# HashEx
BLOCKCHAIN SECURITY

# ShimmerSea FairLaunch & TangleSeaBoosterV2

smart contracts
preliminary audit report
for internal use only

December 2022

hashex.org

contact@hashex.org

# Contents

1. Disclaimer 3
2. Overview 4
3. Found issues 6
4. Contracts 8
5. Conclusion 16
Appendix A. Issues' severity classification 17
Appendix B. List of examined issue types 18

# 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 29/11/2022 and 09/12/2022

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 scope of this audit includes 2 files available at

https://github.com/ShimmerSea/shimmersea-magiclum/blob/a4fd7c51f433ab1d48a5b0c9cfa8d4dddf006432/contracts/TangleSeaBoosterV2.sol and

https://github.com/ShimmerSea/shimmersea-fairlaunch/blob/fbd63ceeed5f77fbc47cb4d56d1671fd1a7e717e/contracts/FairLaunchERC20Merkle.sol.

**Update.** The ShimmerSea team has responded to this report. The updated code is located in the same repository:

https://github.com/ShimmerSea/shimmersea-magiclum/blob/6261bc2b7e0f4cee0ea41242c0b423abeeecc108/contracts/TangleSeaBoosterV2.sol and

https://github.com/ShimmerSea/shimmersea-fairlaunch/blob/eb0bb22c8e019d58fc60cb48daa3104141fbfa83/contracts/FairLaunchERC20Merkle.sol.

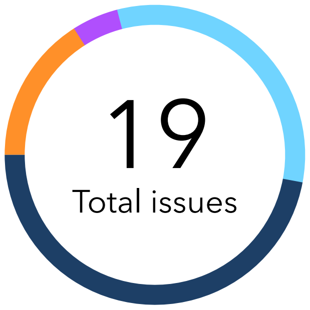
## 2.1 Summary

| | |
|---|---|
| Project name | ShimmerSea FairLaunch & TangleSeaBoosterV2 |
| URL | https://shimmersea.finance/ |
| Platform | Shimmer Network |
| Language | Solidity |

## 2.2 Contracts

| Name | Address |
|---|---|
| FairLaunchERC20Merkle | |
| TangleSeaBoosterV2 | |

## 3. Found issues



| | | |
|---|---|---|
| ● | High | 3 (16%) |
| ● | Medium | 1 (5%) |
| ● | Low | 6 (32%) |
| ● | Info | 9 (47%) |

### C1. FairLaunchERC20Merkle

| ID | Severity | Title | Status |
|---|---|---|---|
| C1-01 | ● High | No pair check | ✓ Resolved |
| C1-02 | ● High | Owner access | ✓ Resolved |
| C1-03 | ● Low | Using SafeERC20 library | ✓ Resolved |
| C1-04 | ● Low | Gas optimization | ✓ Resolved |
| C1-05 | ● Info | No emergency withdraw of LP tokens | ✓ Resolved |
| C1-06 | ● Info | Token support | ✓ Acknowledged |
| C1-07 | ● Info | Redundant compiler options | ✓ Acknowledged |
| C1-08 | ● Info | Default visibility | ✓ Resolved |

## C2. TangleSeaBoosterV2

| ID | Severity | Title | Status |
|---|---|---|---|
| C2-01 | ● High | No apropriate checking of burnAdmin | ✓ Resolved |
| C2-02 | ● Medium | Error with bonusEndTime variable | ✓ Resolved |
| C2-03 | ● Low | Absence of function that returns length of an array | ✓ Resolved |
| C2-04 | ● Low | Wrong return value of pendingBurnReward() function | ✓ Resolved |
| C2-05 | ● Low | Gas optimization | ✓ Resolved |
| C2-06 | ● Low | No appropiate event for the addRewards() function | ✓ Resolved |
| C2-07 | ● Info | Staked funds may be burned before ending of lock period | ✓ Resolved |
| C2-08 | ● Info | Redundancy of the SafeMath | ✓ Acknowledged |
| C2-09 | ● Info | Typos | ✓ Resolved |
| C2-10 | ● Info | No appropriate check of input values in init() function | ✓ Resolved |
| C2-11 | ● Info | Token support | ✓ Acknowledged |

# 4. Contracts

## C1. FairLaunchERC20Merkle

### Overview

A two-round sale contract. Whitelisted private round is fixed upon the deployment by immutable Merkle tree root. The user's funds are split in half to provide liquidity for SMR/LUM pair; another half goes to the Treasury address.

### Issues

#### C1-01 No pair check ● High ✓ Resolved

In the function `buildLP()`, there is no check that the function `getPair()` returns a non-zero account. Therefore, if there is no pair at the moment of calling `buildLP()`, the contract may lose all funds for the pair.

Also, there is no check that the pair returned from the function `getPair()` is empty. If not, tokens will be added in the wrong proportion according to the current pair reserves. This leads to a loss of part of the users' funds. Also, in this case, the `buildLP()` function creates a possibility for a malicious user (who has to acquire a significant amount of tokens bypassing the Fairlaunch sale) to make a sandwich attack and make profits.

```
function buildLP() external virtual nonReentrant onlyOwner {
    ...
    lpToClaim = FACTORY.getPair(LUM, SMR);
    uint256 lumAmount = IERC20(LUM).balanceOf(address(this));
    IERC20(LUM).safeTransfer(lpToClaim, lumAmount);
    IERC20(SMR).safeTransfer(lpToClaim, lpSMRAmount);
    totalLPAmountToClaim = ITangleseaPair(lpToClaim).mint(address(this));
}
```

### Recommendation

Function `buildLP()` should be rewritten, and all appropriate pair checks should be made. It is also possible to drop the direct interaction with a factory/pair in favor of a Router `addLiquidity()` and `addLiquidityETH()` methods.

### Update

Calling the updated `buildLP()` function is impossible if any of pair requirements is unfulfilled , therefore we recommend declaring the emergency withdraw automatically.

## C1-02 Owner access ● High ✓ Resolved

The owner is able to set an `emergencyOperator` address, which has access to all collected funds of the contract before they've been transferred to a liquidity pair.

```
function emergencyWithdrawFunds() external {
    require(
        msg.sender == emergencyOperator,
        "emergencyWithdrawFunds: not allowed"
    );
    uint256 smrAmount = IERC20(SMR).balanceOf(address(this));
    uint256 lumAmount = IERC20(LUM).balanceOf(address(this));
    IERC20(LUM).safeTransfer(msg.sender, lumAmount);
    IERC20(SMR).safeTransfer(msg.sender, smrAmount);

    emit EmergencyWithdraw(lumAmount, smrAmount);
}
```

### Recommendation

According to the comments in the code, only a MultiSig account is supposed to be the emergency operator, although there are no checks of any kind in the `initEmergencyOperator()` function. To add credibility, emergency funds withdrawal could be performed to a separate contract, where users may claim their SMR back in case of an aborted sale.

## C1-03 Using SafeERC20 library ● Low ✓ Resolved

In function `_safeClaimTransfer()` it is better to use the SafeERC20 library for transferring ERC20 tokens. Otherwise, tokens that aren't fully compliant with the ERC20 standard may become non-recoverable by this function.

## C1-04 Gas optimization ● Low ✓ Resolved

In the function `buy()`, global variables `user.maxAllocation` and `user.allocation` can be read multiple times. There is also an unnecessary double call for `hasPublicSaleStarted()` which checks the current time.

In the function `claim()`, the global variable `user.allocation` is read multiple times.

In the function `buildLP()`, the global variable `lpToClaim` is read after the write.

In the function `initEmergencyOperator()`, the global variable `emergencyOperator` is read after writing.

Multiple requirements in the constructor with the same error message could be merged into a single check to reduce the bytecode.

## C1-05 No emergency withdraw of LP tokens ● Info ✓ Resolved

In the `emergencyWithdrawFunds()` function, there is no transfer of LP tokens.

## C1-06 Token support ● Info ✓ Acknowledged

The contract doesn’t support rebasing over time tokens and tokens with commissions on transfers.

### C1-07 Redundant compiler options ● Info ✓ Acknowledged

There is no need for SafeMath starting from the 0.8.0 version of Solidity. It is embedded into the compiler.

ABI coder v2 is also activated by default since the 0.8.0 release.

### C1-08 Default visibility ● Info ✓ Resolved

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

## C2. TangleSeaBoosterV2

### Overview

A staking contract that supposedly allows to stake LumToken for MagicLum reward. Anyone can burn staked tokens for 2% of the burned amount (parameter may be adjusted from 0.1% to 10%), the burnable amount linearly grows in time up to 100% after a fixed time (default value is 30 days, can be changed but not below two days).

### Issues

### C2-01 No apropriate checking of burnAdmin ● High ✓ Resolved

If the global variable `burnAdmin` equals 0x0 address, many ERC20 tokens with OpenZeppelin implementation will fail to transfer to this address and revert transactions. Because of that, the functions `burn()`, and `emergencyUnlock()` may become inaccessible.

```
function updateBurnAdmin(address _newBurnAdmin) external onlyOwner {
    require(_newBurnAdmin == address(0) ||
            _newBurnAdmin == 0x000000000000000000000000000000000000dEaD,
        "update: not valid burn admin");
    ...
}

function burn() external onlyAllowed {
    ...
    _burnTokens(burnAmount.sub(callerReward));
}

function emergencyUnlock() external nonReentrant {
    ...
    _burnTokens(burnAmount.sub(callerReward));
}

function _burnTokens(uint256 amount) private {
    ...
    stakedToken.safeTransfer(burnAdmin, amount);
}
```

### Recommendation

In the functions `init()` and `updateBurnAdmin()` check that `burnAdmin` isn't equal to 0x0 address.

## C2-02 Error with bonusEndTime variable

● Medium ✓ Resolved

In the function `addRewards()` the global variable `bonusEndTime` is equal to `lastSlice.endTime`, but it should be equal to `currentTime-1`. Because of that, after some time, the function `_rewardForSlices()` will always return zero in cases when it shouldn't. The last slice will be inaccessible for rewarding.

```
function addRewards(uint256[] memory monthlyRewardsInWei) external onlyOwner {
    MonthlyReward memory lastSlice = slices[slices.length - 1];
    ...
    bonusEndTime = lastSlice.endTime;
}

function _rewardForSlices(...) internal view returns (uint256 reward) {
    if (lastRewardTime >= bonusEndTime) {
        return 0;
        }
        ...
    }
```

### Recommendation

If this behavior is expected, it should be documented explicitly.

## C2-03 Absence of function that returns length of an array ● Low ✓ Resolved

No function returns the length of the `slices` array. There are no in-built public methods to check the length other than checking the state directly or reconstructing all the `addRewards()`.

## C2-04 Wrong return value of pendingBurnReward() function ● Low ✓ Resolved

In the function `pendingBurnReward()`, there is an error in the calculations. The variable `burnAmount` should equal to `(secSinceLastBurn * totalBurnMultiplier) / PRECISION_FACTOR` instead of `secSinceLastBurn * (totalBurnMultiplier / PRECISION_FACTOR)`. Division before multiplication reduces the calculation precision in general.

Also, in case `burnAmount > stakedSupply`, the function returns zero, which is wrong.

## C2-05 Gas optimization ● Low ✓ Resolved

In the function `_rewardForSlices()` the global variable `lastRewardTime` is read multiple times.

In the function `burn()`, global variables `stakedSupply` and `accTokenPerShare` are read multiple times.

In the `deposit()` function, global variables `userInfo[msg.sender]` (fields of structure), `stakedSupply`, `burnAfterTime`, and `time_locked` are read multiple times.

In the function `_computeSnapshotRewards()`, the global variable `burnSnapshots.length` is read multiple times. Also, a full structure `burnSnapshots[index]` is read instead of reading only the fields that are used in calculations.

In the function `harvest()` global variables `userInfo[msg.sender]` (fields of structure), `stakedSupply`, and `accTokenPerShare` are read multiple times.

In the function `withdrawAndHarvest()` global variables `userInfo[msg.sender]` (fields of structure) and `accTokenPerShare` are read multiple times.

In the function `emergencyUnlock()` global variables `user.rewardClaim` and `stakedSupply` are read multiple times.

In the function `getMonthlyReward()`, global variable `slices.length` is read multiple times.

Using of `PRECISION_FACTOR` for the `totalBurnMultiplier` calculation is questionable, since `stakedSupply` seems to be significantly greater than `burnAfterTime`.

## C2-06 No appropiate event for the addRewards() function

● Low ✓ Resolved

The function `addRewards()` doesn't have an appropriate event, complicating off-chain tracking of important changes.

## C2-07 Staked funds may be burned before ending of lock period

● Info ✓ Resolved

Staked funds are meant to be burned linearly in time to be fully burnt after `burnAfterTime` seconds. It is possible to set this variable lower (min value is 2 days) than the `time_locked` locking period responsible for accounting withdraw penalty (max value is 14 days). These constraints should be documented.

## C2-08 Redundancy of the SafeMath ● Info ✓ Acknowledged

There is no need for SafeMath starting from the 0.8.0 version of Solidity. It is embedded into the compiler.

## C2-09 Typos ● Info ✓ Resolved

Typos reduce the code's readability. Typos found: 'rejustment', 'penality', 'burnsapshots', 'transferd', 'initalized', 'udpate', 'calucated', 'amoutn', 'FUNCIONS', 'raimining'.

## C2-10 No appropriate check of input values in init() function ● Info ✓ Resolved

In the `init()` function, there is no check that `_burnAdmin` is equal to `0xdEaD`.

## C2-11 Token support ● Info ✓ Acknowledged

The contract doesn’t support rebasing over time tokens and tokens with commissions on transfers.

## 5. Conclusion

3 high, 1 medium, 6 low severity issues were found during the audit. 3 high, 1 medium, 6 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.

This audit includes recommendations on improving the code and preventing 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