Cyberscope

Audit Report

TLB Token

March 2024

Network	SEPOLIA
Address	0x7d0c0d6b7f1ffc7c6bfd521b3b714ec020cebd09
Audited by	© cyberscope

Analysis

Critical Medium Minor / Informative Pass

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	ILAR	Incorrect Liquidity Addition Rate	Unresolved
Medium	UBE	Unconditional BurnShiba Event	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	FRV	Fee Restoration Vulnerability	Unresolved
Minor / Informative	IDI	Immutable Declaration Improvement	Unresolved
Minor / Informative	IGCL	Inefficient Gas Check Logic	Unresolved
Minor / Informative	MTEE	Missing Transfer Event Emission	Unresolved
Minor / Informative	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
Minor / Informative	PMRM	Potential Mocked Router Manipulation	Unresolved
Minor / Informative	PSF	Public Swap Function	Unresolved
Minor / Informative	RSML	Redundant SafeMath Library	Unresolved
Minor / Informative	RSW	Redundant Storage Writes	Unresolved
Minor / Informative	RVD	Redundant Variable Declaration	Unresolved
Minor / Informative	RC	Repetitive Calculations	Unresolved

●	TSI	Tokens Sufficiency Insurance	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

Table of Contents

Analysis 1
Diagnostics 2
Table of Contents 4
Review 7
Audit Updates 7
Source Files 7
Findings Breakdown 8
ST - Stops Transactions 9
Description 9
Recommendation 9
BC - Blacklists Addresses 11
Description 11
Recommendation 11
ILAR - Incorrect Liquidity Addition Rate 13
Description 13
Recommendation 14
UBE - Unconditional BurnShiba Event 15
Description 15
Recommendation 15
CR - Code Repetition 17
Description 17
Recommendation 18
FRV - Fee Restoration Vulnerability 20
Description 20
Recommendation 21
IDI - Immutable Declaration Improvement 22
Description 22
Recommendation 22
IGCL - Inefficient Gas Check Logic 23
Description 23
Recommendation 23
MTEE - Missing Transfer Event Emission 24
Description 24
Recommendation 24
PLPI - Potential Liquidity Provision Inadequacy 25
Description 25
Recommendation 25
PMRM - Potential Mocked Router Manipulation 27
Description 27

Recommendation 27
PSF - Public Swap Function 29
Description 29
Recommendation 29
RSML - Redundant SafeMath Library 31
Description 31
Recommendation 31
RSW - Redundant Storage Writes 32
Description 32
Recommendation 32
RVD - Redundant Variable Declaration 33
Description 33
Recommendation 33
RC - Repetitive Calculations 34
Description 34
Recommendation 34
TSI - Tokens Sufficiency Insurance 36
Description 36
Recommendation 36
OCTD - Transfers Contract's Tokens 38
Description 38
Recommendation 38
L02 - State Variables could be Declared Constant 40
Description 40
Recommendation 40
L04 - Conformance to Solidity Naming Conventions 41
Description 41
Recommendation 42
L07 - Missing Events Arithmetic 43
Description 43
Recommendation 43
L09 - Dead Code Elimination 44
Description 44
Recommendation 45
L13 - Divide before Multiply Operation 46
Description 46
Recommendation 46
L14 - Uninitialized Variables in Local Scope 47
Description 47
Recommendation 47
L16 - Validate Variable Setters 48
Description 48

Recommendation 48
L17 - Usage of Solidity Assembly 49
Description 49
Recommendation 49
L19 - Stable Compiler Version 50
Description 50
Recommendation 50
L20 - Succeeded Transfer Check 51
Description 51
Recommendation 51
Functions Analysis **52**
Inheritance Graph **59**
Flow Graph **60**
Summary **61**
Disclaimer **62**
About Cyberscope **63**

Review

Explorer	https://sepolia.etherscan.io/address/0x7d0c0d6b7f1ffc7c6bfd521b3b714ec020cebd09
Address	0x7d0c0d6b7f1ffc7c6bfd521b3b714ec020cebd09
Badge Eligibility	Must Fix Criticals

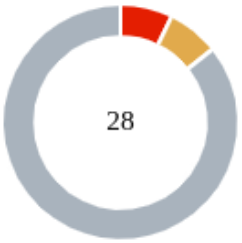
Audit Updates

Initial Audit	13 Mar 2024

Source Files

Filename	SHA256
TLB.sol	95fdb14250590be07033faad7d55cab3ca414be67a1b9dde497a4d39134f748a

Findings Breakdown



Critical	2
Medium	2
Minor / Informative	24

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	2	0	0	0
Minor / Informative	24	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	TLB.sol#L955
Status	Unresolved

Description

The contract owner has the authority to stop the sales for all users excluding the owner. The owner may take advantage of it by setting the `_maxWalletToken` to zero. As a result, the contract may operate as a honeypot.

```
require(
    (heldTokens + amount) <= _maxWalletToken,
    "You are trying to buy too many tokens. You have reached
the limit for one wallet."
);
```

Additionally, the contract owner has the authority to stop transactions, as described in detail in sections `PMRM` and `PLPI`. As a result, the contract might operate as a honeypot.

Recommendation

The contract could embody a check for not allowing setting the `_maxWalletToken` less than a reasonable amount. A suggested implementation could check that the minimum amount should be more than a fixed percentage of the total supply. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.

- Introduce a governance model where users will vote about the actions.

Permanent Solution:

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

BC - Blacklists Addresses

Criticality	Critical
Location	TLB.sol#L907
Status	Unresolved

Description

The contract owner has the authority to stop addresses from transactions. The owner may take advantage of it by calling the `blacklist_Add_Wallets` function.

```
    function blacklist_Add_Wallets(
        address[] calldata addresses
    ) external onlyOwner {
        uint256 startGas;
        uint256 gasUsed;

        for (uint256 i; i < addresses.length; ++i) {
            if (gasUsed < gasleft()) {
                startGas = gasleft();
                if (!_isBlacklisted[addresses[i]]) {
                    _isBlacklisted[addresses[i]] = true;
                }
                gasUsed = startGas - gasleft();
            }
        }
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.

- Introduce a governance model where users will vote about the actions.

Permanent Solution:

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

ILAR - Incorrect Liquidity Addition Rate

Criticality	Medium
Location	TLB.sol#L1050
Status	Unresolved

Description

The contract is designed to perform token swaps and add liquidity to a decentralized exchange (DEX) pool through its `swapAndLiquify` and `force_swapAndLiquify` functions. These functions divide the contract's token balance into two halves. One half is swapped for Ethereum (ETH), and the other half is intended to be added to the liquidity pool alongside the ETH obtained from the swap. However, the function calculates `liquidityAmount` (the portion of tokens to be swapped for ETH) based on a fraction of the first half, determined by the `liquidityTax` rate. This approach leads to a mismatch in the token and ETH amounts being added to the liquidity pool, as otherHalf represents a larger portion of tokens than `liquidityAmount` represents in ETH. Consequently, a portion of tokens remains in the contract after the liquidity addition, which could lead to an imbalance in the intended liquidity provision and leave excess tokens within the contract.

```
    function swapAndLiquify(
        uint256 balance
    ) private lockTheSwap returns (bool) {
        uint256 balanceInContract = balance;
        uint256 initialBalance = address(this).balance;
        uint256 half = balanceInContract.div(2);
        uint256 otherHalf = balanceInContract.sub(half);
        uint256 lf = liquidityTax;
        uint256 bf = burnTax;
        uint256 sf = shibaTax;

        uint256 denominator = lf + bf + sf;

        uint256 liquidityAmount =
half.mul(lf).div(denominator);
        uint256 burnAmount = half.mul(bf).div(denominator);
        uint256 shibaAmount = half.mul(sf).div(denominator);

        swapTokensForETH(liquidityAmount);

        uint256 newBalance =
address(this).balance.sub(initialBalance);

        addLiquidity(otherHalf, newBalance);
    ...
```

Recommendation

It is recommended to adjust the logic within the `swapAndLiquify` function to ensure that the amount of tokens swapped for ETH aligns with the amount of tokens intended for addition to the liquidity pool. This could involve recalculating the `liquidityAmount` to ensure it matches the `otherHalf` of the tokens when added to the liquidity pool, thus preventing any tokens from being inadvertently left in the contract. One approach could be to base the `liquidityAmount` on the total balance intended for liquidity rather than just a fraction of one half. Additionally, ensuring that the token and ETH amounts are equivalent when adding liquidity will help maintain the desired token-ETH ratio in the liquidity pool, enhancing the overall liquidity strategy's effectiveness. This adjustment will not only optimize the liquidity provision process but also ensure that the contract's operations are more aligned with its intended economic mechanisms.

UBE - Unconditional BurnShiba Event

Criticality	Medium
Location	TLB.sol#L1157
Status	Unresolved

Description

The contract is designed to burn a specified amount of Shiba tokens by transferring them to a `deadWallet`. The `_burnShiba` function checks if the contract's balance of Shiba tokens is sufficient for the burn operation. If the condition is met, it proceeds with the transfer and emits the `BurnShiba` event to log the action. However, the function also unconditionally emits a `BurnShiba` event outside of the conditional block, with the burn amount set to half of the requested amount, regardless of whether the initial burn operation was executed. This implementation leads to the `BurnShiba` event being emitted in every invocation of `_burnShiba`, potentially causing confusion and inaccuracies in event logging, especially in scenarios where no tokens are actually burnt due to insufficient balance.

```
    function _burnShiba(uint256 burnAmount) internal {
        if (shiba.balanceOf(address(this)) >= burnAmount) {
            shiba.transfer(deadWallet, burnAmount);
            emit BurnShiba(address(this), deadWallet,
burnAmount);
        }

        emit BurnShiba(address(this), deadWallet, burnAmount /
2);
    }
```

Recommendation

It is recommended to consider removing the second `BurnShiba` event. If the intention is to always emit an event to indicate an attempted burn, the event's parameters should accurately reflect the outcome of the function's execution. For instance, if no tokens are burnt due to an insufficient balance, the function could emit an event with a zero value or not emit an event at all, depending on the desired logic and transparency. Adjusting the

event emission to accurately represent the function's actions will enhance the contract's clarity and reliability, ensuring that event logs correctly reflect the contract's operations.

CR - Code Repetition

Criticality	Minor / Informative
Location	TLB.sol#L1049,1085
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 `swapAndLiquify` and `force_swapAndLiquify` share the same code segmentes.

```
  function swapAndLiquify(
        uint256 balance
    ) private lockTheSwap returns (bool) {
        uint256 balanceInContract = balance;
        uint256 initialBalance = address(this).balance;
        uint256 half = balanceInContract.div(2);
        uint256 otherHalf = balanceInContract.sub(half);
        uint256 lf = liquidityTax;
        uint256 bf = burnTax;
        uint256 sf = shibaTax;

        uint256 denominator = lf + bf + sf;

        uint256 liquidityAmount =
half.mul(lf).div(denominator);
        uint256 burnAmount = half.mul(bf).div(denominator);
        uint256 shibaAmount = half.mul(sf).div(denominator);

        swapTokensForETH(liquidityAmount);

        uint256 newBalance =
address(this).balance.sub(initialBalance);

        addLiquidity(otherHalf, newBalance);

        if (burnAmount > 0) {
            _burn(burnAmount);
        }

        if (shibaAmount > 0) {
            _burnShiba(shibaAmount);
        }

        emit SwapAndLiquify(half, liquidityAmount, burnAmount,
shibaAmount);

        return true;
    }

    function force_swapAndLiquify(
        uint256 balance
    ) external nonReentrant onlyOwner returns (bool) {
        ...
    }
```

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.

FRV - Fee Restoration Vulnerability

Criticality	Minor / Informative
Location	TLB.sol#L958,970,1179
Status	Unresolved

Description

The contract demonstrates a potential vulnerability upon removing and restoring the fees. This vulnerability can occur when the fees have been set to zero. During a transaction, if the fees have been set to zero, then both remove fees and restore fees functions will be executed. The remove fees function is executed to temporarily remove the fees, ensuring the sender is not taxed during the transfer. However, the function prematurely returns without setting the variables that hold the previous fee values.

As a result, when the subsequent restore fees function is called after the transfer, it restores the fees to their previous values. However, since the previous fee values were not properly set to zero, there is a risk that the fees will retain their non-zero values from before the fees were removed. This can lead to unintended consequences, potentially causing incorrect fee calculations or unexpected behavior within the contract.

```
    function removeAllFee() private {
        if (_TotalFee == 0 && _buyFee == 0 && _sellFee == 0)
return;

        _previousBuyFee = _buyFee;
        _previousSellFee = _sellFee;
        _previousTotalFee = _TotalFee;
        _buyFee = 0;
        _sellFee = 0;
        _TotalFee = 0;
    }

    // Restore all fees
    function restoreAllFee() private {
        _TotalFee = _previousTotalFee;
        _buyFee = _previousBuyFee;
        _sellFee = _previousSellFee;
    }

    function _tokenTransfer(
        address sender,
        address recipient,
        uint256 amount,
        bool takeFee
    ) private {
        if (!takeFee) {
            removeAllFee();
        }
        _transferTokens(sender, recipient, amount);

        if (!takeFee) restoreAllFee();
    }
```

Recommendation

The team is advised to modify the remove fees function to ensure that the previous fee values are correctly set to zero, regardless of their initial values. A recommended approach would be to remove the early return when both fees are zero.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	TLB.sol#L562,563,564,565,566
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.

```
treasury
teamVault
ecosystem
rewards
charity
```

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.

IGCL - Inefficient Gas Check Logic

Criticality	Minor / Informative
Location	TLB.sol#L913
Status	Unresolved

Description

The contract is utilizing the `gasleft()` function within a for loop to monitor the remaining gas available during its execution, specifically when iterating over an array of addresses to update their blacklist status. The intention behind this logic is to prevent the contract from running out of gas by comparing `gasUsed` with the result of `gasleft()`. However, the current implementation lacks a mechanism to exit or break the loop when `gasUsed` exceeds the amount of gas left, potentially leading to situations where the loop continues executing even when there is insufficient gas, risking transaction failure without completing the intended operations.

```
for (uint256 i; i < addresses.length; ++i) {
    if (gasUsed < gasleft()) {
        startGas = gasleft();
        if (!_isBlacklisted[addresses[i]]) {
            _isBlacklisted[addresses[i]] = true;
        }
        gasUsed = startGas - gasleft();
    }
```

Recommendation

It is recommended to implement a more robust gas management strategy within the loop. This could involve adding a condition to break out of the loop if the gas left is below a certain threshold, ensuring that the contract does not continue to execute once it becomes clear that there is not enough gas to complete further iterations safely. This modification ensures that the contract conservatively manages its gas usage, preventing execution from proceeding to a point where it might fail due to insufficient gas.

MTEE - Missing Transfer Event Emission

Criticality	Minor / Informative
Location	TLB.sol#L1197
Status	Unresolved

Description

The contract is a missing transfer event emission when fees are transferred to the contract address as part of the transfer process. This omission can lead to a lack of visibility into fee transactions and hinder the ability of decentralized applications (DApps) like blockchain explorers to accurately track and analyze these transactions.

```
_tOwned[address(this)] = _tOwned[address(this)].add(tDev);
```

Recommendation

To address this issue, it is recommended to emit a transfer event after transferring the taxed amount to the contract address. The event should include relevant information such as the sender, recipient, and the amount transferred.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	TLB.sol#L1122
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
    function swapTokensForETH(uint256 tokenAmount) public {
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();
        _approve(address(this), address(uniswapV2Router),
tokenAmount);

uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTok
ens(
            tokenAmount,
            0,
            path,
            address(this),
            block.timestamp
        );
    }
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

PMRM - Potential Mocked Router Manipulation

Criticality	Minor / Informative
Location	TLB.sol#L1215
Status	Unresolved

Description

The contract includes a method that allows the owner to modify the router address and create a new pair. While this feature provides flexibility, it introduces a security threat. The owner could set the router address to any contract that implements the router's interface, potentially containing malicious code. In the event of a transaction triggering the swap functionality with such a malicious contract as the router, the transaction may be manipulated.

```
   function set_New_Router_Address(address newRouter) public
onlyOwner {
        IUniswapV2Router02 _newPCSRouter =
IUniswapV2Router02(newRouter);
        uniswapV2Router = _newPCSRouter;
    }
```

Recommendation

The team should carefully manage the private keys of the owner’s account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

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

Permanent Solution:

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

PSF - Public Swap Function

Criticality	Minor / Informative
Location	TLB.sol#L1123
Status	Unresolved

Description

The contract is exposing the `swapTokensForETH` function as a `public` function, allowing any user to call it and initiate a swap of the contract's tokens for Ethereum (ETH) via the Uniswap V2 Router. This design poses a significant security risk, as it enables potentially malicious actors to manipulate the token's price by executing swaps at opportune moments, possibly affecting the token's market stability and fairness. The unrestricted access to this function could lead to unintended consequences, such as draining the contract's liquidity or causing extreme volatility in the token's price, which could harm legitimate token holders and disrupt the token ecosystem.

```
    function swapTokensForETH(uint256 tokenAmount) public {
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();
        _approve(address(this), address(uniswapV2Router),
tokenAmount);

uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTok
ens(
            tokenAmount,
            0,
            path,
            address(this),
            block.timestamp
        );
    }
```

Recommendation

It is recommended to restrict access to the `swapTokensForETH` function by modifying its visibility from `public` to `internal` or `private`, ensuring that only the contract itself can initiate swaps when certain conditions are met, or through functions that

are securely controlled. Additionally, implementing role-based access control mechanisms, such as requiring that the caller is an owner or has a specific role, can provide further protection against unauthorized access.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	TLB.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	TLB.sol#L884,889
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
    function excludeFromFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = true;
    }

    function includeInFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = false;
    }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

RVD - Redundant Variable Declaration

Criticality	Minor / Informative
Location	TLB.sol#L666,1225
Status	Unresolved

Description

The contract contains the boolean variable `noFeeToTransfer`, which is intended to indicate whether transfers should incur fees. Additionally, the `set_Transfers_Without_Fees` function allows the contract owner to update this variable. However it is observed that the `noFeeToTransfer` variable does not influence any part of the contract's functionality. Specifically, there are no conditional checks or logic within the contract that alter behavior based on the state of `noFeeToTransfer`. This discrepancy suggests that the variable, along with its associated setter function, is redundant, potentially leading to confusion and misinterpretation of the contract's intended behavior.

```
bool public noFeeToTransfer = true;
...
function set_Transfers_Without_Fees(bool true_or_false)
external onlyOwner {
    noFeeToTransfer = true_or_false;
}
```

Recommendation

It is recommended to either integrate the `noFeeToTransfer` variable into the contract's logic where appropriate or remove the variable and its setter function if it serves no purpose. If the intention behind `noFeeToTransfer` is to conditionally apply transfer fees, the contract should implement logic that checks the variable's state during transfer operations and adjusts fees accordingly. Alternatively, if the variable is a remnant of a previous version or feature that is no longer planned or relevant, removing it will simplify the contract and eliminate any confusion regarding its purpose. Ensuring that all variables and functions contribute meaningfully to the contract's operations enhances clarity, maintainability, and efficiency.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	TLB.sol#L770
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.

```
_tOwned[treasury] = _tTotal
    .sub(_tTotal.mul(19).div(100))
    .sub(_tTotal.mul(12).div(100))
    .sub(_tTotal.mul(18).div(100))
    .sub(_tTotal.mul(3).div(100));
_tOwned[teamVault] = _tTotal.mul(19).div(100);
_tOwned[rewards] = _tTotal.mul(12).div(100);
_tOwned[ecosystem] = _tTotal.mul(18).div(100);
_tOwned[charity] = _tTotal.mul(3).div(100);
```

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

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	TLB.sol#L1151
Status	Unresolved

Description

The Shiba 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. Additionally, there is no direct connection between the fees collected by the contract and the Shiba tokens. Consequently, fees designated for specific purposes (shiba burning) merely accumulate within the contract without a clear, automated mechanism for their intended use. This disconnection not only complicates the fee management process but also raises questions about the efficiency and transparency of the contract's operations.

```
    function _burnShiba(uint256 burnAmount) internal {
        if (shiba.balanceOf(address(this)) >= burnAmount) {
            shiba.transfer(deadWallet, burnAmount);
            emit BurnShiba(address(this), deadWallet,
burnAmount);
        }

        emit BurnShiba(address(this), deadWallet, burnAmount /
2);
    }
```

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 Shiba 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. Furthermore, establishing a direct link between the collected fees and the Shiba tokens would ensure that the fees serve their specific purposes effectively. For

instance, the contract could automatically use a portion of the fees to purchase Shiba tokens on the open market for burning, thereby increasing transparency and trust among participants. This could be achieved through smart contract functions that are triggered by certain transactions or at regular intervals, ensuring that the token economy operates smoothly and autonomously. Adopting these measures would not only improve the contract's functionality but also bolster participant confidence in the fairness and efficiency of the token's ecosystem.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	TLB.sol#L1257
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `remove_Stuck_Tokens` function.

```
   function remove_Stuck_Tokens(
        address stuck_Token_Address,
        address send_to_wallet,
        uint256 number_of_tokens
    ) public onlyOwner returns (bool _sent) {
        uint256 stuckBalance =
IERC20(stuck_Token_Address).balanceOf(
            address(this)
        );
        if (number_of_tokens > stuckBalance) {
            number_of_tokens = stuckBalance;
        }
        _sent = IERC20(stuck_Token_Address).transfer(
            send_to_wallet,
            number_of_tokens
        );
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.

- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

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

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	TLB.sol#L460,471,472,473,483,491,492,493,512
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.

```
address private deadWallet =
0x000000000000000000000000000000000000dEaD
string private _name = "TLB Token"
string private _symbol = "TLBS"
uint8 private _decimals = 18
uint256 private maxPossibleFee = 15
uint256 public shibaTax = 2
uint256 public burnTax = 2
uint256 public liquidityTax = 5
uint256 public _maxTxAmount = _tTotal.div(10000)
```

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	TLB.sol#L275,276,289,306,447,450,474,487,488,489,508,512,682,700,727,887,1028,1035,1047,1060,1066,1070,1083
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.

```
function DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint);
function WETH() external pure returns (address);
mapping (address => bool) public _isExcludedFromFee
mapping (address => bool) public _isBlacklisted
uint256 private constant _tTotal = 3 * 10**9 * 10**18
uint256 private _TotalFee = 10
uint256 public _buyFee = 5
uint256 public _sellFee = 5
uint256 public _maxWalletToken = _tTotal.div(100)
uint256 public _maxTxAmount = _tTotal.div(10000)

...
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	TLB.sol#L885,890,1038,1067,1071
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.

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.

```
_isExcludedFromFee[account] = true;
_isExcludedFromFee[account] = false;
_sellFee = Sell_Fee
swapTrigger = _newLimit
_maxWalletToken = _newLimit
```

Recommendation

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 with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	TLB.sol#L75,81,87,91,95,99,106,110,117,121,127
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function isContract(address account) internal view returns
(bool) {
        uint256 size;
        assembly { size := extcodesize(account) }
        return size > 0;
    }

...
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value,
recipient may have reverted");
    }

function functionCall(address target, bytes memory data)
internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level
call failed");
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	TLB.sol#L836,844,845,846,890,899,900,901
Status	Unresolved

Description

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

```
uint256 half = balanceInContract.div(2)
uint256 burnAmount = half.mul(bf).div(denominator)
```

Recommendation

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

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	TLB.sol#L685,687,703,705
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 gasUsed
uint256 i
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	TLB.sol#L562,563,564,565,566
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.

```
treasury = _treasury
teamVault = _teamVault
ecosystem = _ecosystem
rewards = _rewards
charity = _charity
```

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.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	TLB.sol#L77,132
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly { size := extcodesize(account) }

assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata),
returndata_size)
                }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	TLB.sol#L9
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 ^0.8.15;
```

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	TLB.sol#L973
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.

```
shiba.transfer(deadWallet, burnAmount)
```

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**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
SafeMath	Library			
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	sub	Internal		
	div	Internal		
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		

Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		
	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	
	functionDelegateCall	Internal	✓	
	_verifyCallResult	Private		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
ReentrancyGuard	Implementation			
		Public	✓	-

IUniswapV2Factory	Interface			
	feeTo	External		-
	feeToSetter	External		-
	getPair	External		-
	allPairs	External		-
	allPairsLength	External		-
	createPair	External	✓	-
	setFeeTo	External	✓	-
	setFeeToSetter	External	✓	-
IUniswapV2Pair	Interface			
	name	External		-
	symbol	External		-
	decimals	External		-
	totalSupply	External		-
	balanceOf	External		-
	allowance	External		-
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
	DOMAIN_SEPARATOR	External		-
	PERMIT_TYPEHASH	External		-

	nonces	External		-
	permit	External	✓	-
	MINIMUM_LIQUIDITY	External		-
	factory	External		-
	token0	External		-
	token1	External		-
	getReserves	External		-
	price0CumulativeLast	External		-
	price1CumulativeLast	External		-
	kLast	External		-
	burn	External	✓	-
	swap	External	✓	-
	skim	External	✓	-
	sync	External	✓	-
	initialize	External	✓	-
IUniswapV2Router01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
	removeLiquidity	External	✓	-
	removeLiquidityETH	External	✓	-

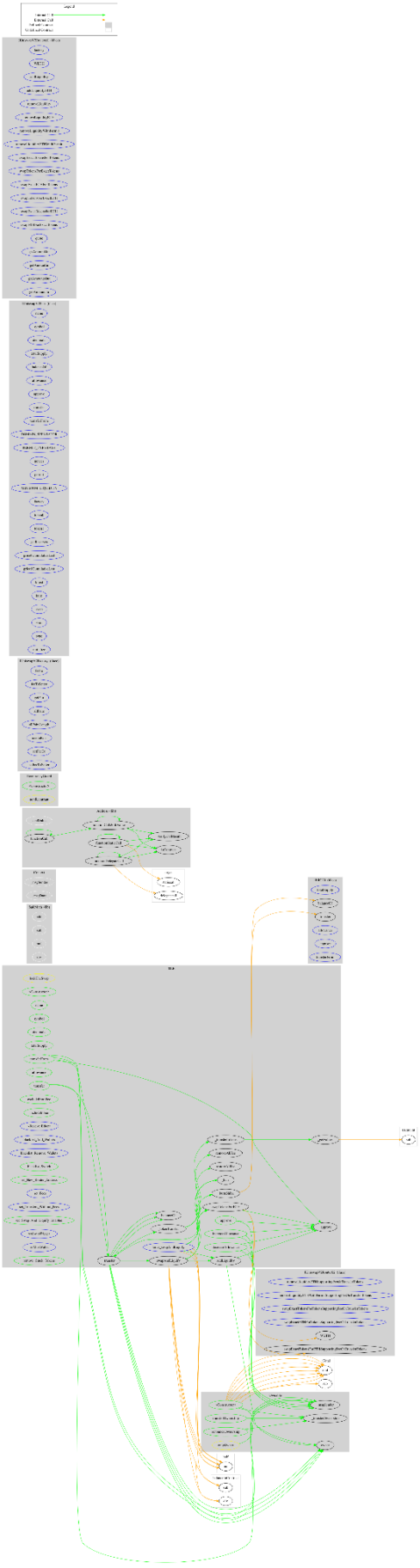
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-
	quote	External		-
	getAmountOut	External		-
	getAmountIn	External		-
	getAmountsOut	External		-
	getAmountsIn	External		-
IUniswapV2Router02	Interface	IUniswapV2 Router01		
	removeLiquidityETHSupportingFeeOnTransferTokens	External	✓	-
	removeLiquidityETHWithPermitSupportingFeeOnTransferTokens	External	✓	-
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External	✓	-
	swapExactETHForTokensSupportingFeeOnTransferTokens	External	Payable	-
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-

TLB	Implementation	Context, IERC20, Ownable, ReentrancyGuard		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	excludeFromFee	Public	✓	onlyOwner
	includeInFee	Public	✓	onlyOwner
		External	Payable	-
	blacklist_Add_Wallets	External	✓	onlyOwner
	blacklist_Remove_Wallets	External	✓	onlyOwner
	blacklist_Switch	Public	✓	onlyOwner
	removeAllFee	Private	✓	
	restoreAllFee	Private	✓	
	_approve	Private	✓	

	_transfer	Private	✓	
	swapAndLiquify	Private	✓	lockTheSwap
	force_swapAndLiquify	External	✓	nonReentrant onlyOwner
	swapTokensForETH	Public	✓	-
	addLiquidity	Private	✓	
	_burnShiba	Internal	✓	
	_burn	Internal	✓	
	_tokenTransfer	Private	✓	
	_transferTokens	Private	✓	
	_getValues	Private		
	set_New_Router_Address	Public	✓	onlyOwner
	_set_Fees	External	✓	onlyOwner
	set_Transfers_Without_Fees	External	✓	onlyOwner
	set_Swap_And_Liquify_Enabled	Public	✓	onlyOwner
	setSwapTrigger	External	✓	onlyOwner
	setMaxWallet	External	✓	onlyOwner
	remove_Stuck_Tokens	Public	✓	onlyOwner

Inheritance Graph

Flow Graph



Summary

TLB Token contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like stop transactions and massively blacklist addresses. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats. There is also a limit of max 15% fee.

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