MEMEHAIN

Smart Contract Audits

Terrance Nibbles - Certified Auditor

November 30, 2024

MEMEHAIN

Smart Contract Audits

Preface

This audit is of the MEMEHAIN presale & main contracts that were provided for detailed analysis on November 29, 2024. The primary & presale solidity smart contracts code are listed at the end of the report. This was manually audited as well as reviewed with other tools.

This token contracts that were audited are on the ETHEREUM Blockchain:

https://etherscan.io/address/0xcce1b79f02ebbf2d7ee8e88f5299de5bc926c5b6#code

https://etherscan.io/address/0x3b62f02ead53b08eb26c0dd922d8c862aa9d15da#code

DISCLAIMER:

This audit report is based on a professional review of the provided smart contract provided. It is important to note that this assessment represents our expert opinion and analysis of the code at the time of the evaluation. The findings and recommendations presented herein are not intended to serve as warranties, guarantees, or assurances of the contract's performance, security, or functionality on any live network, including the Ethereum mainnet.

We expressly disclaim any responsibility for errors, omissions, or inaccuracies in this report, as the assessment is conducted on a non-exhaustive basis and may not cover all possible scenarios or future developments. The audit is conducted in accordance with industry best practices and standards at the time of evaluation.

Furthermore, we are unable to confirm the deployment of this specific contract on the Ethereum mainnet. This report is solely based on the provided code and does not verify the actual deployment status on any live blockchain. It is the responsibility of the contract deployer to ensure the accurate deployment of the contract and adhere to security best practices when deploying to production environments.

Users, developers, and stakeholders are advised to perform additional due diligence and testing before deploying or interacting with the contract on any live network. This report should be considered as a tool for risk assessment rather than a guarantee of the contract's security or performance. In the dynamic and rapidly evolving field of blockchain technology, risks and vulnerabilities may emerge over time, and it is crucial to stay vigilant and up-to-date on security best practices.

By relying on this audit report, the reader acknowledges and accepts that the audit is based on the provided information and that no warranties, guarantees, or assurances are expressed or implied.

Smart Contract Audit Report: MemeHain (HAIN)

Overview:

MemeHain is a basic ERC-20 token contract featuring typical token functionalities such as transfer, approve, and ownership management. The token has a name "MemeHain" with the symbol "HAIN" and a fixed total supply of 3 billion tokens, each divisible up to 18 decimal places.

Key Features:

- **Tokenomics**:
  - **Total Supply**: 3 billion tokens, initially allocated to the contract creator.
  - **Decimals**: 18, allowing for fractional transactions typical of most ERC-20 tokens.

- **Ownership**:
  - Single ownership model with privileges to transfer ownership and manage critical functionalities.

- **Standard ERC-20 Functions**:
  - `transfer`, `approve`, and `transferFrom` for token transfers and delegation.

- **Allowance Management**:
  - `increaseAllowance` and `decreaseAllowance` functions to manage spending limits in a flexible manner.

Security Assessment:

1. Code Correctness and Best Practices:

- The code adheres to the ERC-20 standard.
- The use of explicit state visibility and checks like `require` statements for validations improves readability and security.
- Proper event emissions on state-changing functions provide transparency and traceability of transactions.

2. Access Control:

- The `onlyOwner` modifier restricts sensitive functions to the contract owner, minimizing unauthorized access risks.
- Ownership transfer capability includes safeguards against transferring ownership to the zero address.

3. Potential Risks and Vulnerabilities:

- **Centralization Risk**: The contract's control is centralized in the hands of a single owner, which could lead to risks if the owner's private key is compromised.
- **Lack of Burn Mechanism**: The contract does not support token burning, which could be a limitation for certain economic models aiming for deflationary tokenomics.

4. Gas Optimization:

- Unchecked blocks are used for arithmetic operations post-validation to avoid redundant gas expenditure. This follows the ERC-20 recommendation for safe mathematical operations post-conditions checks.

5. Recommendations for Enhancements:

- **Burn Functionality**: Introducing a token burn function could provide dynamic supply adjustments, enhancing token economic models.
- **Multi-Ownership**: Implementing a multi-signature scheme for ownership could decentralize control, reducing the risks associated with a single owner.
- **Pause Functionality**: Inclusion of pausing mechanisms could be beneficial to halt token transfers in case of detected vulnerabilities or attacks.
- **Automated Testing and Audits**: Ensuring comprehensive unit and integration tests along with regular third-party audits will help maintain security standards and trust.

Conclusion:

The MemeHain smart contract implements the standard functionalities of an ERC-20 token along with basic ownership management. While the implementation is robust for general use, the concentrated control in a single owner and the lack of advanced economic features like burning might limit the adaptability of the token in diverse scenarios. The contract is well-suited for applications requiring a straightforward token implementation with centralized governance. Future iterations should consider integrating additional security and economic features to broaden the utility and security profile of the token.

Smart Contract Audit Report: Presale Contract

Overview:

The Presale contract is designed for conducting token presales, allowing participants to purchase tokens using ETH, USDT, and USDC. The contract interfaces with ERC20 tokens and utilizes an external price feed to convert ETH contributions to USD equivalent values.

Key Features:

- **Token Sale Configuration**:
    - Configurable stages with different token prices per USD.
    - External price feed integration to determine current ETH to USD rates.
    - Provision for token sale in different currencies (ETH, USDT, USDC).
- **Ownership Management**:
    - Ownable contract pattern allowing for owner-specific administrative actions.
- **Token Distribution**:
    - Participants can purchase tokens during the active presale phase.
    - Participants can claim their purchased tokens once the presale ends.

Security Assessment:

1. Security Features and Best Practices:

- **Reentrancy Guard**: The use of non-reentrant modifiers in functions handling external calls or transfers ensures protection against reentrancy attacks.
- **Valid Address Checks**: Functions that handle token transfers or ownership changes include checks to ensure that zero addresses are not allowed.
- **Ownership Restrictions**: Sensitive functions are restricted to the contract owner, mitigating unauthorized access risks.

2. Potential Vulnerabilities and Risks:

- **Centralized Control**: The owner has extensive control over the contract, including ending the presale and updating token prices, which could pose risks if the owner's address is compromised.

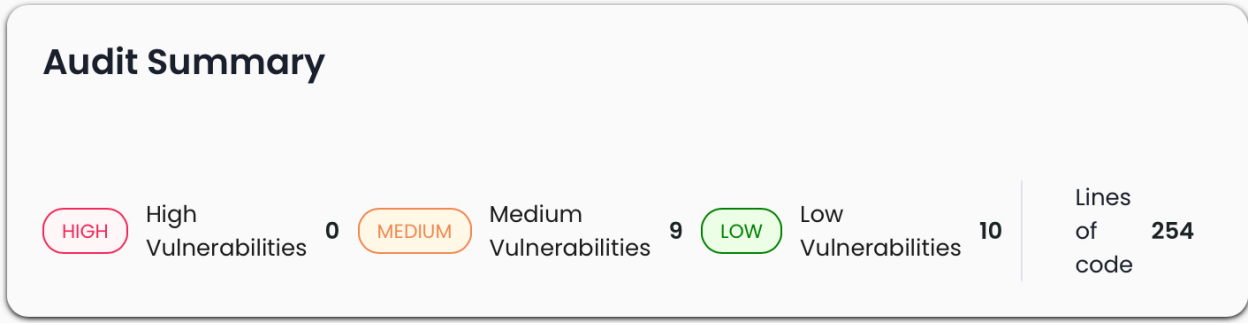
- **Price Feed Dependence**: The contract's functionality heavily depends on the external price feed for ETH to USD conversion. Any manipulation or errors in the price feed can impact the contract's operations and token pricing.

3. Recommendations for Mitigation and Enhancements:

- **Decentralize Ownership**: Implement a multi-signature scheme or a decentralized governance model for critical administrative actions to reduce the risks associated with single-owner control.
- **Audit External Dependencies**: Regular audits and monitoring of the external price feed are recommended to ensure its reliability and integrity.
- **Implement Pause Mechanism**: Introduce a pausing mechanism controlled by the owner or governance body to halt presale operations in case of detected anomalies or attacks.
- **Upgradeability**: Consider adopting a proxy pattern for the contract to facilitate future upgrades and enhancements without losing state or requiring migration.

Conclusion:

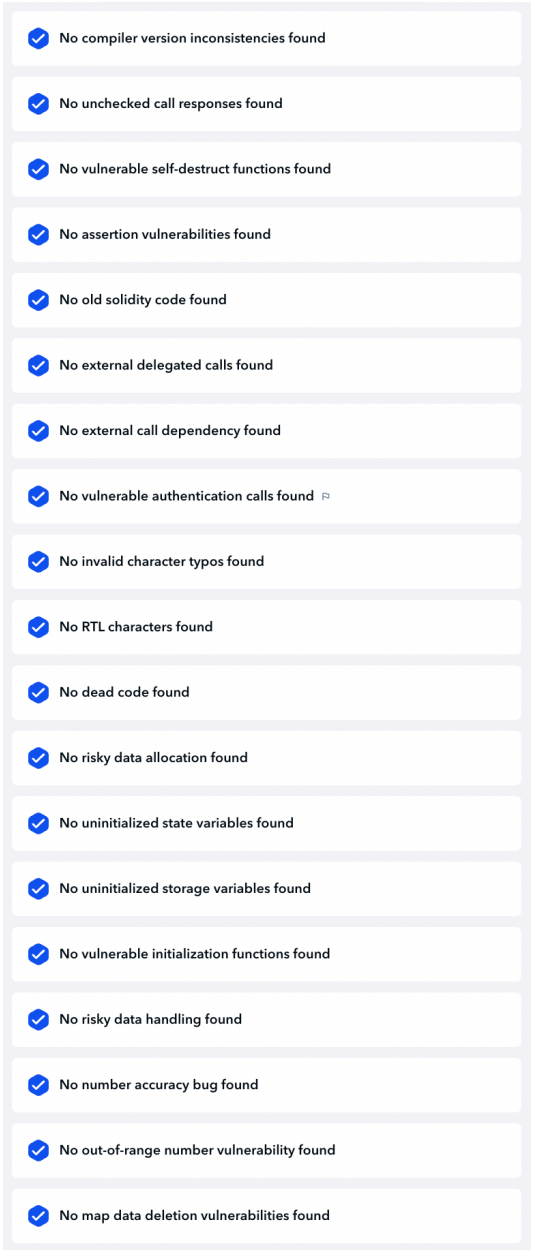
The Presale contract provides a robust framework for conducting token presales with multiple payment options. While the contract implements several security measures, the centralization of control and reliance on external data sources are areas that could be improved. Adopting additional decentralized control mechanisms and ensuring the reliability of external dependencies are crucial steps towards enhancing the contract’s security posture and operational resilience. Regular audits and adherence to best security practices are recommended to maintain and improve the contract's security over time.



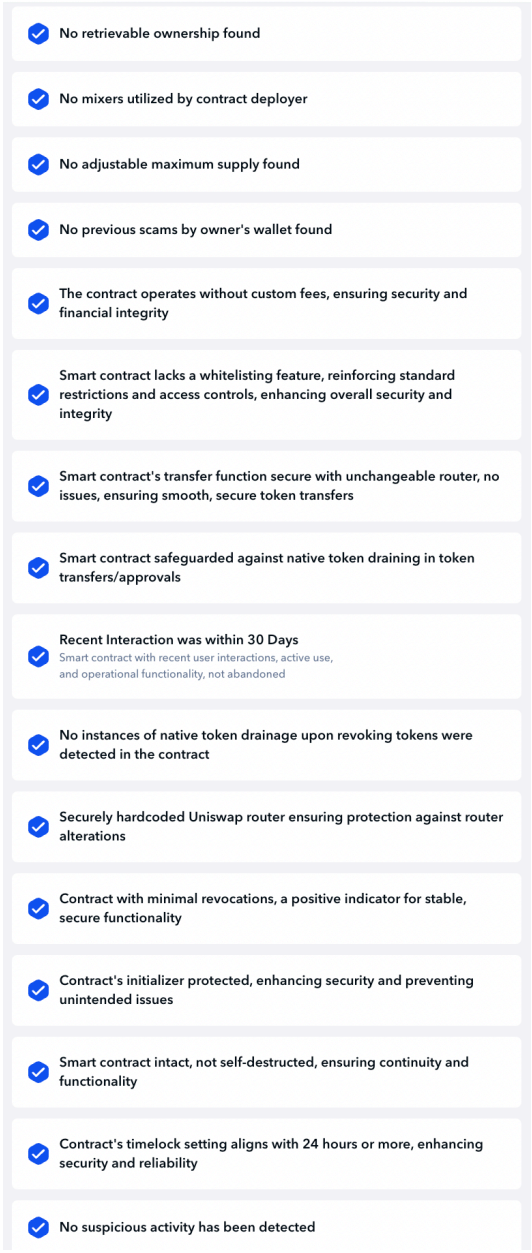
NO HIGH RISK ISSUES IDENTIFIED

- No tautologies or contradictions found
- No faulty true/false values found
- No innacurate divisions found
- No redundant constructor calls found
- No vulnerable transfers found
- No vulnerable return values found
- No uninitialized local variables found
- No default function responses found
- No missing arithmetic events found
- No missing access control events found
- No missing zero address checks found
- No redundant true/false comparisons found
- No state variables vulnerable through function calls found
- No buggy low-level calls found
- No invalid solidity versions found
- No expensive loops found
- No bad numeric notation practices found
- No missing constant declarations found
- No missing external function declarations found

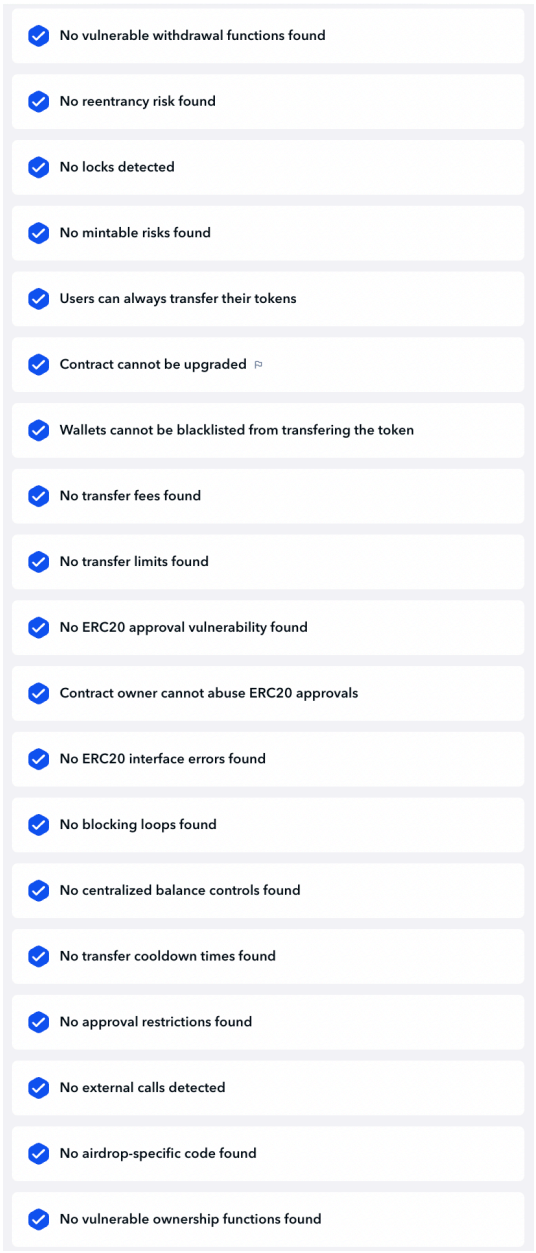
```
/**MAIN CONTRACT

 *Submitted for verification at Etherscan.io on 2024-11-29
*/

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract MemeHain {
    string public name = "MemeHain";
    string public symbol = "HAIN";
    uint8 public decimals = 18;
    uint256 public totalSupply = 3000000000 * 10**uint256(decimals);

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    address public owner;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
    event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

    constructor() {
        owner = msg.sender;
        balanceOf[msg.sender] = totalSupply; // Assign total supply to contract
creator
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can perform this action");
        _;
    }

    modifier validAddress(address _address) {
        require(_address != address(0), "Invalid address");
        _;
}
```




```
  function transfer(address _to, uint256 _value) public validAddress(_to) returns
(bool success) {
      require(balanceOf[msg.sender] >= _value, "Insufficient balance");

      unchecked {
        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;
      }

      emit Transfer(msg.sender, _to, _value);
      return true;
    }

    function approve(address _spender, uint256 _value) public
validAddress(_spender) returns (bool success) {
      allowance[msg.sender][_spender] = _value;
      emit Approval(msg.sender, _spender, _value);
      return true;
    }

    function transferFrom(address _from, address _to, uint256 _value) public
validAddress(_from) validAddress(_to) returns (bool success) {
      require(balanceOf[_from] >= _value, "Insufficient balance");
      require(allowance[_from][msg.sender] >= _value, "Allowance exceeded");

      unchecked {
        balanceOf[_from] -= _value;
        balanceOf[_to] += _value;
        allowance[_from][msg.sender] -= _value;
      }

      emit Transfer(_from, _to, _value);
      return true;
    }

    function increaseAllowance(address _spender, uint256 _addedValue) public
validAddress(_spender) returns (bool success) {
      allowance[msg.sender][_spender] += _addedValue;
     emit Approval(msg.sender, _spender, allowance[msg.sender][_spender]);
```




```
        return true;
    }

    function decreaseAllowance(address _spender, uint256 _subtractedValue)
public validAddress(_spender) returns (bool success) {
        uint256 currentAllowance = allowance[msg.sender][_spender];
        require(currentAllowance >= _subtractedValue, "ERC20: decreased
allowance below zero");

        unchecked {
            allowance[msg.sender][_spender] = currentAllowance -
_subtractedValue;
        }

        emit Approval(msg.sender, _spender, allowance[msg.sender][_spender]);
        return true;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0), "New owner is the zero address");
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
}
/**
 *Submitted for verification at Etherscan.io on 2024-11-29
*/

/**
 *Submitted for verification at Etherscan.io on 2024-11-29
*/PRESALE CONTRACT

// SPDX-License-Identifier: MIT Licensed
pragma solidity ^0.8.17;

abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }
```




```
    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}

contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

    constructor() {
        _transferOwnership(_msgSender());
    }

    function owner() public view virtual returns (address) {
        return _owner;
    }

    modifier onlyOwner() {
        require(owner() == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }

    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        _transferOwnership(newOwner);
    }

    function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
```

```
}

interface IERC20 {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address from, address to, uint256 amount) external returns (bool);
    function decimals() external view returns (uint8);
}

interface AggregatorV3Interface {
    function latestRoundData() external view returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80
answeredInRound);
}

contract Presale is Ownable {
    IERC20 public mainToken;
    IERC20 public USDT = IERC20(0xdAC17F958D2ee523a2206206994597C13D831ec7);
    IERC20 public USDC = IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);

    uint256 public tokensToSell;
    uint256[] public tokenPerUsdPrice;
    uint256 public totalStages;
    uint8 public tokenDecimals;

    AggregatorV3Interface public priceFeed;

    struct Phase {
        uint256 tokenPerUsdPrice;
    }

    uint256 public currentStage;
    uint256 public soldToken;
    uint256 public amountRaised;
    uint256 public amountRaisedUSDT;
    uint256 public amountRaisedUSDC;
    uint256 public totalRaised;
    uint256 public uniqueBuyers;
    address payable public fundReceiver;

    bool public isPresaleEnded;

    mapping(address => User) public users;
    mapping(uint256 => Phase) public phases;
    mapping(address => bool) public isExist;
```

```
struct User {
    uint256 native_balance;
    uint256 usdt_balance;
    uint256 usdc_balance;
    uint256 claimedAmount;
    uint256 claimAbleAmount;
    uint256 purchasedToken;
}

event BuyToken(address indexed _user, uint256 indexed _amount);
event ClaimToken(address _user, uint256 indexed _amount);
event UpdatePrice(uint256 _oldPrice, uint256 _newPrice);

constructor(IERC20 _token, address _fundReceiver) {
    mainToken = _token;
    fundReceiver = payable(_fundReceiver);
    priceFeed = AggregatorV3Interface(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419);
    tokenDecimals = mainToken.decimals();
    tokensToSell = 1_000_000_000 * 10**tokenDecimals;

    tokenPerUsdPrice = [
        100 * 10**tokenDecimals,    // Phase 1: 0.001 $
        250 * 10**tokenDecimals,    // Phase 2: 0.0025 $
        500 * 10**tokenDecimals,    // Phase 3: 0.005 $
        750 * 10**tokenDecimals,    // Phase 4: 0.0075 $
        1000 * 10**tokenDecimals,   // Phase 5: 0.01 $
        1250 * 10**tokenDecimals,   // Phase 6: 0.0125 $
        1500 * 10**tokenDecimals,   // Phase 7: 0.015 $
        1750 * 10**tokenDecimals,   // Phase 8: 0.0175 $
        2000 * 10**tokenDecimals,   // Phase 9: 0.02 $
        2250 * 10**tokenDecimals,   // Phase 10: 0.0225 $
        2500 * 10**tokenDecimals,   // Phase 11: 0.025 $
        2750 * 10**tokenDecimals    // Phase 12: 0.0275 $
    ];

    totalStages = tokenPerUsdPrice.length;

    for (uint256 i = 0; i < totalStages; i++) {
        phases[i].tokenPerUsdPrice = tokenPerUsdPrice[i];
    }
}

function updatePresale(uint256 _phaseId, uint256 _tokenPerUsdPrice) public onlyOwner {
    uint256 oldPrice = phases[_phaseId].tokenPerUsdPrice;
    phases[_phaseId].tokenPerUsdPrice = _tokenPerUsdPrice;
    emit UpdatePrice(oldPrice, _tokenPerUsdPrice);
}

function getLatestPrice() public view returns (uint256) {
```

```
    (, int256 price, , , ) = priceFeed.latestRoundData();
    return uint256(price * 1e10); // Adjust decimal precision for price feed
}

function buyToken() public payable {
    require(!isPresaleEnded, "Presale ended!");
    require(msg.value > 0, "Send ETH to buy tokens");

    if (!isExist[msg.sender]) {
        isExist[msg.sender] = true;
        uniqueBuyers++;
    }

    uint256 usdAmount = (msg.value * getLatestPrice()) / 1e18; // Convert ETH to USD
    uint256 numberOfTokens = nativeToToken(msg.value, currentStage);

    require(soldToken + numberOfTokens <= tokensToSell, "Phase Limit Reached");
    soldToken += numberOfTokens;
    amountRaised += msg.value;
    totalRaised += usdAmount;

    users[msg.sender].native_balance += msg.value;
    users[msg.sender].claimAbleAmount += numberOfTokens;
    users[msg.sender].purchasedToken += numberOfTokens;

    fundReceiver.transfer(msg.value);

    emit BuyToken(msg.sender, numberOfTokens);
}

function buyTokenUSDT(uint256 amount) public {
    require(!isPresaleEnded, "Presale ended!");

    if (!isExist[msg.sender]) {
        isExist[msg.sender] = true;
        uniqueBuyers++;
    }

    uint256 numberOfTokens = usdtToToken(amount, currentStage);
    require(soldToken + numberOfTokens <= tokensToSell, "Phase Limit Reached");
    soldToken += numberOfTokens;
    amountRaisedUSDT += amount;

    users[msg.sender].usdt_balance += amount;
    users[msg.sender].claimAbleAmount += numberOfTokens;
    users[msg.sender].purchasedToken += numberOfTokens;

    USDT.transferFrom(msg.sender, fundReceiver, amount);

    emit BuyToken(msg.sender, numberOfTokens);
```

```
    }

    function buyTokenUSDC(uint256 amount) public {
        require(!isPresaleEnded, "Presale ended!");

        if (!isExist[msg.sender]) {
            isExist[msg.sender] = true;
            uniqueBuyers++;
        }

        uint256 numberOfTokens = usdcToToken(amount, currentStage);
        require(soldToken + numberOfTokens <= tokensToSell, "Phase Limit Reached");
        soldToken += numberOfTokens;
        amountRaisedUSDC += amount;

        users[msg.sender].usdc_balance += amount;
        users[msg.sender].claimAbleAmount += numberOfTokens;
        users[msg.sender].purchasedToken += numberOfTokens;

        USDC.transferFrom(msg.sender, fundReceiver, amount);

        emit BuyToken(msg.sender, numberOfTokens);
    }

    function claimTokens() public {
        require(isPresaleEnded, "Presale is not ended yet");
        require(users[msg.sender].claimAbleAmount > 0, "No claimable tokens");

        uint256 amount = users[msg.sender].claimAbleAmount;
        users[msg.sender].claimedAmount += amount;
        users[msg.sender].claimAbleAmount = 0;

        mainToken.transfer(msg.sender, amount);

        emit ClaimToken(msg.sender, amount);
    }

    function endPresale() public onlyOwner {
        isPresaleEnded = true;
    }

    function getUserInfo(address user) public view returns (uint256 nativeBalance, uint256 usdtBalance, uint256 usdcBalance, uint256
claimAbleAmount, uint256 purchasedToken) {
        return (users[user].native_balance, users[user].usdt_balance, users[user].usdc_balance, users[user].claimAbleAmount,
users[user].purchasedToken);
    }

    // Conversion Functions
    function nativeToToken(uint256 amount, uint256 stage) public view returns (uint256) {
        return (amount * phases[stage].tokenPerUsdPrice) / 1e6;
```

```
    }

    function usdtToToken(uint256 amount, uint256 stage) public view returns (uint256) {
        return (amount * phases[stage].tokenPerUsdPrice) / 1e6;
    }

    function usdcToToken(uint256 amount, uint256 stage) public view returns (uint256) {
        return (amount * phases[stage].tokenPerUsdPrice) / 1e6;
    }
}
```

Terrence Nibbles, CCE, CCA
Auditor #17865