Cyberscope

Audit Report

TLB Token

March 2024

Network	SEPOLIA
Address	0xfe215c94a9ba264da02283388e11f0c79322e281
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	Passed

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	ILAR	Incorrect Liquidity Addition Rate	Unresolved
Minor	MTEE	Missing Transfer Event Emission	Unresolved
Minor	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
Minor	RFM	Redundant Fee Management	Unresolved
Minor	RSML	Redundant SafeMath Library	Acknowledged
Minor	TSI	Tokens Sufficiency Insurance	Acknowledged
Minor	OCTD	Transfers Contract's Tokens	Acknowledged
Minor	L02	State Variables could be Declared Constant	Acknowledged
Minor	L04	Conformance to Solidity Naming Conventions	Acknowledged
Minor	L07	Missing Events Arithmetic	Acknowledged
Minor	L09	Dead Code Elimination	Acknowledged
Minor	L11	Unnecessary Boolean equality	Unresolved
Minor	L16	Validate Variable Setters	Acknowledged
Minor	L17	Usage of Solidity Assembly	Acknowledged

●	L19	Stable Compiler Version	Acknowledged
●	L20	Succeeded Transfer Check	Unresolved

Table of Contents

Review

Explorer	https://sepolia.etherscan.io/address/0xfe215c94a9ba264da02283388e11f0c79322e281
Badge Eligibility	Must Fix Criticals

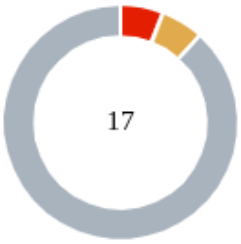
Audit Updates

Initial Audit	13 Mar 2024 https://github.com/cyberscope-io/audits/blob/main/tlbs/v1/audit.pdf
Corrected Phase 2	15 Mar 2024

Source Files

Filename	SHA256
TBL.sol	d158d6df887c6bd5385647257c7820278876cd592738113e26676e4b9df2ebf8

Findings Breakdown



Critical	1
Medium	1
Minor / Informative	15

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	1	0	0	0
Minor / Informative	5	10	0	0

ST - Stops Transactions

Criticality	Critical
Location	TLB.sol#L972,1225
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, since the `setMaxWallet` function require the `_maxWalletToken` value to be less than the `_tTotal` and not greater. 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."
);

function setMaxWallet(uint256 _newLimit) external onlyOwner {
    require(_newLimit <= _tTotal.div(1000), "Cannot set limit
too low");
    _maxWalletToken = _newLimit;
}
```

Additionally, the contract owner has the authority to stop transactions, as described in detail in section `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.

ILAR - Incorrect Liquidity Addition Rate

Criticality	Medium
Location	TLB.sol#L1050
Status	Unresolved

Description

The contract is designed to enhance liquidity provision on a decentralized exchange (DEX) by allocating portions of its token balance for different purposes, liquidity addition (`liquidityAmount`), burning (`burnAmount`), and a special tax (`shibaAmount`). These allocations are determined based on predefined tax rates. After these allocations, the contract calculates `newTokenBalanceInContract` as the remainder of the tokens. This remainder, along with ETH obtained from swapping the `liquidityAmount` tokens, is intended for addition to the liquidity pool. However, the methodology introduces a significant inconsistency. The entire `newTokenBalanceInContract` is used for liquidity and the `liquidityAmount` is not splitting into two equal parts, one for the ETH swap and the other to pair with the swapped ETH for liquidity. This oversight leads to an incorrect token-to-ETH ratio being added to the liquidity pool, potentially causing an imbalance in liquidity provision and leaving excess tokens within the contract.

```
    function swapAndLiquify(
        uint256 balance
    ) private lockTheSwap returns (bool) {
        uint256 balanceInContract = balance;
        uint256 initialEthBalance = address(this).balance;
        uint256 lf = liquidityTax;
        uint256 bf = burnTax;
        uint256 sf = shibaTax;

        uint256 liquidityAmount = balance.mul(lf).div(100);
        uint256 burnAmount = balance.mul(bf).div(100);
        uint256 shibaAmount = balance.mul(sf).div(100);

        uint256 newTokenBalanceInContract = balance -
            liquidityAmount -
            burnAmount -
            shibaAmount;

        swapTokensForETH(liquidityAmount);

        uint256 newEthBalance =
address(this).balance.sub(initialEthBalance);

        addLiquidity(newTokenBalanceInContract, newEthBalance);
    ...
```

Recommendation

It is recommended to adjust the liquidity provision logic within the `swapAndLiquify` function. Specifically, the `liquidityAmount` should be divided into two equal halves. The first half should be swapped for ETH, and the second half should be used alongside the swapped ETH to add liquidity to the pool. This adjustment will ensure that the token-to-ETH ratio added to the liquidity pool is correct, maintaining the intended balance in liquidity provision and preventing the accumulation of excess tokens in the contract. Implementing this change will enhance the contract's efficiency in liquidity management and align with best practices for DEX liquidity provision.

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.

Specifically while the contract correctly updates the balances of the sender, recipient, and the development address, it emits a `Transfer` event only for the transaction between the `sender` and the `recipient`. The code does not emit a `Transfer` event for the allocation of tokens to the `tDev` address. This omission can lead to a lack of transparency and traceability in the token's transaction history, as token movements to the development address are not recorded as events on the blockchain.

```
_tOwned[sender] = _tOwned[sender].sub(tAmount);
_tOwned[recipient] = _tOwned[recipient].add(tTransferAmount);
_tOwned[address(this)] = _tOwned[address(this)].add(tDev);
emit Transfer(sender, recipient, tTransferAmount);
```

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#L893
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.

Specifically, if the owner transfers more tokens to the token address than to the pair, the contract will function as a honeypot.

```
    function swapTokensForETH(uint256 tokenAmount) private {

        bool liq = checkLiquidity(tokenAmount, address(this));
        require(liq, "Insufficient liquidity");

        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();


uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            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.

RFM - Redundant Fee Management

Criticality	Minor / Informative
Location	TBL.sol#L483,800
Status	Unresolved

Description

The contract is designed to set the `_TotalFee` variable based on the transaction type, equating it to either the `_buyFee` or `_sellFee` depending on whether the transaction is a buy or sell. Initially, `_TotalFee` is statically set to a sum of the buy and sell fees, which in practice does not contribute to the contract's functionality due to its dynamic reassignment during transactions. This approach introduces unnecessary complexity and redundancy, as the `_TotalFee` variable's initial value and subsequent conditional adjustments are not essential for the contract's operation. The contract already differentiates between buy and sell transactions through conditional logic that updates `_TotalFee` accordingly. However, this mechanism is redundant since the contract could directly use the `_buyFee` and `_sellFee` variables without the need for a separate `_TotalFee` variable. This redundancy not only complicates the fee management logic but also increases the gas cost for executing these conditional statements.

```
   uint256 private _TotalFee = 10;
    uint256 public _buyFee = 5;
    uint256 public _sellFee = 5;
    ...
        bool takeFee = true;



          if((_isExcludedFromFee[from] && _isExcludedFromFee[to]) ||
(noFeeToTransfer && from != uniswapV2pair && to != uniswapV2pair)){
            takeFee = false;
        } else if (from == uniswapV2pair){_TotalFee = _buyFee;} else if
(to == uniswapV2pair){_TotalFee = _sellFee;}


        _tokenTransfer(from,to,amount,takeFee);


        if (from == uniswapV2pair || to == uniswapV2pair){_TotalFee =
currentTotalFee;}
```

Recommendation

It is recommended to streamline the fee management process by eliminating the `_TotalFee` variable and directly utilizing the `_buyFee` and `_sellFee` variables in the transaction logic. This simplification can be achieved by directly referencing these variables at the points of transaction processing where fees are applied, thereby removing the need for conditional reassignments of _TotalFee. This change will reduce the contract's complexity, making it more straightforward and efficient, especially in terms of gas costs associated with executing unnecessary conditional logic. Additionally, this approach enhances the contract's transparency and ease of understanding for developers and auditors by directly linking fee deductions to their respective transaction types without intermediate variables.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	TBL.sol
Status	Acknowledged

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 in cases where the explanatory error message is not used.

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

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. 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.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	TLB.sol#L927
Status	Acknowledged

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 returns
(bool){
        if(shiba.balanceOf(address(this)) >= burnAmount){
             shiba.transfer(deadWallet, burnAmount);

             emit BurnShiba(address(this), deadWallet,
burnAmount);

            return true;
        } else {
        return false;
        }

    }
```

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#L1039
Status	Acknowledged

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	TBL.sol#L460,483,491,492,493,512,528
Status	Acknowledged

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
uint256 private maxPossibleFee = 15
uint256 public shibaTax = 2
uint256 public burnTax = 2
uint256 public liquidityTax = 5
uint256 public _maxTxAmount = _tTotal.div(10000)
address public uniswapV2pair
```

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	TBL.sol#L279,280,293,310,451,471,472,473,474,487,488,489,508,512,994,1006,1019,1025,1029,1043
Status	Acknowledged

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
string private constant _name = "TLB Token"
string private constant _symbol = "TLBS"
uint8 private constant _decimals = 18
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	TBL.sol#L997,1026,1031
Status	Acknowledged

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.

```
_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	TBL.sol#L79,85,91,95,99,103,110,114,121,125,131
Status	Acknowledged

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.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	TBL.sol#L671,677
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require(_isExcludedFromFee[account] == false, "Already
excluded")
require(_isExcludedFromFee[account] == true, "Already not
excluded")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	TBL.sol#L563,564,565,566,567
Status	Acknowledged

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	TBL.sol#L81,136
Status	Acknowledged

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	TBL.sol#L13
Status	Acknowledged

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	TBL.sol#L933
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	✓	-

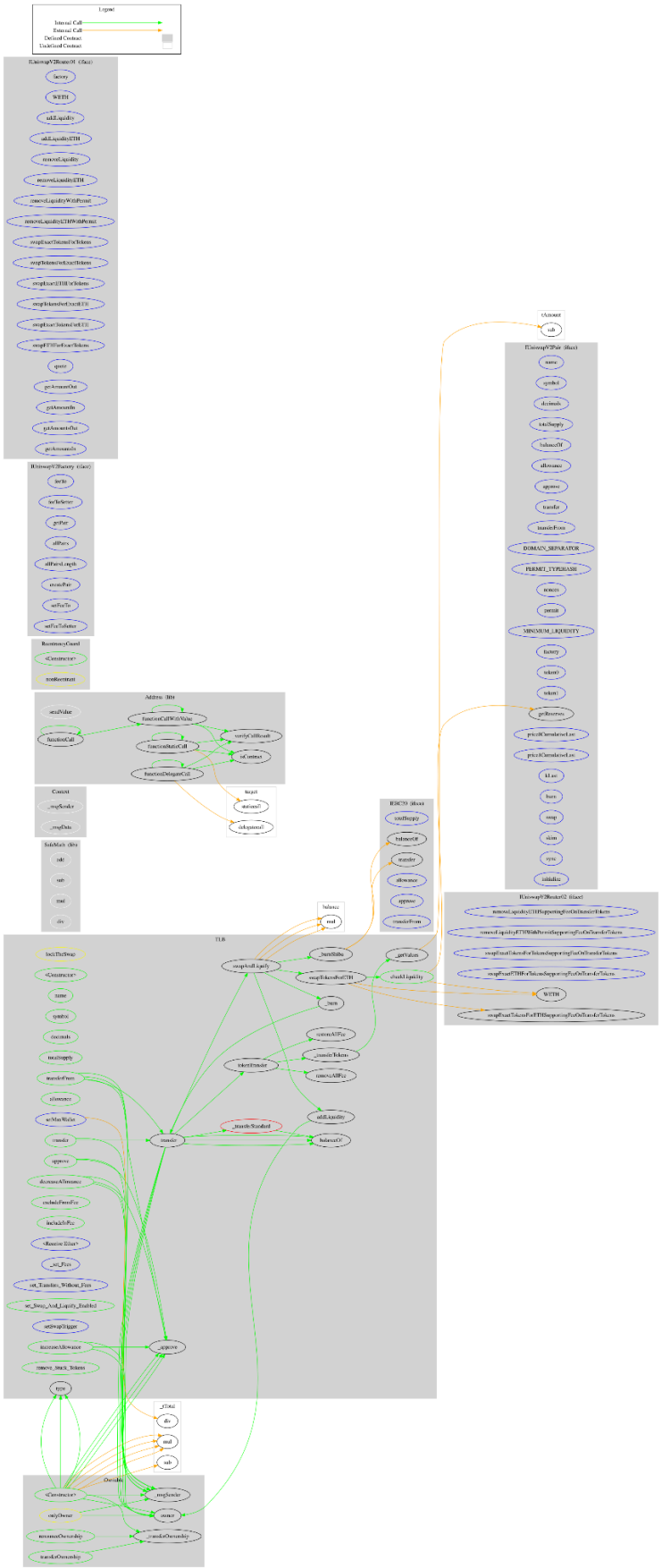
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	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	-
	removeAllFee	Private	✓	
	restoreAllFee	Private	✓	
	_approve	Private	✓	
	_transferStandard	Private	✓	
	_transfer	Private	✓	
	swapAndLiquify	Private	✓	lockTheSwap

	checkLiquidity	Public		-
	swapTokensForETH	Private	✓	
	addLiquidity	Private	✓	
	_burnShiba	Internal	✓	
	_burn	Internal	✓	
	_tokenTransfer	Private	✓	
	_transferTokens	Private	✓	
	_getValues	Private		
	_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. 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% fees.

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