Company DAO

smart contracts
final audit report

December 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 Company DAO team to perform an audit of their smart contracts. The audit was conducted between 29/11/2022 and 08/12/2022.

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 code is available at @company-dao/mvp-tge-v1 GitHub repository after the 475de0e commit.

The audited contracts are designed to be deployed with proxies. Users have no choice but to trust the owners, who can update the contracts at their will.

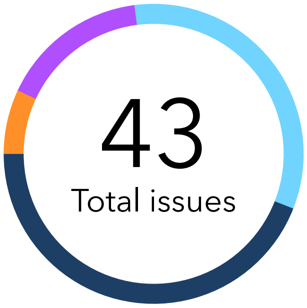
2.1 Summary

Project name	Company DAO
URL	https://companydao.org
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
Service	
Directory	
Metadata	
Pool	
Governor	
ProposalGateway	
GovernanceToken	
TGE	
WhitelistedTokens	
Libraries and interfaces	
All contracts	

3. Found issues



● High	3 (7%)
● Medium	7 (16%)
● Low	14 (33%)
● Info	19 (44%)

C1. Service

ID	Severity	Title	Status
C1-01	● High	Fees are not limited	Open
C1-02	● Medium	User projects can't be paused separately	Open
C1-03	● Low	Gas optimisations	Open
C1-04	● Info	Typos	Open
C1-05	● Info	Wrapped ETH address is not checked	Open
C1-06	● Info	Zero address is used as ETH address	Open
C1-07	● Info	Truncated protocol fee	Open
C1-08	● Info	ETH transfer method	Open
C1-09	● Info	Pool gains ownership of secondary TGE	Open

C2. Directory

ID	Severity	Title	Status
C2-01	Low	Gas optimisations	Open

C3. Metadata

ID	Severity	Title	Status
C3-01	Medium	Possible gas limit problem	Open
C3-02	Low	Gas optimisations	Open

C4. Pool

ID	Severity	Title	Status
C4-01	High	Price manipulation in getTVL()	Open
C4-02	Medium	Snapshots are not used for voting	Open
C4-03	Low	getTVL() should have view visibility	Open
C4-04	Low	Gas optimisations	Open
C4-05	Info	Typos	Open
C4-06	Info	TODO comments	Open

C5. Governor

ID	Severity	Title	Status
C5-01	High	Typo in conditions in proposalState()	Open
C5-02	Medium	ERC20 transfers without SafeERC20	Open
C5-03	Low	Gas optimisations	Open
C5-04	Low	Try catch without logging	Open
C5-05	Info	Typos	Open
C5-06	Info	Target for _executeBallot() is not fixed	Open
C5-07	Info	TODO and irrelevant comments	Open
C5-08	Info	isDelayCleared() should have view visibility	Open

C6. ProposalGateway

ID	Severity	Title	Status
C6-01	Info	Typos	Open

C7. GovernanceToken

ID	Severity	Title	Status
C7-01	Medium	Possible gas limit problem	Open
C7-02	Low	Gas optimisations	Open
C7-03	Info	Typos	Open

C8. TGE

ID	Severity	Title	Status
C8-01	● Medium	ERC20 transfers without SafeERC20	Open
C8-02	● Low	Initialisation parameters aren't checked	Open
C8-03	● Low	Gas optimisations	Open
C8-04	● Low	Unclear calculation of locked amount in purchase	Open
C8-05	● Info	Typos	Open
C8-06	● Info	Implicit rounding up of locked amount in purchase()	Open
C8-07	● Info	Ownable functionality is not in use	Open
C8-08	● Info	Locked funds are claimable only with admin of Service	Open

C9. WhitelistedTokens

ID	Severity	Title	Status
C9-01	● Low	Gas optimisations	Open
C9-02	● Info	Input parameters aren't checked	Open

C10. Libraries and interfaces

ID	Severity	Title	Status
C10-01	● Low	Not used code	? Open

C11. All contracts

ID	Severity	Title	Status
C11-01	● Medium	Lack of automated tests	? Open
C11-02	● Low	Events parameters are not indexed	? Open

4. Contracts

C1. Service

Overview

Main factory contract to deploy separate company pools (DAOs) and their eco-system contracts: governance tokens, primary and secondary token sales.

Issues

C1-01 Fees are not limited ● High Open

There are 2 different fees: the first one `fee` is taken upon the creation of the user project (pool, governance token, and primary TGE) and taken in form of native EVM currency, the second one `protocolTokenFee` is imposed on every successful TGE, primary or secondary, and taken in form of project token to be minted as a percentage of the sold amount. Both of these fees can be updated without any limitations; setting them to unreasonable values may harm the user projects: the creation of new pools may be blocked, successful TGE may suffer from liquidity withdrawals if the protocol fee is comparable to or exceeding the amount of governance tokens in TGE participants possession.

```
function setProtocolTokenFee(uint256 _protocolTokenFee) public onlyOwner {
    require(_protocolTokenFee <= 1000000, ExceptionsLibrary.INVALID_VALUE);
    protocolTokenFee = _protocolTokenFee;
    emit ProtocolTokenFeeChanged(protocolTokenFee);
}

function setFee(uint256 fee_) external onlyManager {
    fee = fee_;
    emit FeeSet(fee_);
}
```

Recommendation

Ownership of the Service contract should be transferred to MultiSig or Governance contract behind the Timelock.

C1-02 User projects can't be paused separately ● Medium Open

Any TGE and Pool actions, e.g. proposal voting/execution or token purchases/claims/redeems check if the Service contract is not on pause, meaning the owner can pause all user projects at once. However, it's impossible to pause them separately, so even the already successful and established DAO may suffer from the problems of the new ones.

```
    /**
     * @dev Pause entire protocol
     */
    function pause() public onlyOwner {
        _pause();
    }

//Pool.sol:L596
    modifier whenServiceNotPaused() {
        require(!service.paused(), ExceptionsLibrary.SERVICE_PAUSED);
        _;
    }
```

Recommendation

Consider introducing separate pausing for different pools in the Service contract.

C1-03 Gas optimisations ● Low Open

1. The `metadata` and `directory` variables are read multiple times in the `createPool()` function.
2. The `protocolTokenFee` variable is read twice in the `setProtocolTokenFee()` function.
3. Unnecessary mathematical operations with a 1000000 factor in the `getProtocolTokenFee()` function.

4. Code for testing purposes could be removed in order to reduce mainnet bytecode.

C1-04 Typos ● Info Open

Typos reduce the code's readability. Typos in 'Proocol', 'theshold'.

C1-05 Wrapped ETH address is not checked ● Info Open

UniswapQuoter contract is used to estimate the token proposal values in terms of WETH and/or USD currencies. However, the `uniswapQuoter` and `uniswapRouter` variables have no update functions besides the initialisation, while weth variable is set separately and can be updated, regardless of the actual `uniswapRouter.WETH()` or `uniswapQuoter.WETH9()` addresses.

C1-06 Zero address is used as ETH address ● Info Open

The `createPool()` and `createSecondaryTGE()` functions create a TGE contract and initialise it with a payment token address. If it is set to `address(0)`, the sale would use ETH as payment. We recommend avoiding using zero parameters in favour of a few non-zero bits constant address, e.g. `address(1)`, to reduce the possibility of an error.

C1-07 Truncated protocol fee ● Info Open

The `getProtocolTokenFee()` function returns the protocol fee amount based on the number of sold tokens in the TGE. The returned value is truncated by the last 12 digits; it should be documented if it's the desired behaviour.

C1-08 ETH transfer method ● Info Open

ETH transfers with `transfer()` and `send()` methods are discouraged in favour of `call()` with re-entrancy protection and possible gas management.

C1-09 Pool gains ownership of secondary TGE

● Info Open

The `createSecondaryTGE()` deploys a new TGE contract and initialises it with `msg.sender` as the owner. Since this function is callable only by the Pool contract, it becomes the owner of any secondary TGE deployed by Pool's proposal execution.

```
function createSecondaryTGE(ITGE.TGEInfo calldata tgeInfo) ... {
...
    ITGE tge = ITGE(address(new BeaconProxy(tgeBeacon, "")));
    tge.initialize(msg.sender, address(IPool(msg.sender).token()), tgeInfo);
}
```

C2. Directory

Overview

A register contract that records all the contracts deployed via Service factory with their types, addresses, and descriptions.

Issues

C2-01 Gas optimisations

● Low Open

1. The `lastProposalRecordIndex` variable is read in for loop in the `getGlobalProposalId()` function.
2. The `proposalRecordAt[i]` is read as a full struct in the `getGlobalProposalId()` function, which uses only 2 items of the structure.
3. Code for testing purposes could be removed in order to reduce the mainnet bytecode. There are a few commented functions to be removed before verification.

C3. Metadata

Overview

An authorisation contract that allows the project owner to add empty slots, without which it is impossible to create your own pool (DAO) via the Service factory.

Issues

C3-01 Possible gas limit problem ● Medium ⓘ Open

Reading data from storage inside the possibly very long loop may cause a problem with the transaction gas limit. Affected functions: `createRecord()`, `lockRecord()`, `jurisdictionAvailable()`.

Recommendation

Use mappings for efficient search.

C3-02 Gas optimisations ● Low ⓘ Open

The contract is extremely inefficient in terms of gas.

1. Checking the `queueInfo[i].jurisdiction` and `queueInfo[i].EIN` in a loop inside the `createRecord()` function should be transformed into a single mapping(bytes32=>bool) to check if hashed parameters were already used.
2. The `currentId`, `queueInfo[i].jurisdiction`, and `queueInfo[i].status` variables are read multiple times in the `createRecord()`, `lockRecord()` , and `jurisdictionAvailable()` functions.
3. Code for testing purposes could be removed in order to reduce the mainnet bytecode. There are a few commented functions to be removed before verification.

C4. Pool

Overview

A governance contract with its own owner and governance token. The primary token sale (TGE - token generation event) is mandatory and deployed simultaneously with the Pool, while any secondary TGEs are optional. Proposal creation is allowed only from ProposalGateway contract, most of the state-changing functions can be paused by the Service factory, i.e. owners of the Company DAO.

Issues

C4-01 Price manipulation in getTVL() ● High ⍰ Open

The `getTVL()` function evaluates the pool in terms of listed tokens from the WhitelistedTokens contract. All the pool's balances are quoted in USDT tokens, which address is stored in the Service contract. UniswapQuoter contract quotes the assets based on the instant price-pair reserves, making the whole getTVL() process susceptible to price manipulation in one or several listed token pairs.

```
    function getTVL() public returns (uint256) {
        IQuoter quoter = service.uniswapQuoter();
        IWhitelistedTokens whitelistedTokens = service.whitelistedTokens();
        address[] memory tokenWhitelist = whitelistedTokens.tokenWhitelist();

        for (uint256 i = 0; i < tokenWhitelist.length; i++) {
            if (tokenWhitelist[i] == address(0)) {
                tvl += address(this).balance;
            } else {
                uint256 balance = IERC20Upgradeable(tokenWhitelist[i])
                    .balanceOf(address(this));
                tvl += quoter.quoteExactInput(
                    whitelistedTokens.tokenSwapPath(tokenWhitelist[i]),
                    balance
                );
            }
        }
        return tvl;
    }
```

Recommendation

A reputable oracle system, centralised or de-centralised, should be implemented to reduce the risk of mis-evaluation. One should know that there's no such entity as the impeccable oracle, so additional checks of the oracle state and returned values may be introduced.

C4-02 Snapshots are not used for voting ● Medium ? Open

Despite GovernanceToken inheriting ERC20Votes from OpenZeppelin, its snapshot functionality is not used for voting in the Pool/Governor, which instead introduces a temporary locking model for each proposal separately. While this approach doesn't imply additional security risks, it increases gas costs and complicates the interaction between the user and the contract.

Recommendation

Consider returning to a classic OpenZeppelin/Compound voting scheme based on historical snapshots, which are already supported by GovernanceToken.

C4-03 getTVL() should have view visibility ● Low ? Open

The `getTVL()` function is a `view` function by its nature but doesn't have this modifier. This concern may complicate using the contract with the project website.

C4-04 Gas optimisations ● Low ? Open

1. Getter functions may have external visibility instead of public: `getPoolRegisteredName()`, `getBallotQuorumThreshold()`, `getBallotDecisionThreshold()`, `getBallotLifespan()`, `getPoolJurisdiction()`, `getPoolEIN()`, `getPoolDateOfIncorporation()`, `getPoolEntityType()`, `getPoolMetadataIndex()`, `maxProposalId()`, `isDAO()`, `getTGEList()`, `owner()`, and `getProposalType()`.

C4-05 Typos

● Info ⍰ Open

Typos reduce the code's readability. Typos in 'incorporatio', 'theshold', 'propsal'.

C4-06 TODO comments

● Info ⍰ Open

TODO comments should be removed before the mainnet launch either by implementing and testing additional functionality or by dropping it for later updates.

C5. Governor

Overview

Base contract for Pool containing proposal's logic and votes counting.

Issues

C5-01 Typo in conditions in proposalState()

● High ⍰ Open

The `proposalState()` function returns a wrong value in a common case of a successful vote after its ending: missing 10000 factor in the `totalCastVotes >= quorumVotes` check may lead to a Failed returned state of a successful proposal.

```
function proposalState(uint256 proposalId)
    public
    view
    returns (ProposalState)
```

```
{
    Proposal memory proposal = _proposals[proposalId];
    ...
    uint256 totalAvailableVotes = _getTotalSupply() -
        _getTotalTGELockedTokens();
    uint256 quorumVotes = (totalAvailableVotes *
        proposal.ballotQuorumThreshold);
    uint256 totalCastVotes = proposal.forVotes + proposal.againstVotes;
    ...
    if (block.number > proposal.endBlock) {
        if (
            totalCastVotes >= quorumVotes &&
            proposal.forVotes * 10000 >
            totalCastVotes * proposal.ballotDecisionThreshold
        ) {
            return ProposalState.Successful;
        } else return ProposalState.Failed;
    }
    return ProposalState.Active;
}
```

Recommendation

Fix the calculations and increase testing coverage.

C5-02 ERC20 transfers without SafeERC20 ● Medium ? Open

The `_executeBallot()` function has 4 types of proposed actions available: create a secondary TGE, change voting parameters (quorum, execution delay, etc.), transfer ETH from the pool, and transfer ERC20 token from the pool. The ERC20 token transfers are performed directly with `IERC20.transfer()` function, which may cause a problem if the token to be transferred is not fully compliant with the ERC20 standard, e.g. USDT. In this case, the pool will not be fully operable without updating the proxy implementation, effectively locking the collected funds.

```
function _executeBallot(
    uint256 proposalId,
    IService service,
    IPool pool
    ) internal {
        ...
        success = IERC20Upgradeable(proposal.token).transfer(proposal.targets[i],
proposal.values[i]);
        require(success, ExceptionsLibrary.EXECUTION_FAILED);
        ...
    }
```

Recommendation

Use the SaferERC20 library from OpenZeppelin for operation with arbitrary ERC20 tokens.

C5-03 Gas optimisations

● Low ⓘ Open

1. The proposal could be stored in hashed form (single bytes32 slot) to save gas. Otherwise, its structure should be rearranged to reduce the storage slots needed.
2. The `proposalState()` function performs unnecessary checks `proposal.forVotes * 10000 > totalCastVotes * proposal.ballotDecisionThreshold` and `proposal.againstVotes * 10000 > totalCastVotes * proposal.ballotDecisionThreshold`, which could be removed since their next adjacent requirements are stricter.
3. Enum values `ProposalExecutionState.Accomplished`, `ProposalExecutionState.Rejected`, `ProposalState.None` are never used in the code.

C5-04 Try catch without logging

● Low ⓘ Open

The function `isDelayCleared()` calculates the USDT equivalent amount of the tokens to be transferred in a proposal via `uniswapQuoter` contract. The calculation is wrapped in a `try/catch` structure. In case of failure of calculation, there's no indication that it failed.

```
    function isDelayCleared(IPool pool, uint256 proposalId, uint256 index)
        public
        returns (bool)
    {
      ...
            try
                pool.service().uniswapQuoter().quoteExactInput(
            abi.encodePacked(from, uint24(3000), pool.service().usdt()),
            amount
          )
        returns (uint256 v) {
          valueUSDT = v;
        } catch (
          bytes memory /*lowLevelData*/
        ) {}
    ...
    }
```

We can't ensure the logic of a non-failing transaction if a try call fails.

Recommendation

We recommend emitting an event in the `catch` code block to easier track failures of the calculation.

C5-05 Typos

● Info ? Open

Typos reduce the code's readability. Typo in 'decsision'.

C5-06 Target for _executeBallot() is not fixed

● Info ? Open

The `_executeBallot()` function receives the target pool address to check the delay clearance. But the Governor is inherited by the Pool contract, so it should use `address(this)` instead of the input parameter address. While this is not an issue with the current implementation, a possible future update may lead to a problem.

C5-07 TODO and irrelevant comments

● Info ? Open

TODO and irrelevant comments (see L67) should be removed for the mainnet launch.

C5-08 isDelayCleared() should have view visibility ● Info ⓘ Open

The `isDelayCleared()` function is a `view` function by its nature but doesn't have this modifier. This concern may complicate using the contract with the project website.

C6. ProposalGateway

Overview

A contract for user interaction with the Pool contract, limiting the possible actions that are allowed to be included in proposals: ETH and ERC20 transfers, voting parameters update and starting secondary TGEs.

Issues

C6-01 Typos ● Info ⓘ Open

Typos reduce the code's readability. Typo in 'transfered'.

C7. GovernanceToken

Overview

An ERC20 standard token with Votes and Capped extensions by OpenZeppelin. It is used for proposal voting in the corresponding Pool.

Issues

C7-01 Possible gas limit problem ● Medium ⓘ Open

Reading data from storage inside the possibly very long loop may cause a problem with the transaction gas limit. Affected function - `minUnlockedBalanceOf()`. This function is used in every token transfer.

```
    function minUnlockedBalanceOf(address user) public view returns (uint256) {
        uint256 min = balanceOf(user);
        for (uint256 i = 0; i <= IPool(pool).maxProposalId(); i++) {
            uint256 current = unlockedBalanceOf(user, i);
            if (current < min) {
                min = current;
            }
        }
        return min;
    }
```

Recommendation

Refactor the code or move to snapshots voting.

C7-02 Gas optimisations ● Low ? Open

1. Useless the checks of `0 <= uintVar <= type(uint).max` are performed in `increaseTotalTGELockedTokens()` and `decreaseTotalTGELockedTokens()`, they also double-read the `totalTGELockedTokens` variable from storage.

C7-03 Typos ● Info ? Open

Typos reduce the code's readability. Typos in 'dealine', 'Prposal'.

C8. TGE

Overview

A sale contract to be deployed with each Pool via Service factory as a primary token generation event (TGE). The success of the sale is defined by the hard- and softcap; after a successful sale part of the purchased tokens is locked in the TGE contract and can be claimed with time and TVL terms reached; in case of failure TGE users may redeem their payments back. Company DAO's fee is minted as part of the successfully sold tokens.

Issues

C8-01 ERC20 transfers without SafeERC20 ● Medium ? Open

1. The function `claim()` checks if the token's `transfer()` function has returned `true`. This check may fail on non-fully ERC20 conformant tokens such as USDT in the Ethereum network which the `transfer()` function does not return as a boolean.

2. The function `redeem()` transfers tokens in L322 and does not check the result of the ERC20 token transfer.

```
function redeem() {
  ...
          if (_unitOfAccount == address(0)) {
            payable(msg.sender).transfer(refundValue);
        } else {
            IERC20Upgradeable(_unitOfAccount).transfer(msg.sender, refundValue);
        }
}
```

Recommendation

Use OpenZeppelin's SafeERC20 library to handle token transfers.

C8-02 Initialisation parameters aren't checked ● Low ? Open

Input data is not filtered during the initialisation. `lockupPercent` can be set up to 100%, which may leave a primary TGE without tokens for liquidity pair creation. `lockupDuration` can be set lower than the duration of the TGE. All the input parameters don't have sanity checks against their max value.

C8-03 Gas optimisations ● Low ? Open

1. The `initialize()` function performs an unnecessary check `info.hardcap <= remainingSupply`, which could be removed since the next requirement with `HARDCAP_AND_PROTOCOL_FEE_OVERFLOW_REMAINING_SUPPLY` error code is stricter.
2. The `lockupTVL` variable is read from storage unnecessary in the `initialize()` function.
3. The `_unitOfAccount`, `_totalPurchased`, and `token` variables are read multiple times in the `purchase()` function.
4. The `_unitOfAccount` variable is read multiple times in the `transferFunds()` function.
5. The `_totalPurchased` and `token` variables are read multiple times in the `claimProtocolTokenFee()` function.
6. A fixed whitelist of users in the `initialize()` function could be implemented with a Merkle tree, available in the OpenZeppelin library.
7. Code for testing purposes could be removed in order to reduce the mainnet bytecode.

C8-04 Unclear calculation of locked amount in purchase ● Low ? Open

The locked amount is the function purchase is calculated as follows:

```
uint256 lockedAmount = (amount * lockupPercent + 99) / 100;
```

It's unclear why additional 99 argument is needed. As it's divided by 100, a zero amount is added to the `lockedAmount` value.

C8-05 Typos

● Info ? Open

Typos reduce the code's readability. Typos in 'puchase', 'avilable'.

C8-06 Implicit rounding up of locked amount in purchase()

● Info ? Open

Implicit rounding up of the locked amount is implemented in the `purchase()` function. It should be documented if this is the desired behaviour.

```
uint256 lockedAmount = (amount * lockupPercent + 99) / 100
```

C8-07 Ownable functionality is not in use

● Info ? Open

While the TGE contract inherits the Ownable from openZeppelin, its functionality is not in use.

C8-08 Locked funds are claimable only with admin of Service

● Info ? Open

Locked tokens can be unlocked only by the manager of the Service contract (appointed by the owners of the Company DAO project). Users have to trust them, if whatever reason they refuse to cooperate, the locked part of sold-in TGE tokens will remain locked in the contract.

```
    function setLockupTVLReached() external whenServiceNotPaused onlyManager {
        lockupTVLReached = true;
    }
```

C9. WhitelistedTokens

Overview

A list of supported valid tokens that are used for total value locked (TVL) calculation. Managed by the owners of the project (Company DAO).

Issues

C9-01 Gas optimisations ● Low ⓘ Open

1. Strict requirements for return values of `_tokenWhitelist.add()` and `_tokenWhitelist.remove()` in the `addTokensToWhitelist()` and `removeTokensFromWhitelist()` functions could be replaced with an event for tokens that haven't changed their status or simply ignoring such tokens.

C9-02 Input parameters aren't checked ● Info ⓘ Open

Lengths of the `tokens[]`, `swapPaths[]`, and `swapReversePaths[]` arrays aren't checked in the `addTokensToWhitelist()` and `removeTokensFromWhitelist()` functions. We recommend checking them for matching or using an array of structs that contains all 3 addresses.

C10. Libraries and interfaces

Overview

Error codes are shared with all the contracts as a library, various interfaces contain structs and enums for the above-mentioned contracts.

No issues were found.

Issues

C10-01 Not used code

● Low ? Open

The enum value `ProposalType.None` is never used in the project.

```
interface IProposalGateway {
    enum ProposalType {
        None,
        TransferETH,
        TransferERC20,
        TGE,
        GovernanceSettings
    }
}
```

Recommendation

We recommend removing all unused enums, variables, and functions to make the code more readable and minimise the risk of error on refactoring the code later.

C11. All contracts

Overview

Issues regarding all contact or the whole project.

Issues

C11-01 Lack of automated tests

● Medium ? Open

The project has several files with unit tests, but the code of these tests does not compile. Unit testing has crucial importance in smart contract development ensuring that the code works as expected.

Recommendation

We strongly recommend fixing the test suite and ensuring the test coverage of at least 90%.

C11-02 Events parameters are not indexed

● Low Open

Indexed parameters are not used in the events. Parameter indexing makes it easier to filter and finds events with specific parameters.

Example of events with non-indexed parameters in the Service.sol smart contract.

```
/**
 * @dev Event emitted on change in user's whitelist status.
 * @param account User's account
 * @param whitelisted Is whitelisted
 */
event UserWhitelistedSet(address account, bool whitelisted);

/**
 * @dev Event emitted on change in tokens's whitelist status.
 * @param token Token address
 * @param whitelisted Is whitelisted
 */
event TokenWhitelistedSet(address token, bool whitelisted);
```

Recommendation

We recommend making address parameters in the events indexed.

5. Conclusion

3 high, 7 medium, 14 low severity issues were found during the audit. No issues were resolved in the update.

The reviewed contract is 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.

The audited contracts are designed to be deployed with proxies. Users have no choice but to trust the owners, who can update the contracts at their will.

The code needs to be refactored and cleaned up before the launch: there are todos, commented code, and unused variables.

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

Appendix A. Issues' severity classification

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

Appendix B. List of examined issue types

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

✉ contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY