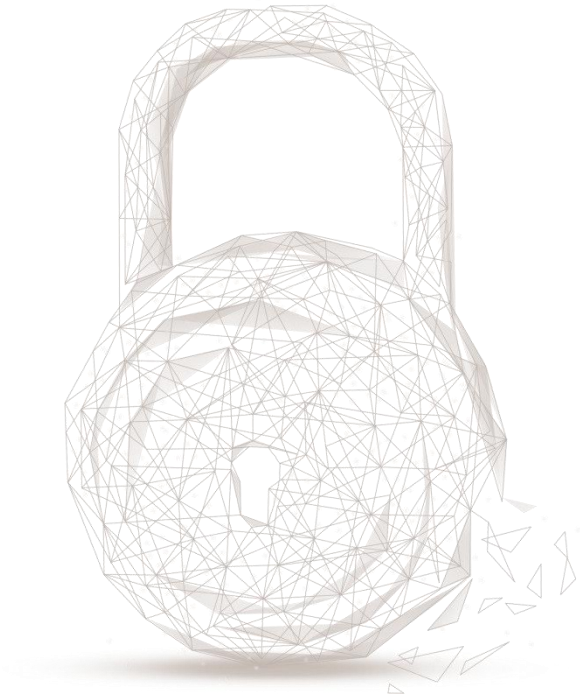
BEOSIN
Blockchain Security

Smart contract security audit report



Audit Number: 202109061830

Report Inquiry Name: LAVASWAP

Smart Contract Name:

MasterChef

Smart Contract Address:

0x155291E78823C53CA6995d114B5FfaCc42dd6c2c

Smart Contract Address Link:

https://bscscan.com/address/0x155291E78823C53CA6995d114B5FfaCc42dd6c2c#code

Start Date: 2021.08.30

Completion Date: 2021.09.06

Overall Result: Pass

Audit Team: Beosin (Chengdu LianAn) Technology Co. Ltd.

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	Compiler Version Security	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		SafeMath Features	Pass
		require/assert Usage	Pass
		Gas Consumption	Pass
		Visibility Specifiers	Pass
		Fallback Usage	Pass
2	General Vulnerability	Integer Overflow/Underflow	Pass
		Reentrancy	Pass
		Pseudo-random Number Generator (PRNG)	Pass

		Transaction-Ordering Dependence	Pass
		DoS (Denial of Service)	Pass
		Access Control of Owner	Pass
		Low-level Function (call/delegatecall) Security	Pass
		Returned Value Security	Pass
		tx.origin Usage	Pass
		Replay Attack	Pass
		Overriding Variables	Pass
3	Business Security	Business Logics	Pass
		Business Implementations	Pass

Note: Audit results and suggestions in code comments

Disclaimer: This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin (Chengdu LianAn) Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin (Chengdu LianAn) Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin (Chengdu LianAn) Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin (Chengdu LianAn) Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin (Chengdu LianAn) Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin (Chengdu LianAn). Due to the technical limitations of any organization, this report conducted by Beosin (Chengdu LianAn) still has the possibility that the entire risk cannot be completely detected. Beosin (Chengdu LianAn) disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin (Chengdu LianAn).

Audit Results Explained:

Beosin (Chengdu LianAn) Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of smart contracts LAVASWAP projects, including Coding Standards, Security, and Business Logic. **The LAVASWAP contracts projects passed all audit items. The overall result is Pass. The smart contract is able to function properly.**

1. Coding Conventions

Check the code style that does not conform to Solidity code style.

1.1 Compiler Version Security

- Description: Check whether the code implementation of current contract contains the exposed solidity compiler bug.
- Result: Pass

1.2 Deprecated Items

- Description: Check whether the current contract has the deprecated items.
- Result: Pass

1.3 Redundant Code

- Description: Check whether the contract code has redundant codes.
- Result: Pass

1.4 SafeMath Features

- Description: Check whether the SafeMath has been used. Or prevents the integer overflow/underflow in mathematical operation.
- Result: Pass

1.5 require/assert Usage

- Description: Check the use reasonability of 'require' and 'assert' in the contract.
- Result: Pass

1.6 Gas Consumption

- Description: Check whether the gas consumption exceeds the block gas limitation.
- Result: Pass

1.7 Visibility Specifiers

- Description: Check whether the visibility conforms to design requirement.
- Result: Pass

1.8 Fallback Usage

- Description: Check whether the Fallback function has been used correctly in the current contract.
- Result: Pass

2. General Vulnerability

Check whether the general vulnerabilities exist in the contract.

2.1 Integer Overflow/Underflow

- Description: Check whether there is an integer overflow/underflow in the contract and the calculation result is abnormal.
- Result: Pass

2.2 Reentrancy

- Description: An issue when code can call back into your contract and change state, such as withdrawing BNB.
- Result: Pass

2.3 Pseudo-random Number Generator (PRNG)

- Description: Whether the results of random numbers can be predicted.
- Result: Pass

2.4 Transaction-Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Pass

2.5 DoS (Denial of Service)

- Description: Whether exist DoS attack in the contract which is vulnerable because of unexpected reason.
- Result: Pass

2.6 Access Control of Owner

- Description: Whether the owner has excessive permissions, such as malicious issue, modifying the balance of others.
- Result: Pass

2.7 Low-level Function (call/delegatecall) Security

- Description: Check whether the usage of low-level functions like call/delegatecall have vulnerabilities.
- Result: Pass

2.8 Returned Value Security

- Description: Check whether the function checks the return value and responds to it accordingly.
- Result: Pass

2.9 tx.origin Usage

- Description: Check the use secure risk of 'tx.origin' in the contract. In this project, the contract.
- Result: Pass

2.10 Replay Attack

- Description: Check whether the implement possibility of Replay Attack exists in the contract.
- Result: Pass

2.11 Overriding Variables

- Description: Check whether the variables have been overridden and lead to wrong code execution.
- Result: Pass

3. Business Security

3.1 add function

- Description: As shown in the figure below, the contract implements the *add* function to add a pool, and the contract owner can call this function to add a pool for users to LP and obtain rewards. When calling *add* to add a pool, *lastRewardBlock* and *totalAllocPoint* will be updated, and pool related information will be stored. Please do not add the same LP tokens, this will cause the reward to be messed.

```
function add(uint256 _allocPoint, IERC20 _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,
        accSushiPerShare: 0
    }));
}
```

Figure 1 source code of *add* function

- Related functions: *massUpdatePools, add*

- Audit Recommendations: The MasterChef type stake pool was designed at a time when there were no deflationary tokens, so the developers did not consider the impact that such tokens could have. Some project owners used the old MasterChef tokens and added deflationary tokens or rewards as stake tokens when developing the code, which led to various malicious attacks or anomalies. As of now, there are two types of problems with the MasterChef type stake pool: first, there is no special handling of inflation-deflation tokens, and the actual number of tokens transferred to the contract is not checked to be the same as the number filled in when the function is called; second, reward tokens are added as stake tokens, resulting in anomalies in the reward calculation. The root cause of both types of problems still lies in the fact that the balanceOf function is used to obtain the amount of stake when calculating the reward. It is recommended that the project owner use a separate variable as the record of stake quantity when developing the code of MasterChef type stake pool, and then use this variable to get the stake token quantity when calculating the reward, instead of using the balanceOf function.

- Result: Pass

3.2 set function

- Description: As shown in the figure below, the contract implements the *set* function to set the reward distribution ratio of the pool. The contract owner can call this function to set the reward distribution ratio of the pool. The modification of the pool reward distribution ratio will affect the lava reward when the user withdraws the LP tokens.

```
function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public
onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add
    (_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
}
```

Figure 2 source code of *set* function

- Related functions: *set, massUpdatePools*
- Audit Recommendations: It is recommended to use governance contracts to manage owner permissions.
- Result: Pass

3.3 updateSushiPerBlock function

- Description: As shown in the figure below, the contract implements the *updateSushiPerBlock* function to set the reward for each block, which can only be called through the owner.

```
function updateSushiPerBlock(uint256 _sushiPerBlock) public onlyOwner{
    massUpdatePools();
    sushiPerBlock = _sushiPerBlock;
}
```

Figure 3 source code of *updateSushiPerBlock* function

- Related functions: *updateSushiPerBlock, massUpdatePools*
- Result: Pass

3.4 deposit function

- Description: As shown in the figure below, the contract implements the *deposit* function for users to deposit LP tokens. The user pre-authorizes the contract address and calls this function to LP tokens. When the *deposit* function deposits tokens, if the user's balance stored in the pool is greater than zero, the reward will be calculated and sent to the user's address. (Note that if the lava token balance in the contract is less than the reward value, only remaining the lava tokens in the contract will be sent to the user address) At the same time, the LP token will be sent to the pool, And the user's *amount* and *rewardDebt* will be updated.

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

Figure 4 source code of *deposit* function

```
function safeLavaTransfer(address _to, uint256 _amount) internal {
    uint256 lavaBal = lava.balanceOf(address(this));
    if (_amount > lavaBal) {
        lava.transfer(_to, lavaBal);
    } else {
        lava.transfer(_to, _amount);
    }
}
```

Figure 5 source code of *safeLavaTransfer* function

- Related functions: *deposit, safeLavaTransfer, updatePool, safeTransferFrom, transfer, balanceOf*
- Result: Pass

3.5 withdraw function

- Description: As shown in the figure below, the contract implements the *withdraw* function for users to withdraw LP tokens and lava rewards. Users can call this function to withdraw a specified number of LP tokens and lava rewards from the specified pool (requires the specified pool to exist and the number of LP tokens is greater than or equal to Withdrawal quantity). When users withdraw LP tokens and rewards, update the pool information, and transfer the designated LP tokens and lava to the user address. (Note that if the lava token balance in the contract is less than the reward value, only Remaining the lava tokens in the contract will be sent to the user address)

```
// Withdraw LP tokens from MasterChef.
function withdraw(uint256 _pid, uint256 _amount) public {
    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.accSushiPerShare).div(1e12).sub
    (user.rewardDebt);
    safeLavaTransfer(msg.sender, pending);
    user.amount = user.amount.sub(_amount);
    user.rewardDebt = user.amount.mul(pool.accSushiPerShare).div(1e12);
    pool.lpToken.safeTransfer(address(msg.sender), _amount);
    emit Withdraw(msg.sender, _pid, _amount);
}
```

Figure 6 source code of *withdraw* function

```
// Safe lava transfer function, just in case if rounding error causes pool to not have enough lavas.
function safeLavaTransfer(address _to, uint256 _amount) internal {
    uint256 lavaBal = lava.balanceOf(address(this));
    if (_amount > lavaBal) {
        lava.transfer(_to, lavaBal);
    } else {
        lava.transfer(_to, _amount);
    }
}
```

Figure 7 source code of *safeLavaTransfer* function

- Related functions: *updatePool, safeLavaTransfer, safeTransfer, transfer, balanceOf, withdraw*
- Result: Pass

3.6 updatePool function

- Description: As shown in the figure below, the contract implements the *updatePool* function to update the lava reward and pool information of the current block pool. Any user can call this function to update the pool's latest lava reward and information.(*block.number* must be greater than *lastRewardBlock* to call)

```
// 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 sushiReward = multiplier.mul(sushiPerBlock).mul(pool.allocPoint)
    .div(totalAllocPoint);
    pool.accSushiPerShare = pool.accSushiPerShare.add(sushiReward.mul(1e12).
    div(lpSupply));
    pool.lastRewardBlock = block.number;
}
```

Figure 8 source code of *updatePool* function

- Related functions: *updatePool, balanceOf, getMultiplier*
- Result: Pass

3.7 massUpdatePools function

- Description: The contract implements the *massUpdatePools* function to update all pools, first by obtaining the length of the pool, and then calling the function *updatePool* to update the pool in turn.

```
// Update reward vairables for all pools. Be careful of gas spending!
function massUpdatePools() public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        updatePool(pid);
    }
}
```

Figure 9 source code of *massUpdatePools* function

- Related functions: *massUpdatePools, updatePool*
- Result: Pass

4. Conclusion

Beosin(ChengduLianAn) conducted a detailed audit on the design and code implementation of the smart contracts LAVASWAP. The contracts LAVASWAP passed all audit items, The overall audit result is **Pass**.

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com