


ShimmerSea

smart contracts
final audit report

September 2022

 hashex.org

 contact@hashex.org

Contents

1. Disclaimer	3
2. Overview	4
3. Found issues	6
4. Contracts	9
5. Conclusion	23
Appendix A. Issues' severity classification	24
Appendix B. List of examined issue types	25

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.

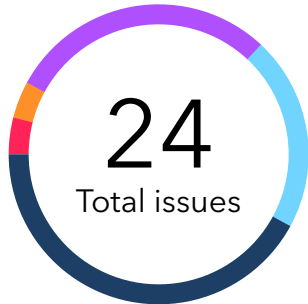
2.1 Summary

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

3. Found issues



● Critical	1 (4%)
● High	1 (4%)
● Medium	7 (29%)
● Low	5 (21%)
● Info	10 (42%)

Ce6. DEX contracts

ID	Severity	Title	Status
Ce613b	● Low	TangleseaLibrary: code with no effect	✔ Resolved
Ce613a	● Info	TangleseaFactory: fees distribution	✔ Acknowledged

Ce7. FarmUniV2Zap

ID	Severity	Title	Status
Ce713c	● Medium	zapOut() function doesn't perform safety checks	✔ Resolved
Ce713d	● Info	Typos	✔ Resolved

Cec. RewardPool

ID	Severity	Title	Status
Cecl3e	● Medium	External functions can be used for phishing	✔ Resolved
Cecl40	● Low	Gas optimizations	✔ Resolved
Cecl3f	● Info	Rewards aren't guaranteed	⊖ Acknowledged

Ced. RestrictedLumPool

ID	Severity	Title	Status
Cedl69	● High	Staked funds may be transferred as reward	✔ Resolved
Cedl46	● Medium	External functions can be used for phishing	✔ Resolved
Cedl44	● Low	Gas optimizations	✔ Resolved
Cedl43	● Info	Typos	✔ Resolved

Cee. TangleSeaMasterChef

ID	Severity	Title	Status
Ceel48	● Medium	Unfair distribution of awards without massUpdatePool()	⊖ Acknowledged
Ceel47	● Medium	External functions can be used for phishing	✔ Resolved
Ceel4d	● Low	Gas optimizations	✔ Resolved

Ceel4a	● Info	Typos	✔ Resolved
Ceel4c	● Info	Lack of events	✔ Resolved
Ceel49	● Info	emergencywithdraw() doesn't notify the rewarder	✔ Resolved
Ceel4b	● Info	Tokens with fees on transfers aren't supported	✔ Resolved
Ceel4e	● Info	Lack of safety checks on input values	✔ Resolved

Cf0. TimeBasedMasterChefRewarder

ID	Severity	Title	Status
Cf014f	● Critical	Rewarder is exposed to emergencyWithdraw() exploit	✔ Resolved
Cf0150	● Medium	Rewards aren't guaranteed	✔ Resolved
Cf016b	● Medium	Unfair distribution of awards without massUpdatePool()	✔ Resolved
Cf0152	● Low	Gas optimizations	✔ Resolved
Cf0151	● Info	Typos	✔ Resolved

4. Contracts

Ce6. DEX contracts

Overview

A typical fork of a [UniswapV2](#) DEX with minor changes of customizable fee for each pair.

Issues

Ce6I3b TangleseaLibrary: code with no effect ● Low ✔ Resolved

Code with no effect in `getReserves()`:

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

Ce6I3a 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.

Ce7. FarmUniV2Zap

Overview

Auxiliary contract for simplifying user interaction with the TangleSeaMasterChef contract.

Issues

Ce713c **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 {
    ...
    _removeLiquidity(address(pair), address(this));
    ...
    _returnAssets(tokens);
}

function _removeLiquidity(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.

Ce713d **Typos** ● Info ✔ Resolved

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

Ce8. Interfaces

Overview

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

Ce9. 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.

Cea. 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.

Ceb. 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.

Cec. RewardPool

Overview

A single pool contract inspired by [MasterChefV2](#) from Sushiswap.

Issues

Cec13e 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.

Cecl40 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.

Cecl3f 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()`.

Ced. 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

Ced169 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()`.

Ced146 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.

Ced144 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.

Ced143 Typos

● Info

✔ Resolved

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

Cee. TangleSeaMasterChef

Overview

A contract inspired by [MasterChefV2](#) from Sushiswap with additional optional Rewarder contracts for each pool.

Issues

Ceel48 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.

Ceel47 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.

Ceel4d 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()`;

Ceel4a Typos

● Info

✔ Resolved

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

Ceel4c Lack of events

● Info

✔ Resolved

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

Ceel49 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.

Ceel4b 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.

Ceel4e 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.

Cef. 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.

Cf0. 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

Cf0I4f **Rewarder is exposed to emergencyWithdraw()** ● Critical ✔ Resolved exploit

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.

Cf0150 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(accTokenPrecision).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.

Cf016b 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.

Cf0152 Gas optimizations

● Low

✔ Resolved

Several gas optimizations should be implemented:

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

Cf0151 Typos

● Info

✔ Resolved

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

5. Conclusion

1 critical, 1 high, 7 medium, 5 low severity issues were found during the audit. 1 critical, 1 high, 6 medium, 5 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](https://t.me/hashex_manager)

 blog.hashex.org

 [linkedin](https://www.linkedin.com/company/hashex)

 [github](https://github.com/hashex)

 [twitter](https://twitter.com/hashex)

#HashEx
BLOCKCHAIN SECURITY