#HashEx
BLOCKCHAIN SECURITY

Company DAO

smart contracts final audit report

March 2023


hashex.org


contact@hashex.org

Contents

1. Disclaimer 3

2. Overview 4

3. Found issues 6

4. Contracts 10

5. Conclusion 27

Appendix A. Issues' severity classification 28

Appendix B. List of examined issue types 29

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 24/01/2023 and 06/02/2023.

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 f4eea4b 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.

The audit was done based on the code published in Guthub. Users need to check if they are interacting with the same implementations as were audited.

Update. Recheck was done after the commit b5e43.

Update 2. Second recheck was done after the commit 3cbfd64.

2.1 Summary

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

Language	Solidity

2.2 Contracts

Name	Address
Service	
Pool	
Governor	
GovernorProposals	
GovernanceSettings	
Token	
TGE	
Registry	
CompaniesRegistry	
RecordsRegistry	
TokensRegistry	
RegistryBase	
Libraries and interfaces	
All contracts	

3. Found issues

High	2 (6%)
Medium	4 (13%)
Low	12 (39%)
Info	13 (42%)

C1. Service

ID	Severity	Title	Status
C1-01	Low	Confusing math	Acknowledged
C1-02	Info	Typos	Acknowledged
C1-03	Info	Unclear functionality of Preference tokens	Acknowledged
C1-04	Info	Multiple active secondary TGE can be created	Acknowledged

C2. Pool

ID	Severity	Title	Status
C2-01	Low	Gas optimisations	Acknowledged
C2-02	Info	Typos	Acknowledged

C3. Governor

ID	Severity	Title	Status
C3-01	High	No voting delay	Resolved
C3-02	High	Potential voting manipulation	Resolved
C3-03	Medium	Function getBallot returns votes for incorrect block	Resolved
C3-04	Low	Incorrect function documentation	Resolved
C3-05	Low	Gas optimisations	Acknowledged
C3-06	Info	Typos	Acknowledged

C4. GovernorProposals

ID	Severity	Title	Status
C4-01	Info	Gas optimisations	Acknowledged

C5. GovernanceSettings

ID	Severity	Title	Status
C5-01	Low	Unused variable	Acknowledged
C5-02	Low	Insufficient validation of governance parameters	Acknowledged

C6. Token

ID	Severity	Title	Status
C6-01	Medium	Possible gas limit problem	Acknowledged
C6-02	Low	Max supply is not validated	Acknowledged
C6-03	Low	Inconsistent documentation	Acknowledged
C6-04	Low	Gas optimisations	Partially fixed
C6-05	Low	Flawed authorisation	Acknowledged
C6-06	Info	Typos	Acknowledged

C7. TGE

ID	Severity	Title	Status
C7-01	Medium	Implicit rounding up of locked amount in purchase()	Acknowledged
C7-02	Low	Gas optimisations	Acknowledged
C7-03	Info	Price format is not documented	Acknowledged
C7-04	Info	Allowed excessive payments	Acknowledged
C7-05	Info	Claiming can be available in active TGE state	Acknowledged
C7-06	Info	Typos	Acknowledged
C7-07	Info	Instant mining of purchased tokens	Acknowledged

C11. TokensRegistry

ID	Severity	Title	Status
C11-01	Low	Function isTokenWhitelisted never used	Acknowledged

C14. All contracts

ID	Severity	Title	Status
C14-01	Medium	Lack of automated tests	Acknowledged
C14-02	Info	Events parameters are not indexed	Acknowledged

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 Confusing math

Low Acknowledged

The `getMinSoftCap()` function is used for calculating the minimum allowed caps for creating tokens. We can't ensure the validity of the used math:

```
function getMinSoftCap() public view returns (uint256) {
    return (DENOM + protocolTokenFee - 1) / protocolTokenFee;
}
```

C1-02 Typos

Info Acknowledged

Typos reduce the code's readability. Typos in 'recepient', 'cacellation'.

C1-03 Unclear functionality of Preference tokens

Info Acknowledged

Secondary TGEs for Preference tokens are created with the same Token contract, which supports snapshots used for voting. However, there are no sings of those properties being intentional.

C1-04 Multiple active secondary TGE can be created ● Info ✓ Acknowledged

The `createSecondaryTGE()` function can be used for the creation of secondary TGE with a Preference token type. It allows creating multiple active secondary TGEs of Preference type, but only the last one is stored as a Preference token.

```
function createSecondaryTGE(
    ITGE.TGEInfo calldata tgeInfo,
    IToken.TokenInfo calldata tokenInfo,
    string memory metadataURI
) external override onlyPool nonReentrant whenNotPaused {
    ...
    } else if (tokenInfo.tokenType == IToken.TokenType.Preference) {
        // Case of Preference token

        // Check if it's new token or additional TGE
        if (
            address(token) == address(0) ||
            ITGE(token.getTGEList()[0]).state() == ITGE.State.Failed
        ) {
          ...
  ...
}
```

C2. 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. Most of the state-changing functions can be paused by the Service factory, i.e. owners of the Company DAO. Inherits Governor, GovernorProposals, and GovernanceSettings contracts.

Issues

C2-01 Gas optimisations

Low Acknowledged

1. The `lastProposalIdForAddress` mapping is never read on-chain in reviewed contracts, so an event could be used instead.

C2-02 Typos

Info Acknowledged

Typos reduce the code's readability. Typo in 'propo-nodes'.

C3. Governor

Overview

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

Issues

C3-01 No voting delay

High Resolved

The `_propose()` function creates a new proposal with a fixed start in the next block. Voters have no time for preparing, so the proposal can possibly be determined from the start, e.g. if a major holder mistakenly delegates his share.

```
    /**
     * @dev Creating a proposal and assigning it a unique identifier to store in the list
of proposals in the Governor contract.
     * @param core Proposal core data
     * @param meta Proposal meta data
     * @param votingDuration Voting duration in blocks
     */
    function _propose(
        ProposalCoreData memory core,
```

```
        ProposalMetaData memory meta,
        uint256 votingDuration
    ) internal returns (uint256 proposalId) {
        // Increment ID counter
        proposalId = ++lastProposalId;

        // Create new proposal
        proposals[proposalId] = Proposal({
            core: core,
            vote: ProposalVotingData({
                startBlock: block.number + 1,
                endBlock: block.number + 1 + votingDuration,
                availableVotes: _getCurrentTotalVotes(),
                forVotes: 0,
                againstVotes: 0,
                executionState: ProposalState.None
            }),
            meta: meta
        });
        ...
}
```

Recommendation

Consider adding a voting delay parameter to the GgovernanceSettings.

C3-02 Potential voting manipulation

High Resolved

The quorum and decision threshold are calculated against the number of available votes on the block when the proposal was created. The number of available votes is calculated with the `_getCurrentTotalVotes()` function which returns a current number of all the votes that could be used in voting.

```
    function _propose(
        ProposalCoreData memory core,
        ProposalMetaData memory meta,
        uint256 votingDuration
    ) internal returns (uint256 proposalId) {
        // Increment ID counter
```

```
        proposalId = ++lastProposalId;

        // Create new proposal
        proposals[proposalId] = Proposal({
            core: core,
            vote: ProposalVotingData({
                startBlock: block.number + 1,
                endBlock: block.number + 1 + votingDuration,
                availableVotes: _getCurrentTotalVotes(),
                forVotes: 0,
                againstVotes: 0,
                executionState: ProposalState.None
            }),
            meta: meta
        });
        ...
    }
```

If an attacker has a number of vested tokens, he can create a proposal withdrawing funds from the pool (company) and manipulate votes calculation to execute the proposal. For example, let the current number of available votes be 1000 and the attacker has 1000 tokens vested in a TGE event. After the proposal is created, he can send a backrun transaction that will withdraw his vested tokens in the same block but after the proposal transaction. In such a case the quorum and decision threshold would be calculated considering that the maximum number of all votes is 1000 while the attacker has 1000 votes at the same time, so he will be able to finish voting only by himself.

Recommendation

Calculate the available votes considering all the tokens that can be withdrawn from the vesting.

C3-03 Function getBallot returns votes for incorrect block

● Medium ✓ Resolved

The power is calculated on the block number equal to `startBlock - 1`.

```
    function _castVote(uint256 proposalId, bool support) internal {
```

```
        ...
                // Get number of votes
    uint256 votes = _getPastVotes(
        msg.sender,
        proposals[proposalId].vote.startBlock - 1
    );

    // Account votes
    if (support) {
        proposals[proposalId].vote.forVotes += votes;
        ballots[msg.sender][proposalId] = Ballot.For;
    } else {
        proposals[proposalId].vote.againstVotes += votes;
        ballots[msg.sender][proposalId] = Ballot.Against;
    }
        ...
}
```

But the function `getBallot()` uses startBlock to get the voting power.

```
function getBallot(address account, uint256 proposalId)
    public
    view
    returns (Ballot ballot, uint256 votes)
{
    return (
        ballots[account][proposalId],
        _getPastVotes(account, proposals[proposalId].vote.startBlock)
    );
}
```

Recommendation

Use `startBlock - 1` for the `getBallot()` function to get the voting power.

C3-04 Incorrect function documentation

Low Resolved

The function `_setLastProposalIdForAddress` has incorrect documentation.

```
    /**
     * @dev Function that returns the amount of votes for a client address at any given
block
     * @param proposer Proposer's address
     * @param proposalId Proposal id
     */
    function _setLastProposalIdForAddress(address proposer, uint256 proposalId) internal
virtual;
```

C3-05 Gas optimisations

Low Acknowledged

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 full Proposal structure is read from storage in the `proposalState()`, `_executeProposal()`, and `_checkProposalVotingEarlyEnd()` functions. Only needed parts should be read or the `storage` keyword should be used.

C3-06 Typos

Info Acknowledged

Typos reduce the code's readability. Typos in 'reched', 'methos', 'adrress'.

C4. GovernorProposals

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

C4-01 Gas optimisations

Info Acknowledged

In the `proposeTGE()` function `IPool(address(this))` could be reduced to `IPool(this)`.

C5. GovernanceSettings

Overview

A contract used as a storage for governance settings like quorum and decision thresholds, voting duration, and minimum amount of votes required to create a proposal.

Issues

C5-01 Unused variable

Low Acknowledged

The variable `transferValueForDelay` is never used in the code. We recommend removing all unused variables from the code to improve its readability.

C5-02 Insufficient validation of governance parameters

Low Acknowledged

The function `_validateGovernanceSettings` only makes the most basic validation checks on parameters.

```
    function _validateGovernanceSettings(NewGovernanceSettings memory settings)
        internal
        pure
    {
        // Check all values for sanity
        require(
            settings.quorumThreshold < DENOM,
```

```
            ExceptionsLibrary.INVALID_VALUE
        );
        require(
            settings.decisionThreshold <= DENOM,
            ExceptionsLibrary.INVALID_VALUE
        );
        require(settings.votingDuration > 0, ExceptionsLibrary.INVALID_VALUE);
    }
```

For example, a decision threshold of zero value can be set.

Recommendation

Add more sanity checks for the governance parameters to eliminate setting invalid or risky values.

C6. Token

Overview

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

Issues

C6-01 Possible gas limit problem

Medium Acknowledged

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

```
    /**
     * @dev The given getter returns the total balance of the address that is not locked
for transfer, taking into account all the TGEs with which this token was distributed. Is
the difference.
```

```
     * @param account Account address
     * @return Unlocked balance of account
     */
    function unlockedBalanceOf(address account) public view returns (uint256) {
        // Get total account balance
        uint256 balance = balanceOf(account);

        // Iterate through TGE list to get locked balance
        address[] memory _tgeList = tgeList;
        uint256 totalLocked = 0;
        for (uint256 i; i < _tgeList.length; i++) {
            if (ITGE(_tgeList[i]).state() != ITGE.State.Failed)
                totalLocked += ITGE(tgeList[i]).lockedBalanceOf(account);
        }

        // Return difference
        return balance - totalLocked;
    }
```

Recommendation

Refactor the code or remove instant minting functionality in favour of claiming only in case of successful TGE.

C6-02 Max supply is not validated

Low Acknowledged

The token inherits from the ERC20CappedUpgradeable contract which has a constraint of maximum tokens of 2^224^ - 1. It's never checked if this constraint is not exceeded.

C6-03 Inconsistent documentation

Low Acknowledged

The documentation of the `description` field says it has a maximum length of 5000 characters, but his constraint is never checked.

C6-04 Gas optimisations

Low Partially fixed

1. Unnecessary read from storage of `tgeList[i]` in the `unlockedBalanceOf()` function.
2. Some other contracts read the full array only for its first element: `getTGEList()`.

C6-05 Flawed authorisation

Low Acknowledged

The `burn()` and `onlyTGE()` functions share the same authorization model: it checks if the caller is of type TGE in the Registry contract. While the current implementations are safe, the possible future updates (contracts are meant to be deployed behind proxies) may cause an authorization problem for the TGE of pool A to be able to call restricted functions of the token of pool B.

```
    modifier onlyTGE() {
        require(
            service.registry().typeOf(msg.sender) ==
                IRecordsRegistry.ContractType.TGE,
            ExceptionsLibrary.NOT_TGE
        );
        _;
    }
```

C6-06 Typos

Info Acknowledged

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

C7. TGE

Overview

A sale contract to be deployed with each Pool via the 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

C7-01 Implicit rounding up of locked amount in purchase()

Medium Acknowledged

Implicit rounding up of the payment and vested amounts is implemented in the `purchase()` function. It should be documented if this is the desired behaviour. Users may be confused with not enough approvals if they calculate the needed value as `amount * info.price`.

```
uint256 purchasePrice = (amount * info.price + (1 ether - 1)) / 1 ether;

uint256 vestedAmount = (amount * info.vestingPercent + (DENOM - 1)) /
            DENOM;
```

Recommendation

We recommend avoiding rounding up.

C7-02 Gas optimisations

Low Acknowledged

1. The `purchaseOf[msg.sender]`, `info.unitOfAccount`, and `token` variables are read from storage multiple times in the `redeem()` function.
2. A fixed whitelist of users in the `initialize()` function could be implemented with a Merkle tree, available in the OpenZeppelin library.

C7-03 Price format is not documented

Info Acknowledged

The `TGEInfo.price` parameter is supposedly stored in form of mantissa with 1e18 exponent, but never documented.

C7-04 Allowed excessive payments

Info Acknowledged

The `purchase()` function allows excessive payments in ETH form, not returning the remaining.

```
function purchase(uint256 amount)
    ...
        require(
            msg.value >= purchasePrice,
    ...
}
```

C7-05 Claiming can be available in active TGE state

Info Acknowledged

The `claimAvailable()` function allows claiming in an active state of the TGE.

```
function claimAvailable() public view returns (bool) {
    return
        vestingTVLReached &&
        block.number >= createdAt + info.vestingDuration &&
        (state()) != State.Failed;
}
```

C7-06 Typos

Info Acknowledged

Typos reduce the code's readability. Typos in 'Retrun', 'successfull'.

C7-07 Instant mining of purchased tokens

Info Acknowledged

The `purchase()` function could be mining part of purchased tokens to the buyer, which may complicate the further process of adding liquidity to DEX or other protocols since there would be already added user tokens.

C8. Registry

Overview

A registry contract that stores all the contracts, events, tokens, and available companies of the project. Inherits RegistryBase, CompaniesRegistry, RecordsRegistry, and TokensRegistry contracts.

No issues were found.

C9. CompaniesRegistry

Overview

A registry contract that stores and manages available companies of the project.

C10. RecordsRegistry

Overview

A registry contract that stores and manages the contracts and events of the project.

No issues were found.

C11. TokensRegistry

Overview

A registry contract that stores and manages whitelisted tokens of the project.

Issues

C11-01 Function isTokenWhitelisted never used

Low Acknowledged

The function `isTokenWhitelisted()` is never used in the project's code. We recommend removing all unused code to improve reability.

C12. RegistryBase

Overview

A registry contract that stores and manages the contracts and events of the project.

No issues were found.

C13. Libraries and interfaces

Overview

A registry contract that stores and manages whitelisted tokens of the project.

No issues were found.

C14. All contracts

Overview

A contract that stores and manages Governor settings, i.e. proposal voting parameters. Can be modified only by the Pool itself.

Issues

C14-01 Lack of automated tests

Medium Acknowledged

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%.

C14-02 Events parameters are not indexed

Info Acknowledged

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

2 high, 4 medium, 12 low severity issues were found during the audit. 2 high, 1 medium, 1 low 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.

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

The audit was done based on the code published in Guthub. Users need to check if they are interacting with the same implementations as were audited.

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