#HashEx
BLOCKCHAIN SECURITY

ShimmerSea

smart contracts final audit report

September 2022


hashex.org


contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the ShimmerSea team to perform an audit of their smart contract. The audit was conducted between 24/08/2022 and 05/09/2022.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at @ShimmerSea/shimmersea-contracts and @ShimmerSea/shimmersea-magiclum GitHub repositories after the commits fc5e952 and 20cbb9c respectively. Documentation was provided via pre-production website.

Update: the ShimmerSea team has responded to this report. The updated code is located in the same GitHub repositories after the cd03053 and 40dbbd4 commits.

2.1 Summary

Project name	ShimmerSea
URL	https://shimmersea.finance/
Platform	Shimmer Network
Language	Solidity

2.2 Contracts

Name	Address
DEX contracts	https://github.com/ShimmerSea/shimmersea-contracts/tree/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/dex
FarmUniV2Zap	https://github.com/ShimmerSea/shimmersea-contracts/blob/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/zaps/FarmUniV2Zap.sol
Interfaces	https://github.com/ShimmerSea/shimmersea-contracts/tree/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/interfaces
Misc contracts	https://github.com/ShimmerSea/shimmersea-contracts/tree/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/misc
PearlToken	https://github.com/ShimmerSea/shimmersea-contracts/blob/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/PearlToken.sol
LumToken	https://github.com/ShimmerSea/shimmersea-contracts/blob/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/LumToken.sol
RewardPool	https://github.com/ShimmerSea/shimmersea-contracts/blob/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/pools/RewardPool.sol
RestrictedLumPool	https://github.com/ShimmerSea/shimmersea-contracts/blob/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/pools/RestrictedLumPool.sol
TangleSeaMasterChef	https://github.com/ShimmerSea/shimmersea-contracts/blob/fc5e9527a8499d1d0eea151fe2d6118a5a8e7927/contracts/TangleSeaMasterChef.sol
MagicLum	https://github.com/ShimmerSea/shimmersea-magiclum/blob/20cbb9c1011c20a8b30620f057445e4b700c0d91/contracts/MagicLum.sol
TimeBasedMasterChefRewarder	https://github.com/ShimmerSea/shimmersea-magiclum/blob/20cbb9c1011c20a8b30620f057445e4b700c0d91/contracts/rewarder/TimeBasedMasterChefRewarder.sol

TangleSeaBooster	https://github.com/ShimmerSea/shimmersea-magiclum/blob/20cbb9c1011c20a8b30620f057445e4b700c0d91/contracts/TangleSeaBooster.sol

3. Found issues

Critical	2 (5%)
High	1 (3%)
Medium	10 (26%)
Low	8 (21%)
Info	18 (45%)

C1. DEX contracts

ID	Severity	Title	Status
C1-01	Low	TangleseaLibrary: code with no effect	Resolved
C1-02	Info	TangleseaFactory: fees distribution	Acknowledged

C2. FarmUniV2Zap

ID	Severity	Title	Status
C2-01	Medium	zapOut() function doesn't perform safety checks	Resolved
C2-02	Info	Typos	Resolved

C7. RewardPool

ID	Severity	Title	Status
C7-01	Medium	External functions can be used for phishing	Resolved
C7-02	Low	Gas optimizations	Resolved
C7-03	Info	Rewards aren't guaranteed	Acknowledged

C8. RestrictedLumPool

ID	Severity	Title	Status
C8-01	High	Staked funds may be transferred as reward	Resolved
C8-02	Medium	External functions can be used for phishing	Resolved
C8-03	Low	Gas optimizations	Resolved
C8-04	Info	Typos	Resolved

C9. TangleSeaMasterChef

ID	Severity	Title	Status
C9-01	Medium	Unfair distribution of awards without massUpdatePool()	Acknowledged
C9-02	Medium	External functions can be used for phishing	Resolved
C9-03	Low	Gas optimizations	Resolved
C9-04	Info	Typos	Resolved
C9-05	Info	Lack of events	Resolved

C9-06	Info	emergencywithdraw() doesn't notify the rewarder	Resolved
C9-07	Info	Tokens with fees on transfers aren't supported	Resolved
C9-08	Info	Lack of safety checks on input values	Resolved

C11. TimeBasedMasterChefRewarder

ID	Severity	Title	Status
C11-01	Critical	Rewarder is exposed to emergencyWithdraw() exploit	Resolved
C11-02	Medium	Rewards aren't guaranteed	Resolved
C11-03	Medium	Unfair distribution of awards without massUpdatePool()	Resolved
C11-04	Low	Gas optimizations	Resolved
C11-05	Info	Typos	Resolved

C12. TangleSeaBooster

ID	Severity	Title	Status
C12-01	Critical	emergencyWithdraw() problem	Resolved
C12-02	Medium	Unreliable EOA check	Resolved
C12-03	Medium	Rewards aren't guaranteed	Resolved
C12-04	Medium	Locked rewards	Resolved

C12-05	Low	Precision factor for float point calculations	Resolved
C12-06	Low	Error in EmergencyWithdraw()	Resolved
C12-07	Low	Gas optimizations	Resolved
C12-08	Info	Inconsistent comment	Resolved
C12-09	Info	Typos	Resolved
C12-10	Info	Gas consumption infinitely grows over time	Acknowledged
C12-11	Info	Total burn time can be set below lock time	Resolved
C12-12	Info	Discontinuity in _getBurnDebt() function	Resolved
C12-13	Info	Default visibility	Resolved
C12-14	Info	Lack of events	Resolved
C12-15	Info	Initialization problem	Resolved

4. Contracts

C1. DEX contracts

Overview

A typical fork of a UniswapV2 DEX with minor changes of customizable fee for each pair.

Issues

C1-01 TangleseaLibrary: code with no effect

Low Resolved

Code with no effect in `getReserves()`:

```
    function getReserves(...) internal view returns (...) {
        ...
        pairFor(factory, tokenA, tokenB);
        ...
    }
```

C1-02 TangleseaFactory: fees distribution

Info Acknowledged

Fee distribution differs from the documentation. 50% goes to the project owner instead of 25% going to the owner and 25% to the buyback of the LUM token.

C2. FarmUniV2Zap

Overview

Auxiliary contract for simplifying user interaction with the TangleSeaMasterChef contract.

Issues

C2-01 zapOut() function doesn't perform safety checks

Medium Resolved

The zap-out function doesn't perform security checks of output amounts after removing liquidity. Any calls of these functions can be sandwiched.

```
function zapOut(address lpToken, uint256 withdrawAmount) external {
    ...
    _removeLiqudity(address(pair), address(this));
    ...
    _returnAssets(tokens);
}

function _removeLiqudity(address pair, address to) private {
    IERC20(pair).safeTransfer(pair, IERC20(pair).balanceOf(address(this)));
    (uint256 amount0, uint256 amount1) = ITangleseaPair(pair).burn(to);

    require(amount0 >= minimumAmount, "UniswapV2Router: INSUFFICIENT_A_AMOUNT");
    require(amount1 >= minimumAmount, "UniswapV2Router: INSUFFICIENT_B_AMOUNT");
}
```

Recommendation

Any interaction between a user and pair should include the safety limits in the parameters.

C2-02 Typos

Info Resolved

Typos reduce the code's readability.Typos in 'directy', 'liqudity'.

C3. Interfaces

Overview

IMasterChef, IRewarder,
IRewardToken,IZap, IPermitToken, ITangleseaFactory, ITangleseaPair, ITangleseaRouter, and IWETH interfaces. No issues were found.

C4. Misc contracts

Overview

FeeVault is a simple vault contract with 2 different withdrawal functions: one for the `feeAddr` and the other for the owner.

Multicall and Multicall2 are well-known contracts for aggregated calls.

WETH9 is a wrapped ETH token contract.

No issues were found.

C5. PearlToken

Overview

An implementation of EIP-20 token standard built on the ERC20Permit extension from OpenZeppelin, which supports the EIP-712 signing. PearlToken is mintable by the owner (supposedly, TangleSeaMasterChef). No issues were found.

C6. LumToken

Overview

An implementation of the EIP-20 token standard built on the ERC20Permit extension from OpenZeppelin, which supports the EIP-712 signing. LumToken is mintable by the owner (supposedly, TangleSeaMasterChef). No issues were found.

C7. RewardPool

Overview

A single pool contract inspired by MasterChefV2 from Sushiswap.

Issues

C7-01 External functions can be used for phishing

Medium Resolved

`deposit()`, `withdrawAndHarvest()`, and `harvest()` functions receive `_to` address in parameters to deposit, withdraw or harvest reward. These functions can be used for phishing in case of hacked front-end.

```
function deposit(uint256 _amount, address _to) external nonReentrant {
    ...
    stakedToken.safeTransferFrom(address(msg.sender), address(this), _amount);
    ...
}

function harvest(address _to) public {
    ...
    rewardToken.safeTransfer(address(_to), _pending);
    ...
}

function withdrawAndHarvest(uint256 _amount, address _to) external nonReentrant {
```

```
        ...
        stakedToken.safeTransfer(address(_to), _amount);
        rewardToken.safeTransfer(address(_to), _pending);
        ...
    }
```

Recommendation

Add 2 external functions: one with `_to = msg.sender` and other with `require(msg.sender == desiredContract)`, alongside with the single internal function containing all the logic.

C7-02 Gas optimizations

Low Resolved

Several gas optimizations could be implemented:

1. `stakedToken`, `rewardToken`, `PRECISION_FACTOR` variables should be declared as `immutable`
2. unnecessary reads from storage: `user.amount` in `deposit()`;`user.amount`, `accTokenPerShare` in `withdrawAndHarvest()`;`user.amount` in `emergencyWithdraw()`;`bonusEndTime` in `stopReward()`;`poolLimitPerUser` in `updatePoolLimitPerUser()`;`startTime` in `updateStartAndEndTime()`;`lastRewardTime` in `pendingRewards()`;`lastRewardTime` in `_updatePool()`;`bonusEndTime` in `_getMultiplier()`;
3. In the updated code `msg.sender` is checked against `trustee` variable twice in the `harvestOnBehalf()` function.

C7-03 Rewards aren't guaranteed

Info Acknowledged

Several gas optimizations could be implemented:

1. `stakedToken`, `rewardToken`, `PRECISION_FACTOR` variables should be declared as `immutable`
2. unnecessary reads from storage: `user.amount` in `deposit()`;`user.amount`, `accTokenPerShare` in `withdrawAndHarvest()`;`user.amount` in `emergencyWithdraw()`;`bonusEndTime` in `stopReward()`;`poolLimitPerUser` in `updatePoolLimitPerUser()`;`startTime` in `updateStartAndEndTime()`;`lastRewardTime` in `pendingRewards()`;`lastRewardTime` in `_updatePool()`;`bonusEndTime` in `_getMultiplier()`.

C8. RestrictedLumPool

Overview

A single pool contract inspired by MasterChefV2 from Sushiswap. Accessible only for LUMI NFT holders. Works closely with the single pool of TangleSeaMasterChef contract.

Issues

C8-01 Staked funds may be transferred as reward

High Resolved

The contract's calculated reward from MasterChef may differ from actually pending. The discrepancy will lead to the rewards payments including users' staked tokens. There are two inaccuracies that can cause this discrepancy, i.e. `accRewardPerShare` being greater than it should.

Firstly, `rewardsPerSec` and/or `pool.allocPoint & totalAllocPoint` could have been adjusted between 2 `_updatePool()` calls.

Secondly, it's supposed all rewards from the `masterChefPool` are received by the RestrictedLumPool contract, which is generally not true if `masterChefPool` has other stakes.

```
    function _updatePool() internal {
        IMasterChef.PoolInfo memory masterChefPool = MASTERCHEF.poolInfo(MASTERCHEF_PID);
        ...
        uint256 totalAllocPoint = MASTERCHEF.totalAllocPoint();
        uint256 masterChefRewardsPerSec = MASTERCHEF.rewardsPerSec();
        uint256 rewards = nbSeconds.mul(masterChefRewardsPerSec).mul(masterChefPool.allocPo
int).div(totalAllocPoint);
        accRewardPerShare =
accRewardPerShare.add(rewards.mul(PRECISION_FACTOR).div(lpSupply));
        ...
    }

    function harvest(address _to) public {
        ...
```

```
        _updatePool();

        uint256 accRewards = (user.amount.mul(accRewardPerShare)).div(PRECISION_FACTOR);
        uint256 _pending = accRewards.sub(user.rewardDebt);
        safeTransfer(_to, _pending);
        ...
    }
```

Recommendation

We advise replacing pending reward amount estimation with `MASTERCHEF.pendingRewards(MASTERCHEF_PID, address(this))` and `MASTERCHEF.harvest()` them:

```
    function _updatePool() internal {
        uint256 rewards = MASTERCHEF.pendingRewards(MASTERCHEF_PID, address(this));
        MASTERCHEF.harvest()
        accRewardPerShare =
accRewardPerShare.add(rewards.mul(PRECISION_FACTOR).div(lpSupply));
        ...
    }
```

Also, this solution requires adding `pendingRewards()` to `IMasterChef` interface, removing `harvestFromMasterChef()`, modifying `safeTransfer()`, `pendingRewards()` and `init()`.

C8-02 External functions can be used for phishing

Medium Resolved

`deposit()`, `withdrawAndHarvest()`, and `harvest()` functions receive `_to` address in parameters to deposit, withdraw or harvest reward. These functions can be used for phishing in case of hacked front-end.

```
function deposit(uint256 _amount, address _to) external nonReentrant {
 ...
 UserInfo storage user = userInfo[_to];
 user.amount = user.amount.add(_amount);
 ...
 }
```

```
function harvest(address _to) public {
...
safeTransfer(_to, _pending);
...
}

function withdrawAndHarvest(uint256 _amount, address _to) external nonReentrant {
...
safeTransfer(_to, _pending);
safeTransfer(_to, _amount);
...
}
```

Recommendation

Add 2 external functions: one with `_to = msg.sender` and other with `require(msg.sender == desiredContract)`, alongside the single internal function containing all the logic.

C8-03 Gas optimizations

Low Resolved

Several gas optimizations could be implemented:

1. `MASTERCHEF_PID` and `MASTERCHEF` variables should be declared as `immutable`
2. `PRECISION_FACTOR` variable should be declared as `constant`
3. unnecessary reads from storage: `user.amount` in `deposit()`;`user.amount`, `accTokenPerShare` in `withdrawAndHarvest()`;`lastRewardTime`, `accRewardPerShare` in `pendingRewards()`; `lastRewardTime`, `accRewardPerShare` in `_updatePool()`;
4. L229 and L232 transfers can be done in one transaction.

C8-04 Typos

Info Resolved

Typos reduce the code's readability.Typos in 'alreay'.

C9. TangleSeaMasterChef

Overview

A contract inspired by MasterChefV2 from Sushiswap with additional optional Rewarder contracts for each pool.

Issues

C9-01 Unfair distribution of awards without massUpdatePool()

Medium Acknowledged

The reward distribution for pools, where the `updatePool()` function is rarely called, can become too small (unfair) if new pools are added or updated without the `_withUpdate` flag.

Recommendation

Force mass update without the flag.

C9-02 External functions can be used for phishing

Medium Resolved

`deposit()`, `withdrawAndHarvest()`, and `harvest()` functions receive `_to` address in parameters to deposit, withdraw or harvest reward. These functions can be used for phishing in case of hacked front-end.

```
function deposit(uint256 _amount, address _to) external nonReentrant {
    ...
    UserInfo storage user = userInfo[_pid][_to];
    user.amount = user.amount.add(_amount);
    ...
}

function harvest(address _to) public {
    ...
    safeRewardTransfer(_to, _pending);
```

```
        ...
    }

    function withdrawAndHarvest(uint256 _amount, address _to) external nonReentrant {
        ...
        safeRewardTransfer(_to, _pending);
        pool.lpToken.safeTransfer(_to, _amount);
        ...
    }
```

Recommendation

Add 2 external functions: one with `_to = msg.sender` and other with `require(msg.sender == desiredContract)`, alongside the single internal function containing all the logic.

C9-03 Gas optimizations

Low Resolved

Several gas optimizations could be implemented:

1. `checkPoolDuplicate()` is inefficient: mapping `address=>bool` saves gas or even duplicated pools could be allowed as `pool.lpSupply` is tracked
2. requirements in L145, L187 should be switched places, i.e. first check the address for zero then its `extcodesize`
3. unnecessary reads from storage: `startTime` in `add()`; `user.amount`, `pool.depositFeeBP` in `deposit()`; `user.amount`, `pool.accRewardPerShare` in `withdrawAndHarvest()`; `pool.lastRewardTime` in `pendingRewards()`;`pool.lastRewardTime`, `pool.accRewardPerShare`, `pool.lpSupply` in `_updatePool()`;

C9-04 Typos

Info Resolved

Typos reduce the code's readability.Typos in 'PRECSION', 'vairables', 'FUNCIONS', 'FUNCIONTS'

C9-05 Lack of events

Info Resolved

No events are emitted in the constructor section except for the contract's parameters being changed.

C9-06 emergencywithdraw() doesn't notify the rewarder

Info Resolved

The `emergencyWithdraw()` function doesn't call the `Rewarder.onNativeReward()`. This may result in users' loss of additional rewards and staked amounts inconsistency. Better to use `try/catch` or `.call()` without success check to properly notify the Rewarder contract without compromising the possibility of emergency withdrawal.

C9-07 Tokens with fees on transfers aren't supported

Info Resolved

The `deposit()` function doesn't check the actual transferred amount, which is mandatory in the case of tokens with fees on transfers. The owner must not add pools with such tokens.

C9-08 Lack of safety checks on input values

Info Resolved

Constructor setters aren't subjected to max value filtering, unlike the separate set functions for the same parameters.

There's no `validatePool()` call in `set()` function.

C10. MagicLum

Overview

A governance token, an implementation of the EIP-20 token standard built on ERC20Permit and ERC20Votes extensions from OpenZeppelin, which supports the EIP-712 signing and snapshots for voting. MagicLum is mintable by the owner (supposedly, TangleSeaBooster). No issues were found.

C11. TimeBasedMasterChefRewarder

Overview

A rewarder contract for additional optional rewards for users of TangleSeaMasterChef contract. It can operate with multiple pools of MasterChef with the same reward token.

Issues

C11-01 Rewarder is exposed to emergencyWithdraw() exploit

Critical Resolved

`TangleSeaMasterChef.emergencyWithdraw()` doesn't notify the rewarder about changed user amount, making it possible to

```
TangleSeaMasterChef.deposit(pid, amount, aliceAddr);
emergencyWithdraw(pid);
deposit(pid, amount, bobAddr)
emergencyWithdraw(pid);
...
```

scheme to acquire additional rewards for the same amount of `lpTokens`. The problem with the `onNativeReward()` function is it doesn't use the `_pending` input parameter but only the `newLpAmount`.

Recommendation

See the possible solutions in 'emergencywithdraw() doesn't notify the rewarder issue' in the 'C9.TangleSeaMasterChef' section.

C11-02 Rewards aren't guaranteed

Medium Resolved

Rewards are calculated linearly in time with the `rewardPerSecond` parameter. However, the actual reward balance of the contract is going to be replenished from an external source that is out of the scope of this audit. But the `onNativeReward()` function cuts the rewarded amount if the balance is low, so users may lose part of their rewards.

```
    function onNativeReward(
        uint256 pid,
        address userAddress,
        address recipient,
        uint256,
        uint256 newLpAmount
    ) external override onlyMasterChef {
        ...
        pending = userPoolInfo.amount.mul(pool.accRewardTokenPerShare).div(accTokenPrecisio
n).sub(userPoolInfo.rewardDebt);
        if (pending > rewardToken.balanceOf(address(this))) {
            pending = rewardToken.balanceOf(address(this));
        }
        rewardToken.safeTransfer(recipient, pending);
        ...
    }
```

Recommendation

Store the unpaid rewards for each user and allow claiming them later.

C11-03 Unfair distribution of awards without massUpdatePool()

Medium Resolved

Changes in allocation points by calling the `add()` and `set()` functions also cause a re-distribution of historical rewards from the last pool's update time. If a new pool is added, the total allocation sum of existed pools usually becomes lower than total allocation points, so historical rewards become partially locked.

Recommendation

Call `massUpdatePools(masterchefPoolIds)` in the `add()` and `set()` functions.

C11-04 Gas optimizations

Low Resolved

Several gas optimizations should be implemented:

1. unnecessary reads from storage: `userPoolInfo.amount`, `rewardToken` in `onNativeReward()`;

C11-05 Typos

Info Resolved

Typos reduce the code's readability.Typos in 'withdraaw', 'dont'.

C12. TangleSeaBooster

Overview

A staking contract that supposedly allows to stake LumToken for MagicLum reward. Staked tokens may be burned by anyone for 2% of the burned amount, burnable amount linearly grows in time up to 100% after a fixed time (default value is 30 days, can be changed but not below 3 days).

Issues

C12-01 emergencyWithdraw() problem

Critical Resolved

It's claimed in the documentation of the booster contract that deposited tokens are locked for 72h, although they still can be withdrawn but with a 20% burn penalty. In fact, provided code contains two emergency withdraw functions `emergencyUnlock()` and `emergencyWithdraw()`, if the first one behaves as intended, whereas the `emergencyWithdraw()` allows to withdraw assets without any check and fee disregarding lock period. Users may even save the rewards if call `harvest()` before. The function may be abused to reset last `lastUserBurnTime`.

```
function emergencyWithdraw() external nonReentrant {
  UserInfo storage user = userInfo[msg.sender];

  uint256 amountToTransfer = user.amount;

 stakedSupply = stakedSupply.sub(amountToTransfer);

  // clear all
  user.amount = 0;
  user.rewardDebt = 0;
  user.rewardClaim = 0;
  user.burnMultiplier = 0;
  user.lastUserBurnTime = block.timestamp;
  user.changedAt = block.timestamp;


  if (amountToTransfer > 0) {
    stakedToken.safeTransfer(address(msg.sender), amountToTransfer);
  }

  emit EmergencyWithdraw(msg.sender, user.amount);
}
```

Recommendation

We believe the function was added by accident and can be simply deleted from the code without any adverse impact on the existing logic.

C12-02 Unreliable EOA check

Medium Resolved

`notContract()` check is not strict enough, it can be bypassed by an 'under construction' contract. See more in OpenZeppelin's docs.

Recommendation

Use `require(msg.sender == tx.origin)` to check for EOA.

C12-03 Rewards aren't guaranteed

Medium Resolved

Rewards aren't guaranteed, `initialize()` and `addRewards()` functions don't ensure the available rewards, and `emergencyRewardWithdraw()` allows the owner to seize the rewards before the distribution. `safeRewardTransfer()` would erase accumulated user's rewards if there's not enough tokens on the balance.

Recommendation

Impossible to claim the amount of rewards, they should be stored individually in `user.rewardClaim`.

C12-04 Locked rewards

Medium Resolved

The `_updatePool()` function mints zero reward if called after the `bonusEndTime` timestamp (see `_rewardForSlices()`), all accumulated since the last update rewards would be lost. It's impossible to collect all the rewards as the `bonusEndTime` equals the last slice's end time, so at least the last second reward is always locked.

```
function _updatePool() internal {
```

```
    uint256 blockTime = block.timestamp;
    if (blockTime <= lastRewardTime) {
      return;
    }
    ...
    uint256 gpReward = _rewardForSlices(blockTime, fromSlice, toSlice);
    rewardToken.mint(address(this), gpReward);
    ...
  }

  function _rewardForSlices(
    uint256 blockTime,
    MonthlyReward memory fromSlice,
    MonthlyReward memory toSlice
  ) internal view returns (uint256 reward) {
    if (blockTime >= bonusEndTime) {
      return 0;
    }
    ...
  }
```

Recommendation

Consider allowing reward collection after the `bonusEndTime`. In that case, `_rewardForSlices()` may return the calculated amount without changes after the end of the last slice.

C12-05 Precision factor for float point calculations

Low Resolved

`PRECISION_FACTOR` could be initialized down to the value of `10**1`, which is too low to be used with float calculations. Consider tightening the requirements for decimals, e.g. require it less than 20:

```
require(decimalsRewardToken < 30, "Must be inferior to 30");
```

C12-06 Error in EmergencyWithdraw()

Low Resolved

A wrong value is emitted in `EmergencyWithdraw()` inside the `emergencyWithdraw()` function.

```
function emergencyWithdraw() external nonReentrant {
  ...
  uint256 amountToTransfer = user.amount;
  user.amount = 0;
  emit EmergencyWithdraw(msg.sender, user.amount);
  ...
}
```

C12-07 Gas optimizations

Low Resolved

Several gas optimizations could be implemented:

1. `stakedToken` variable should be declared as `immutable`
2. `REWARD_FEE`, `WITHDRAW_FEE` variables should be declared as `constant`
3. unnecessary reads from storage: `slices[slices.length - 1].endTime` in `addRewards()`; `isAllowListEnabled` in `enableAllowlist()`;`user.amount`, `time_locked` in `deposit()`; `user.amount`, `accTokenPerShare` in `harvest()`;`user.amount`, `accTokenPerShare` in `withdrawAndHarvest()`;`user.amount` in `emergencyWithdraw()`;`user.amount` in `emergencyUnlock()`;`burnBatchCursor`, `burnBatchSize` in `burnNextBatch()`;`lastRewardTime` in `pendingRewards()`;`slices.length` in `getMonthlyReward()`; `user.amount`, `accTokenPerShare` in `_burnAmountAndClaimReward()`;`lastRewardTime` in `_updatePool()`; full `MonthlyReward` struct inside the loop in `getMonthlyReward()`.

C12-08 Inconsistent comment

Info Resolved

An inconsistent comment on L66. Either the value or the comment should be updated.

```
MIN_AFTER_BURN_TIME = 60 * 60 * 24 * 3; // Two days
```

C12-09 Typos

Info Resolved

Typos reduce the code's readability.Typos in 'penality', 'mutliplier', 'transferd', 'completly', 'FUNCIONS', 'Reciever', 'raimining'

C12-10 Gas consumption infinitely grows over time

Info Acknowledged

`_updatePool()` gas consumption linearly increases over time.

```
function _updatePool() internal {
  ...
  MonthlyReward memory fromSlice = getMonthlyReward(lastRewardTime);
  MonthlyReward memory toSlice = getMonthlyReward(blockTime);
  ...
}

function getMonthlyReward(uint256 blockTime)
  public
  view
  returns (MonthlyReward memory)
{
  for (uint256 i = 0; i < slices.length; i++) {
    MonthlyReward memory slice = slices[i];
    if (blockTime >= slice.startTime && blockTime <= slice.endTime) {
      return slice;
    }
  }
  return slices[slices.length - 1];
}
```

C12-11 Total burn time can be set below lock time

Info Resolved

The lower limit of `burnAfterTime` parameter is 3 days (`MIN_AFTER_BURN_TIME = 60 * 60 * 24 * 3`), while the maximum for lock time is 2 weeks (`MAX_LOCK_TIME = 60 * 60 * 24 * 14`). Setting the `burnAfterTime` less than `time_locked` may cause users' staked funds to be prematurely burned.

We suggest adding the corresponding safety checks to the `updateTimeLocked()` and `updateBurnAfterTime()` functions to ensure `burnAfterTime > time_locked` invariant.

C12-12 Discontinuity in _getBurnDebt() function

Info Resolved

Discontinuity in the `_getBurnDebt()` return value: if `user.lockedAt < user.lastUserBurnTime`, the returned value increases linearly in time from 0 to `(user.lockedAt - user.lastUserBurnTime + burnAfterTime) * user.burnMultiplier` and then abruptly jumps to `user.amount.mul(PRECISION_FACTOR)`. We recommend adding the NatSpec description with assumptions.

C12-13 Default visibility

Info Resolved

No visibility for the `time_locked` variable is specified. `internal` visibility is used by default.

C12-14 Lack of events

Info Resolved

Not enough events in onlyOwner parameter-changing functions. This complicates the debugging and possible recreation of the contract's state.

C12-15 Initialization problem

Info Resolved

There's no safety check of `_monthlyRewardsInWei[]` parameter of the `init()` function. Setting it to zero-length array would cause a complete contract malfunction as `addRewards()` will be non-callable.

5. Conclusion

2 critical, 1 high, 10 medium, 8 low severity issues were found during the audit. 2 critical, 1 high, 9 medium, 8 low issues were resolved in the update.

The reviewed contracts are highly dependent on the owner’s account. Users using the project have to trust the owner and that the owner's account is properly secured.

We strongly suggest adding documentation as well as increasing unit and functional tests coverage for all contracts.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY