Readme Document

Vesting Contract with Role-Based Access Control

Table of Contents

Overview

This smart contract implements a sophisticated token vesting system with role-based access control. It allows for different vesting schedules based on three distinct roles: User, Partner, and Team. The contract is designed to work with any ERC20 token, providing flexibility for various token distribution scenarios.

Features

- Role-based vesting schedules (User, Partner, Team)
- Customizable cliff and vesting durations for each role
- Secure beneficiary management system
- Token claiming functionality with built-in vesting calculations
- Real-time vested amount queries
- Comprehensive event logging for transparency
- Owner-controlled vesting initiation
- SafeMath integration for arithmetic safety

Technical Specifications

- Solidity Version: ^0.8.0
- OpenZeppelin Dependencies:
  - IERC20.sol
  - Ownable.sol
  - SafeMath.sol
- Compiler Optimizations: Recommended
- Estimated Gas Usage: Varies by function (detailed breakdown recommended)

Contract Structure

The contract consists of:

- State variables for storing the token, vesting start time, and vesting status
- An enum `Role` defining User, Partner, and Team roles
- A struct `VestingSchedule` containing vesting parameters for each beneficiary
- Mappings to store vesting schedules and role allocations
- Functions for vesting management, token claiming, and vested amount calculations

Roles and Vesting Schedules

1. User:
   - Allocation: 50% of total tokens
   - Cliff: 10 months (300 days)
   - Vesting Duration: 2 years (730 days)

2. Partner:
   - Allocation: 25% of total tokens
   - Cliff: 2 months (60 days)
   - Vesting Duration: 1 year (365 days)

3. Team:
   - Allocation: 25% of total tokens
   - Cliff: 2 months (60 days)
   - Vesting Duration: 1 year (365 days)

Main Functions

1. `constructor(IERC20 _token, address initialOwner)`

   - Initializes the contract with the ERC20 token address and sets the initial owner.

2. `startVesting()`

   - Begins the vesting period. Can only be called by the owner.

3. `addBeneficiary(address _beneficiary, Role _role, uint256 _allocation)`

   - Adds a beneficiary with a specific role and token allocation. Can only be called by the owner before vesting starts.

4. `claimTokens(Role _role)`

   - Allows beneficiaries to claim their vested tokens. Calculates and transfers the claimable amount.

5. `getVestedAmount(address _beneficiary, Role _role)`

   - Returns the vested amount for a specific beneficiary and role.

6. `getTotalVestedAmount(address _beneficiary)`

   - Returns the total vested amount across all roles for a beneficiary.

7. `calculateVestedAmount(VestingSchedule memory _schedule)`

   - Internal function to calculate the vested amount based on the current timestamp.

8. `getCliffDuration(Role _role)` and `getVestingDuration(Role _role)`

   - Internal functions to get cliff and vesting durations for each role.

Events

1. `VestingStarted(uint256 startTime)`

   - Emitted when the vesting period starts.

2. `BeneficiaryAdded(address beneficiary, Role role, uint256 allocation)`

   - Emitted when a new beneficiary is added.

3. `TokensClaimed(address beneficiary, Role role, uint256 amount)`

   - Emitted when tokens are claimed by a beneficiary.

Usage Guide

1. Deployment:

   - Deploy the contract by providing the ERC20 token address and the initial owner address.

2. Beneficiary Management:

   - As the owner, use `addBeneficiary` to add beneficiaries before starting the vesting period.

3. Start Vesting:

   - Call `startVesting` to initiate the vesting process.

4. Token Claiming:

   - Beneficiaries can use `claimTokens` to claim their vested tokens after their cliff period.

5. Vesting Queries:

   - Use `getVestedAmount` or `getTotalVestedAmount` to check vesting progress.

Security Considerations

- Owner privileges are restricted to critical functions.
- SafeMath is used to prevent arithmetic overflow and underflow.
- Checks are in place to prevent double allocation and premature claiming.
- Ensure sufficient token balance in the contract before starting vesting.

Development and Testing

1. Environment Setup:

   ```
   npm init -y
   npm install @openzeppelin/contracts
   npm install --save-dev truffle
   ```

2. Compile:

```
truffle compile
```

3. Testing:

   - Write comprehensive tests covering all functions and edge cases.
   - Use Truffle's time manipulation functions to test different vesting scenarios.

   ```
   truffle test
   ```

Deployment

1. Set up a `.env` file with your network details and private key.
2. Configure `truffle-config.js` with your network settings.
3. Deploy:

   ```
   truffle migrate --network <your_network>
   ```

License

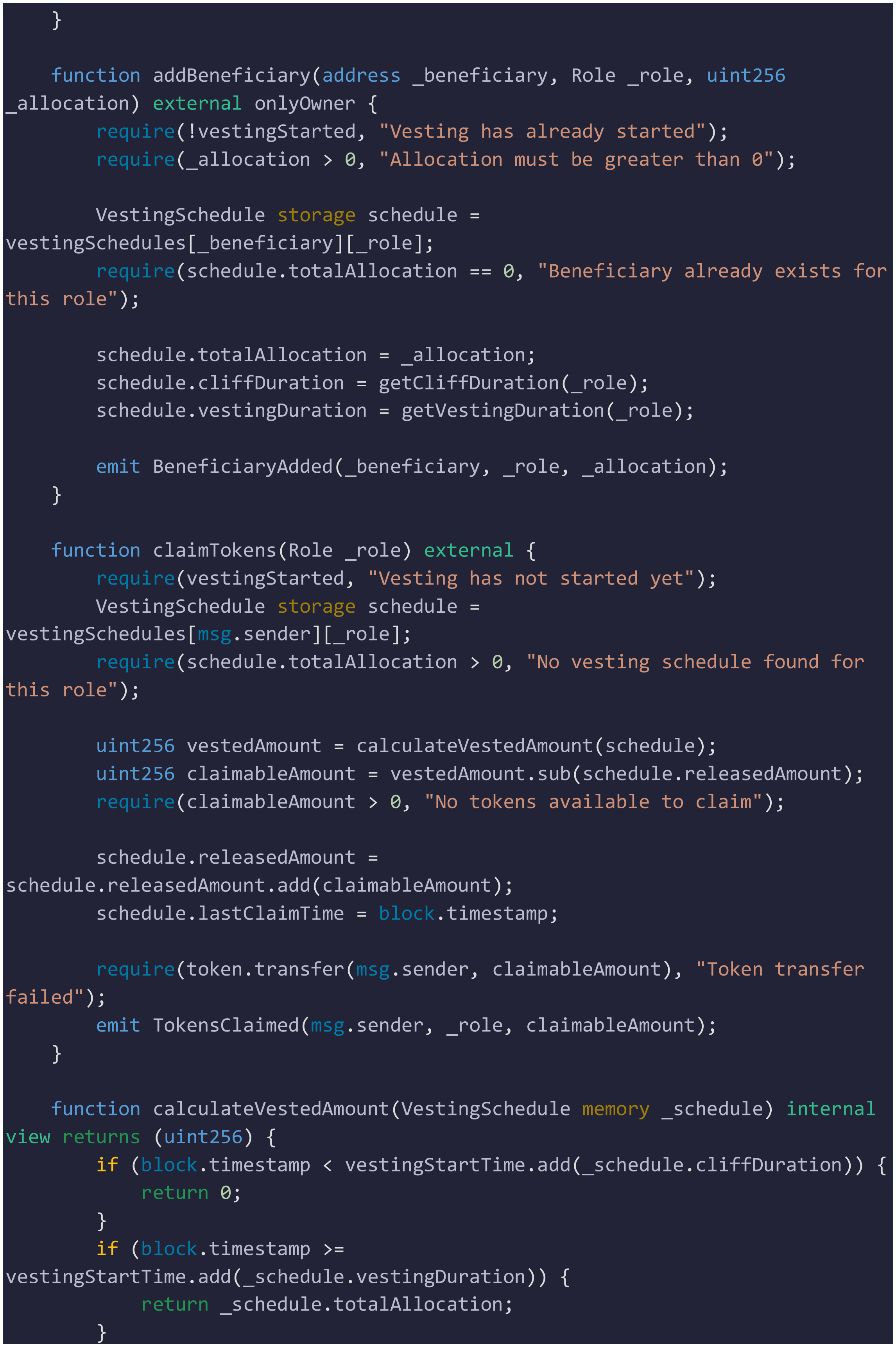
This project is licensed under the MIT License. See the LICENSE file for details.

Disclaimer

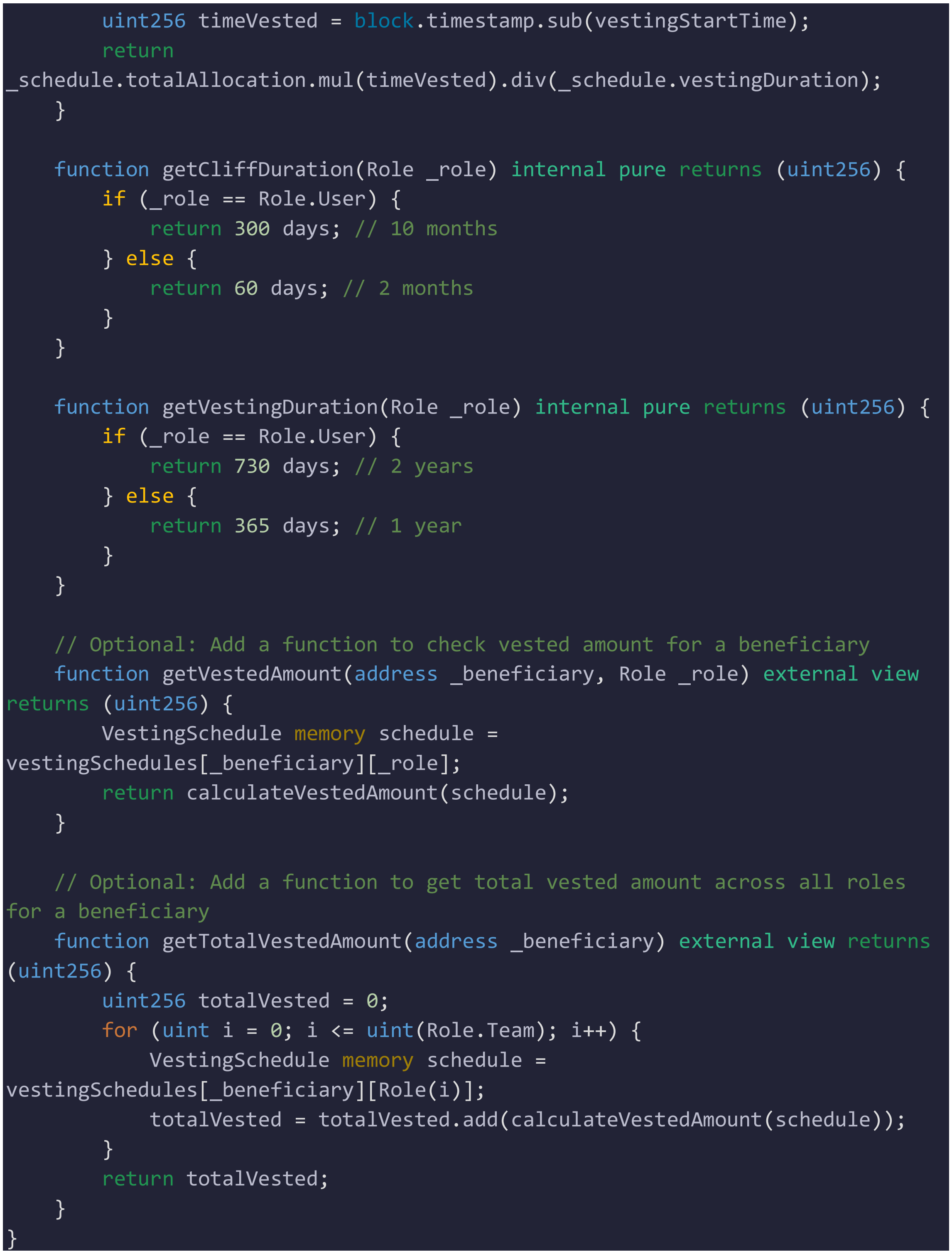
This smart contract is provided as-is. While efforts have been made to ensure its correctness and security, a thorough audit and testing are strongly recommended before any live deployment. The authors and contributors are not liable for any losses or damages arising from the use of this contract.

Smart Contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract VestingContract is Ownable {
    using SafeMath for uint256;

    IERC20 public token;
    uint256 public vestingStartTime;
    bool public vestingStarted;

    enum Role { User, Partner, Team }

    struct VestingSchedule {
        uint256 totalAllocation;
        uint256 cliffDuration;
        uint256 vestingDuration;
        uint256 releasedAmount;
        uint256 lastClaimTime;
    }

    mapping(address => mapping(Role => VestingSchedule)) public
vestingSchedules;
    mapping(Role => uint256) public roleTotalAllocation;

    event VestingStarted(uint256 startTime);
    event BeneficiaryAdded(address beneficiary, Role role, uint256
allocation);
    event TokensClaimed(address beneficiary, Role role, uint256 amount);

    constructor(IERC20 _token, address initialOwner) Ownable(initialOwner) {
        token = _token;
        roleTotalAllocation[Role.User] = 50;
        roleTotalAllocation[Role.Partner] = 25;
        roleTotalAllocation[Role.Team] = 25;
    }

    function startVesting() external onlyOwner {
        require(!vestingStarted, "Vesting has already started");
        vestingStartTime = block.timestamp;
        vestingStarted = true;
        emit VestingStarted(vestingStartTime);
```



```
    }

    function addBeneficiary(address _beneficiary, Role _role, uint256
_allocation) external onlyOwner {
        require(!vestingStarted, "Vesting has already started");
        require(_allocation > 0, "Allocation must be greater than 0");

        VestingSchedule storage schedule =
vestingSchedules[_beneficiary][_role];
        require(schedule.totalAllocation == 0, "Beneficiary already exists for
this role");

        schedule.totalAllocation = _allocation;
        schedule.cliffDuration = getCliffDuration(_role);
        schedule.vestingDuration = getVestingDuration(_role);

        emit BeneficiaryAdded(_beneficiary, _role, _allocation);
    }

    function claimTokens(Role _role) external {
        require(vestingStarted, "Vesting has not started yet");
        VestingSchedule storage schedule =
vestingSchedules[msg.sender][_role];
        require(schedule.totalAllocation > 0, "No vesting schedule found for
this role");

        uint256 vestedAmount = calculateVestedAmount(schedule);
        uint256 claimableAmount = vestedAmount.sub(schedule.releasedAmount);
        require(claimableAmount > 0, "No tokens available to claim");

        schedule.releasedAmount =
schedule.releasedAmount.add(claimableAmount);
        schedule.lastClaimTime = block.timestamp;

        require(token.transfer(msg.sender, claimableAmount), "Token transfer
failed");
        emit TokensClaimed(msg.sender, _role, claimableAmount);
    }

    function calculateVestedAmount(VestingSchedule memory _schedule) internal
view returns (uint256) {
        if (block.timestamp < vestingStartTime.add(_schedule.cliffDuration)) {
            return 0;
        }
        if (block.timestamp >=
vestingStartTime.add(_schedule.vestingDuration)) {
            return _schedule.totalAllocation;
        }
```



```
        uint256 timeVested = block.timestamp.sub(vestingStartTime);
        return
_schedule.totalAllocation.mul(timeVested).div(_schedule.vestingDuration);
    }

    function getCliffDuration(Role _role) internal pure returns (uint256) {
        if (_role == Role.User) {
            return 300 days; // 10 months
        } else {
            return 60 days; // 2 months
        }
    }

    function getVestingDuration(Role _role) internal pure returns (uint256) {
        if (_role == Role.User) {
            return 730 days; // 2 years
        } else {
            return 365 days; // 1 year
        }
    }

    // Optional: Add a function to check vested amount for a beneficiary
    function getVestedAmount(address _beneficiary, Role _role) external view
returns (uint256) {
        VestingSchedule memory schedule =
vestingSchedules[_beneficiary][_role];
        return calculateVestedAmount(schedule);
    }

    // Optional: Add a function to get total vested amount across all roles
for a beneficiary
    function getTotalVestedAmount(address _beneficiary) external view returns
(uint256) {
        uint256 totalVested = 0;
        for (uint i = 0; i <= uint(Role.Team); i++) {
            VestingSchedule memory schedule =
vestingSchedules[_beneficiary][Role(i)];
            totalVested = totalVested.add(calculateVestedAmount(schedule));
        }
        return totalVested;
    }
}
```