#HashEx
BLOCKCHAIN SECURITY

Evmoswap

smart contracts
preliminary audit report
for internal use only

May 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 Evmoswap team to perform an audit of their smart contract. The audit was conducted between 16/05/2022 and 23/05/2022.

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 the @evmoswap/evmoswap-contract GitHub repository and was audited after the commit 21ae857. The updated code was rechecked after 4b62de1 commit in the same repository.

2.1 Summary

Project name	Evmoswap
URL	https://evmoswap.org
Platform	Evmos
Language	Solidity

2.2 Contracts

Name	Address
EMOToken	
MasterChef	
VotingEscrow	
RewardPool	
MultiFeeDistribution	
StakingPoolInitializable	
StakingPoolFactory	
FeeDistributor	
SimpleIncentivesController	
EvmoSwapLibrary	
SafeDecimal, Math, SafeERC20, TransferHelper, SafeMath, UQ112x112	

3. Found issues

High	5 (15%)
Medium	3 (9%)
Low	15 (45%)
Info	10 (31%)

C1. EMOToken

ID	Severity	Title	Status
C1-01	High	Voting vulnerabilities	Acknowledged
C1-02	Low	Gas optimization	Resolved

C2. MasterChef

ID	Severity	Title	Status
C2-01	High	100% fee	Resolved
C2-02	High	Huge payouts to DAO and referrals	Resolved
C2-03	High	Fail in emergencyWithdraw()	Resolved
C2-04	Medium	Unfair distribution of awards without massUpsatePool()	Acknowledged
C2-05	Low	_isContract() check	Acknowledged

C2-06	Low	Lack of validation	Partially fixed
C2-07	Low	Using of SafeMath library	Partially fixed
C2-08	Low	Gas optimizations	Partially fixed
C2-09	Info	Constructor lacks validation of input parameters	Resolved
C2-10	Info	Reward minting may reach limit	Acknowledged
C2-11	Info	Lack of events	Acknowledged

C3. VotingEscrow

ID	Severity	Title	Status
C3-01	Low	Gas optimization	Resolved
C3-02	Info	Lack of events	Resolved
C3-03	Info	Inconsistent comments	Resolved

C4. RewardPool

ID	Severity	Title	Status
C4-01	High	enterStaking() in _withdraw()	Resolved
C4-02	Low	Redundant address conversion	Resolved
C4-03	Low	Variable default visibility	Resolved
C4-04	Low	Using of SafeMath library	Resolved

C5. MultiFeeDistribution

ID	Severity	Title	Status
C5-01	Low	Lack of events	Resolved
C5-02	Low	Constructor lacks validation of input parameters	Resolved
C5-03	Low	Gas optimization	Resolved

C6. StakingPoolInitializable

ID	Severity	Title	Status
C6-01	Medium	Rewards of the contract	Resolved
C6-02	Low	Gas optimization	Resolved
C6-03	Info	Withdrawal of rewards by the owner	Acknowledged

C7. StakingPoolFactory

ID	Severity	Title	Status
C7-01	Info	Lack of error messages	Resolved

C8. FeeDistributor

ID	Severity	Title	Status
C8-01	Low	Gas optimization	Resolved
C8-02	Info	History actions is limited	Acknowledged

C9. SimpleIncentivesController

ID	Severity	Title	Status
C9-01	Medium	Reward rate isn't limited	Partially fixed
C9-02	Low	Gas optimization	Resolved
C9-03	Info	Pending rewards	Acknowledged
C9-04	Info	onlyOperator() modifier	Acknowledged

4. Contracts

C1. EMOToken

Overview

The ERC20-like contract with delegation of voting rights. The contract has a limit of minting and also serves as the reward token in other contracts.

Issues

C1-01 Voting vulnerabilities

High Acknowledged

The contract has functionality for voting and for delegating voting rights. At the same time, delegation does not take into account the transfer of tokens between users, which leads to voting vulnerabilities.

Delegation of votes is carried out by the functions `_delegate()` and `_moveDelegates()`.

```
function _delegate(address delegator, address delegatee)
   internal
   {
       address currentDelegate = _delegates[delegator];
       uint256 delegatorBalance = balanceOf(delegator);
       _delegates[delegator] = delegatee;

       emit DelegateChanged(delegator, currentDelegate, delegatee);

       _moveDelegates(currentDelegate, delegatee, delegatorBalance);
   }

function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
       if (srcRep != dstRep && amount > 0) {
           if (srcRep != address(0)) {
               // decrease old representative
               uint32 srcRepNum = numCheckpoints[srcRep];
```

```
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum -
1].votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum -
1].votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }
```

Possible vulnerabilities:

a. When delegating votes, these votes are not blocked and can be delegated again after being transferred to another account. For example:

1. Alice has 100 tokens on her balance and delegates 100 votes to Bob.

2. Bob collects 100 votes.

3. Then Alice transfers her 100 tokens to Carol. And Carol delegates 100 votes to Bob (this step can be repeated).

4. Finally, Bob has 200 votes (from 100 tokens).

b. Theft of other people's votes can be performed.

Example:

1. Alice, and Carol delegate 100 and 250 tokens respectively to Bob.

2. Bob collects 350 votes.

3. Attacker_1 has 1 token and delegates his vote to Bob. Now Bob has 351 votes.

4. Attacker_2 has 350 tokens and transfers all to Attacker_1. Attacker_1 now has 351 tokens.

5. Attacker_1 redelegate his votes from Bob to Attacker_2. Since the `_moveDelegates()` function takes `amount` value from the current user balance on L195, all Bob votes will go to Attacker_2.

6. In the end, Bob has 0 votes, and Attacker_2 has 351 votes.

c. Unable to redelegate if the balance has increased.

Example:

1. Alice has 100 tokens and delegates them to Bob.

2. Bob collects 100 votes.

3. Alice earns 1 (or more) token and has a balance of 101 tokens.

4. Alice can't redelegate her votes to Carol, due to underflow in L209.

Recommendation

We recommend using the ERC20Votes contract, where delegated votes count towards the transfer.

Developer's response

The voting functionality will not be used.

C1-02 Gas optimization

Low Resolved

a. The `mint()`, `addMinter()`, `delMinter()`, `getMinter()` functions can be declared as `external` to save gas.

b. The variable `votes` of the `Checkpoint` structure can be declared as `uint128` to store entire structure data in one slot.

C2. MasterChef

Overview

The contract allows to stake tokens in the pools, and get rewards. The reward amounts depend on the size of users' stake in the VotingEscrow contract.

Issues

C2-01 100% fee

High Resolved

The contract owner has the ability to set a 100% `depositFeePercent` using the `add()` or `set()` functions for each pool.

```
 function set(uint256 _pid,
        uint256 _allocPoint,
        uint256 _depositFeePercent,
        IOnwardIncentivesController _incentivesController,
        bool _withUpdate) public onlyOwner {
        require(_depositFeePercent <= percentDec, "set: invalid deposit fee basis
points");
        ...
    }
```

require(_depositFeePercent <= percentDec, "set: invalid deposit fee basis points");

This causes the user's entire deposit in the `depositFor()` function to be used to pay fees.

```
function depositFor(address _user, uint256 _pid, uint256 _amount) public nonReentrant {
        ...
        if (_amount > 0) {
```

```
            uint256 balanceBefore = pool.lpToken.balanceOf(address(this));
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            _amount = pool.lpToken.balanceOf(address(this)).sub(balanceBefore);
            if (pool.depositFeePercent > 0) {
                uint256 depositFee = _amount.mul(pool.depositFeePercent).div(percentDec);
                pool.lpToken.safeTransfer(feeAddr, depositFee);
                _amount = _amount.sub(depositFee);
            }
            user.amount = user.amount.add(_amount);
        }
        ...
    }
```

Recommendation

It is necessary to limit the fee percentage that the owner can set.

C2-02 Huge payouts to DAO and referrals

High Resolved

The function `withdrawDevAndRefFee()` performs the payout to `daoAddr`, `safuAddr`, `refAddr` addresses.

```
    function withdrawDevAndRefFee() public {
        require(lastTimeDaoWithdraw < block.timestamp, 'wait for new block');
        uint256 multiplier = getMultiplier(lastTimeDaoWithdraw, block.timestamp);
        uint256 emoReward = multiplier.mul(emoPerSecond);
        emo.mint(daoAddr, emoReward.mul(daoPercent).div(percentDec));
        emo.mint(safuAddr, emoReward.mul(safuPercent).div(percentDec));
        emo.mint(refAddr, emoReward.mul(refPercent).div(percentDec));
        lastTimeDaoWithdraw = block.timestamp;
    }
```

The variable `lastTimeDaoWithdraw` is not initialized in the contract and therefore equals zero. Thus, when calculating payments, the period since 1970 will be taken. Depending on the values of `emoPerSecond` and `BONUS_MULTIPLIER`, total payments can bring the total emission of EMOTokens closer to the limit of the `MAX_TOTAL_SUPPLY()` (or exceeds it).

This leads to the fact that it will no longer be possible to mint EMOTokens as rewards.

Recommendation

Define the initial value of the `lastTimeDaoWithdraw` variable.

C2-03 Fail in emergencyWithdraw()

High Resolved

The `emergencyWithdraw()` function allows withdrawing user's tokens without caring about rewards if something breaks in the `withdraw()` function.

```
function emergencyWithdraw(uint256 _pid) public nonReentrant {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
        user.amount = 0;
        user.rewardDebt = 0;

        // working amount
        if (pool.workingSupply >= user.workingAmount) {
            pool.workingSupply = pool.workingSupply - user.workingAmount;
        } else {
            pool.workingSupply = 0;
        }
        user.workingAmount = 0;

        // Interactions
        IOnwardIncentivesController _incentivesController = pool.incentivesController;
        if (address(_incentivesController) != address(0)) {
            _incentivesController.onReward(msg.sender, 0);
        }
    }
```

But also `emergencyWithdraw()` tries to get rewards in all `incentivesControllers` L470-473. If an error occurs inside contract IncentivesController, the user will not be able to withdraw his tokens.

Recommendation

We recommend placing the call to contract IncentivesController (L470-473) into a try-catch statement.

C2-04 Unfair distribution of awards without massUpsatePool()

Medium Acknowledged

The reward distribution for pools where the `updatePool()` function is rarely called and can become too small (unfair) if new pools are added L182 (or updated L206) without the `_withUpdate` flag.

Recommendation

Consider not using the option for calling `massUpdatePools()` function inside the `add()`, `set()` functions, but force it.

C2-05 _isContract() check

Low Acknowledged

The check for `!_isContract(user)` on L290can be bypassed if the `depositFor()` function is called from the constructor of the other contract.

C2-06 Lack of validation

Low Partially fixed

a. There is no validation for the `BONUS_MULTIPLIER` parameter of the function `updateMultiplier()`.

b. The functions `setDaoAddress()`, `setRefAddress()`, `setSafuAddress()` don't check the input address for a non-zero value.

C2-07 Using of SafeMath library

Low Partially fixed

Part of the calculations in the contract don't use the imported SafeMath library: L318, L320, L323, L326, L383, L385, L388, L391, L423, L446.

We recommend upgrading `pragma` to the latest major release with internal safety checks, or refactoring to implement already imported SafeMath library.

C2-08 Gas optimizations

Low Partially fixed

a. The state variable `percentDec` can be declared as `constant` to save gas.

b. The state variables `stakingPercent`, `daoPercent`, `safuPercent`, `refPercent`, `rewardMinter`, `votingEscrow` can be declared as `immutable` to save gas.

c. The `setStartTime()`, `updateMultiplier()`, `withdrawDevAndRefFee()`, `add()`, `set()`, `deposit()`, `withdraw()`, `enterStaking()`, `leaveStaking()`, `emergencyWithdraw()`, `setEmoPerSecond()`, `setDaoAddress()`, `setRefAddress()`, `setSafuAddress()`, `setFeeAddress()` functions can be declared as `external` to save gas.

d. The conversion of `msg.sender` and `_user` variables to `address` type is redundant on L304, L376, L419, L443, L456.

e. The variable `pool.lpToken` is read 5 times from storage in the `depositFor()` function. The local variable can be used instead to save gas.

f. Since the argument `_users` of the functions `setWhitelist()`, `setPool0Staker()` is read-only, it can be declared as `calldata` instead of `memory` to save gas.

g. `user.workingAmount` variable is not in use in staking `pid=0` pool. It's always equal to `user.amount` and should not be updated in `enterStaking()` and `leaveStaking()` functions.

C2-09 Constructor lacks validation of input parameters

Info Resolved

The contract constructor does not check the addresses `rewardMinter` and `votingEscrow` for non-zero value.

Also, consider adding validation for `percentDec`, `stakingPercent`, `daoPercent`, `safuPercent`, `refPercent` variables.

C2-10 Reward minting may reach limit

Info Acknowledged

The `depositFor()`, `_withdraw()` functions cause the minting of tokens in the MultiFeeDistribution contract.

The work of these functions can be blocked if the reward token has a supply limit, and it's finished.

C2-11 Lack of events

Info Acknowledged

The functions `setStartTime()`, `updateMultiplier()`, `setWhitelist()`, `setPool0Staker()`, `setEmoPerSecond()`, `setDaoAddress()`, `setRefAddress()`, `setSafuAddress()`, `setFeeAddress()` don't emit events, which complicates the tracking of important off-chain changes.

C3. VotingEscrow

Overview

Allows users to stake their EMOTokens. Size and time of stake correlate with earning the amount of rewards tokens in the pools of the MasterChef contract.

Staked users' EMOTokens will be redirected to the RewardPool contract, where they will be staked in the zero pool of the MasterChef contract.

Issues

C3-01 Gas optimization

Low Resolved

a. The state variable `token` L38 can be declared as `immutable` to save gas.

b. The variable `end` of the `LockedBalance` structure can be declared as `uint128` to store entire structure data in one slot.

C3-02 Lack of events

Info Resolved

The functions `setWhitelist()`, `setMasterchef()`, and `setEmergency()` don't emit events, which complicates the tracking of important off-chain changes.

C3-03 Inconsistent comments

Info Resolved

The functions `balanceOfT()` and `balanceOf()` use address from function parameters, but the descriptions state that it should be taken from `msg.sender` variable.

C4. RewardPool

Overview

The RewardPool contract is used to deposit staked users' EMOTokens in the zero pool of the MasterChef contract.

Issues

C4-01 enterStaking() in _withdraw()

High Resolved

The `_withdraw()` function should withdraw tokens from the MasterChef contract. But instead (as in `depositFor()` function), the function `enterStaking()` is called in L143.

```
function _withdraw(address from, address to, uint256 _amount) internal returns (bool) {
    ...
        // adjust amount
        _amount = _amount > _userAmountBeforeWithdraw ? _userAmountBeforeWithdraw :
_amount;
        masterchef.enterStaking(_amount);
        if (_amount > 0) {
            stakingToken.safeTransfer(to, _amount);
            user.amount = _userAmountBeforeWithdraw - _amount;
        }
```

```
    ...
}
```

Thus, tokens cannot be withdrawn.

Recommendation

It is necessary to call `leaveStaking()` instead of the `enterStaking()` function on L143.

C4-02 Redundant address conversion

Low Resolved

The conversion of the `msg.sender` value in L125 is redundant.

C4-03 Variable default visibility

Low Resolved

The variable `incentivesController` has default visibility. Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

C4-04 Using of SafeMath library

Low Resolved

Part of the calculations in the `depositFor()` and `_withdraw()` functions don't use the imported SafeMath library.

We recommend upgrading `pragma` to the latest major release with internal safety checks, or refactoring to implement the already imported SafeMath library.

C5. MultiFeeDistribution

Overview

Minter contract for EMO token with one-time setter for the list of external minters. Minted amounts could be locked for 4 weeks with a 50% penalty for early withdrawals.

Issues

C5-01 Lack of events

Low Resolved

The function `setMinters()` doesn't emit events, which complicates the tracking of important off-chain changes.

C5-02 Constructor lacks validation of input parameters

Low Resolved

The contract constructor does not check the addresses `_stakingToken` and `_penaltyReceiver` for a non-zero value. Since variables `stakingToken` and `penaltyReceiver` cannot be changed later, consider adding validation.

C5-03 Gas optimization

Low Resolved

a. Since the argument `_minters` of the function `setMinters()` is read-only, it can be declared as `calldata` instead of `memory` to save gas.

b. The variable `earnings.length` is read on every loop step and did it twice (L90, L100) in function execution. Consider using a local variable instead to save gas.

c. The 2nd `for()` loop of `earnedBalances()` is redundant. It's gas-wise to fill the dynamic array during the 1st loop, and then copy it to a static one.

d. The function `withdrawableBalance()`, `withdraw()`, `withdrawByIndex()` can be declared as `external` to save gas.

e. The state variable `penaltyReceiver` can be declared as `immutable` to save gas.

C6. StakingPoolInitializable

Overview

Single staking pool with MasterChef-like logic that has an external source of reward tokens. It's meant to be deployed and initialized via StakingPoolFactory.

Without documentation, it is impossible to accurately determine the relationship of the contract with the rest of the repository contracts.

Issues

C6-01 Rewards of the contract

Medium Resolved

The `deposit()` and `withdraw()` functions calculate the rewards and transfer them (L136, L168) to users. All rewards must be on the balance of the contract. But there is no guarantee that the contract has enough rewards on his balance at any time. Thus, the execution of the functions `deposit()` and `withdraw()` will be blocked.

Update

Rewards transfers are made with `_safeRewardTransfer()` function that reduces the actual transferred amount down to 0 according to the contract's balance. In other words, `withdraw()` function works as `emergencyWithdraw()` in case of insufficient balance.

C6-02 Gas optimization

Low Resolved

a. The state variable `SMART_CHEF_FACTORY` can be declared as `immutable` to save gas.

b. The conversion of `msg.sender` variable to address type is redundant in L136, L142, L164, L168, L187, L198.

C6-03 Withdrawal of rewards by the owner

Info Acknowledged

The contract owner has the ability to withdraw all rewards from the contract at any time using the `emergencyRewardWithdraw()` function.

Thus, users will not be able to receive rewards, as well as make new deposits.

Recommedation

Consider restricting the owner's ability to withdraw rewards or adding a cooldown period for withdrawal. It's also a good practice to transfer exaggerated owner rights to a Timelock contract.

C7. StakingPoolFactory

Overview

A simple onlyOwner factory to deploy StakingPoolInitializable contracts.

Issues

C7-01 Lack of error messages

Info Resolved

Require statements in `deployPool()` function lack revert reasons.

C8. FeeDistributor

Overview

A distributor contract that allows users to claim their rewards based on the FeeDistributor balance and VotingEscrow math.

Issues

C8-01 Gas optimization

Low Resolved

a. The state variables `startTime`, `votingEscrow`, `token`, `emergencyReturn` can be declared as immutable to save gas.

C8-02 History actions is limited

Info Acknowledged

`for()` loops in `_checkpointToken()`, `_checkpointTotalSupply()`, and `_claim()` functions are limited in step numbers meaning that users may lose some rewards if these functions are called rarely.

C9. SimpleIncentivesController

Overview

Secondary reward contract that works with MasterChef. Can be called only by a single immutable operator or by the owner but only for emergency withdrawal. The Source of rewards is unclear, reward amounts are not guaranteed. Multiple instances of SimpleIncentivesController can be configured into a chain of contracts that trigger at once upon a single `onReward()` call of first chain element.

Issues

C9-01 Reward rate isn't limited

Medium Partially fixed

The owner is able to use `setRewardRate()` function to update the `tokenPerSec` reward rate without any restrictions. If the owner acts maliciously or being hacked, all the SimpleIncentivesController's reward balance goes to unpaid rewards locked into the contract.

Recommendation

Consider adding a sanity check of the new value in the `setRewardRate()` function.

Update

Token per second reward rate was limited to be lesser than 1e30 wei per second. For a typical token with decimals of 18, it's an unreasonably large value.

C9-02 Gas optimization

Low Resolved

a. The state variable `ACC_TOKEN_PRECISION` can be declared as `immutable` to save gas.

b. The conversion of `msg.sender` variable to address type is redundant in L188.

c. The function `emergencyWithdraw()` can be declared as `external` to save gas.

C9-03 Pending rewards

Info Acknowledged

The `onReward()` function allows paying rewards to users. Moreover, the number of rewards on the balance of the contract at some point in time may not be enough. In this case, the contract debt is written to the `user.unpaidRewards` variable with no guarantee of subsequent payment.

Together with this, the `pendingTokens()` function only shows all the rewards earned, not those that are available for withdrawal.

C9-04 onlyOperator() modifier

Info Acknowledged

The `onlyOperator()` modifier serves to restrict the ability to call the `onReward()` function. In the context of this contract, the MasterChef contract is the operator.

At the same time, a contract with the same interface (IOnwardIncentivesController) is called in the `onReward()` function L154 itself. If the `onlyOperator()` modifier is also present in the `_nextIncentivesController` contract, then the operator for it must be the current contract.

C10. EvmoSwapLibrary

Overview

Fork of UniswapV2Library with variable swap fee for different pairs. No issues were found.

C11. SafeDecimal, Math, SafeERC20, TransferHelper, SafeMath, UQ112x112

Overview

Standard libraries forked from Uniswap and OpenZeppelin. No issues were found.

5. Conclusion

5 high, 3 medium, 15 low, and 10 informational severity issues were found, of which 4 high, 1 medium, 11 low, and 4 info severity issues were fixed, while 1 medium and 3 low severity issues were fixed partially.

The 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 unit and functional tests for all contracts.

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

 @hashexbot

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY