

Locations

```
243 pool.lpToken.safeTransfer(address(msg.sender), _amount);
244 }
245 user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);
246 emit Withdraw(msg.sender, _pid, _amount);
247 }
```

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/MasterChef.sol

Locations

```
243 pool.lpToken.safeTransfer(address(msg.sender), _amount);
244 }
245 user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);
246 emit Withdraw(msg.sender, _pid, _amount);
247 }
```

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/MasterChef.sol

Locations

```
243 pool.lpToken.safeTransfer(address(msg.sender), _amount);
244 }
245 user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);
246 emit Withdraw(msg.sender, _pid, _amount);
247 }
```

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/MasterChef.sol

Locations

```
105 massUpdatePools();
106 }
107 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
108 totalAllocPoint = totalAllocPoint.add(_allocPoint);
109 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

contracts/MasterChef.sol

Locations

```
105 massUpdatePools();
106 }
107 uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
108 totalAllocPoint = totalAllocPoint.add(_allocPoint);
109 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

contracts/MasterChef.sol

Locations

```
170 uint256 accXttPerShare = pool.accXttPerShare;
171 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
172 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
173 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
174 uint256 cakeReward = multiplier.mul(xttPerBlock).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

contracts/MasterChef.sol

Locations

```
171 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
172 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
173 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
174 uint256 cakeReward = multiplier.mul(xttPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
175 accXttPerShare = accXttPerShare.add(cakeReward.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/MasterChef.sol

Locations

```
190 function updatePool(uint256 _pid) public {
191 PoolInfo storage pool = poolInfo[_pid];
192 if (block.number <= pool.lastRewardBlock) {
193 return;
194 }
```

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/MasterChef.sol

Locations

```
195 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
196 if (lpSupply == 0) {
197 pool.lastRewardBlock = block.number;
198 return;
199 }
```

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/MasterChef.sol`

Locations

```
198 return;
199 }
200 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
201 uint256 xttReward = multiplier.mul(xttPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
202 pool.accXttPerShare = pool.accXttPerShare.add(xttReward.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/MasterChef.sol`

Locations

```
201 uint256 xttReward = multiplier.mul(xttPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
202 pool.accXttPerShare = pool.accXttPerShare.add(xttReward.mul(1e12).div(lpSupply));
203 pool.lastRewardBlock = block.number;
204 }
205
```

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

contracts/MasterChef.sol

Locations

```
193 return;
194 }
195 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
196 if (lpSupply == 0) {
197 pool.lastRewardBlock = block.number;
```

Source file

contracts/MasterChef.sol

Locations

```
18 //
19 // Have fun reading it. Hopefully it's bug-free. God bless.
20 contract MasterChef is Ownable {
21 using SafeMath for uint256;
22 using SafeBEP20 for IBEP20;
23
24 // Info of each user.
25 struct UserInfo {
26 uint256 amount; // How many LP tokens the user has provided.
27 uint256 rewardDebt; // Reward debt. See explanation below.
28 //
29 // We do some fancy math here. Basically, any point in time, the amount of XTTs
30 // entitled to a user but is pending to be distributed is:
31 //
32 // pending reward = (user.amount * pool.accXttPerShare) - user.rewardDebt
33 //
34 // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
35 // 1. The pool's `accXttPerShare` (and `lastRewardBlock`) gets updated.
36 // 2. User receives the pending reward sent to his/her address.
37 // 3. User's `amount` gets updated.
38 // 4. User's `rewardDebt` gets updated.
39 }
40
41 // Info of each pool.
42 struct PoolInfo {
43 IBEP20 lpToken; // Address of LP token contract.
44 uint256 allocPoint; // How many allocation points assigned to this pool. XTTs to distribute per block.
45 uint256 lastRewardBlock; // Last block number that XTTs distribution occurs.
46 uint256 accXttPerShare; // Accumulated XTTs per share, times 1e12. See below.
47 }
48
49 // The XTT TOKEN!
50 IBEP20 public xtt;
51
52 // XTT tokens created per block.
53 uint256 public xttPerBlock;
54 // Bonus muliplier for early xtt makers.
55 uint256 public BONUS_MULTIPLIER = 1;
56 // The migrator contract. It has a lot of power. Can only be set through governance (owner).
57 IMigratorChef public migrator;
58
59 // Info of each pool.
60 PoolInfo[] public poolInfo;
61
```

```
// 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 XTT 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(
IBEP20 _xtt,
uint256 _xttPerBlock,
uint256 _startBlock
) public {
xtt = _xtt;
xttPerBlock = _xttPerBlock;
startBlock = _startBlock;

// staking pool
poolInfo.push(PoolInfo({
lpToken: _xtt,
allocPoint: 1000,
lastRewardBlock: startBlock,
accXttPerShare: 0
}));

totalAllocPoint = 1000;

}

function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
BONUS_MULTIPLIER = multiplierNumber;
}

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, bool _withUpdate) public onlyOwner {
if (_withUpdate) {
massUpdatePools();
}
uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
totalAllocPoint = totalAllocPoint.add(_allocPoint);
poolInfo.push(PoolInfo({
lpToken: _lpToken,
allocPoint: _allocPoint,
lastRewardBlock: lastRewardBlock,
accXttPerShare: 0
}));
updateStakingPool();
}

// Update the given pool's XTT allocation point. Can only be called by the owner.
function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
if (_withUpdate) {
massUpdatePools();
```

```
}
uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
poolInfo[_pid].allocPoint = _allocPoint;
if (prevAllocPoint != _allocPoint) {
totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
updateStakingPool();
}
}

function updateStakingPool() internal {
uint256 length = poolInfo.length;
uint256 points = 0;
for (uint256 pid = 1; pid < length; ++pid) {
points = points.add(poolInfo[pid].allocPoint);
}
if (points != 0) {
points = points.div(3);
totalAllocPoint = totalAllocPoint.sub(poolInfo[0].allocPoint).add(points);
poolInfo[0].allocPoint = points;
}
}

// Set the migrator contract. Can only be called by the owner.
function setMigrator(IMigratorChef _migrator) public onlyOwner {
migrator = _migrator;
}

// Migrate lp token to another lp contract. Can be called by anyone. We trust that migrator contract is good.
function migrate(uint256 _pid) public {
require(address(migrator) != address(0), "migrate: no migrator");
PoolInfo storage pool = poolInfo[_pid];
IBEP20 lpToken = pool.lpToken;
uint256 bal = lpToken.balanceOf(address(this));
lpToken.safeApprove(address(migrator), bal);
IBEP20 newLpToken = migrator.migrate(lpToken);
require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
pool.lpToken = newLpToken;
}

// 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 XTTs on frontend.
function pendingCake(uint256 _pid, address _user) external view returns (uint256) {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][_user];
uint256 accXttPerShare = pool.accXttPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 cakeReward = multiplier.mul(xttPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
accXttPerShare = accXttPerShare.add(cakeReward.mul(1e12).div(lpSupply));
}
return user.amount.mul(accXttPerShare).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.lastRewardBlock = block.number;
return;
}
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 xttReward = multiplier.mul(xttPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
pool.accXttPerShare = pool.accXttPerShare.add(xttReward.mul(1e12).div(lpSupply));
pool.lastRewardBlock = block.number;
}

// Deposit LP tokens to MasterChef for XTT allocation.
function deposit(uint256 _pid, uint256 _amount) public {

require (_pid != 0, 'deposit XTT by staking');

PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
updatePool(_pid);
if (user.amount > 0) {
uint256 pending = user.amount.mul(pool.accXttPerShare).div(1e12).sub(user.rewardDebt);
if(pending > 0) {
safeXttTransfer(msg.sender, pending);
}
}
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
user.amount = user.amount.add(_amount);
}
user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}

// Withdraw LP tokens from MasterChef.
function withdraw(uint256 _pid, uint256 _amount) public {

require (_pid != 0, 'withdraw XTT by unstaking');
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.accXttPerShare).div(1e12).sub(user.rewardDebt);
if(pending > 0) {
safeXttTransfer(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.accXttPerShare).div(1e12);
    emit Withdraw(msg.sender, _pid, _amount);
    }

    // Stake XTT tokens to MasterChef
    function enterStaking(uint256 _amount) public {
    PoolInfo storage pool = poolInfo[0];
    UserInfo storage user = userInfo[0][msg.sender];
    updatePool(0);
    if (user.amount > 0) {
    uint256 pending = user.amount.mul(pool.accXttPerShare).div(1e12).sub(user.rewardDebt);
    if(pending > 0) {
    safeXttTransfer(msg.sender, pending);
    }
    }
    if(_amount > 0) {
    pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
    user.amount = user.amount.add(_amount);
    }
    user.rewardDebt = user.amount.mul(pool.accXttPerShare).div(1e12);

    // syrup.mint(msg.sender, _amount);
    emit Deposit(msg.sender, 0, _amount);
    }

    // Withdraw XTT tokens from STAKING.
    function leaveStaking(uint256 _amount) public {
    PoolInfo storage pool = poolInfo[0];
    UserInfo storage user = userInfo[0][msg.sender];
    require(user.amount >= _amount, "withdraw: not good");
    updatePool(0);
    uint256 pending = user.amount.mul(pool.accXttPerShare).div(1e12).sub(user.rewardDebt);
    if(pending > 0) {
    safeXttTransfer(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.accXttPerShare).div(1e12);

    // syrup.burn(msg.sender, _amount);
    emit Withdraw(msg.sender, 0, _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];
    pool.lpToken.safeTransfer(address(msg.sender), user.amount);
    emit EmergencyWithdraw(msg.sender, _pid, user.amount);
    user.amount = 0;
    user.rewardDebt = 0;
    }

    // Safe xtt transfer function, just in case if rounding error causes pool to not have enough XTTs.
    function safeXttTransfer(address _to, uint256 _amount) internal {
    xtt.safeTransfer(_to, _amount);
    }
    }
```