Cyberscope

Audit Report

TLB Vault

March 2024

Network	ETH
Address	0x166b322e25ea3577b310C1c893c754546a018e5F
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://etherscan.io/address/0x166b322e25ea3577b310c1c893c754546a018e5f

Audit Updates

Initial Audit	26 Mar 2024

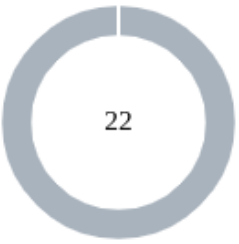
Source Files

Filename	SHA256
Vault.sol	55105aab4a2c3d0c52510a937feb7277f193e3bf9c7f6db4ae211884857950bf

Overview

The Vault contract enables specific users, as determined by the owner, to claim tokens through distinct functions catered to their respective roles, either as team members or partner members. These individuals are granted the ability to claim tokens, with the quantity and schedule of these claims being set by the owner, according to a specific percentage over a defined period. This mechanism ensures a systematic distribution of tokens, allowing eligible participants to receive their allocated shares at predetermined intervals, thereby promoting a methodical and transparent approach to token vesting.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	22

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	22	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	ITV	Inadequate TotalTLB Validation	Unresolved
	CR	Code Repetition	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	DDS	Duplicate Data Structure	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	MC	Missing Check	Unresolved
	MEE	Missing Events Emission	Unresolved
	MU	Modifiers Usage	Unresolved
	PBV	Percentage Boundaries Validation	Unresolved
	RAS	Redundant Address Storage	Unresolved
	RLC	Redundant Limit Check	Unresolved
	RNRM	Redundant No Reentrant Modifier	Unresolved
	RC	Repetitive Calculations	Unresolved
	TUU	Time Units Usage	Unresolved

●	TSI	Tokens Sufficiency Insurance	Unresolved
●	UCC	Unnecessary Claim Calls	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L06	Missing Events Access Control	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

ITV - Inadequate TotalTLB Validation

Criticality	Minor / Informative
Location	Vault.sol#L164,173,287
Status	Unresolved

Description

The contract is designed to allocate a certain percentage of the total `totalTLB` tokens to team members and partners. Initially, it sets allocations for seven team members, totaling 13% of the `totalTLB`. However, the contract lacks sufficient validation in the `addPartner` function to ensure that the cumulative allocation does not exceed the `totalTLB`. This oversight allows for the potential addition of partners in a manner that could collectively bypass the intended total allocation limit of `totalTLB`. Specifically, while the function checks that the added percentage does not exceed the `partnerLimit` and that the cumulative percentage of all partners does not surpass this limit, it does not verify against the actual `totalTLB` amount remaining. Consequently, this could lead to scenarios where the sum of allocations to partners and team members exceeds the total available `totalTLB`, undermining the integrity of the token distribution mechanism.

```
    uint256 public totalTLB = 3 * 10 **9 * 10 **18;
     constructor(
          ...
          address _teamMember7
     ) {
          ...
          team[_owner] = Team(
               _owner,
               (totalTLB * 8)/100,
               0,
               tge,
               0
          );
          team[_dev] = Team(
               _dev,
               (totalTLB * 1)/100,
               0,
               tge,
               0
          );
          team[_teamMember1] = Team(
               _teamMember1,
               (totalTLB * 5)/1000,
               0,
               tge,
               0
          );
          ...
     }

     function addPartner(address _partner, uint8
_percentInBasePoints) external onlyOwner {
          require(0 < _percentInBasePoints && _percentInBasePoints
<= partnerLimit, "Added amount exceeds total partner limit");
          require(_percentInBasePoints + partnerTotal <=
partnerLimit, "Added amount exceeds partner allocation limit");
          require(partner[_partner].member == address(0), "Partner
already added!");
          partner[_partner] = Partner (
               _partner,
               (totalTLB * _percentInBasePoints)/1000,
               0,
               tge,
               0
          );
          partnerTotal+= _percentInBasePoints;

     }
```

Recommendation

It is recommended to introduce additional checks within the `addPartner` function to ensure that the total allocation for partners, when combined with the allocations for team members, does not exceed the available `totalTLB`. This could involve maintaining a running total of all `totalTLB` allocations and comparing this against the `totalTLB` before approving new partner additions. Implementing such checks will safeguard against the possibility of over-allocation, thereby preserving the contract's intended token distribution framework and ensuring fairness and transparency in the allocation process.

CR - Code Repetition

Criticality	Minor / Informative
Location	Vault.sol#L304,383
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 `teamClaim` and `partnerClaim` functions share similar code segments.

```
   function teamClaim() external nonReentrant {
        require(
            msg.sender == team[msg.sender].member,
            "Team Member not found!"
        );
        ...
        }

        else {
            uint256 amountReceived =
(team[msg.sender].amountDue *
                teamVestingPercent) / 1000;
            require(
                token.balanceOf(address(this)) >=
amountReceived,
                "Insufficient balance of TLB held in contract
to complete claim"
            );

            token.transfer(msg.sender, amountReceived);

            team[msg.sender].nextClaim = block.timestamp +
2629743; //2629743 1 month;
            team[msg.sender].amountClaimed += amountReceived;
            team[msg.sender].timesClaimed++;
        }
    }


    function partnerClaim() external nonReentrant {
        ....
        }
    }
```

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	Vault.sol#L287,461,465,470
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 centralizes significant authority in the address of the owner, enabling them to unilaterally alter critical parameters such as the token to be claimed, the timing and percentage of claims, and the eligibility of addresses to participate in the vault process.

```
    function setTge(uint256 _newTge) external onlyOwner{
        tge = _newTge;
    }

    function setTgePercent(uint8 _percent) external onlyOwner {
        require(_percent > 0 && _percent <= 100, "Invalid
percentage entered");
        tgePercent = _percent * 10;
    }

    function setToken(address _token) external onlyOwner {
        token = IERC20(_token);
    }

    function addPartner(address _partner, uint8
_percentInBasePoints) external onlyOwner {
        ...
        require(partner[_partner].member == address(0), "Partner
already added!");
        partner[_partner] = Partner (
            _partner,
            (totalTLB * _percentInBasePoints)/1000,
            0,
            tge,
            0
        );
        partnerTotal+= _percentInBasePoints;

    }
```

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.

DDS - Duplicate Data Structure

Criticality	Minor / Informative
Location	Vault.sol#L127,135
Status	Unresolved

Description

The contract utilized two data structures `Team` and `Partner` to illustrate the same information.

```
struct Team {
  address member;
  uint256 amountDue;
  uint256 amountClaimed;
  uint256 nextClaim;
  uint8 timesClaimed;
 }

 struct Partner {
  address member;
  uint256 amountDue;
  uint256 amountClaimed;
  uint256 nextClaim;
  uint8 timesClaimed;
 }
```

Recommendation

The team is advised to merge the two data structures. That way it will enhance the efficiency, readability, and performance of the source code, while also decreasing the cost of executing it.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	Vault.sol#L190,191,192,193,194,195,196,197
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
dev
teamMember1
teamMember2
teamMember3
teamMember4
teamMember5
teamMember6
teamMember7
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MC - Missing Check

Criticality	Minor / Informative
Location	Vault.sol#L461
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically the contract is missing a check in order to verify that the `_newTge` parameter is greater than the current time.

```
    function setTge(uint256 _newTge) external onlyOwner{
        tge = _newTge;
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Vault.sol#L461
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 setTge(uint256 _newTge) external onlyOwner{
        tge = _newTge;
    }

    function setTgePercent(uint8 _percent) external onlyOwner {
        require(_percent > 0 && _percent <= 100, "Invalid
percentage entered");
        tgePercent = _percent * 10;
    }

    function setToken(address _token) external onlyOwner {
        token = IERC20(_token);
    }

    function setVesting(
        ...
    }

    function setPartnerLimit (uint8 _percent) external
onlyOwner {
        require(_percent > 0 && _percent <= 100, "Invalid
percentage entered");
        partnerLimit = _percent * 10;
    }

    function transferOwnership(address payable _newOwner)
external onlyOwner {
        owner = _newOwner;
    }
```

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	Vault.sol#L305,363
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(
            msg.sender == team[msg.sender].member,
            "Team Member not found!"
        );
        require(
            block.timestamp > team[msg.sender].nextClaim,
            "Not time for next vesting"
        );
        require(
            team[msg.sender].timesClaimed < teamVestingPeriods,
            "You're already fully vested!"
        );
        require(
            team[msg.sender].amountDue > 0,
            "You are not due to collect anymore."
        );
```

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.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	Vault.sol#L476
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 setVesting(
        ...
        uint8 _newTeamPercent,
        ...
        uint8 _newPartnerPercent
        )
        external
        onlyOwner
    {
        ...
        teamVestingPercent = _newTeamPercent;
        ...
        partnerVestingPercent = _newPartnerPercent;
    }
```

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.

RAS - Redundant Address Storage

Criticality	Minor / Informative
Location	Vault.sol#L127,135,159
Status	Unresolved

Description

The contract is designed to manage team and partner allocations through two separate structs, `Team` and `Partner`, each containing a member's address, the amount due, the amount claimed, the next claim date, and the number of times claimed. Additionally, it utilizes mappings from member addresses to their respective Team or Partner struct, effectively creating a direct link between an address and its associated data. Given that the `member` address is the key to accessing each struct within the mappings, storing the member address again within the structs themselves is redundant. This redundancy does not contribute to the functionality of the contract and leads to unnecessary storage usage, which, in the context of blockchain, translates to wasted space and increased costs. Moreover, the practice of storing the address both in the mapping key and within the struct could lead to inconsistencies if the data within the struct were ever to be mistakenly modified.

```
struct Team {
    address member;
    uint256 amountDue;
    uint256 amountClaimed;
    uint256 nextClaim;
    uint8 timesClaimed;
   }

   struct Partner {
    address member;
    uint256 amountDue;
    uint256 amountClaimed;
    uint256 nextClaim;
    uint8 timesClaimed;
   }

    mapping (address => Team) public team;
    mapping (address => Partner) public partner;
```

Recommendation

It is recommended to remove the `member` field from both the `Team` and `Partner` structs, as the member address is already implicitly stored as the key in the respective mappings. This change will reduce storage redundancy, decrease the contract's deployment and execution gas costs, and simplify the data structure for easier maintenance and lower risk of inconsistencies. To check if a member exists or to access a member's data, the contract should directly utilize the mappings with the member's address as the key. This approach streamlines data management within the contract and adheres to best practices for efficient smart contract development.

RLC - Redundant Limit Check

Criticality	Minor / Informative
Location	Vault.sol#L267
Status	Unresolved

Description

The contract manages the addition of partners through the `addPartner` function, which includes checks to ensure that the percentage of the total allocated to each partner does not exceed predefined limits. Specifically, the function includes two require statements related to the `_percentInBasePoints`. The one checks if `_percentInBasePoints` is within a valid range (greater than 0 and less than or equal to partnerLimit), and the other verifies if the addition of `_percentInBasePoints` to `partnerTotal` does not exceed `partnerLimit`. However the first check is redundant because if `_percentInBasePoints` is greater than `partnerLimit`, the second condition (`_percentInBasePoints + partnerTotal <= partnerLimit`) would always fail. This redundancy could lead to unnecessary gas costs for executing the contract and makes the code less efficient without providing additional security or functional benefits.

```
    function addPartner(
        address _partner,
        uint8 _percentInBasePoints
    ) external onlyOwner {
        require(
            0 < _percentInBasePoints && _percentInBasePoints <=
partnerLimit,
            "Added amount exceeds total partner limit"
        );
        require(
            _percentInBasePoints + partnerTotal <=
partnerLimit,
            "Added amount exceeds partner allocation limit"
        );
        ...
        partnerTotal += _percentInBasePoints;
    }
```

Recommendation

It is recommended to consider removing the check `_percentInBasePoints <= partnerLimit` since it is superfluous. The second condition, `_percentInBasePoints + partnerTotal <= partnerLimit`, sufficiently ensures that the addition of a new partner's percentage does not exceed the overall `partnerLimit`. Eliminating the redundant check will streamline the function, potentially reduce gas costs, and maintain the contract's integrity by ensuring that the total allocation does not surpass the designated limit. Simplifying the conditionals in this manner will also enhance code readability and maintainability without compromising the contract's security or functionality.

RNRM - Redundant No Reentrant Modifier

Criticality	Minor / Informative
Location	Vault.sol#L304,383
Status	Unresolved

Description

The contract uses the `nonReentrant` modifier to the `teamClaim` and `partnerClaim` functions, which suggests an intention to prevent potential reentrancy attacks. However, neither of these functions deals with the transfer of the native token or any other value. As such, the risk of reentrancy attacks in these specific functions is minimal to non-existent.

```
    function teamClaim() external nonReentrant {
      ...
    }

    function partnerClaim() external nonReentrant {
      ...
    }
```

Recommendation

To address this finding and enhance code simplicity and clarity, it is recommended to remove the unnecessary `nonReentrant` modifier from the `teamClaim` and `partnerClaim` functions. By removing the modifier, the code becomes more streamlined and easier to comprehend, reducing the gas consuption.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	Vault.sol#L352,431
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

```
} else if ((team[msg.sender].amountDue -
team[msg.sender].amountClaimed) < (team[msg.sender].amountDue *
teamVestingPercent)/1000) {
        uint256 remainder = team[msg.sender].amountDue -
        team[msg.sender].amountClaimed;
...
} else if ((partner[msg.sender].amountDue -
partner[msg.sender].amountClaimed) <
(partner[msg.sender].amountDue * partnerVestingPercent)/1000) {
    uint256 remainder = partner[msg.sender].amountDue -
        partner[msg.sender].amountClaimed;
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and

gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

TUU - Time Units Usage

Criticality	Minor / Informative
Location	Vault.sol#L334,376,413
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
team[msg.sender].nextClaim = block.timestamp + 2629743;
//2629743 1 month
team[msg.sender].nextClaim = block.timestamp + 2629743;
//2629743 1 month;
partner[msg.sender].nextClaim = block.timestamp + 2629743; //1
month
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	Vault.sol#L470
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.

The contract currently lacks a mechanism to verify the adequacy of token amounts during the claim process. This oversight means that when members or partners attempt to claim their allocated tokens, there is no preliminary check to ensure that the contract possesses a sufficient amount of tokens to fulfill these claims. Without such validation, there is a risk of attempting to distribute more tokens than the contract holds, which could lead to failed transactions, wasted gas fees, and potential trust issues among the participants. Ensuring the contract has an adequate supply of tokens before proceeding with claims is crucial for maintaining the integrity of the distribution process and for upholding the contract's obligations to its members and partners.

```
    function setToken(address _token) external onlyOwner {
        token = IERC20(_token);
    }
```

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. It is recommended that the contract implement checks to verify the availability of tokens before proceeding with any balance updates, particularly during the claim process. This verification should be incorporated not only within the claim functions

but also during initial setup phases such as within the constructor and the `addPartner` function. By checking that the contract holds a sufficient amount of tokens before setting the address balances, the contract can prevent attempts to claim or allocate tokens beyond its current supply. This proactive approach ensures that all transactions can be completed successfully and that the contract remains in a valid state, thereby maintaining trust and reliability among its participants. Implementing these checks will enhance the contract's robustness and its adherence to best practices in smart contract development.

UCC - Unnecessary Claim Calls

Criticality	Minor / Informative
Location	Vault.sol#L304,383
Status	Unresolved

Description

The contract contains the `teamClaim` and `partnerClaim` functions designed to allow team members and partners to claim their allocated tokens in a vesting process. These functions include several checks to ensure that claims are made by valid members, at the correct time, and that members have not already claimed all their due tokens. However, if the contract owner changes the configuration of the variables that set the percentage of tokens due to each member or partner, it could potentially alter the `amountDue`, for individuals. As a result in a case where a user's `amountClaimed` equals their `amountDue`, indicating they have already claimed their total allocated amount, the functions do not explicitly prevent these users from invoking the claim functions again. As a result, users can still call the claim functions even when they are due to receive zero tokens. This oversight can lead to unnecessary gas expenditures for users and unwarranted computational load on the network, as these transactions do not alter the state in any meaningful way.

```
    function teamClaim() external nonReentrant {
        require(
            msg.sender == team[msg.sender].member,
            "Team Member not found!"
        );
        require(
            block.timestamp > team[msg.sender].nextClaim,
            "Not time for next vesting"
        );
        require(
            team[msg.sender].timesClaimed < teamVestingPeriods,
            "You're already fully vested!"
        );
        require(
            team[msg.sender].amountDue > 0,
            "You are not due to collect anymore."
        );



         ...
        } else if ((team[msg.sender].amountDue -
team[msg.sender].amountClaimed) < (team[msg.sender].amountDue *
teamVestingPercent)/1000) {
            uint256 remainder = team[msg.sender].amountDue -
                team[msg.sender].amountClaimed;
             require(
                token.balanceOf(address(this)) >= remainder,
                "Insufficient balance of TLB held in contract
to complete claim"
            );

            token.transfer(msg.sender, remainder);

            team[msg.sender].amountClaimed += remainder;
            team[msg.sender].timesClaimed++;
        }

      ...
        }
    }


    function partnerClaim() external nonReentrant {
       ...
        } else if ((partner[msg.sender].amountDue -
partner[msg.sender].amountClaimed) <
(partner[msg.sender].amountDue * partnerVestingPercent)/1000) {
            uint256 remainder = partner[msg.sender].amountDue -
                partner[msg.sender].amountClaimed;
```

```
                require(
                    token.balanceOf(address(this)) >= remainder,
                    "Insufficient balance of TLB held in contract
to complete claim"
                );

                token.transfer(msg.sender, remainder);

                partner[msg.sender].amountClaimed += remainder;
                partner[msg.sender].timesClaimed++;
            }
        ...
            }
    }
```

Recommendation

It is recommended to add an additional check in both the `teamClaim` and `partnerClaim` functions to prevent users from invoking these functions when their `amountClaimed` equals their `amountDue`. Specifically, a require statement should be introduced to assert that the difference between `amountDue` and `amountClaimed` is greater than zero before proceeding with the rest of the function's logic. This check will ensure that only users with unclaimed tokens can execute the claim functions, thereby optimizing contract efficiency and reducing unnecessary network and user costs. Implementing this safeguard will enhance the contract's performance and user experience by preventing futile transactions.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	Vault.sol#L168
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
uint256 public totalTLB = 3 * 10 **9 * 10 **18
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Vault.sol#L271,275,279,282,291,465,469,474,481,482,483,484,495,502
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.

```
address _addr
uint8 _percentInBasePoints
address _partner
uint256 _newTge
uint8 _percent
address _token
uint8 _newTeamPeriod
uint8 _newTeamPercent
uint8 _newPartnerPeriod
uint8 _newPartnerPercent
address payable _newOwner
```

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/v0.8.17/style-guide.html#naming-convention.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	Vault.sol#L503
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
owner = _newOwner
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	Vault.sol#L189,190,191,192,193,194,195,196,197,503
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
owner = _owner
dev = _dev
teamMember1 = _teamMember1
teamMember2 = _teamMember2
teamMember3 = _teamMember3
teamMember4 = _teamMember4
teamMember5 = _teamMember5
teamMember6 = _teamMember6
teamMember7 = _teamMember7
owner = _newOwner
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	Vault.sol#L6
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 ^0.8.17;
```

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	Vault.sol#L336,352,364,378,415,431,443,457
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.

```
token.transfer(msg.sender, amountReceived )
token.transfer(msg.sender, remainder)
token.transfer(msg.sender, amountReceived)
```

Recommendation

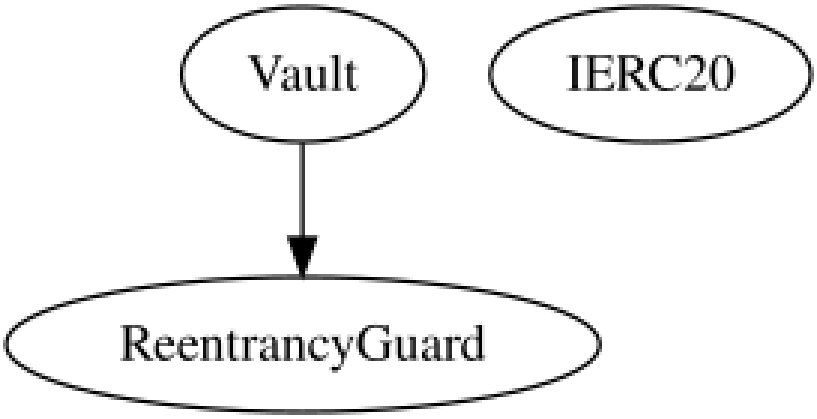
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

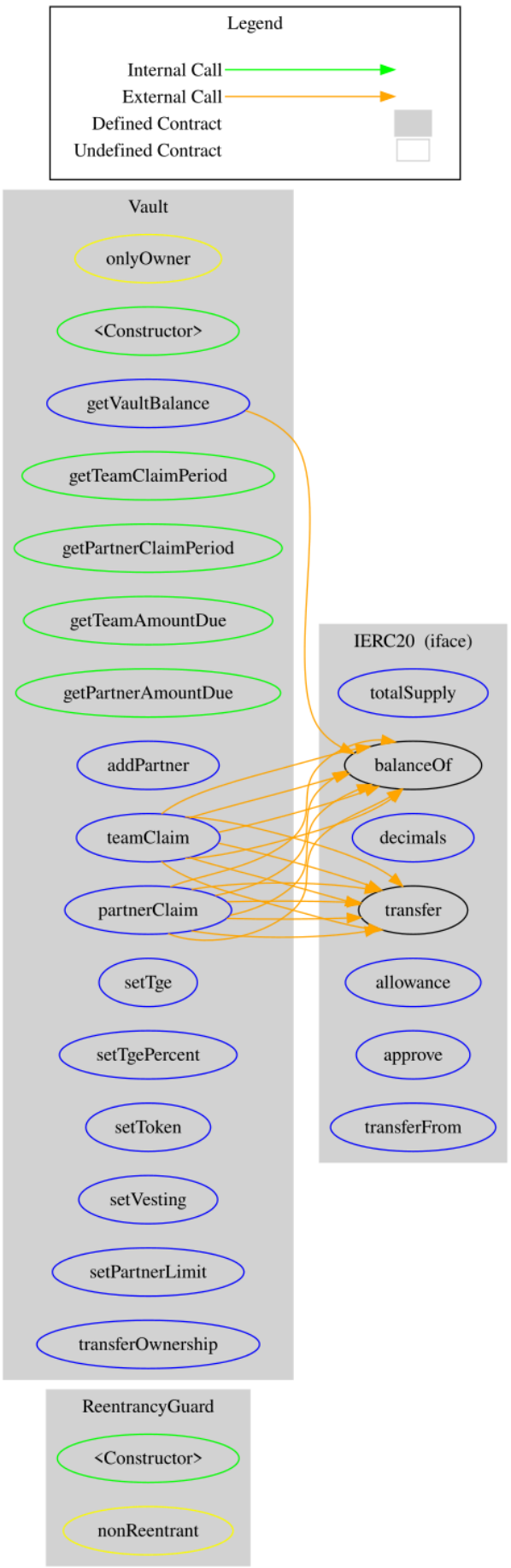
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ReentrancyGuard	Implementation			
		Public	✓	-
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	decimals	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
Vault	Implementation	ReentrancyGuard		
		Public	✓	-
	getVaultBalance	External		-
	getTeamClaimPeriod	Public		-
	getPartnerClaimPeriod	Public		-
	getTeamAmountDue	Public		-

	getPartnerAmountDue	Public		-
	addPartner	External	✓	onlyOwner
	teamClaim	External	✓	nonReentrant
	partnerClaim	External	✓	nonReentrant
	setTge	External	✓	onlyOwner
	setTgePercent	External	✓	onlyOwner
	setToken	External	✓	onlyOwner
	setVesting	External	✓	onlyOwner
	setPartnerLimit	External	✓	onlyOwner
	transferOwnership	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Vault
onlyOwner
<Constructor>
getVaultBalance
getTeamClaimPeriod
getPartnerClaimPeriod
getTeamAmountDue
getPartnerAmountDue
addPartner
teamClaim
partnerClaim
setTge
setTgePercent
setToken
setVesting
setPartnerLimit
transferOwnership
IERC20 (iface)
totalSupply
balanceOf
decimals
transfer
allowance
approve
transferFrom
ReentrancyGuard
<Constructor>
nonReentrant

Summary

TLB Token contract implements a vault 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