Cyberscope

Audit Report

MemeStax

December 2024

Repository https://github.com/thexchange-pty-ltd/memstax-ERC20/tree/main

Commit 64d19880a4df15d323bfca1f77798432dc774f67

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Unresolved
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	ALM	Array Length Mismatch	Unresolved
	CR	Code Repetition	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	DPI	Decimals Precision Inconsistency	Unresolved
	MMN	Misleading Method Naming	Unresolved
	MEE	Missing Events Emission	Unresolved
	MZEM	Missing Zero Error Message	Unresolved
	ODM	Oracle Decimal Mismatch	Unresolved
	PBV	Percentage Boundaries Validation	Unresolved
	POSD	Potential Oracle Stale Data	Unresolved
	PPFL	Potential Permanent Funds Lock	Unresolved
	PRE	Potential Reentrance Exploit	Unresolved
	UVE	Uniform Vesting Endtime	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved

●	L13	Divide before Multiply Operation	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

Table of Contents

Analysis 1
Diagnostics 2
Table of Contents 4
Risk Classification 6
Review 7
Audit Updates 7
Source Files 7
Overview 8
MEMEStaxToken 8
VestingContract 8
Roles 9
Owner 9
Users 9
Findings Breakdown 10
MT - Mints Tokens 11
Description 11
Recommendation 12
ALM - Array Length Mismatch 13
Description 13
Recommendation 14
CR - Code Repetition 15
Description 15
Recommendation 15
CCR - Contract Centralization Risk 16
Description 16
Recommendation 17
DPI - Decimals Precision Inconsistency 18
Description 18
Recommendation 19
MMN - Misleading Method Naming 21
Description 21
Recommendation 22
MEE - Missing Events Emission 23
Description 23
Recommendation 23
MZEM - Missing Zero Error Message 24
Description 24
Recommendation 24
ODM - Oracle Decimal Mismatch 25

Description 25
Recommendation 26
PBV - Percentage Boundaries Validation 27
Description 27
Recommendation 28
POSD - Potential Oracle Stale Data 29
Description 29
Recommendation 29
PPFL - Potential Permanent Funds Lock 31
Description 31
Recommendation 33
PRE - Potential Reentrance Exploit 34
Description 34
Recommendation 35
UVE - Uniform Vesting Endtime 36
Description 36
Recommendation 37
L04 - Conformance to Solidity Naming Conventions 38
Description 38
Recommendation 39
L13 - Divide before Multiply Operation 40
Description 40
Recommendation 40
L19 - Stable Compiler Version 41
Description 41
Recommendation 41
L20 - Succeeded Transfer Check 42
Description 42
Recommendation 42
Functions Analysis **43**
Inheritance Graph **45**
Flow Graph **46**
Summary **47**
Disclaimer **48**
About Cyberscope **49**

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/thexchange-pty-ltd/memstax-ERC20/tree/main
Commit	64d19880a4df15d323bfca1f77798432dc774f67
Testing Deploy	https://testnet.bscscan.com/address/0x7cf14f9fc2784358fd00a8f80cef2465f2ca3030
Badge Eligibility	Must Fix MT Finding

Audit Updates

Initial Audit	25 Dec 2024

Source Files

Filename	SHA256
contracts/MEMEStaxToken.sol	c28cac923edcb1437b067add7de189e55752bf914ddd50a02530d9b2561a90aa

Overview

The MemeStax smart contracts collectively manage the creation, distribution, and vesting of the MEMEStaxToken, a token adhering to the ERC20 standard. The primary goal of the system is to facilitate token sales, minting, and a structured vesting mechanism for token holders. These contracts integrate with Chainlink oracles for price conversions and support various forms of token payments, including native token, stablecoins and SHIB tokens. They also ensure that minted tokens adhere to a maximum supply limit and provide withdrawal functionality for collected payments.

MEMEStaxToken

The `MEMEStaxToken` contract is an ERC20 token implementation that supports minting, burning, and payment handling during a token sale phase. It integrates with Chainlink oracles to dynamically calculate token prices in ETH, SHIB, or stablecoins, ensuring accurate conversions. The contract enforces a maximum supply of 888 billion tokens and provides mechanisms for minting tokens in exchange for payments, where excess ETH is refunded. It manages token distribution by transferring minted tokens to a vesting contract, which ensures structured release over monthly time. Additionally, the contract supports administrative features such as changing token price, withdrawing collected funds, and airdropping tokens.

VestingContract

The `VestingContract` is designed to manage the structured release of MEMEStaxTokens to investors or other stakeholders based on predefined vesting schedules. Shareholders are assigned a maximum token allocation, with tokens released in monthly increments as specified by the contract's parameters. The contract calculates the amount of tokens that can be withdrawn at any given time, factoring in the elapsed time since the sale ended. Only shareholders can initiate token withdrawals, and the owner can add or update shareholder allocations. This ensures a controlled and transparent distribution of tokens after the sale is completed.

Roles

Owner

The owner can interact with the following functions:

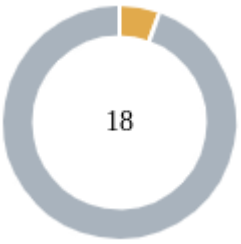
- `changePrice`
- `endPrivateListing`
- `addGroup`
- `addShareholder`
- `withdraw`
- `withdrawShib`
- `withdrawStableCoins`
- `airdrop`

Users

Users can interact with the following functions:

- `burn`
- `mint`
- `mintForShib`
- `mintForStableCoin`
- `withdraw` (in `VestingContract`)
- `calculateAllowedAmount` (in `VestingContract`)

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	17

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	17	0	0	0

MT - Mints Tokens

Criticality	Medium
Location	contracts/MEMEStaxToken.sol#L143,
Status	Unresolved

Description

The contract owner has the authority to mint tokens. The owner may take advantage of it by calling the `mint` or the `addShareholder` function. These functions allow the owner to mint tokens until the `maxSupply` limit is reached. As a result, the contract tokens will be highly inflated.

```
    function airdrop(address to, uint256 amount) public onlyOwner {
        require(
            totalSupply() + amount <= maxSupply * 10**decimals(),
            "Max supply exceeded"
        );
        _mint(to, amount);

    }

    function addShareholder(
        address vestingGroupAddress,
        address account,
        uint256 amount
    ) public onlyOwner {
        require(
            totalSupply() + amount <= maxSupply * 10**decimals(),
            "Max supply exceeded"
        );
        VestingContract vestingContract =
VestingContract(vestingGroupAddress);...
        _mint(vestingGroupAddress, amount);
        vestingContract.addShareholder(account, amount);
    }
```

Recommendation

It is recommended to mint the full `maxSupply` during contract deployment and allocate tokens transparently to predefined accounts. Additionally, the team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

ALM - Array Length Mismatch

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L164,278
Status	Unresolved

Description

The contract is designed to handle the process of elements from arrays through functions that accept multiple arrays as input parameters. These functions are intended to iterate over the arrays, processing elements from each array in a coordinated manner. However, there are no explicit checks to verify that the lengths of these input arrays are equal. This lack of validation could lead to scenarios where the arrays have differing lengths, potentially causing out-of-bounds access if the function attempts to process beyond the end of the shorter array. Such situations could result in unexpected behavior or errors during the contract's execution, compromising its reliability and security.

```
   function addGroup(
        address[] memory shareholderAddresses,
        uint256[] memory shareholderMaxAmount,
        ...
    ) public onlyOwner {
        uint256 amount = 0;
        for (uint256 i = 0; i < shareholderMaxAmount.length;
i++)
            amount += shareholderMaxAmount[i];
        ...
        }

    constructor(
        address[] memory shareholderAddresses,
        uint256[] memory shareholderAmount,
       ...
    ) Ownable(msg.sender) {
        ...
        for (uint256 i = 0; i < shareholderAddresses.length;
i++) {
            shareholders[shareholderAddresses[i]] =
ShareholderInfo(
                shareholderAmount[i],
                0
            );
        }
    }
```

Recommendation

To mitigate this, it is recommended to incorporate a validation check at the beginning of the function that accepts multiple arrays to ensure that the lengths of these arrays are identical. This can be achieved by implementing a conditional statement that compares the lengths of the arrays, and reverts the transaction if the lengths do not match. Such a validation step will prevent out-of-bounds errors and ensure that elements from each array are processed in a paired and coordinated manner, thus preserving the integrity and intended functionality of the contract.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L46,67,90
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specifically the `mint` , `mintForShib` and `mintForStableCoin` functions contain similar code segments.

```
function mint(uint256 numOfBillions) public payable
mintingIsAllowed {
        ...
    }

    function mintForShib(uint256 numOfBillions) public
mintingIsAllowed {
        ...
    }

    function mintForStableCoin(uint256 numOfBillions, address
stableCoinAddress) public mintingIsAllowed {
        ...
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L120,152,164,189
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Additionally, the contract owner has the authority to terminate the sale early without enforcing soft cap criteria or adhering to time restrictions. This flexibility centralises control, allowing the owner to make significant changes to the sale process, which could affect participant confidence and the overall integrity of the system.

```
    function changePrice(uint256 _price) public onlyOwner {
        usdtPricePerBillion = _price;
    }

    function endPrivateListing(address treasuryVestingGroup,
address account)
        public
        onlyOwner
    {
        ...
    }

    function addGroup(
        address[] memory shareholderAddresses,
        uint256[] memory shareholderMaxAmount,
        uint256 initialVestingPeriod,
        uint256 percentage
    ) public onlyOwner {
        ...
        _mint(address(vestingContract), amount);
        vestingGroups.push(address(vestingContract));
    }

    function addShareholder(
        address vestingGroupAddress,
        address account,
        uint256 amount
    ) public onlyOwner {
        ...
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L216
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

```
    function convertUsdtToShib(uint256 usdtAmount)
        public
        view
        returns (uint256)
    {
        uint256 ethPerShib = getShibPriceInEth();
        uint256 ethAmount = convertUsdtToEth(usdtAmount);
        uint256 shibAmount = (ethAmount / ethPerShib) * 10**18;

        return shibAmount;
    }

    function convertUsdtToEth(uint256 usdtAmount)
        public
        view
        returns (uint256)
    {
        uint256 usdtPerEth = getEthPriceInUSDT();

        uint256 ethAmount = (usdtAmount * usdtPerEth) / 10**6;

        return ethAmount;
    }
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L152,209
Status	Unresolved

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

Specifically, the contract is designed to manage token listings but contains a function, `endPrivateListing`, whose implementation does not align with its name. While the name implies that the function ends a private listing and sets the `privateListingEndTime`, its actual functionality impacts the public minting process. Specifically, the function is disabling public minting for users who would otherwise pay the required amount of funds. This mismatch between the function name and its operational scope could cause confusion for integrators or users of the contract and may lead to unintended consequences in the minting process.

```
    function endPrivateListing(address treasuryVestingGroup,
address account)
        public
        onlyOwner
    {
        privateListingEndTime = block.timestamp;
        uint256 maximumSupply = maxSupply * 10**decimals();
        if (totalSupply() < maximumSupply) {
            uint256 unmintedTokens = maximumSupply -
totalSupply();
            addShareholder(treasuryVestingGroup, account,
unmintedTokens);
        }
    }
```

Additionally, the contract is using a function named `getEthPriceInUSDT`, which implies that the function returns the price of ETH in terms of USDT. However, the naming is misleading because the actual purpose of the function is to return the price of 1 USDT per ETH, calculated using the `ethUsdtPriceFeed` data. This inconsistency can create confusion for developers or integrators using the contract, potentially leading to incorrect assumptions about the data returned by the function.

```
    function getEthPriceInUSDT() public view returns (uint256) {
        (, int256 price, , , ) = ethUsdtPriceFeed.latestRoundData();

        return uint256(price);
    }
```

Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

Additionally, it is recommended to rename the function to more accurately reflect its purpose, such as `getUSDTperETH`. This adjustment would align the function name with its intended behavior, improving clarity and reducing the likelihood of misinterpretation.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	MEMEStaxToken.sol#L120,164
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
    function changePrice(uint256 _price) public onlyOwner {
        usdtPricePerBillion = _price;
    }

    function addGroup(
        address[] memory shareholderAddresses,
        uint256[] memory shareholderMaxAmount,
        uint256 initialVestingPeriod,
        uint256 percentage
    ) public onlyOwner {
        ...
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MZEM - Missing Zero Error Message

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L341
Status	Unresolved

Description

The contract is missing a check in the `withdraw` function to ensure that the `allowedAmount` calculated for the caller is greater than zero before proceeding with the token transfer. If `allowedAmount` is zero, the function will attempt to execute a transfer of zero tokens and update the `withdrawnTokens` state unnecessarily. This could lead to confusion for users and integrators, as the function call will not revert with a clear error message, leaving users uncertain about the reason for their inability to withdraw funds.

```
    function withdraw() public onlyShareholder {
        uint256 allowedAmount =
calculateAllowedAmount(msg.sender);
        uint256 withdrawnTokens =
shareholders[msg.sender].withdrawnTokens;
        MEMEStaxToken staxToken =
MEMEStaxToken(staxTokenAddress);
        uint256 amount = allowedAmount - withdrawnTokens;
        staxToken.transfer(msg.sender, amount);
        shareholders[msg.sender].withdrawnTokens += amount;
    }
```

Recommendation

It is recommended to add a check at the beginning of the `withdraw` function to revert the call with a clear and descriptive error message if the `allowedAmount` is zero. This ensures that the function provides immediate feedback to the caller, improving the user experience and reducing potential misunderstandings or unnecessary state updates. Clear error messages also help maintain transparency and facilitate debugging for developers and users.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L216
Status	Unresolved

Description

The contract relies on data retrieved from an external Oracle to make critical calculations. However, the contract does not include a verification step to align the decimal precision of the retrieved data with the precision expected by the contract's internal calculations. This mismatch in decimal precision can introduce substantial errors in calculations involving decimal values.

```
    function convertUsdtToShib(uint256 usdtAmount)
        public
        view
        returns (uint256)
    {
        uint256 ethPerShib = getShibPriceInEth();
        uint256 ethAmount = convertUsdtToEth(usdtAmount);
        uint256 shibAmount = (ethAmount / ethPerShib) * 10**18;

        return shibAmount;
    }

    function convertUsdtToEth(uint256 usdtAmount)
        public
        view
        returns (uint256)
    {
        uint256 usdtPerEth = getEthPriceInUSDT();

        uint256 ethAmount = (usdtAmount * usdtPerEth) / 10**6;

        return ethAmount;
    }
```

Recommendation

The team is advised to retrieve the decimals precision from the Oracle API in order to proceed with the appropriate adjustments to the internal decimals representation.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L164,323
Status	Unresolved

Description

The contract utilizes variables for percentage-based calculations that are required for its operations. These variables are involved in multiplication and division operations to determine proportions related to the contract's logic. If such variables are set to values beyond their logical or intended maximum limits, it could result in incorrect calculations. This misconfiguration has the potential to cause unintended behavior or financial discrepancies, affecting the contract's integrity and the accuracy of its calculations.

```
    function addGroup(
        ...
        uint256 percentage
    ) public onlyOwner {
        ...
        VestingContract vestingContract = new VestingContract(
            shareholderAddresses,
            shareholderMaxAmount,
            initialVestingPeriod,
            amount,
            percentage
        );
        ...
        }

    function calculateAllowedAmount(address shareholderAddress)
        public
        view
        returns (uint256)
    {
        ...
        if (
            privateListingEndTime > 0 &&
            initialVestingPeriodEnd <= block.timestamp
        ) {
            allowedAmountInPercentage += percentage;
        }

        ...

        uint256 allowedAmount = (maxWithdrawableAmount *
            allowedAmountInPercentage) / TOTAL_PERCENTAGE;

        return allowedAmount;
    }
```

Recommendation

To mitigate risks associated with boundary violations, it is important to implement validation checks for variables used in percentage-based calculations. Ensure that these variables do not exceed their maximum logical values. This can be accomplished by incorporating `require` statements or similar validation mechanisms whenever such variables are assigned or modified. These safeguards will enforce correct operational boundaries, preserving the contract's intended functionality and preventing computational errors.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L203
Status	Unresolved

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
   function getShibPriceInEth() public view returns (uint256) {
        (, int256 price, , , ) =
shibEthPriceFeed.latestRoundData();

        return uint256(price);
    }

    function getEthPriceInUSDT() public view returns (uint256)
{
        (, int256 price, , , ) =
ethUsdtPriceFeed.latestRoundData();

        return uint256(price);
    }
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

PPFL - Potential Permanent Funds Lock

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L125,133,242
Status	Unresolved

Description

The contract is designed to allow the owner to withdraw funds raised in ETH, SHIB tokens, and stablecoins (USDT and USDC) using the `withdraw`, `withdrawShib`, and `withdrawStableCoins` functions, respectively. However, these functions are restricted by the `onlyOwner` modifier, which requires the caller to be the current contract owner. If ownership of the contract is renounced, these functions will become inaccessible, leaving the funds permanently locked in the contract. This creates a risk, particularly in scenarios where renouncing ownership might be considered part of the contract lifecycle or governance.

```
  function withdraw() public onlyOwner {
        uint256 balance = address(this).balance;
        require(balance > 0, "No ETH to withdraw");

        (bool success, ) = payable(owner()).call{value:
balance}("");
        require(success, "Transfer failed");
    }

    function withdrawShib() public onlyOwner {
        uint256 contractBalance =
shibToken.balanceOf(address(this)); // Get SHIB balance of the
contract
        require(contractBalance > 0, "No SHIB tokens in the
contract");

        bool success = shibToken.transfer(owner(),
contractBalance); // Transfer all SHIB tokens to the owner
        require(success, "SHIB transfer failed");
    }

    function withdrawStableCoins() public onlyOwner {
        uint256 usdtBalance =
usdtToken.balanceOf(address(this));
        uint256 usdcBalance =
usdcToken.balanceOf(address(this));

        require(usdtBalance > 0 || usdcBalance > 0, "No
stablecoins to withdraw");

        if (usdtBalance > 0) {
            bool usdtSuccess = usdtToken.transfer(owner(),
usdtBalance);
            require(usdtSuccess, "USDT transfer failed");
        }

        if (usdcBalance > 0) {
            bool usdcSuccess = usdcToken.transfer(owner(),
usdcBalance);
            require(usdcSuccess, "USDC transfer failed");
        }
    }
```

Recommendation

It is recommended to implement a mechanism that allows the withdrawal functions to operate independently of the contract ownership. One approach is to utilize a specific address, such as a treasury or multisig wallet, as the recipient of the funds. This address should not rely on ownership permissions, ensuring that withdrawals remain functional even if ownership is renounced. Additionally, clearly document the behaviour and risks associated with renouncing ownership to provide transparency to users and stakeholders.

PRE - Potential Reentrance Exploit

Criticality	Minor / Informative
Location	MEMEStaxToken.sol#L46
Status	Unresolved

Description

The contract makes an external call to transfer funds to recipients using the payable transfer method. The recipient could be a malicious contract that has an untrusted code in its fallback function that makes a recursive call back to the original contract. The re-entrance exploit could be used by a malicious user to drain the contract's funds or to perform unauthorized actions. This could happen because the original contract does not update the state before sending funds.

Specifically, the contract is vulnerable to a reentrancy attack in its `mint` function, which processes refunds for excess payment to the `msg.sender`. A malicious contract could exploit this vulnerability by using reentrant calls to bypass the `require` check that enforces the `maxSupply` limit. This would allow the attacker to mint more tokens than permitted, breaching the intended token supply constraints and potentially disrupting the tokenomics of the system. The refund logic does not adequately protect against recursive calls, leaving the contract susceptible to exploitation.

```
function mint(uint256 numOfBillions) public payable mintingIsAllowed {
        ...
        require(
            totalSupply() + amount <= maxSupply * 10**decimals(),
            "Max supply exceeded"
        );
        require(msg.value >= totalPrice, "Insufficient payment");
        // Refund excess ETH
        if (msg.value > totalPrice) {
            uint256 excessAmount = msg.value - totalPrice;
            payable(msg.sender).transfer(excessAmount);
        }
        ...
        _mint(privateRoundAddress, amount);
        privateRoundContract.addShareholder(msg.sender, amount);
    }
```

Recommendation

The team is advised to prevent the potential re-entrance exploit as part of the solidity best practices. Some suggestions are:

- Add lockers/mutexes in the method scope. It is important to note that mutexes do not prevent cross-function reentrancy attacks.
- Do Not allow contract addresses to receive funds.
- Proceed with the external call as the last statement of the method, so that the state will have been updated properly during the re-entrance phase.

Additioanlly, it is recommended to implement a `nonReentrant` modifier on the functions to prevent reentrancy attacks. This modifier ensures that no recursive calls can be made to the vulnerable function, safeguarding the contract against exploitation and maintaining the integrity of the `maxSupply` limit enforcement.

UVE - Uniform Vesting Endtime

Criticality	Minor / Informative
Location	MEMEStaxToken.sol#L164,309
Status	Unresolved

Description

The contract is using the `addGroup` function, which allows the owner to create new vesting contracts dynamically. However, within the `calculateAllowedAmount` function, all groups are designed to calculate the allowed vesting amounts based on the same `privateListingEndTime`. This means that groups added later are still subjected to the original vesting schedule, applying the same sale end time across all groups. This implementation could lead to discrepancies in the expected behaviour, as newly added groups might not be intended to adhere to the same timeline as the original groups.

```
    function addGroup(
        address[] memory shareholderAddresses,
        uint256[] memory shareholderMaxAmount,
        uint256 initialVestingPeriod,
        uint256 percentage
    ) public onlyOwner {
        ...
        VestingContract vestingContract = new VestingContract(
            shareholderAddresses,
            shareholderMaxAmount,
            initialVestingPeriod,
            amount,
            percentage
        );
        _mint(address(vestingContract), amount);
        vestingGroups.push(address(vestingContract));
    }

    function calculateAllowedAmount(address shareholderAddress)
        public
        view
        returns (uint256)
    {
        uint256 allowedAmountInPercentage = 0;
        uint256 maxWithdrawableAmount = shareholders[shareholderAddress]
            .maximumTokens;
        MEMEStaxToken staxToken = MEMEStaxToken(staxTokenAddress);

        uint256 privateListingEndTime =
staxToken.privateListingEndTime();

        uint256 initialVestingPeriodEnd = privateListingEndTime +
            (initialVestingPeriod * ONE_MONTH);
    ...
    }
```

Recommendation

It is recommended that the team clarify whether applying the same `privateListingEndTime` to all groups is the intended functionality. If this is the case, it should be clearly documented to prevent misunderstandings. Otherwise, it is recommended to modify the implementation to ensure that each group operates under its own specific or configurable vesting schedule to accommodate varying start and end times.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L14,15,16,120
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
IERC20 internal constant shibToken =
IERC20(0x95aD61b0a150d79219dCF64E1E6Cc01f0B64C4cE)
IERC20 internal constant usdtToken =
IERC20(0xdAC17F958D2ee523a2206206994597C13D831ec7)
IERC20 internal constant usdcToken =
IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48)
uint256 _price
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L223,325,327
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 months = vestingPeriodElapsedTime / ONE_MONTH
allowedAmountInPercentage += months * percentage
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	contracts/MEMEStaxToken.sol#L78,103,346
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
shibToken.transferFrom(msg.sender, address(this),
totalPriceInShib)
stableCoin.transferFrom(msg.sender, address(this),
totalPriceInStableCoin)
staxToken.transfer(msg.sender, amount)
```

Recommendation

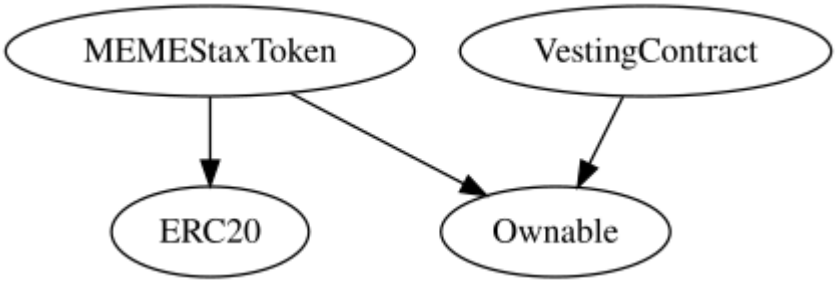
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

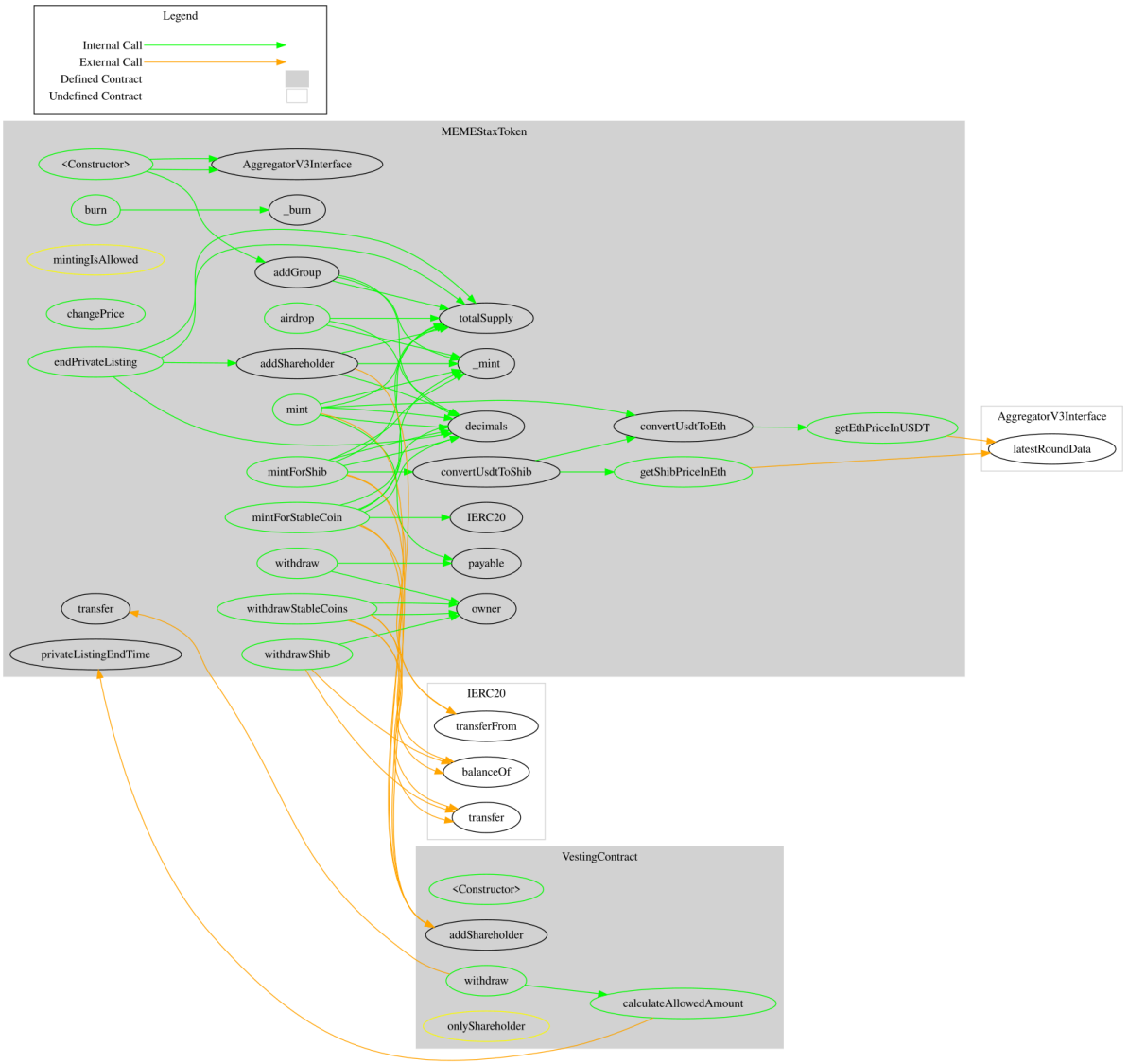
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
MEMEStaxToken	Implementation	ERC20, Ownable		
		Public	✓	ERC20 Ownable
	burn	Public	✓	-
	mint	Public	Payable	mintingIsAllowed
	mintForShib	Public	✓	mintingIsAllowed
	mintForStableCoin	Public	✓	mintingIsAllowed
	decimals	Public		-
	changePrice	Public	✓	onlyOwner
	withdraw	Public	✓	onlyOwner
	withdrawShib	Public	✓	onlyOwner
	airdrop	Public	✓	onlyOwner
	endPrivateListing	Public	✓	onlyOwner
	addGroup	Public	✓	onlyOwner
	addShareholder	Public	✓	onlyOwner
	getShibPriceInEth	Public		-
	getEthPriceInUSDT	Public		-
	convertUsdtToShib	Public		-
	convertUsdtToEth	Public		-

	withdrawStableCoins	Public	✓	onlyOwner
VestingContract	Implementation	Ownable		
		Public	✓	Ownable
	calculateAllowedAmount	Public		-
	withdraw	Public	✓	onlyShareholder
	addShareholder	Public	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
MEMEStaxToken
<Constructor>
AggregatorV3Interface
burn
_burn
mintingIsAllowed
addGroup
changePrice
airdrop
totalSupply
endPrivateListing
addShareholder
_mint
mint
decimals
convertUsdtToEth
getEthPriceInUSDT
AggregatorV3Interface
latestRoundData
mintForShib
convertUsdtToShib
getShibPriceInEth
mintForStableCoin
IERC20
withdraw
payable
transfer
withdrawStableCoins
owner
privateListingEndTime
withdrawShib
IERC20
transferFrom
balanceOf
transfer
VestingContract
<Constructor>
addShareholder
withdraw
calculateAllowedAmount
onlyShareholder

Summary

The MEMEStaxToken contract implements a comprehensive token management and distribution system that includes minting, burning, and a robust vesting mechanism through the integration of a dedicated VestingContract. This mechanism ensures structured and gradual release of tokens to stakeholders based on predefined schedules, supporting transparency and preventing premature withdrawals. The contract also supports various payment methods for minting, including ETH, SHIB, and stablecoins, with price conversions handled via Chainlink oracles for accuracy. This audit investigates potential security vulnerabilities, evaluates the correctness of business logic, and identifies areas for potential improvement.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope’s prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope’s position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io