Morpheus Labs

Two Monkey Juice Bar (TMON) SC Audit

Final

Security Audit as a Service

Morpheus Labs has a proven IT Expertise in the Financial Industry as with different IT Solution Providers within their sectors. With us, as Certified Security Expert, we can bring you beyond your Security Challenges before and after an Audit issue.

TABLE OF CONTENTS

Content

DISCLAIMER

The client agrees to fully indemnify Morpheus Labs from all liabilities, costs, expenses, damages, financial losses, reputation embarrassment including the following but not limited to;
Hacks by any party, security incursion by any means and /or parties, any form of malware, ransomware, DDOS attack, malicious intent by internal and/or external parties.

METHODOLOGY

Combining the knowledge of the world's common and uncommon smart contract's known bugs and hacks with our own experience of developing and reviewing dozens of lines of code of productive applications and smart contracts, we create a detailed security audit checklist.
When auditing a smart contract, we perform 6 steps:

1. We do receive your packaged Source code:
To ensure the possibility to have a consistent view on the to be reviewed source code, we do ask you to provide to us a Truffle project in a compressed archive, or the link to a source code repository related to the audit, to obtain changes, which may lead to inconsistency of the audit results. If your smart contracts been already verified and deployed on a test network, they can be audited as well. We do ensure the integrity of the audited files can be confirmed after the audit. This means we do require a kind of fingerprint for the exact source code version in our audit report. If we do receive the source code repository, we will take note of the commit number. In other cases, we compute a SHA-256 hash of every file where a audit is requested.

2. We will get familiar with your project and do our 1st Code Review:
Before we will go deep into the code, we familiarize ourselves with the purpose of the smart contract and his architecture 1st. For that the provisioning of documentation is requested. If this won't be detailed enough this will may lead into a discussion with the development too, for them to explain their design and architecture.
After this a pre-liminary code review will be executed. We do read through the source code and try to understand the main design decisions, we do look at the libraries being used and doe very the test coverage

3. Static code, Quality, Vulnerability, and functional Analysis:

We run our in-house tools to analyze both the source code and the compiled byte code of the contract(s). This results in a list of suspicious security vulnerabilities. The tools also check for coding style and best practices.

Our experts manually go through every of the tools' output items to determine if it is an actual issue. In addition, the experts manually scan every line of the provided source code and re-check very item of checklist to find additional issues.

Additionally as well as more general software engineering guidelines, such as commenting, variable naming, code structure and layout, function visibility and avoidance of replicated code.

A line by line code analysis is performed against a checklist of know vulnerabilities.

Issues will be labeled critical, major and minor, according to severity.

For the functional Analysis we do again a line by line code analysis to verify the correct behavior of the code.

4. Test about Efficiency, Life test and Gas usage:

Our experts assess the test scripts and test reports provided by the development team to determine if testing was done properly and sufficiently. We also deploy the contract to testnet network and manual test ourselves to ensure that it works as expected. This will be followed by a line by line code analysis to look at efficiency and the analysis of the Gas usage.

5. 1st Pre-Audit Report, Time for you to fix your findings:

A draft security audit report is then created and sent to the development team for discussion. We will work closely with the development team until every issue is resolved, each was either fixed or decided that no action is needed.

6. We do a 2nd Round Test and you do will receive your final Audit Report.

SCOPE OF AUDIT

This audit assesses the given token contract(s) source code to uncover security vulnerabilities and to confirm that it works according the given specification.

The audit does not assess the token-sale contract nor the token-sale process. The audit does not review whether the given smart contract source code conform to any external statement including but not limited to those which appear on whitepaper or websites.

The files which contain the source code of the smart contracts to be audited has the following names and hash:

Commit Date: 10th of March 2022

File name: **TMON.sol**
Version 1:
(Mar 10,2022)

File name: **VestingContract.sol**
Version 1:
(Mar 10,2022)

EXECUTIVE SUMMARY

The smart contract conforms to ERC-20 standard. It implements all functions, events, and state variables required by the ERC-20 specification.

Overall, the token smart contract(s) source code is clearly written, readable, well-documented, and demonstrates effective use of abstraction, separation of concerns, and modularity. The contract source code was originally inherited from some contracts from the OpenZeppelin repository and modified and simplified to adapt to the project's requirements. The audit itself won't include the contracts inherited from OpenZepplin.

The feedback provided is based on best practice for securely and clean written source code.

The audit team cannot verify that the development team has tested the smart contract thoroughly. We recommend that the development team resolves the issues and tests the smart contract thoroughly and makes the test report accessible.
We do recommend checking properly the given feedback and testing the given smart contracts regards the mentioned issues before deploying into production.

Below is a summary of the issue we found during the auditing process.

High severity	None
Medium severity	None
Low severity	1
Informational findings	None

VULNERABILITY LEVELS

The possible vulnerabilities are classified according to the levels described below:

High severity – A vulnerability that affects the desired outcome when using a contract or provides the opportunity to use a contract in an unintended way, or a vulnerability that can disrupt the contract functioning in several scenarios or creates the risk that the contract may be broken.
Medium severity – A vulnerability that could affect the desired outcome of executing the contract in certain scenarios.
Low severity – A vulnerability that does have a significant impact on the use of the contract and is probably subjective.
Informational findings – Recommendations about coding styles and issues that do not need to be fixed immediately but may consider fixing in the future.

ISSUES BY TYPE

High Severity

No findings marked with high severity found.

Medium Severity

No findings marked with medium severity found.

Low Severity

File Processed: **VestingContract.sol**

```
20      constructor (IERC20 _token, uint _amount, uint _distributed,
        address _beneficiary, uint _percentage, uint _cliffTime, uint _releasePeriod) {
```

Specify constructor's visibility – Severity: LOW

Recommendation: Define the constructor's visibility properly – this a recommendation regarding well written code

Informational Findings

No findings marked with medium severity found.

STATIC ANALYSIS

CONTRACTS/VESTINGCONTRACT.SOL

Security

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.morePos: 21:27:
startReleaseDate = block.timestamp + _cliffTime;

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.morePos: 34:67:
require(startReleaseDate + distributedTime*releasePeriod < block.timestamp, "Token is still in lock");

Gas & Economy

Gas costs:
Gas requirement of function VestingContract.release is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 31:4:

```
function release() public …
```

Gas costs:
Gas requirement of function VestingContract.vestingInfo is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 45:4:

```
function vestingInfo() public view returns …
```

Miscellaneous

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.moreMore Pos: 33:8:

```
require(amount > distributed, "Vesting is fully released") ;
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.moreMore Pos: 34:8:

```
require(startReleaseDate + distributedTime*releasePeriod < block.timestamp,
"Token is still in lock");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.moreMore Pos: 43:8:

```
require(owner() == _msgSender(), "Ownable: caller is not the owner");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.moreMore Pos: 63:8:

```
require(newOwner != address(0), "Ownable: new owner is the zero address");
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 36:34:

```
uint periodReleaseToken = percentage*amount/10000;
```

CONTRACTS/TMON.SOL

Security

Block timestamp:
Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.morePos: 21:27:

```
startReleaseDate = block.timestamp + _cliffTime;
```

Block timestamp:
Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.morePos: 34:67:

```
require(startReleaseDate + distributedTime*releasePeriod < block.timestamp,
"Token is still in lock");
```

Gas & Economy

Gas costs:
Gas requirement of function TMON.pause is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 33:4:

```
function pause() public onlyOwner
```

Gas costs:
Gas requirement of function TMON.unpause is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 37:4:

```
function unpause() public onlyOwner
```

Gas costs:
Gas requirement of function TMON.getVestings is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 41:4:

```
function getVestings() public view returns (address[] memory vestings)
```

Gas costs:
Gas requirement of function VestingContract.release is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 31:4:

```
function release() public
```

Gas costs:
Gas requirement of function VestingContract.vestingInfo is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 45:4:

```
function vestingInfo() public view returns
```

Gas costs:
Gas requirement of function TMON.name is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 62:4:

```
function name() public view virtual override returns (string memory)
```

CONTRACTS/TMON.SOL

Gas costs:
Gas requirement of function TMON.symbol is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 70:4:
```
function symbol() public view virtual override returns (string memory)
```

Gas costs:
Gas requirement of function TMON.transfer is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 113:4:
```
function transfer(address recipient, uint256 amount) public virtual override
returns (bool)
```

Gas costs:
Gas requirement of function TMON.allowance is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 121:4:
```
function allowance(address owner, address spender) public view virtual
override returns (uint256)
```

Gas costs:
Gas requirement of function TMON.approve is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 132:4:
```
function approve(address spender, uint256 amount) public virtual override
returns (bool)
```

Gas costs:
Gas requirement of function TMON.transferFrom is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 150:4:
```
function transferFrom
```

Gas costs:
Gas requirement of function TMON.increaseAllowance is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 178:4:
```
function increaseAllowance(address spender, uint256 addedValue) public
virtual returns (bool)
```

Gas costs:
Gas requirement of function TMON.decreaseAllowance is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 197:4:
```
function decreaseAllowance(address spender, uint256 subtractedValue) public
virtual returns (bool)
```

Gas costs:
Gas requirement of function TMON.burn is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 20:4:
```
function burn(uint256 amount) public virtual
```

Gas costs:
Gas requirement of function TMON.burnFrom is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)Pos: 35:4:
```
function burnFrom(address account, uint256 amount) public virtual
```

CONTRACTS/TMON.SOL

Miscellaneous

Constant/View/Pure functions:
TMON._beforeTokenTransfer(address,address,uint256) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.morePos: 52:4:

```
function _beforeTokenTransfer(
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 59:8:

```
require(!paused(), "ERC20Pausable: token transfer while paused");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 158:8:

```
require(currentAllowance >= amount, "ERC20: transfer amount exceeds
allowance");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 199:8:

```
require(currentAllowance >= subtractedValue, "ERC20: decreased allowance
below zero");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 226:8:

```
require(sender != address(0), "ERC20: transfer from the zero address");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 227:8:

```
require(recipient != address(0), "ERC20: transfer to the zero address");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 232:8:

```
require(senderBalance >= amount, "ERC20: transfer amount exceeds balance");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 253:8:

```
require(account != address(0), "ERC20: mint to the zero address");
```

CONTRACTS/TMON.SOL

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 276:8:

```
require(account != address(0), "ERC20: burn from the zero address");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 281:8:

```
require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 310:8:

```
require(owner != address(0), "ERC20: approve from the zero address");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 311:8:

```
require(spender != address(0), "ERC20: approve to the zero address");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 37:8:

```
require(currentAllowance >= amount, "ERC20: burn amount exceeds allowance");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 31:8:

```
require(!paused(), "ERC20Pausable: token transfer while paused");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 43:8:

```
require(owner() == _msgSender(), "Ownable: caller is not the owner");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 63:8:

```
require(newOwner != address(0), "Ownable: new owner is the zero address");
```

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.morePos: 52:8:

```
require(!paused(), "Pausable: paused");
```

CONTRACTS/TMON.SOL

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.moreInfo
Pos: 64:8:

```
require(paused(), "Pausable: not paused");
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 13:31:

```
uint internal token4Seed = 4*TOTAL_SUPPLY/100; //4%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 14:38:

```
uint internal token4PrivateSale = 5*TOTAL_SUPPLY/100; //5%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 15:34:

```
uint internal token4PreSale = 8*TOTAL_SUPPLY/100; //8%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 16:37:

```
uint internal token4PublicSale = 1*TOTAL_SUPPLY/100; //1%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 17:31:

```
uint internal token4Team = 16*TOTAL_SUPPLY/100; // 16%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 18:34:

```
uint internal token4Rewards = 30*TOTAL_SUPPLY/100; // 30%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 19:37:

```
uint internal token4Foundation = 20*TOTAL_SUPPLY/100; // 20%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 20:36:

```
uint internal token4Marketing = 10*TOTAL_SUPPLY/100; // 10%
```

CONTRACTS/TMON.SOL

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 21:36:

```
uint internal token4Liquidity = 3*TOTAL_SUPPLY/100; // 3%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 22:34:

```
uint internal token4Reserve = 3*TOTAL_SUPPLY/100; // 3%
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 119:24:

```
uint tgeToken = _tgePercent*_totalVesting/10000;
```

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.Pos: 36:34:

```
uint periodReleaseToken = percentage*amount/10000;
```

„Unit“ Testing

A successful test couldn’t be conducted following the provided readme file. Some corrections have been performed to get tests successfully:

Code:
Class: tmon/test/vesting-test.js
Deleted line 46: await blocktime();
which was causing an exception (Should has name is TMON: Error: `Invalid JSON RPC response: ""`) and the failure of the first of the 29 tests).

The final outcome of the truffle test is as follows:

Using network 'development'.

Compiling your contracts...
==========
> Everything is up to date, there is nothing to compile.

```
  Contract: TMON
    ✓ Should has name is TMON (46ms)
    ✓ Should has symbol is TMON
    ✓ Should has decimals is 18 (43ms)
    ✓ Should has 1 billion token in total (52ms)
    ✓ Seed address 0x95e2390c53fe9b202bd71ce87BECcCe568b8502F should has 2000000
token  (60ms)
    ✓ Private Sale address 0x9Dfb2C2F89d77cE2bFb7EabB2F611A6Da6b8f0dc should has 5000000
token  (46ms)
    ✓ Presale address 0xA1D201ac57dA5692c30F32B0f51677988c1F429E should has 0 token  (58ms)
    ✓ Public sale address 0x91d7963362caB391Cda66d9a3BbfA95F014607A1 should has 2000000
token  (63ms)
    ✓ Team address 0xEB5754a30E7906Da49B218219944fE31CE3467cE should has 0 token  (47ms)
    ✓ Rewards address 0x1066A4AaBa6f3F4d8e2172090d28f50dbeEc0809 should has 6000000
token  (62ms)
    ✓ Foundation address 0x8681970709757c692D999fA2307FfD752fb6BF55 should has 10000000
token  (59ms)
    ✓ Marketing address 0x1a69F0f5Dd9fBF578717ceE198aC45690b1E0F49 should has 0
token  (49ms)
    ✓ Liquidity address 0x245ab458f29453Ee74Bea85f3b08d17a7Cb5ACc5 should has 9000000
token  (59ms)
    ✓ Reserve address 0x6ed021fBb8Aaf3cEc159A3Ab44c946dF42dF7fA8 should has 3000000
token  (58ms)
    ✓ Should not pause as default and able to transfer token (159ms)
    ✓ Should able to pause (150ms)
    ✓ Should able to burn (274ms)
    ✓ Should has 10 vesting contracts (57ms)
    ✓ Checking vesting/cliff for seed (94ms)
    ✓ Checking vesting/cliff for private sale (90ms)
    ✓ Checking vesting/cliff for pre sale (80ms)
    ✓ Checking vesting/cliff for public sale (80ms)
    ✓ Checking vesting/cliff for team (106ms)
    ✓ Checking vesting/cliff for reward (80ms)
    ✓ Checking vesting/cliff for foudation (94ms)
    ✓ Checking vesting/cliff for marketing (108ms)
    ✓ Checking vesting/cliff for liquidity (75ms)
    ✓ Checking vesting/cliff for reserve (93ms)
    ✓ Release token after cliff time  (181879ms)


  29 passing (3m)
```

Environmental Reference:
Truffle: 5.4.32
Solidity: 0.8.7
Node: 17.4.0
Web3.js: 1.5.3
Ganache-UI: 2.5.4

CONCLUSION

The pre-audit of the TMON.sol, VestingContract.sol smart contract(s) has found several high severity issues from a security point of view. These were all properly resolved by the development team.
The static source code analysis is providing warnings which should be checked by the development team if they are applicable, but they don't stop to proceed.
Before deployment to production, we do recommend a proper re-test.

Morpheus Labs