Audit Report

# MyVolt

March 2024

| | |
|---|---|
| Repository | https://github.com/MyVoltEnergy/MyVolt-Solidity- |
| Commit | 752c8a6206192fd76faaa5d52705070e723f426a |
| Audited by | © cyberscope |

# Table of Contents

## Review

| | |
|---|---|
| **Repository** | https://github.com/MyVoltEnergy/MyVolt-Solidity- |
| **Commit** | 752c8a6206192fd76faaa5d52705070e723f426a |
| **Testing Deploy** | https://testnet.bscscan.com/address/0x178fcfbb57790b441c646535cff995c1afd7f04c |

## Audit Updates

| | |
|---|---|
| **Initial Audit** | 15 Mar 2024 |

## Source Files

| **Filename** | SHA256 |
|---|---|
| **contracts/MyVoltVesting.sol** | 8e8a7a7c8da0b6bf517f6e786cabc53bb77d6bf1f14b280d4b5342b20d4fbe14 |

# Overview

The `MyVoltVesting` contract is designed to manage a vesting schedule for tokens, allowing specific addresses (beneficiaries) to claim tokens over time based on predefined criteria. The core functionality revolves around creating vesting schedules for beneficiaries, determining the amount of tokens available for the claim at any given time based on the passage of time and predefined `cliffs`, and allowing beneficiaries to claim their vested tokens.

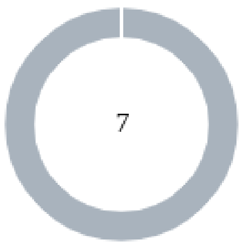
## Owner Functionality

The contract owner has the ability to add beneficiaries to the vesting schedule, withdraw tokens and native currency from the contract, and set or update the token contract address used for the vesting process. The addition of beneficiaries to the vesting schedule is a critical function, allowing the owner to specify the amount of tokens allocated per cliff and the timing of these cliffs. The owner also has the flexibility to withdraw funds from the contract, ensuring control over the contract's assets and the ability to manage the token supply effectively.

## User Functionality

The users who are set as beneficiaries of the vesting schedule have the ability to claim vested tokens. The contract calculates the amount of tokens that a beneficiary is eligible to claim based on the current time and their specific vesting schedule. This calculation takes into account how many cliffs have passed and the amount of tokens per cliff, ensuring that beneficiaries can only claim tokens that have vested according to the predetermined schedule. This mechanism ensures a fair and transparent distribution of tokens, aligning with the intended vesting strategy.

## Findings Breakdown



| | | |
|---|---|---|
| ● | Critical | 0 |
| ● | Medium | 0 |
| ● | Minor / Informative | 7 |

| Severity | | Unresolved | Acknowledged | Resolved | Other |
|---|---|---|---|---|---|
| ● | Critical | 0 | 0 | 0 | 0 |
| ● | Medium | 0 | 0 | 0 | 0 |
| ● | Minor / Informative | 7 | 0 | 0 | 0 |

## Diagnostics

● Critical ● Medium ● Minor / Informative

| Severity | Code | Description | Status |
|---|---|---|---|
| ● | CCR | Contract Centralization Risk | Unresolved |
| ● | MEM | Misleading Error Messages | Unresolved |
| ● | MEE | Missing Events Emission | Unresolved |
| ● | MU | Modifiers Usage | Unresolved |
| ● | TAV | Token Address Validation | Unresolved |
| ● | TSI | Tokens Sufficiency Insurance | Unresolved |
| ● | UCO | Unverified Cliff Ordering | Unresolved |

## CCR - Contract Centralization Risk

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/MyVoltVesting.sol#L113,177,182,189 |
| **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.

Specifically, the contract grants the owner the capability to modify the token contract utilized for the vesting process, withdraw tokens and native tokens from the contract, and include specific participants in the vesting mechanism.

```
   function addVestingSchedule(
        address[] memory receivers,
        uint256[] memory tokens,
        uint256[] memory cliffs
    ) external onlyOwner {
        ...
            vestingSchedules[receivers[i]].tokensPerCliff = tokens;
            vestingSchedules[receivers[i]].cliffs = cliffs;
        }
    }

    function setTokenContract(address newContract) external onlyOwner
{
        require(newContract != address(0), "Invalid Address!");
        tokenContract = newContract;
    }

    function withdraw(address recipient, uint256 amount) external
onlyOwner {
        require(recipient != address(0), "Invalid Address!");

        (bool sent, ) = recipient.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }

    function withdraw(
        address recipient,
        uint256 amount,
        address token
    ) external onlyOwner {
        ...

        require(
            IERC20(token).transfer(recipient, amount),
            "Unsuccessful Transfer!"
        );
    }
```

## 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.

## MEM - Misleading Error Messages

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/MyVoltVesting.sol#L |
| **Status** | Unresolved |

### Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

Specifically, the contract includes a `require` check designed to validate whether `tokensPerCliff.length` equals 0, and if `lastCliffClaimed` matches the total number of `cliffs` (cliffs.length). This check is crucial for determining the initiation of a vesting schedule for a receiver. However, the contract reverts with the same error message, `"Vesting Schedule already active!"` , for both conditions. This implementation can lead to confusion, as a `tokensPerCliff.length` of zero indicates that a vesting schedule does not exist for the receiver, contrary to what the error message suggests. The current error message is misleading, especially in scenarios where the vesting schedule has not been set up, leading to incorrectly error that a schedule is already active.

```
require(
    vestingSchedules[receivers[i]].tokensPerCliff.length == 0
||
        vestingSchedules[receivers[i]].lastCliffClaimed ==
        vestingSchedules[receivers[i]].cliffs.length,
    "Vesting Schedule already active!"
);
```

### Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract. Specifically, it is recommended to refactor the error messages to accurately reflect the specific condition that triggers the revert. Separate error messages should be used for each condition to provide clear and actionable feedback.Implementing distinct error messages for these scenarios will enhance clarity and user understanding of the contract's operations.

## MEE - Missing Events Emission

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/MyVoltVesting.sol#L176 |
| **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 setTokenContract(address newContract) external
onlyOwner {
    require(newContract != address(0), "Invalid Address!");
    tokenContract = newContract;
}
```

### 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.

## MU - Modifiers Usage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/MyVoltVesting.sol#L178,183,194 |
| **Status** | Unresolved |

### Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(newContract != address(0), "Invalid Address!");
require(recipient != address(0), "Invalid Address!");
require(recipient != address(0), "Invalid Address!");
```

### Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

## TAV - Token Address Validation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/MyVoltVesting.sol#L181 |
| **Status** | Unresolved |

### Description

The contract provides the owner with the capability to modify the `tokenContract` address through the `setTokenContract` function. This feature allows for the substitution of the token used in the vesting process. Consequently, such changes necessitate careful consideration of the token's decimal places and any potential side effects that altering the token address might entail. The flexibility to switch the token contract address introduces a layer of complexity and risk, particularly if the new token's characteristics (such as decimals) differ significantly from the original or if the change impacts the vesting process.

```
    function setTokenContract(address newContract) external
onlyOwner {
        require(newContract != address(0), "Invalid Address!");
        tokenContract = newContract;
    }
```

### Recommendation

It is recommended to implement safeguards and additional checks when changing the `tokenContract` address to ensure compatibility and mitigate any adverse effects. This could include verifying the decimal places of the new token and assessing any other token-specific attributes that could affect the vesting calculations or logic. Additionally, it may be prudent to include a mechanism for handling potential side effects or discrepancies that could arise from changing the token address. Ensuring thorough validation and compatibility checks will help maintain the integrity and functionality of the vesting process, even when the underlying token contract is updated.

## TSI - Tokens Sufficiency Insurance

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/MyVoltVesting.sol#L115 |
| **Status** | Unresolved |

### Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

Specifically, the contract is designed to manage vesting schedules for its participants, allowing the addition of new vesting schedules through the `addVestingSchedule` function and enabling participants to claim their vested tokens via the `claimVestedTokens` function. However, it lacks a verification step to ensure that the necessary tokens are available within the contract before adding new vesting schedules or allowing claims. This omission poses a risk of creating vesting schedules that cannot be fulfilled due to insufficient tokens in the contract or permitting claims when the contract does not hold enough tokens to satisfy these claims. Such scenarios could lead to failed transactions, wasted gas fees, and potentially undermine the trust in the contract's ability to manage and distribute tokens according to the vesting schedules.

```
    function addVestingSchedule(
            address[] memory receivers,
            uint256[] memory tokens,
            uint256[] memory cliffs
        ) external onlyOwner {
            ...

                vestingSchedules[receivers[i]].tokensPerCliff = tokens;
                vestingSchedules[receivers[i]].cliffs = cliffs;
            }
        }

    function setTokenContract(address newContract) external
onlyOwner {
        require(newContract != address(0), "Invalid Address!");
        tokenContract = newContract;
    }
```

## Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the presale tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process. Specifically, it is recommended to implement checks within the `addVestingSchedule` function to verify the availability of tokens in the contract. Before adding new vesting schedules, the contract should ensure that the sum of all tokens allocated to existing and new schedules does not exceed the contract's current token balance. Similarly, before allowing claims, the contract should verify that it has enough tokens to fulfill the claim according to the vesting schedule. These verifications can be achieved by querying the token balance of the contract and comparing it against the total tokens required for all vesting schedules and claims. Implementing these checks will enhance the contract's reliability and trustworthiness by ensuring that all vesting schedules and claims can be successfully fulfilled, aligning with best practices for smart contract development and token distribution management.

## UCO - Unverified Cliff Ordering

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/MyVoltVesting.sol#L113,141 |
| **Status** | Unresolved |

### Description

The contract is designed to manage token vesting schedules, allowing the addition of new schedules through the `addVestingSchedule` function and determining available tokens for claiming through the `vestedTokensAvailable_` function. The `vestedTokensAvailable_` function checks if the current time has surpassed the timestamps defined for each cliff in a vesting schedule. If it encounters a cliff whose time has not yet passed, it breaks from the loop, implying that all subsequent `cliffs` are also unmet. This logic assumes that the cliff times within a vesting schedule are provided in ascending order during the schedule's creation. However, the contract does not enforce or verify this ordering when `cliffs` are added via the `addVestingSchedule` function. The absence of such a verification could lead to inconsistencies or errors in calculating available tokens if the cliffs are not strictly increasing, potentially affecting the integrity of the vesting process.

```
   function addVestingSchedule(
        address[] memory receivers,
        uint256[] memory tokens,
        uint256[] memory cliffs
    ) external onlyOwner {
        require(tokens.length == cliffs.length, "Array sizes do not
match!");

        for (uint256 i = 0; i < receivers.length; i++) {
            ...
            vestingSchedules[receivers[i]].tokensPerCliff = tokens;
            vestingSchedules[receivers[i]].cliffs = cliffs;
        }
    }

    function vestedTokensAvailable_(
        address beneficiary
    ) internal view returns (uint256, uint256) {
        VestingSchedule memory vestingSchedule_ =
vestingSchedules[beneficiary];
        uint256 availableTokens;
        uint256 lastCliff = vestingSchedule_.cliffs.length;
        for (
            uint256 i = vestingSchedule_.lastCliffClaimed;
            i < lastCliff;
            i++
        ) {
            if (block.timestamp >= vestingSchedule_.cliffs[i]) {
                availableTokens +=
vestingSchedule_.tokensPerCliff[i];
            } else {
                lastCliff = i;
                break;
            }
        }
```

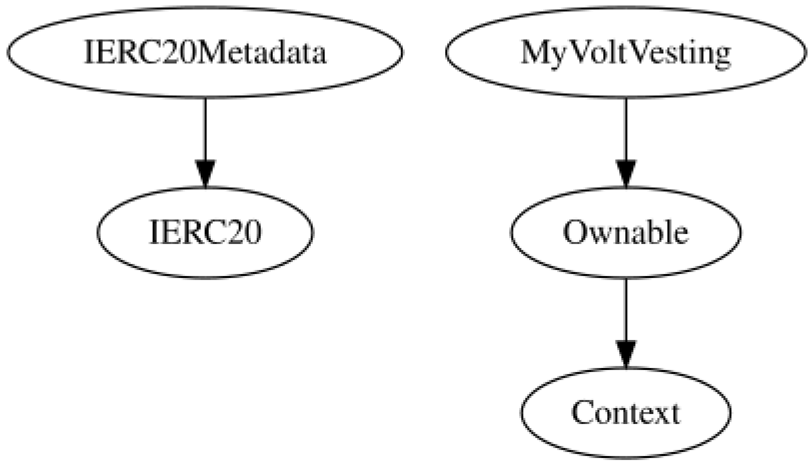
## Recommendation

It is recommended to include additional checks within the `addVestingSchedule` function to verify that the cliff values are provided in an incremental manner. This can be achieved by iterating through the `cliffs` array and ensuring that each subsequent cliff time is greater than the previous one before assigning them to a vesting schedule. Implementing this verification ensures the integrity of the vesting logic, as it relies on the assumption that each cliff occurs strictly after the previous one. By enforcing this requirement, the contract can prevent potential errors in token distribution and maintain the intended functionality and fairness of the vesting process. This addition will enhance the contract's robustness and reliability, aligning with best practices for smart contract development in token vesting scenarios.

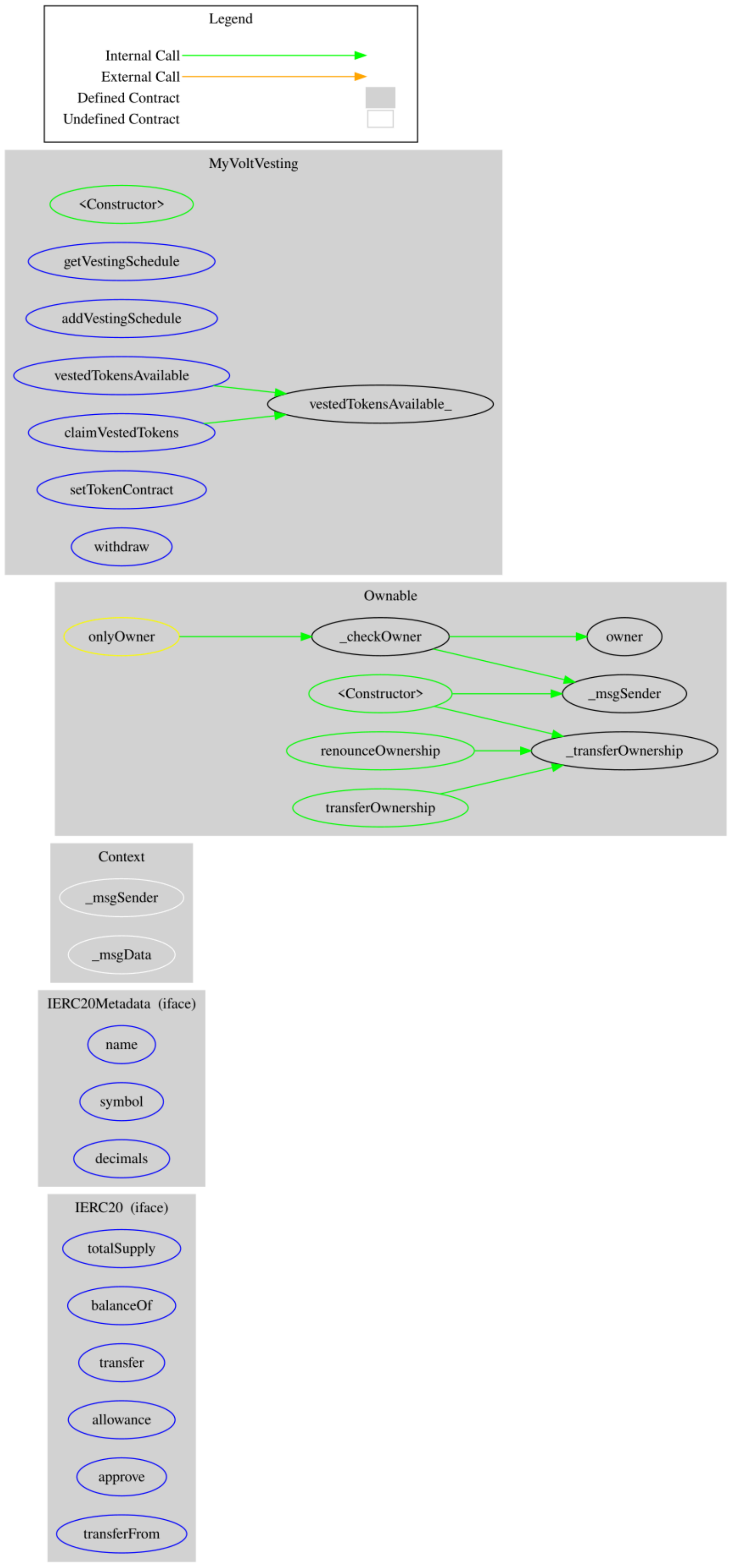
## Functions Analysis

| Contract | Type | Bases | | |
|---|---|---|---|---|
| | Function Name | Visibility | Mutability | Modifiers |
| | | | | |
| MyVoltVesting | Implementation | Ownable | | |
| | | Public | ✓ | - |
| | getVestingSchedule | External | | - |
| | addVestingSchedule | External | ✓ | onlyOwner |
| | vestedTokensAvailable | External | | - |
| | vestedTokensAvailable_ | Internal | | |
| | claimVestedTokens | External | ✓ | - |
| | setTokenContract | External | ✓ | onlyOwner |
| | withdraw | External | ✓ | onlyOwner |
| | withdraw | External | ✓ | onlyOwner |

## Inheritance Graph



## Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
MyVoltVesting
<Constructor>
getVestingSchedule
addVestingSchedule
vestedTokensAvailable
vestedTokensAvailable_
claimVestedTokens
setTokenContract
withdraw
Ownable
onlyOwner
_checkOwner
owner
<Constructor>
_msgSender
renounceOwnership
_transferOwnership
transferOwnership
Context
_msgSender
_msgData
IERC20Metadata (iface)
name
symbol
decimals
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom

## Summary

MyVolt contract implements a vesting mechanism. This audit investigates security issues, business logic concerns and potential improvements.

## 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**

https://www.cyberscope.io