MythX

REPORT 615D0DA89B09440018F13B81

Created	Wed Oct 06 2021 02:44:56 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	615cfffe43f2c3497112e999

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
ee8ac71c-5343-4443-8ffb-9962708b4560	MasterChef.sol	48

Analysis ee8ac71c-5343-4443-8ffb-9962708b4560 MythX

Started	Wed Oct 06 2021 02:45:03 GMT+0000 (Coordinated Universal Time)
Finished	Wed Oct 06 2021 03:30:36 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	MasterChef.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	1	47

ISSUES

MEDIUM

SWC-113

Multiple calls are executed in the same transaction.

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

```

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

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

```


pragma solidity >=0.6.0 <0.8.0;

/**
```

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

```
218 }
219
220 pragma solidity >=0.6.2 <0.8.0;
221
222 /**
```

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

```
407
408
409 pragma solidity >=0.6.0 <0.8.0;
410
411 /*
```

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

```
431
432
433 pragma solidity >=0.6.0 <0.8.0;
434
435 /**
```

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

```


pragma solidity >=0.4.0;

interface IBEP20 {
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is "">=0.5.0"". 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

```
}

pragma solidity >=0.5.0;

interface IUniswapV2ERC20 {
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is "">=0.5.0"". 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

```
}

pragma solidity >=0.5.0;

interface IUniswapV2Pair {
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is "">=0.5.0"". 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

```
}

pragma solidity >=0.5.0;

interface IUniswapV2Factory {
```

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

```


pragma solidity >=0.4.0;

/**
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is "">=0.6.2"". 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

```
}

pragma solidity >=0.6.2;

interface IUniswapV2Router01 {
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is "">=0.6.2"". 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

```


pragma solidity >=0.6.2;

```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.6.0"". 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

```


pragma solidity ^0.6.0;

```

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

```


pragma solidity >=0.6.0 <0.8.0;

/**
```

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

```
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if (address(pool.lpToken) == address(PUMPKIN)) {
uint256 transferTax = _amount.mul(PUMPKIN.transferTaxRate()).div(10000);
_amount = _amount.sub(transferTax);
```

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

```
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if (address(pool.lpToken) == address(PUMPKIN)) {
uint256 transferTax = _amount.mul(PUMPKIN.transferTaxRate()).div(10000);
_amount = _amount.sub(transferTax);
```

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

```
_amount = _amount.sub(transferTax);
}
if (pool.depositFeeBP > 0) {
uint256 depositFee = _amount.mul(pool.depositFeeBP).div(10000);
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

```
}
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);
```

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

```
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 {
```

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

```
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 {
```

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

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

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

```
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);
```

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

```
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);
```

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

```
}
}
user.rewardDebt = user.amount.mul(pool.accPUMPKINPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _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

```
2108 }
2109 }
2110 user.rewardDebt = user.amount.mul(pool.accPUMPKINPerShare).div(1e12);
2111 emit Deposit(msg.sender, _pid, _amount);
2112 }
```

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

```
2108 }
2109 }
2110 user.rewardDebt = user.amount.mul(pool.accPUMPKINPerShare).div(1e12);
2111 emit Deposit(msg.sender, _pid, _amount);
2112 }
```

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

```
2088 PUMPKINReferral.recordReferral(msg.sender, _referrer);
2089 }
2090 if (user.amount > 0) {
2091 uint256 pending = user.amount.mul(pool.accPUMPKINPerShare).div(1e12).sub(user.rewardDebt);
2092 if (pending > 0) {
```

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

```
}
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if (address(pool.lpToken) == address(PUMPKIN)) {
uint256 transferTax = _amount.mul(PUMPKIN.transferTaxRate()).div(10000);
```

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

```
user.amount = user.amount.add(_amount).sub(depositFee);
} else {
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

```
user.amount = user.amount.add(_amount).sub(depositFee);
} else {
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

```
2131  pool.lpToken.safeTransfer(address(msg.sender), _amount);
2132  }
2133  user.rewardDebt = user.amount.mul(pool.accPUMPKINPerShare).div(1e12);
2134  emit Withdraw(msg.sender, _pid, _amount);
2135  }
```

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

```
2131  pool.lpToken.safeTransfer(address(msg.sender), _amount);
2132  }
2133  user.rewardDebt = user.amount.mul(pool.accPUMPKINPerShare).div(1e12);
2134  emit Withdraw(msg.sender, _pid, _amount);
2135  }
```

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

```
2131  pool.lpToken.safeTransfer(address(msg.sender), _amount);
2132  }
2133  user.rewardDebt = user.amount.mul(pool.accPUMPKINPerShare).div(1e12);
2134  emit Withdraw(msg.sender, _pid, _amount);
2135  }
```

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

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

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

```
}
if (user.amount > 0) {
uint256 pending = user.amount.mul(pool.accPUMPKINPerShare).div(1e12).sub(user.rewardDebt);
if (pending > 0) {
safePUMPKINTransfer(msg.sender, pending);
```

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

```
}
if (user.amount > 0) {
uint256 pending = user.amount.mul(pool.accPUMPKINPerShare).div(1e12).sub(user.rewardDebt);
if (pending > 0) {
safePUMPKINTransfer(msg.sender, pending);
```

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

```
2089 }
2090 if (user.amount > 0) {
2091 uint256 pending = user.amount.mul(pool.accPUMPKINPerShare).div(1e12).sub(user.rewardDebt);
2092 if (pending > 0) {
2093 safePUMPKINTransfer(msg.sender, pending);
```

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

```
1602 returns (uint256)
1603 {
1604 require(blockNumber < block.number, "PUMPKIN::getPriorVotes: not yet determined");
1605
1606 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

```
1675 internal
1676 {
1677 uint32 blockNumber = safe32(block.number, "PUMPKIN::_writeCheckpoint: block number exceeds 32 bits");
1678
1679 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

```
massUpdatePools();
}
uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
totalAllocPoint = totalAllocPoint.add(_allocPoint);
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

```
massUpdatePools();
}
uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
totalAllocPoint = totalAllocPoint.add(_allocPoint);
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

```
uint256 accPUMPKINPerShare = pool.accPUMPKINPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 PUMPKINReward = multiplier.mul(PUMPKINPerBlock).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

```
2043 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
2044 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
2045 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
2046 uint256 PUMPKINReward = multiplier.mul(PUMPKINPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
2047 accPUMPKINPerShare = accPUMPKINPerShare.add(PUMPKINReward.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

```
2064 function updatePool(uint256 _pid) public {
2065 PoolInfo storage pool = poolInfo[_pid];
2066 if (block.number <= pool.lastRewardBlock) {
2067 return;
2068 }
```

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

```
2069 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
2070 if (lpSupply == 0 || pool.allocPoint == 0) {
2071 pool.lastRewardBlock = block.number;
2072 return;
2073 }
```

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

```
2072  return;
2073  }
2074  uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
2075  uint256 PUMPKINReward = multiplier.mul(PUMPKINPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
2076  PUMPKIN.mint(devAddress, PUMPKINReward.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

`MasterChef.sol`

Locations

```
2077  PUMPKIN.mint(address(this), PUMPKINReward);
2078  pool.accPUMPKINPerShare = pool.accPUMPKINPerShare.add(PUMPKINReward.mul(1e12).div(lpSupply));
2079  pool.lastRewardBlock = block.number;
2080  }
```

LOW

SWC-123

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).

Source file

`MasterChef.sol`

Locations

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

Source file

`MasterChef.sol`

Locations

```
//
// Have fun reading it. Hopefully it's bug-free. God bless.
contract MasterChef is Ownable, ReentrancyGuard {
using SafeMath for uint256;
using SafeBEP20 for IBEP20;

// Info of each user.
struct UserInfo {
uint256 amount; // How many LP tokens the user has provided.
uint256 rewardDebt; // Reward debt. See explanation below.

//
// We do some fancy math here. Basically, any point in time, the amount of PUMPKINs
// entitled to a user but is pending to be distributed is:
//
// pending reward = (user.amount * pool.accPUMPKINPerShare) - user.rewardDebt
//
// Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
// 1. The pool's `accPUMPKINPerShare` (and `lastRewardBlock`) gets updated.
// 2. User receives the pending reward sent to his/her address.
// 3. User's `amount` gets updated.
// 4. User's `rewardDebt` gets updated.

}

// Info of each pool.
struct PoolInfo {
IBEP20 lpToken; // Address of LP token contract.
uint256 allocPoint; // How many allocation points assigned to this pool. PUMPKINs to distribute per block.
uint256 lastRewardBlock; // Last block number that PUMPKINs distribution occurs.
uint256 accPUMPKINPerShare; // Accumulated PUMPKINs per share, times 1e12. See below.
uint16 depositFeeBP; // Deposit fee in basis points

}

// The PumpkinToken!
PumpkinToken public PUMPKIN;
// Dev address.
address public devAddress;

// Deposit Fee address
address public feeAddress;
// PumpkinTokens created per block.
uint256 public PUMPKINPerBlock;
// Bonus muliplier for early PUMPKIN makers.
```

```
uint256 public constant BONUS_MULTIPLIER = 1;



// Info of each pool.
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 PUMPKIN mining starts.
uint256 public startBlock;



// PUMPKIN referral contract address.
IPUMPKINReferral public PUMPKINReferral;
// Referral commission rate in basis points.
uint16 public referralCommissionRate = 500;
// Max referral commission rate: 10%.
uint16 public constant MAXIMUM_REFERRAL_COMMISSION_RATE = 1000;


event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);
event EmissionRateUpdated(address indexed caller, uint256 previousAmount, uint256 newAmount);
event ReferralCommissionPaid(address indexed user, address indexed referrer, uint256 commissionAmount);

constructor(
PumpkinToken _PUMPKIN,
address _devaddr,
address _feeAddress,
uint256 _PUMPKINPerBlock,
uint256 _startBlock

) public {
PUMPKIN = _PUMPKIN;

devAddress = _devaddr;
feeAddress = _feeAddress;

PUMPKINPerBlock = _PUMPKINPerBlock;
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, uint256 _harvestInterval, bool _withUpdate) public onlyOwner {
function add(uint256 _allocPoint, IBEP20 _lpToken, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {

// max 5% deposit fee allowed
require(_depositFeeBP <= 500, "add: invalid deposit fee basis points, 5% max");


if (_withUpdate) {
massUpdatePools();
}
uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
```

```
totalAllocPoint = totalAllocPoint.add(_allocPoint);
poolInfo.push(PoolInfo({
lpToken: _lpToken,
allocPoint: _allocPoint,
lastRewardBlock: lastRewardBlock,
accPUMPKINPerShare: 0,
depositFeeBP: _depositFeeBP

}));
}

// Update the given pool's PUMPKIN allocation point and deposit fee. Can only be called by the owner.
//function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, uint256 _harvestInterval, bool _withUpdate) public onlyOwner {
function set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP, bool _withUpdate) public onlyOwner {

require(_depositFeeBP <= 500, "set: invalid deposit fee basis points, 5% max");
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 pure returns (uint256) {
return _to.sub(_from).mul(BONUS_MULTIPLIER);
}

// View function to see pending PUMPKINs on frontend.
function pendingPUMPKIN(uint256 _pid, address _user) external view returns (uint256) {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][_user];
uint256 accPUMPKINPerShare = pool.accPUMPKINPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 PUMPKINReward = multiplier.mul(PUMPKINPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
accPUMPKINPerShare = accPUMPKINPerShare.add(PUMPKINReward.mul(1e12).div(lpSupply));
}
return user.amount.mul(accPUMPKINPerShare).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) {
```

```solidity
pool.lastRewardBlock = block.number;
return;
}
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 PUMPKINReward = multiplier.mul(PUMPKINPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
PUMPKIN.mint(devAddress, PUMPKINReward.div(10));
PUMPKIN.mint(address(this), PUMPKINReward);
pool.accPUMPKINPerShare = pool.accPUMPKINPerShare.add(PUMPKINReward.mul(1e12).div(lpSupply));
pool.lastRewardBlock = block.number;
}

// Deposit LP tokens to MasterChef for PUMPKIN allocation.
function deposit(uint256 _pid, uint256 _amount, address _referrer) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
updatePool(_pid);
if (_amount > 0 && address(PUMPKINReferral) != address(0) && _referrer != address(0) && _referrer != msg.sender) {
PUMPKINReferral.recordReferral(msg.sender, _referrer);
}
if (user.amount > 0) {
uint256 pending = user.amount.mul(pool.accPUMPKINPerShare).div(1e12).sub(user.rewardDebt);
if (pending > 0) {
safePUMPKINTransfer(msg.sender, pending);
}
}
if (_amount > 0) {
pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
if (address(pool.lpToken) == address(PUMPKIN)) {
uint256 transferTax = _amount.mul(PUMPKIN.transferTaxRate()).div(10000);
_amount = _amount.sub(transferTax);
}
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.accPUMPKINPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}

// Withdraw LP tokens from MasterChef.
function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
require(user.amount >= _amount, "withdraw: not good");
updatePool(_pid);

uint256 pending = user.amount.mul(pool.accPUMPKINPerShare).div(1e12).sub(user.rewardDebt);

if (pending > 0) {
// send rewards
safePUMPKINTransfer(msg.sender, pending);
payReferralCommission(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.accPUMPKINPerShare).div(1e12);
```

```
emit Withdraw(msg.sender, _pid, _amount);
}

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

pool.lpToken.safeTransfer(address(msg.sender), amount);
emit EmergencyWithdraw(msg.sender, _pid, amount);
}


// Safe PUMPKIN transfer function, just in case if rounding error causes pool to not have enough PUMPKINs.
function safePUMPKINTransfer(address _to, uint256 _amount) internal {
uint256 PUMPKINBal = PUMPKIN.balanceOf(address(this));
if (_amount > PUMPKINBal) {
PUMPKIN.transfer(_to, PUMPKINBal);
} else {
PUMPKIN.transfer(_to, _amount);
}
}

// Update dev address by the previous dev.
function setDevAddress(address _devAddress) public {
require(msg.sender == devAddress, "setDevAddress: FORBIDDEN");
require(_devAddress != address(0), "setDevAddress: ZERO");
devAddress = _devAddress;
}

function setFeeAddress(address _feeAddress) public {
require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
require(_feeAddress != address(0), "setFeeAddress: ZERO");
feeAddress = _feeAddress;
}


// Pancake has to add hidden dummy pools in order to alter the emission, here we make it simple and transparent to all.
function updateEmissionRate(uint256 _PUMPKINPerBlock) public onlyOwner {
massUpdatePools();
emit EmissionRateUpdated(msg.sender, PUMPKINPerBlock, _PUMPKINPerBlock);
PUMPKINPerBlock = _PUMPKINPerBlock;
}

// Update the PUMPKIN referral contract address by the owner
function setPUMPKINReferral(IPUMPKINReferral _PUMPKINReferral) public onlyOwner {
PUMPKINReferral = _PUMPKINReferral;
}

// Update referral commission rate by the owner
function setReferralCommissionRate(uint16 _referralCommissionRate) public onlyOwner {
require(_referralCommissionRate <= MAXIMUM_REFERRAL_COMMISSION_RATE, "setReferralCommissionRate: invalid referral commission rate basis points");
referralCommissionRate = _referralCommissionRate;
}

// Pay referral commission to the referrer who referred this user.
function payReferralCommission(address _user, uint256 _pending) internal {
if (address(PUMPKINReferral) != address(0) && referralCommissionRate > 0) {
address referrer = PUMPKINReferral.getReferrer(_user);
uint256 commissionAmount = _pending.mul(referralCommissionRate).div(10000);

if (referrer != address(0) && commissionAmount > 0) {
PUMPKIN.mint(referrer, commissionAmount);
PUMPKINReferral.recordReferralCommission(referrer, commissionAmount);
emit ReferralCommissionPaid(_user, referrer, commissionAmount);
}
}
}

//Only update before start of farm if not ready - emergency only
function updateStartBlock(uint256 _startBlock) public onlyOwner {
startBlock = _startBlock;

}

}
```