#HashEx
BLOCKCHAIN SECURITY

# Trinity DAO

## smart contracts final audit report

November 2023

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 Trinity DAO team to perform an audit of their smart contracts. The audit was conducted between 25/10/2023 and 31/10/2023.

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 was provided directly in .sol file with very limited tests and documentation. The SHA-1 hashes of audited contracts:

GTToken.sol `5365440375af18917e3fc154eaf5058459bd2216`

Distributor.sol `7ced4ce7ae73ba790aeaaca3e1fbbd37c97bc950`

LocalMatrix.sol `bf87b2f95c393254aa2e6dfae5022709a2250e7d`

LocalMatrixLib.sol `b82c6940db593f1a6d9b65ab6649d266c47e9922`

Core.sol `015e4750f599b590609ed38c1ebae347bd206442`

ILevelsContract.sol `032bd45a38a060aab1765a114918876c0127d5ea`

IDistributor.sol `282fe9cc02e7a13236b007d47d2ada8db8a4ed99`

IERC20.sol `90ebd78dd3051ecb612f01f57888986142102b3e`

IGTToken.sol `20fcaada386542113fba227b0d1b44a02d482a45`

IMatrix.sol `b08f637ba4434d7f4fd147d2f86471a1a35b7fc0`

**Update**: the Trinity DAO team has responded to this report. The updated contracts have

SHA-1:

GTToken.sol `eb7ae753591c5a274bffca05c92425521e699f73`

Distributor.sol `ff31f34c2dedc4537ea07adf2d736a04dfdd9a64`

LocalMatrix.sol `f223599ba3d6f746ac8799e54eca9615f970d6dc`

LocalMatrixLib.sol `b82c6940db593f1a6d9b65ab6649d266c47e9922`

Core.sol `015e4750f599b590609ed38c1ebae347bd206442`

ILevelsContract.sol `032bd45a38a060aab1765a114918876c0127d5ea`

IDistributor.sol `6d1e6baea8165f638483ad6013b74b3ff309845c`

IERC20.sol `90ebd78dd3051ecb612f01f57888986142102b3e`

IGTToken.sol `20fcaada386542113fba227b0d1b44a02d482a45`

IMatrix.sol `cd9eb252eabcc7271ea5d78b345e45914084c1ff`

## 2.1 Summary

| Project name | Trinity DAO |
|---|---|
| URL | https://trinity-dao.com |
| Platform | Binance Smart Chain |
| Language | Solidity |
| Centralization level | High |
| Centralization risk | High |

## 2.2 Contracts

| Name | Address |
| --- | --- |
| GTToken | |
| TrinityDAO | |
| LocalMatrix | |
| AccountsLocalMatrixes | |
| Core | |
| Interface and imports | |

# 3. Project centralization risks

The project owner has indirect ability to withdraw user's funds without explicit permission.

# 4. Found issues


| | |
|---|---|
| High | 4 (22%) |
| Medium | 2 (11%) |
| Low | 8 (44%) |
| Info | 4 (23%) |

## C41. GTToken

| ID | Severity | Title | Status |
|---|---|---|---|
| C41Ic9 | High | ERC20 standard violation | Resolved |
| C41Ica | Low | Default visibility of state variables | Resolved |
| C41Icb | Low | Inconsistent behavior between transfer() and transferFrom() functions | Acknowledged |
| C41Icd | Low | Gas optimizations | Acknowledged |
| C41Icc | Info | Possible transfer blocking by donating tokens to next level | Acknowledged |

## C42. TrinityDAO

| ID | Severity | Title | Status |
|---|---|---|---|
| C42Id0 | Low | Low entropy randomization in selecting random referrer | Acknowledged |

| | | | |
|---|---|---|---|
| C42Id3 | Low | Gas optimizations | Acknowledged |
| C42Ice | Low | Default visibility of state variables | Resolved |
| C42Id2 | Info | Trinary bonus can exceed balance of the Matrix contract | Resolved |

## C43. LocalMatrix

| ID | Severity | Title | Status |
|---|---|---|---|
| C43Id8 | High | Refund amount can exceed input amount | Acknowledged |
| C43Id5 | High | The getTrinaryBonus() function is callable with zero amount, leading to bonus being lost | Partially fixed |
| C43Id6 | High | Refund success flag is not updated | Resolved |
| C43Id9 | Low | Gas optimizations | Partially fixed |
| C43Id4 | Low | Default visibility of state variables | Resolved |
| C43Ie2 | Info | Typographical error | Open |

## C44. AccountsLocalMatrixes

| ID | Severity | Title | Status |
|---|---|---|---|
| C44Idb | Medium | Possible math problem | Acknowledged |
| C44Idc | Info | Possible surrogate handling logic problem | Resolved |

# C45. Core

| ID | Severity | Title | Status |
| --- | --- | --- | --- |
| C45Ibf | Medium | No mechanism to transfer contract ownership | Acknowledged |

# 5. Contracts

## C41. GTToken

### Overview

An ERC20 standard token with minting functionality for a selected address - TrinityDAO contract.

### Issues

#### C41Ic9 ERC20 standard violation

High Resolved

The contract's `transfer` function exhibits behavior that is not compliant with the ERC20 standard:

1. The function returns `false` upon a successful transfer, which is contrary to the standard that expects a `true` value.
2. Transfers of a zero amount are prohibited. The ERC20 standard does not enforce such a restriction, and zero transfers should be treated as valid (often used to trigger side effects).

Non-compliance with established standards can lead to incompatibilities with third-party tools, services, and dApps that expect standard behavior. It also can introduce unexpected behaviors for users and developers familiar with ERC20 standard.

```
    function transfer(address to, uint256 amount) public virtual override returns (bool) {
        if (amount == 0) { return false; }
        super.transfer(to, amount);
        emit TokenTransfer(to, _msgSender(), amount, 0, 0);
    }
```

## C41Ica Default visibility of state variables

Low Resolved

The `tokenLimit`, `tokenAmount`, and `distributor` state variables in the contract have been declared without an explicit visibility specifier. In Solidity, if no visibility is specified, the default is `internal`. This means that while these variables are not directly accessible from external calls, they can be accessed and potentially modified by derived contracts. Not specifying visibility explicitly can lead to confusion about the intended accessibility of these variables and may inadvertently expose them to unintended modifications in future contract iterations.

## C41Icb Inconsistent behavior between transfer() and transferFrom() functions

Low Acknowledged

The `transfer` and `transferFrom` functions in the provided ERC20 implementation exhibit different behaviors:

1. The `transfer` function restricts transfers of a zero amount, whereas `transferFrom` (as per the provided information) does not enforce such a restriction.
2. The `transfer` function emits a custom `TokenTransfer` event, while `transferFrom` does not.

This inconsistency can lead to confusion for developers and users, potential issues in dApps or services built on top of this token, and makes the token's behavior deviate from standard ERC20 expectations.

```
    function transfer(address to, uint256 amount) public virtual override returns (bool) {
        if (amount == 0) { return false; }
        super.transfer(to, amount);
        emit TokenTransfer(to, _msgSender(), amount, 0, 0);
    }
```

## C41Icd Gas optimizations

Low Acknowledged

1. The `accountStatuses` data should be moved to immutable state.

2. Unnecessary read of user's balance in call of `getAccountStatusLevel()` inside the `_beforeTokenTransfer()` function.

3. Multiple reads from storage in the `getStatusLevelViaBalance()` function: `accountStatuses.length` and `accountStatuses[i].lowBorder` variables.

## C41Icc Possible transfer blocking by donating tokens to next level

● Info ⊘ Acknowledged

The contract's mechanism, which assigns status levels based on token balances and prohibits transfers that would decrease a user's status level, introduces a potential risk. A malicious actor or even an uninformed user could send tokens to an account, pushing it to a higher status level. Once this happens, the recipient becomes effectively "locked" from transferring any tokens out, as any transfer would lead to a level decrease and is thereby prohibited. This could lead to scenarios where token balances are inadvertently frozen, disrupting user activities and potentially leading to a loss of trust in the contract's functionality.

```
function _beforeTokenTransfer(address from, address to, uint256 amount)
internal
override
{
    if (contractOwner != msg.sender && from != address(0)) {
        uint balanceOf = balanceOf(from);
        uint8 currentAccountStatusLevel = getAccountStatusLevel(from);
        uint8 newAccountStatusLevel;
        if (currentAccountStatusLevel != 0) {
            newAccountStatusLevel = getStatusLevelViaBalance(balanceOf - amount);
        }
        require(newAccountStatusLevel == currentAccountStatusLevel, "GTE: 1");
    }
    super._beforeTokenTransfer(from, to, amount);
}
```

# C42. TrinityDAO

## Overview

Main contract of the project - an MLM staking for a fixed ERC20 token. The referral multilayer scheme is configured via instances of LocalMatrix contracts. Has access to minting GT tokens as an additional reward for staking.

## Issues

### C42Id0 Low entropy randomization in selecting random referrer

Low Acknowledged

The `getRandomReferer` function attempts to generate a pseudo-random number by hashing a combination of `block.timestamp`, `block.difficulty`, and `msg.sender`. While this method does provide some level of randomness, it lacks sufficient entropy and is potentially vulnerable to manipulation.

1. `block.timestamp`: Miners can slightly manipulate timestamps, which could influence the result.
2. `block.difficulty`: Predictable and can't be used as a major source of randomness. For EVM versions after the Paris, it behaves as an alias for `block.prevrandao.`
3. `msg.sender`: Publicly known information.

Given the above inputs, a well-informed and well-resourced attacker could predict or influence the outcome of the `getRandomReferer` function.

### C42Id3 Gas optimizations

Low Acknowledged

1. The `GTToken`, `IERC20Token`, `gtTokenDecimal`, `gtTokenSize` variables can be declared immutable or constant.

2. Multiple reads from storage in the `isSubMatrix()` function: `matrixCount`.

3. The `isSubMatrix()` function should break the loop upon the first found condition.

### C42Ice Default visibility of state variables

Low Resolved

The contract contains several variables (`players`, `accountsByAddress`, `IERC20Token`, `GTToken`, `gtTokenDecimal`, `gtTokenSize`, and `levelInfos`) that use the default visibility. By not explicitly specifying a visibility (like `public`, `internal`, or `private`), it can lead to confusion about the intended accessibility of these variables, possibly exposing them to unintended access patterns or making it harder to discern the contract's intended behavior.

### C42Id2 Trinary bonus can exceed balance of the Matrix contract

Info Resolved

The `TrinityDAO.getTrinaryBonus()` function calls for `LocalMatrix.getTrinaryBonus()` and transfers up to `account.balance[1] + account.balance[2] + account.balance[3]` amount to the user. In general, the Matrix contract's balance can be insufficient to perform such transfer due to possibility of adding a user without incoming payment via `onlyOwner buyProgram()` function.

## C43. LocalMatrix

## Overview

A referral model contract that stores staked user's funds. Some functions of LocalMatrix rely on the AccountsLocalMatrixes library. It can be called only via the Distributor contract.

# Issues

## C43Id8 Refund amount can exceed input amount

High Acknowledged

The general nature of the TrinityDAO as a Ponzi scheme may cause an insufficient balance to complete a withdrawal. In particular, the `TrinityDAO.refundRequest()` can't be completed for a single user in overall due to refund amount being doubled against the input amount for the `buyProgram()` function.

```
  function buyProgram(
      address accountAddress,
      address referer,
      uint8 level
  ) external isLevelTreeInitialized(level) nonReentrant {
      ...
      uint levelPrice = localMatrixes[level].getLevelPrice();
      _sendInLocalMatrix(level, levelPrice);\
  }

    function refundRequest(address accountAddress)
    external
    onlyDistributor
    levelTreeInitialized
    accountInTree(accountAddress)
    nonReentrant {
      ...
      uint refundAmount = levelPrice;
      if (tree.accounts[accountAddress].balance[1] == 0) { refundAmount += levelPrice /
2; }
      else { refundAmount += tree.accounts[accountAddress].balance[1]; }
      if (tree.accounts[accountAddress].balance[2] == 0) { refundAmount += levelPrice /
2; }
      else { refundAmount += tree.accounts[accountAddress].balance[2]; }
      delete tree.accounts[accountAddress];
      IERC20Token.transfer(accountAddress, refundAmount);
  }
```

## Recommendation

Increase test coverage to eliminate possible problems with insufficient funds.

## C43Id5 The getTrinaryBonus() function is callable with zero amount, leading to bonus being lost

High Partially fixed

The `getTrinaryBonus()` function allows the caller to specify an amount, which is used to transfer the bonus to the `receiverAddress`. There is no check in place to ensure that this amount is greater than zero. If called with a zero amount, the function will effectively delete the bonus for the specified `accountAddress` without actually transferring any bonus to the receiver. This could lead to unintentional loss of bonuses, especially if the function is called programmatically.

```
    function getTrinaryBonus(address accountAddress, uint amount, address receiverAddress)
external onlyDistributor {
        AccountsLocalMatrixes.LevelTree storage tree = matrixLevelTrees[0];
        require(
            roots[accountAddress] != address(0),
            "LME: 21"
        );
        require(
            tree.getAccountBalance(accountAddress) >= amount,
            "LME: 22"
        );
        IERC20Token.transfer(receiverAddress, amount);
        delete tree.accounts[accountAddress];
    }
```

## Recommendation

Add a `require` check to avoid invertedly passing not initialized parameter to the function.

## C43Id6 Refund success flag is not updated

High Resolved

The `refundRequest()` function checks whether a refund has been processed for an account

using the condition `if (!tree.accounts[accountAddress].isRefundDone)`. However, within this conditional block, there's no code that updates the `isRefundDone` attribute for the given account. This omission allows the condition to always be met, potentially allowing users to request multiple refunds which can be exploited to drain funds from the contract.

```
    function refundRequest(address accountAddress)
    external
    onlyDistributor
    levelTreeInitialized
    accountInTree(accountAddress)
    nonReentrant {
        AccountsLocalMatrixes.LevelTree storage tree = matrixLevelTrees[0];
        require(
            tree.accounts[accountAddress].programPurchaseTime + 365 days <=
block.timestamp,
            "LME: 9"
        );
        require(
            tree.accounts[accountAddress].reinvestCount == 0,
            "LME: 10"
        );
        require(
            roots[accountAddress] == address(0),
            "LME: 20"
        );
        if (!tree.accounts[accountAddress].isRefundDone) {
            uint refundAmount = levelPrice;
            address[3] memory childs;
            (childs,) = tree.getAccountMatrix(accountAddress);
            if (tree.accounts[accountAddress].balance[1] == 0) { refundAmount +=
levelPrice / 2; }
            else { refundAmount += tree.accounts[accountAddress].balance[1]; }
            if (tree.accounts[accountAddress].balance[2] == 0) { refundAmount +=
levelPrice / 2; }
            else { refundAmount += tree.accounts[accountAddress].balance[2]; }
            delete tree.accounts[accountAddress];
            IERC20Token.transfer(accountAddress, refundAmount);
            distributor.emitRefundsEvent(accountAddress, refundAmount, level);
        }
    }
```

### Recommendation

Update the `tree.accounts[accountAddress].isRefundDone` instead of fully deleting mapping item.

## C43Id9 Gas optimizations

Low Partially fixed

1. The `distributor`, `IERC20Token`, `level`, `levelPrice`, `activationPrice` variables can be declared immutable.

2. The `selfAddress` variable is not initialized nor used anywhere.

3. The `childs[]` array is not used in the `refundRequest()` function.

## C43Id4 Default visibility of state variables

Low Resolved

The contract contains several variables (`distributor`, `IERC20Token`, `selfAddress`, and `matrixLevelTrees`) that use the default visibility. By not explicitly specifying a visibility (like `public`, `internal`, or `private`), it can lead to confusion about the intended accessibility of these variables, possibly exposing them to unintended access patterns or making it harder to discern the contract's intended behavior.

## C43Ie2 Typographical error

Info Open

The term "referer" is used in the contract, which is presumably a typographical error.
The correct term should be "referrer". Misnaming variables can lead to confusion for developers,
maintainers, and auditors, potentially obscuring the intent and functionality of the code.

# C44. AccountsLocalMatrixes

## Overview

A library contract for LocalMatrix referral tree functions.

## Issues

### C44Idb Possible math problem

Medium Acknowledged

The Matrix contract receives not more than `levelPrice` amount of payment tokens for `LocalMatrix.addAccountInLevelTree()`, and `_initiatePayment()` spends at most full `levelPrice` amount. Then `_proceedReinvest()` tries to spend more, and these payments may cause a general withdrawal failure due to insufficient funds.

```
function paste(
    LevelTree storage self,
    address accountAddress,
    address refererAddress,
    bool isRefererSurrogate,
    address levelsContractAddress,
    address distributorAddress
) internal {
    ...
    uint8 position = _findFreePosition(parentMatrix.connections);
    _initiatePayment(
        self,
        accountAddress,
        parentMatrixOwner.addr,
        position,
        levelsContract,
        distributorAddress
    );
    if (_isReinvest(position)) {
        IDistributor distributorContract = IDistributor(distributorAddress);
        _proceedReinvest(
            self,
            parentMatrixOwner,
```

```
                levelsContract,
                distributorAddress
            );
        }
    }

function _initiatePayment(
        LevelTree storage self,
        address sender,
        address receiver,
        uint position,
        ILevelsContract levelsContract,
        address distributorAddress
    ) private {
        uint amount;
        Account storage receiverAccount = self.accounts[receiver];
        ...
        if (position == 3) {
            amount = levelsContract.getLevelPrice();
            ...
        }
        if (amount != 0) {
            levelsContract.pay(receiver, amount);
            levelsContract.emitEvent(sender, receiver, position,
receiverAccount.reinvestCount, amount);
        }
    }

    function _proceedReinvest(
        LevelTree storage self,
        Account storage reinvestAccount,
        ILevelsContract levelsContract,
        address distributorAddress
    ) private {
        ...
        paste(
            self,
            reinvestAccount.addr,
            reinvestReferer,
            isRefererSurrogate,
            address(levelsContract),
            distributorAddress
        );
```

```
    }
```

## Recommendation

Increase test coverage to eliminate possible problems with insufficient funds.

## C44Idc Possible surrogate handling logic problem

Info Resolved

The function designed for reinvestment has inconsistent logic for handling surrogate referrers. Initially, it checks if the referrer is a surrogate, and subsequently, it calls `_closeAccountMatrix()`, which seems to be designed for non-surrogate referrers. Then, when the function tries to get the reinvest referrer, and if this referrer isn't a surrogate, it ends up calling `_closeAccountMatrixWithSurrogateReferer()`, which appears to be designed specifically for surrogate referrers. This logic can lead to potential discrepancies in how matrices are closed based on the referrer type.

```
function _proceedReinvest(
    LevelTree storage self,
    Account storage reinvestAccount,
    ILevelsContract levelsContract,
    address distributorAddress
) private {
    IDistributor distributorContract = IDistributor(distributorAddress);
    if (!reinvestAccount.isRefererSurrogate) {
        _closeAccountMatrix(self, reinvestAccount);
        paste(
            self,
            reinvestAccount.addr,
            reinvestAccount.refererAddress,
            false,
            address(levelsContract),
            distributorAddress
        );
        return;
    }
    (
        address reinvestReferer,
        bool isRefererSurrogate
```

```
        ) = _getReferer(
            self,
            reinvestAccount.addr,
            levelsContract
        );
        if (!isRefererSurrogate) {
            _closeAccountMatrixWithSurrogateReferer(
                self, reinvestAccount, reinvestReferer
            );
        }
        ...
    }
```

### Update

The Trinity team answered that this behaviour is intended.

## C45. Core

## Overview

A simple authorization model for a single address ownership.

## Issues

### C45Ibf No mechanism to transfer contract ownership

Medium Acknowledged

The `Core` contract designates the contract deployer as the `contractOwner` upon deployment. However, there is no function available to transfer the ownership to another address after the contract has been deployed. This limitation means that if the original deployer's address gets compromised or if there's a need to delegate administrative control to another address, there's no way to do so, leaving the contract potentially stuck or at risk.

### Recommendation

Use OpenZeppelin's Ownable contract for managing ownership.

## C47. Interface and imports

## Overview

Various interface contracts and direct forks from OpenZeppelin repository:

Context, IERC20, IERC20Metadata, IGTToken, IDistributor, ILevelsContract, IMatrix.

No issues of any severity were found.

# 6. Conclusion

4 high, 2 medium, 8 low severity issues were found during the audit. 2 high, 3 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. Vulnerabilities checked

| | |
|---|---|
| Unchecked math | passed |
| Reentrancy attacks | passed |
| Front-run attacks | passed |
| Flashloat attacks | passed |
| DoS with (unexpected) revert | passed |
| DoS with block gas limit | passed |
| Transaction-ordering dependence | passed |
| ERC/BEP and other standards violation | passed |
| Implicit visibility levels | passed |
| Excessive gas usage | passed |
| Timestamp dependence | passed |
| Forcibly sending ether to a contract | passed |
| Weak sources of randomness | not passed |
| Shadowing state variables | passed |
| Usage of deprecated code | passed |

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter


#HashEx
BLOCKCHAIN SECURITY