freshcoins

# SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Weebs

## Smart Contract

MasterChef - stake contract

06/09/2024

# TABLE OF CONTENTS

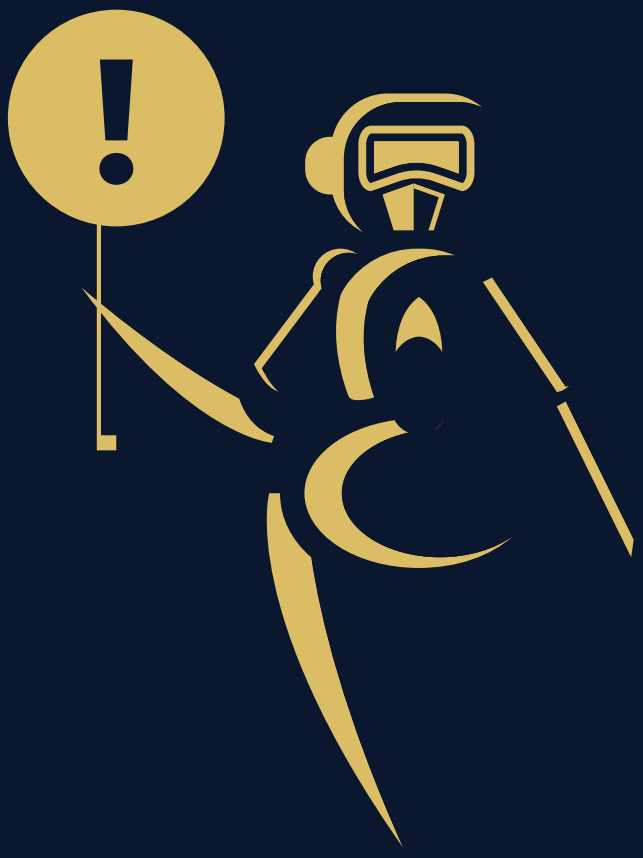


# DISCLAIMER

The information provided on this analysis document is only for general information and should not be used as a reason to invest.

FreshCoins Team will take no payment for manipulating the results of this audit.

The score and the result will stay on this project page information on our website https://freshcoins.io
FreshCoins Team does not guarantees that a project will not sell off team supply, or any other scam strategy ( RUG or Honeypot etc )



# INTRODUCTION

FreshCoins (Consultant) was contracted by
Weebs MasterChef - Smart Contract (Customer) to conduct a Smart Contract Code Review and Security Analysis.

0xd9D5Cc9ABD009F3F65fFC2A4043aC878Df2ecEC7

Network: TESTNET Binance Smart Chain (BSC)

This report presents the findings of the security assessment of Customer's smart contract and its code review conducted on 06/09/2024

# AUDIT OVERVIEW

71

Security Score

98

**Static Scan**
Automatic scanning for common vulnerabilities

82

**ERC Scan**
Automatic checks for ERC's conformance

0 High

2 Medium

2 Low

0 Optimizations

0 Informational

| No. | Issue description | Checking Status |
|---|---|---|
| 1 | Compiler Errors / Warnings | Passed |
| 2 | Reentrancy and Cross-function | Medium |
| 3 | Front running | Low |
| 4 | Timestamp dependence | Passed |
| 5 | Integer Overflow and Underflow | Passed |
| 6 | Reverted DoS | Low |
| 7 | DoS with block gas limit | Passed |
| 8 | Methods execution permissions | Passed |
| 9 | Exchange rate impact | Passed |
| 10 | Malicious Event | Passed |
| 11 | Scoping and Declarations | Passed |
| 12 | Uninitialized storage pointers | Passed |
| 13 | Design Logic | Passed |
| 14 | Safe Zeppelin module | Passed |

# PRIVILEGES

## deposit function line 222

uint256 public maxPoolSize;

```
function deposit(uint256 _pid, uint256 _amount) public {

    require (totalStaked + _amount < maxPoolSize, 'This pool is full');

    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    if (user.amount > 0) {
      uint256 pending = (user.amount * pool.accWeebsPerShare / 1e12) - user.rewardDebt;
      if(pending > 0) {
        _safeWeebsTransfer(msg.sender, pending);
      }
    }
    if (_amount > 0) {
      pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
      user.amount += _amount;
    }
    user.rewardDebt = user.amount * pool.accWeebsPerShare / 1e12;
    user.depositBlock = block.number;
    pool.totalStaked += _amount;
    emit Deposit(msg.sender, _pid, _amount);
}
```

**Purpose: Allows users to deposit LP (Liquidity Provider) tokens into a specified pool, staking them in return for rewards.**

### How it works:

**require(totalStaked + _amount < maxPoolSize, 'This pool is full'): This condition ensures that users can only deposit tokens if the total staked amount does not exceed the pool's maximum capacity (maxPoolSize is set to 600000000000000000000000000).**

**updatePool(_pid): This updates the pool's reward variables before proceeding to ensure rewards are correctly calculated for the user.**

**Pending Reward Calculation: If the user has a previously deposited amount (user.amount > 0), their pending reward is calculated using the formula:**

**pending = (user.amount * pool.accWeebsPerShare / 1e12) - user.rewardDebt.**

**If a pending reward exists, it is transferred to the user (_safeWeebsTransfer(msg.sender, pending)).**

**Token Transfer: The deposited LP tokens are transferred from the user to the contract using safeTransferFrom, and the user's amount is updated (user.amount += _amount).**

**Update Reward Debt: The rewardDebt is updated to reflect the user's new position in the pool: user.rewardDebt = user.amount * pool.accWeebsPerShare / 1e12.**

**Tracking Deposit Block: The block number of the deposit is recorded (user.depositBlock = block.number), which is important for penalty calculations during withdrawals.**

**Event Emission: The Deposit event is emitted to log the deposit action.**

## General Security Assessment:

### 1. Zero Amount Handling

The function doesn't prevent zero-value deposits. Allowing _amount == 0 would still incur gas fees for the user without making any meaningful deposit. This might also cause unexpected behaviors elsewhere in the contract, especially if not all functions handle zero-value deposits correctly.

Recommendation: Add a check to disallow zero-value deposits:

```
require(_amount > 0, "Deposit amount must be greater than zero");
```

**2. Potential for Reentrancy**: The function calls an external contract (pool.lpToken.safeTransferFrom) to transfer tokens from the user to the contract. This external call could potentially introduce a reentrancy vulnerability if the external contract or token has malicious behavior.

Recommendation: Implement a reentrancy guard by updating user balances and state before making any external calls. In this case, moving user.amount += _amount; and user.rewardDebt = user.amount * pool.accWeebsPerShare / 1e12; before the safeTransferFrom call would mitigate this risk. Alternatively, you can use the ReentrancyGuard contract from OpenZeppelin.

**3. Pending Reward Calculation**: The formula to calculate the pending reward is correct and follows common practice in staking contracts. However, there is a potential issue with large values leading to overflow, though this is less of a concern if you are using Solidity version >=0.8, as overflow protection is automatically enforced. In earlier versions of Solidity, you would need to use SafeMath.

### 4. Zero Reward Edge Case

In some cases, pending could be zero, and in that case, _safeWeebsTransfer(msg.sender, pending) would result in a gas-wasting call. While this is a minor inefficiency, it could be improved by adding a check before transferring

### 5. Pool Size Check

The line require(pool.totalStaked + _amount < pool.maxPoolSize, 'This pool is full'); ensures that the pool's maximum size is not exceeded. However, this check might result in a situation where the pool is slightly overfilled due to block mining timing.

Potential Issue: If the exact max pool size is needed, use <= instead of < to prevent deposits that cause overflows

## withdraw function line 246

uint256 public maxPoolSize;

```
function deposit(uint256 _pid, uint256 _amount) public {

    require (totalStaked + _amount < maxPoolSize, 'This pool is full');

    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    if (user.amount > 0) {
      uint256 pending = (user.amount * pool.accWeebsPerShare / 1e12) - user.rewardDebt;
      if(pending > 0) {
        _safeWeebsTransfer(msg.sender, pending);
      }
    }
    if (_amount > 0) {
      pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
      user.amount += _amount;
    }
    user.rewardDebt = user.amount * pool.accWeebsPerShare / 1e12;
    user.depositBlock = block.number;
    pool.totalStaked += _amount;
    emit Deposit(msg.sender, _pid, _amount);
}
```

**Purpose: Allows users to withdraw their staked LP tokens and claim rewards. If the withdrawal is made early, a penalty is applied.**

### How it works:

require(user.amount >= _amount, "withdraw: not good"): Ensures the user is trying to withdraw an amount they actually have staked.

updatePool(_pid): Updates the pool's reward variables, similar to the deposit function. (check page 5)

Reward Calculation: If rewards are enabled (totalReward > 0), pending rewards are calculated using the same formula as the deposit function, and the user receives them.

Penalty Calculation: If the user is withdrawing early, a penalty is applied based on the time staked. The _calculatePenalty function determines the penalty based on the block duration since the deposit.

Penalty percentages: less than 2 weeks = 25%, less than 4 weeks = 15%, less than 6 weeks = 10%, less than 8 weeks = 5%, 8 weeks or more = 0%

The penalty is sent to the devaddr, and the remaining amount (totalWithdraw = _amount - penalty) is transferred back to the user.

Update Reward Debt: Similar to the deposit function, the user's rewardDebt is updated: user.rewardDebt = user.amount * pool.accWeebsPerShare / 1e12.

Update Pool's Total Staked: The pool's total staked amount is reduced by the withdrawn amount.

Event Emission: The Withdraw event is emitted to log the withdrawal action.

## General Security Assessment:

### 1. Potential for Reentrancy

The function does make external token transfer calls with pool.lpToken.safeTransfer after updating state variables (user.amount and user.rewardDebt). This structure minimizes reentrancy risks because state is updated before external calls.

Good Practice: The function structure follows good practice by updating the user's balance (user.amount) and pool state (pool.totalStaked) before executing the token transfer, reducing the risk of reentrancy.

### 2. Penalty Calculation Logic

The penalty calculation is handled by _calculatePenalty. While the calculation itself seems correct, it's worth reviewing that the block numbers are handled properly to avoid manipulation of penalty conditions.

Potential Issue: If the block numbers are manipulated (e.g., via flash loans or mining manipulation), users might attempt to avoid penalties. Ensure that the block calculation (block.number - user.depositBlock) correctly accounts for this possibility.

Improvement: You could consider using a timestamp-based system instead of block numbers, which can be harder to manipulate. However, this would require significant changes across the contract.

### 3. Pending Reward Logic

This is a common formula used in staking contracts and works correctly as long as Solidity's overflow protection is enforced (Solidity >=0.8 automatically prevents overflows).

### 4. Gas Optimization

The line pool.lpToken.safeTransfer(devaddr, penalty); could be optimized to check if penalty > 0 before making the external transfer, avoiding unnecessary gas usage for zero-penalty cases:

---

## claimReward function line 135

```
function claimReward(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    if (user.amount > 0) {
      uint256 pending = (user.amount * pool.accWeebsPerShare / 1e12) - user.rewardDebt;
      if(pending > 0) {
        _safeWeebsTransfer(msg.sender, pending);
      }
      user.rewardDebt = user.amount * pool.accWeebsPerShare / 1e12;
    }
}
```

Purpose: Allows users to claim their pending rewards without withdrawing their staked LP tokens.

How it works:

updatePool(_pid): Ensures the pool's reward variables are up to date.

Reward Calculation: If the user has staked LP tokens (user.amount > 0), their pending reward is calculated:

pending = (user.amount * pool.accWeebsPerShare / 1e12) - user.rewardDebt.

Reward Transfer: If the user has any pending reward (pending > 0), the reward is transferred using _safeWeebsTransfer.

Update Reward Debt: After claiming the reward, rewardDebt is updated to the current position:

user.rewardDebt = user.amount * pool.accWeebsPerShare / 1e12.

## General Security Assessment:

### 1. Avoid Redundant Reward Debt Updates

The function updates user.rewardDebt even when no reward transfer occurs (i.e., when pending == 0). This is unnecessary and can be optimized to avoid a redundant state update. The reward debt update should only happen if the pending reward is greater than zero.

Recommendation: Update rewardDebt only when a reward transfer occurs

### 2. Resource Exhaustion

Since the function computes rewards based on the user's staked amount and the pool's accumulated rewards per share, there is no direct threat of resource exhaustion attacks. However, users with large balances may consume more gas due to the updatePool and reward calculations.

There is no immediate risk of denial-of-service (DoS) via resource exhaustion in the current implementation. However, in scenarios with many users or very large pools, gas costs could become high during pool updates. Optimizing for gas, especially in pool reward updates, would mitigate this.

## add function line 119

```
function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
    if (_withUpdate) {
      massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint += _allocPoint;
    poolInfo.push(PoolInfo({
      lpToken: _lpToken,
      allocPoint: _allocPoint,
      lastRewardBlock: lastRewardBlock,
      accWeebsPerShare: 0,
      maxPoolSize: maxPoolSize,
      totalStaked: 0
    }));
}
```

**Purpose: Allows the owner to add a new LP token pool to the contract.**

### How it works:

**Optional Mass Update: If _withUpdate is true, the function will call massUpdatePools() to update reward variables across all pools before adding the new pool.**

**Add New Pool: The new pool is added with the provided allocation points (_allocPoint) and LP token address (_lpToken). The allocation points dictate how many WEEBS rewards the pool will receive relative to other pools.**

---

## set function line 150

```
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 - prevAllocPoint + _allocPoint;
      // updateStakingPool();
    }
}
```

**Purpose: Allows the owner to update the allocation points for an existing pool, which affects how rewards are distributed.**

### How it works:

**Optional Mass Update: Similar to add, if _withUpdate is true, all pools are updated first.**

**Update Allocation Points: The pool's allocation points are updated, and the total allocation points are adjusted accordingly.**

## updatePool function line 204

```
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 weebsReward = multiplier * weebsPerBlock * pool.allocPoint / totalAllocPoint;

        pool.accWeebsPerShare = pool.accWeebsPerShare + (weebsReward * 1e12 / lpSupply);
        pool.lastRewardBlock = block.number;
    }
```

**Purpose: Updates a single pool's reward variables based on the latest block number.**

### How it works:

**if (block.number <= pool.lastRewardBlock): If the block number hasn't advanced since the last update, the function returns without making changes.**

**Reward Calculation: The reward is calculated using the multiplier (from the last reward block to the current block), the pool's allocation points, and the total allocation points across all pools. The accWeebsPerShare is updated accordingly to reflect the additional rewards per LP token.**

---

## switchPenalty function line 275

```
bool public penaltyInEffect = true;
```

```
function switchPenalty(bool _penaltyOn) public onlyOwner {
        penaltyInEffect = _penaltyOn;
    }
```

**Purpose: Enables or disables the penalty system for early withdrawals.**

### How it works:

**The owner can toggle the penaltyInEffect flag. If true, penalties will be applied during withdrawals. (check page 6 for more details about penalty values)**

## pendingWeebs function line 181

```
function pendingWeebs(uint256 _pid, address _user) external view returns (uint256) {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][_user];
    uint256 accWeebsPerShare = pool.accWeebsPerShare;
    uint256 lpSupply = pool.lpToken.balanceOf(address(this));
    if (block.number > pool.lastRewardBlock && lpSupply != 0) {
      uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
      uint256 weebsReward = multiplier * weebsPerBlock * pool.allocPoint/ totalAllocPoint;
      accWeebsPerShare += (weebsReward * 1e12 / lpSupply);
    }
    return (user.amount * accWeebsPerShare / 1e12) - user.rewardDebt;
}
```

**Purpose: Provides a view of the user's pending WEEBS rewards that have not been claimed.**

**How it works:**

**Reward Calculation: First, the pool's current accWeebsPerShare is calculated based on the latest reward multiplier and staked LP tokens. Then, the user's pending reward is computed as: (user.amount * accWeebsPerShare / 1e12) - user.rewardDebt.**

---

## updateMultiplier function line 101

```
uint256 public BONUS_MULTIPLIER = 1;
```

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

**Purpose: Allows the owner to adjust the reward multiplier, changing the rate at which WEEBS tokens are distributed.**
**Updating the reward multiplier can significantly impact the reward distribution, potentially leading to issues such as economic impact and potential exploitation**

---

## updateTotalReward function line 105

```
uint256 public totalReward = 0;
```

```
function updateTotalReward(uint256 newTotalReward) public onlyOwner {
    totalReward = newTotalReward;
}
```

**Purpose: Updates the total amount of WEEBS tokens allocated for rewards distribution.**
**Changing the total reward value can significantly affect reward distribution. If not managed properly, it can lead to unintended consequences such as excessive or insufficient rewards and potential exploitation**

# CONCLUSION AND ANALYSIS

Smart Contracts within the scope were manually reviewed and analyzed with static tools.

Audit report overview contains all found security vulnerabilities and other issues in the reviewed code.

Found no HIGH issues during the first review.

# TECHNICAL DISCLAIMER

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The audit can't guarantee the explicit security of the audited project / smart contract.