#HashEx
BLOCKCHAIN SECURITY

Pools Finance

smart contracts final audit report

July 2024

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 Pools Finance team to perform an audit of their smart contract. The audit was conducted between 03/06/2024 and 22/06/2024.

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 in the @Pools-Finance/pools-proxies-contracts Github repository after the af3e3fa commit and in the @Pools-Finance/pools-token-contracts after the a7044b4 commit. The Pools Finance project is a fork of Balancer V2 protocol with several audits available, forked repository is @Pools-Finance/pools-monorepo.

The audited contracts include forked Sushiswap's Masterchef V2 with main token and additional rewarders, token vesting contract from OpenZeppelin v3.0 release, governance contracts with a timelock contract bases on the Timelock by Compound Finance, and reward gauges contracts forked from Curve Finance.

Update. The Pools Finance team has responded to this report. The updated contracts are available in the @Pools-Finance/pools-proxies-contracts Github repository after the 37d3179 commit and in the @Pools-Finance/pools-token-contracts after the 8f2f7a1 commit.

2.1 Summary

Project name	Pools Finance
URL	https://pools.finance/
Platform	IOTA EVM
Language	Solidity
Centralization level	High
Centralization risk	High

2.2 Contracts

Name	Address
PoolsToken	
PoolsMasterChef	
MasterChefRewarderFactory	
TimeBasedMasterChefRewarder	
MasterChefOperator	
Timelock	
MasterChefLpTokenTimelock	
TokenVesting	

Gauges contracts

CopperProxy

PoolsZap

3. Project centralization risks

Some of Balancer contracts contain known vulnerabilities (1, 2). The project owner should refrain from deploying those contracts.

C0eCR25 Owner

The ownership should be transferred to the PoolsMasterChef contract or another contract with clear minting policy.

C10CR26 Owner privileges

The contract's owner can update the reward emission rate within fixed limits, update treasury address, add or update list of pools.

C12CR27 Owner privileges

The contract owner can update reward rate to an arbitrary value or transfer out all rewards.

C19CR28 Owner privileges

The contract owner can update the `_feeRecipient` address. Malicious fee recipient may revert `exitPool()` execution or use `skim()` function to transfer out tokens of exiting user, if fee token contains any form of transfer hooks.

4. Found issues

Critical	1 (5%)
High	2 (10%)
Medium	5 (24%)
Low	10 (48%)
Info	3 (13%)

C10. PoolsMasterChef

ID	Severity	Title	Status
C10Ide	Medium	Possible fail of emergency withdrawal	Resolved
C10Ie2	Low	Small amount of rewards remains locked	Resolved
C10Ie0	Low	Changing reward model without pools updating	Resolved
C10Idf	Low	Gas optimizations	Partially fixed
C10Ie1	Info	Inconsistent comment	Resolved

C11. MasterChefRewarderFactory

ID	Severity	Title	Status
C11Ie3	Low	Gas optimizations	Resolved
C11Ie4	Low	Lack of revert reason	Resolved

C12. TimeBasedMasterChefRewarder

ID	Severity	Title	Status
C12Ie6	Low	Gas optimizations	Resolved
C12Ie7	Info	Limited token support	Resolved

C14. MasterChefOperator

ID	Severity	Title	Status
C14Iec	Low	Gas optimizations	Resolved
C14Ieb	Info	Lack of NatSpec descriptions	Resolved

C18. Gauges contracts

ID	Severity	Title	Status
C18If7	Medium	No possibility of changing reward receiver	Resolved
C18Ied	Medium	Unused variable in ChildChainLiquidityGaugeFactory	Resolved
C18Iee	Low	Lack of input validation in ChildChainStreamer	Resolved
C18If6	Low	Lack of events	Resolved

C19. CopperProxy

ID	Severity	Title	Status
C19If1	High	Lack of tests	Resolved
C19Ief	Medium	Fee charging could be circumvented	Resolved

C1a. PoolsZap

ID	Severity	Title	Status
C1aIf0	Critical	Abusing approvals	Resolved
C1aIf2	High	Lack of tests	Resolved
C1aIf4	Medium	The possibility of token loss	Resolved
C1aIf3	Low	Lack of input validation	Resolved

5. Contracts

C0e. PoolsToken

Overview

The PoolsToken (POOLS) contract is an ERC-20 standard token standard based on OpenZeppelin's implementation. It can be minted with a fixed cap by a single owner, meant to be the PoolsMasterChef contract.

C10. PoolsMasterChef

Overview

This is a yield farming contract based on the MasterChef by Sushiswap with optional external rewarders.

Issues

C10Ide Possible fail of emergency withdrawal

Medium Resolved

The `emergencyWithdraw()` function is used to withdraw user's funds without rewards. If the corresponding pool has an external rewarder, it must be notified about changes in user's balance. The emergencyWithdraw() function should be designed fail-proof, but external call to the rewarder may be reverted.

```
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid, address _to) public {
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 amount = user.amount;
    user.amount = 0;
    user.rewardDebt = 0;
```

```
        IRewarder _rewarder = rewarder[_pid];
        if (address(_rewarder) != address(0)) {
            _rewarder.onPoolsReward(_pid, msg.sender, _to, 0, 0);
        }

        // Note: transfer can fail or succeed if `amount` is zero.
        lpTokens[_pid].safeTransfer(_to, amount);
        emit EmergencyWithdraw(msg.sender, _pid, amount, _to);
    }
```

Recommendation

Use try/catch for the `_rewarder.onPoolsReward()` call without explicit revert in catch section or use direct encoded call without requiring the success.

C10le2 Small amount of rewards remains locked

Low Resolved

The updatePool() function splits minting reward amount to the pool and to the treasury, but small amount is always locked in the contract due to rounding down error.

```
    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public returns (PoolInfo memory pool) {
        pool = poolInfo[_pid];

        if (block.number > pool.lastRewardBlock) {
            // total lp tokens staked for this pool
            uint256 lpSupply = lpTokens[_pid].balanceOf(address(this));
            if (lpSupply > 0) {
                uint256 blocksSinceLastReward = block.number -
                    pool.lastRewardBlock;

                // rewards for this pool based on his allocation points
                uint256 poolsRewards = (blocksSinceLastReward *
                    poolsPerBlock *
                    pool.allocPoint) / totalAllocPoint;

                uint256 poolsRewardsForPool = (poolsRewards * POOL_PERCENTAGE) /
                    1000;
```

```
                pools.mint(
                    treasuryAddress,
                    (poolsRewards * TREASURY_PERCENTAGE) / 1000
                );

                pools.mint(address(this), poolsRewardsForPool);

                pool.accPoolsPerShare =
                    pool.accPoolsPerShare +
                    ((poolsRewardsForPool * ACC_POOLS_PRECISION) / lpSupply);
            }
            pool.lastRewardBlock = block.number;
            poolInfo[_pid] = pool;

            emit LogUpdatePool(
                _pid,
                pool.lastRewardBlock,
                lpSupply,
                pool.accPoolsPerShare
            );
        }
    }
```

Recommendation

Use `poolsRewardsForTreasury = poolsRewards - poolsRewardsForPool`.

C10le0 Changing reward model without pools updating

Low Resolved

Total allocation can be changed without updating other pools, potentially resulting in part of reward being lost.

```
    // Add a new lp to the pool. Can only be called by the owner.
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        IRewarder _rewarder
    ) public onlyOwner {
        require(
```

```
            Address.isContract(address(_lpToken)),
            "add: LP token must be a valid contract"
        );
        require(
            Address.isContract(address(_rewarder)) ||
                address(_rewarder) == address(0),
            "add: rewarder must be contract or zero"
        );
        // we make sure the same LP cannot be added twice which would cause trouble
        require(
            !lpTokenAddresses.contains(address(_lpToken)),
            "add: LP already added"
        );

        // respect startBlock!
        uint256 lastRewardBlock = block.number > startBlock
            ? block.number
            : startBlock;
        totalAllocPoint = totalAllocPoint + _allocPoint;

        // LP tokens, rewarders & pools are always on the same index which translates into
the pid
        lpTokens.push(_lpToken);
        lpTokenAddresses.add(address(_lpToken));
        rewarder.push(_rewarder);

        poolInfo.push(
            PoolInfo({
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accPoolsPerShare: 0
            })
        );
        emit LogPoolAddition(
            lpTokens.length - 1,
            _allocPoint,
            _lpToken,
            _rewarder
        );
    }
```

Recommendation

Update existing pools before changing reward model.

C10Idf Gas optimizations

Low Partially fixed

1. Duplicated stored data in `poolInfo.length, lpTokens.length` and `rewarder.length`: lpTokens and rewarder variables should be declared as mappings `poolId => data`.

2. Duplicated stored data in `lpTokens` and `lpTokenAddresses.values`.

3. The requirements for `_rewarder` parameter should be moved inside `if (overwrite)` clause in the `set()` function.

4. Multiple reads from storage in the `deposit()` function: `user.amount` variable.

5. Multiple reads from storage in the `harvest()` function: `user.amount` variable.

6. Multiple reads from storage in the `withdrawAndHarvest()` function: `user.amount` variable.

C10Ie1 Inconsistent comment

Info Resolved

The comment L70 is not the part of `lpTokens` description.

```
    // Info of each user that stakes LP tokens per pool. poolId => address => userInfo
    /// @notice Address of the LP token for each MCV pool.
    IERC20[] public lpTokens;
```

C11. MasterChefRewarderFactory

Overview

This is a factory contract for deploying external rewarders for the PoolsMasterChef contract. Anyone can submit and deploy rewarder contract, but only privileged accounts can approve it and submit for PoolsMasterChef governance contract.

Issues

C11le3 Gas optimizations

Low Resolved

The `emergencyWithdraw()` function is used to withdraw user's funds without rewards. If the corresponding pool has an external rewarder, it must be notified about changes in user's balance. The `emergencyWithdraw()` function should be designed fail-proof, but external call to the rewarder may be reverted.

```
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid, address _to) public {
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 amount = user.amount;
    user.amount = 0;
    user.rewardDebt = 0;

    IRewarder _rewarder = rewarder[_pid];
    if (address(_rewarder) != address(0)) {
        _rewarder.onPoolsReward(_pid, msg.sender, _to, 0, 0);
    }

    // Note: transfer can fail or succeed if `amount` is zero.
    lpTokens[_pid].safeTransfer(_to, amount);
    emit EmergencyWithdraw(msg.sender, _pid, amount, _to);
}
```

C11le4 Lack of revert reason

Low Resolved

1. Duplicated stored data in `poolInfo.length, lpTokens.length` and `rewarder.length`: lpTokens and rewarder variables should be declared as mappings `poolId => data`.

2. Duplicated stored data in `lpTokens` and `lpTokenAddresses.values`.

3. The requirements for `_rewarder` parameter should be moved inside `if (overwrite)` clause in the `set()` function.

4. Multiple reads from storage in the `deposit()` function: `user.amount` variable.

5. Multiple reads from storage in the `harvest()` function: `user.amount` variable.

6. Multiple reads from storage in the `withdrawAndHarvest()` function: `user.amount` variable.

7. Updated code: inefficient search over existing pools in the `lpTokenAddressAdded()` function.

8. Updated code: lack of getter function for LP token address, since the `poolInfo()` function reads full structure.

C12. TimeBasedMasterChefRewarder

Overview

This is a rewarder contract for the PoolsMasterChef contract. It distributes certain ERC-20 token with constant (but adjustable) rate.

Issues

C12Ie6 Gas optimizations

Low Resolved

1. Multiple reads from storage in the `onPoolsReward()` function: `userPoolInfo.amount`, `accTokenPrecision`, `rewardToken` variables, `rewardToken.balanceOf(address(this))` external call.

C12Ie7 Limited token support

Info Resolved

Tokens with limited ERC-20 standard implementation may be not fully supported, e.g., the `shutDown()` function doesn't use SafeERC20 library for transfer.

C14. MasterChefOperator

Overview

The MasterChefOperator contract is designed to be set as admin account for the Timelock contract. It uses stage-commit model for governance functions of the PoolsMasterChef contract.

Issues

C14Iec Gas optimizations

Low Resolved

1. Multiple reads from storage in the `queueFarmModifications()` function: `farmModifications[eta].length` and `farmAdditions[eta].length` variables.

2. Multiple reads from storage in the `executeFarmModifications()` function: `farmModifications[eta].length` and `farmAdditions[eta].length` variables.

C14leb Lack of NatSpec descriptions

Info Resolved

We recommend writing documentation using NatSpec Format. This would help in development, as well as simplify user interaction with the contract (including using the block explorer).

C15. Timelock

Overview

The Timelock contract is based on the original Timelock by Compound Finance. Delay limits are 6 hours to 30 days. The contract is meant to be set as owner of the PoolsMasterChef contract.

C16. MasterChefLpTokenTimelock

Overview

An ERC-20 token locker contract that deposits tokens in the PoolsMasterChef contract for a locking period. Operates with fixed ERC-20 token, beneficiary, and release time.

C17. TokenVesting

Overview

A vesting contract based on OpenZeppelin's contract from v3.0 release.

C18. Gauges contracts

Overview

The gauges contracts are forked from Balancer V2 repository.

List of contracts:

ChildChainGaugeRewardHelper,

ChildChainLiquidityGaugeFactory,

ChildChainStreamer,

RewardsOnlyGauge,

IChildChainLiquidityGaugeFactory,

IChildChainStreamer,

ILiquidityGauge,

ILiquidityGaugeFactory

IRewardsOnlyGauge.

Issues

C18If7 No possibility of changing reward receiver

Medium Resolved

In the `RewardsOnlyGauge` contract, the `_checkpoint_rewards()` function is called within the `deposit()` and `withdraw()` functions. However, the last argument of `_checkpoint_rewards()` is always set to `ZERO_ADDRESS`, preventing the ability to change the reward receiver to a custom address.

C18led Unused variable in ChildChainLiquidityGaugeFactory

Medium Resolved

The `_isStreamerFromFactory` mapping is not written in the `create()` function. The `isStreamerFromFactory()` function always returns false.

```
    function create(address pool) external override returns (address) {
        require(_poolGauge[pool] == address(0), "Gauge already exists");

        address gauge = Clones.clone(address(_gaugeImplementation));
        address streamer = Clones.clone(
            address(_childChainStreamerImplementation)
        );

        IChildChainStreamer(streamer).initialize(gauge);
        IRewardsOnlyGauge(gauge).initialize(pool, streamer, _CLAIM_SIG);

        _isGaugeFromFactory[gauge] = true;
        _poolGauge[pool] = gauge;
        _gaugeStreamer[gauge] = streamer;
        emit RewardsOnlyGaugeCreated(gauge, pool, streamer);

        return gauge;
    }
```

C18lee Lack of input validation in ChildChainStreamer

Low Resolved

The `_add_reward()` function checks for `reward_data[_token].distributor` against `address(0)`, but actual `_distributor` value is not checked against zero.

```
@internal
def _add_reward(_token: address, _distributor: address, _duration: uint256):
    """
    @notice Add a reward token
    @param _token Address of the reward token
    @param _distributor Address permitted to call `notify_reward_amount` for this token
    @param _duration Number of seconds that rewards of this token are streamed over
    """
```

```
    assert self.reward_data[_token].distributor == ZERO_ADDRESS, "Reward token already
added"

    idx: uint256 = self.reward_count
    self.reward_tokens[idx] = _token
    self.reward_count = idx + 1
    self.reward_data[_token].distributor = _distributor
    self.reward_data[_token].duration = _duration
    log RewardDistributorUpdated(_token, _distributor)
    log RewardDurationUpdated(_token, _duration)
```

C18If6 Lack of events

Low Resolved

In the contract `RewardsOnlyGauge` these functions don't emit any events:

1. `_checkpoint_rewards()`
2. `set_rewards_receiver()`
3. `claim_rewards()`
4. `_set_rewards()`

We recommend adding events to simplify tracking changes of important values in the contract offchain.

C19. CopperProxy

Overview

A helper contract for LiquidityBootstrappingPool creation.

Issues

C19If1 Lack of tests

High Resolved

The project doesn't contain any tests for the CopperProxy contract. We urgently recommend increasing test coverage.

C19Ief Fee charging could be circumvented

Medium Resolved

The `PoolConfig.isCorrectOrder` parameter of the `createAuction()` can be used to select wrong fund token, which balance is checked during execution of the `exitPool()` function. Setting any of two tokens as a funding token increases chances of skipping of the fee charging.

```
struct PoolConfig {
    string name;
    string symbol;
    address[] tokens;
    uint256[] amounts;
    uint256[] weights;
    uint256[] endWeights;
    bool isCorrectOrder;
    uint256 swapFeePercentage;
    bytes userData;
    uint256 startTime;
    uint256 endTime;
}

function createAuction(
    PoolConfig memory poolConfig
) external returns (address) {
    ...
    // 3: store pool data
    _poolData[pool] = PoolData(
        msg.sender,
        poolConfig.isCorrectOrder,
        poolConfig.amounts[poolConfig.isCorrectOrder ? 0 : 1]
    );
}
```

```
function exitPool(
    address pool,
    uint256[] calldata minAmountsOut,
    uint256 maxBPTTokenOut
) external onlyPoolOwner(pool) {
    ...
    // 4. Calculate and transfer fee to recipient
    address fundToken = poolTokens[poolData.isCorrectOrder ? 0 : 1];
    uint256 fundTokenBalance = IERC20(fundToken).balanceOf(address(this));
    if (fundTokenBalance > poolData.fundTokenInputAmount) {
        uint256 feeAmount = ((fundTokenBalance -
            poolData.fundTokenInputAmount) * _feeBPS) / 10_000;
        TransferHelper.safeTransfer(fundToken, _feeRecipient, feeAmount);
        emit TransferedFee(pool, fundToken, _feeRecipient, feeAmount);
    }
}
```

Recommendation

Fees should be taken in both tokens.

C1a. PoolsZap

Overview

A helper contract for combining pool interaction and deposit yield into the PoolsMasterChef contract.

Issues

C1a|f0 Abusing approvals

Critical Resolved

The `enter()` and `enterWithJoinPool()` functions receive user address in parameters. Thus, any user approval can be used by attacker to transfer funds and depositing them to an arbitrary beneficiary.

```
    function enter(
        bytes32 poolId,
        address sender,
        address recipient,
        AmountsData calldata amountsData,
        bytes[] calldata batchRelayerData,
        uint256 masterchefPid
    ) external {
        ...
        for (uint256 i = 0; i < len; i++) {
            TransferHelper.safeTransferFrom(
                amountsData.assets[i],
                sender,
                address(this),
                amountsData.amountsIn[i]
            );
            TransferHelper.safeApprove(
                amountsData.assets[i],
                VAULT,
                amountsData.amountsIn[i]
            );
        }
        ...
        // 4: Deposit BPT tokens obtained in Masterchef
        PoolsMasterChef(MASTERCHEF).deposit(
            masterchefPid,
            balanceToSend,
            recipient
        );
        ...
    }
```

Recommendation

Use `msg.sender` as source of incoming funds.

C1alf2 Lack of tests

High Resolved

The project doesn't contain any tests for the PoolsZap contract. We urgently recommend increasing test coverage.

C1alf4 The possibility of token loss

Medium Resolved

In the `enter()` function a user can loss his tokens if some of his tokens aren't used by the Balancer protocol.

Recommendation

All tokens sent by a user that will remain on the contract after a multicall to the Balancer protocol should be transferred back to the user.

C1alf3 Lack of input validation

Low Resolved

The `enter()` and `enterWithJoinPool()` functions may fail due to `AmountsData.assets` and `AmountsData.amountsIn` lengths mismatch.

6. Conclusion

1 critical, 2 high, 5 medium, 10 low severity issues were found during the audit. 1 critical, 2 high, 5 medium, 9 low issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

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. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. 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

Appendix D. Centralization risks classification

Centralization level

- **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY