Cyberscope

Audit Report

JLT Token

February 2024

Network	MATIC
Address	0xb8b3eef848b91affd4cb8747190d21c6b339fae8
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
●	FRV	Fee Restoration Vulnerability	Unresolved
●	ULF	Unused Liquidity Functionality	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	IFH	Inefficient Fee Handling	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	PVC	Price Volatility Concern	Unresolved
●	RES	Redundant Event Statement	Unresolved
●	RLF	Redundant Liquidity Functionality	Unresolved
●	RRS	Redundant Require Statement	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSW	Redundant Storage Writes	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
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved

Table of Contents

Analysis **1**
Diagnostics **2**
Table of Contents **4**
Review **6**
Audit Updates 6
Source Files 6
Findings Breakdown **7**
ST - Stops Transactions 8
Description 8
Recommendation 8
FRV - Fee Restoration Vulnerability 9
Description 9
Recommendation 10
ULF - Unused Liquidity Functionality 11
Description 11
Recommendation 11
IDI - Immutable Declaration Improvement 13
Description 13
Recommendation 13
IFH - Inefficient Fee Handling 14
Description 14
Recommendation 15
PLPI - Potential Liquidity Provision Inadequacy 16
Description 16
Recommendation 16
PTRP - Potential Transfer Revert Propagation 18
Description 18
Recommendation 18
PVC - Price Volatility Concern 19
Description 19
Recommendation 20
RES - Redundant Event Statement 21
Description 21
Recommendation 21
RLF - Redundant Liquidity Functionality 22
Description 22
Recommendation 23
RRS - Redundant Require Statement 24
Description 24

Recommendation 24
RSML - Redundant SafeMath Library 25
Description 25
Recommendation 25
RSW - Redundant Storage Writes 26
Description 26
Recommendation 26
L02 - State Variables could be Declared Constant 27
Description 27
Recommendation 27
L04 - Conformance to Solidity Naming Conventions 28
Description 28
Recommendation 28
L07 - Missing Events Arithmetic 30
Description 30
Recommendation 30
L09 - Dead Code Elimination 31
Description 31
Recommendation 31
L13 - Divide before Multiply Operation 33
Description 33
Recommendation 33
L16 - Validate Variable Setters 34
Description 34
Recommendation 34
L17 - Usage of Solidity Assembly 35
Description 35
Recommendation 35
L19 - Stable Compiler Version 36
Description 36
Recommendation 36
Functions Analysis 37
Inheritance Graph 45
Flow Graph 46
Summary 47
Disclaimer 48
About Cyberscope 49

Review

Contract Name	JLT
Compiler Version	v0.8.22+commit.4fc1097e
Optimization	200 runs
Explorer	https://polygonscan.com/address/0xb8b3eef848b91affd4cb8747190d21c6b339fae8
Address	0xb8b3eef848b91affd4cb8747190d21c6b339fae8
Network	MATIC
Symbol	JLT
Decimals	18
Total Supply	10,000,000,000
Badge Eligibility	Yes

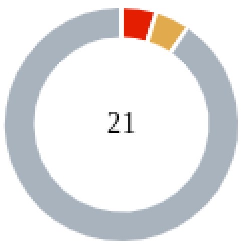
Audit Updates

Initial Audit	05 Feb 2024

Source Files

Filename	SHA256
JLT.sol	752695b2619f4a463fd1730d3eb5bd0479206b8f3da9b3b917bcf40a72ab3f0b

Findings Breakdown



●	Critical	1
●	Medium	1
●	Minor / Informative	19

	Severity	Unresolved	Acknowledged	Resolved	Other
●	Critical	1	0	0	0
●	Medium	1	0	0	0
●	Minor / Informative	19	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	JLT.sol
Status	Unresolved

Description

The contract owner has the authority to stop the sales for all users excluding the owner, as described in detail in sections `PTRP` and `PVC`. As a result, the contract might operate as a honeypot.

Recommendation

The team is strongly encouraged to adhere to the recommendations outlined in the respective sections. By doing so, the contract can eliminate any potential of operating as a honeypot.

FRV - Fee Restoration Vulnerability

Criticality	Medium
Location	JLT.sol#L914,1018
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(_taxFee == 0 && _liquidityFee == 0 && _treasuryFee==0 &&
_burnFee==0) return;

        _previousTaxFee = _taxFee;
        _previousLiquidityFee = _liquidityFee;
        _previousBurnFee = _burnFee;
        _previousTreasuryFee = _treasuryFee;

        _taxFee = 0;
        _liquidityFee = 0;
        _treasuryFee = 0;
        _burnFee = 0;
    }

    function restoreAllFee() private {
       _taxFee = _previousTaxFee;
       _liquidityFee = _previousLiquidityFee;
       _burnFee = _previousBurnFee;
       _treasuryFee = _previousTreasuryFee;
    }

    function _tokenTransfer(address sender, address recipient, uint256
amount) private
    {
        if(_isExcludedFromFee[sender] || _isExcludedFromFee[recipient])
        {
           removeAllFee();
        }
        ...
        if(_isExcludedFromFee[sender] || _isExcludedFromFee[recipient])
        {
            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.

ULF - Unused Liquidity Functionality

Criticality	Minor / Informative
Location	JLT.sol#L1002
Status	Unresolved

Description

The contract contains the `addLiquidity` function designed to facilitate the addition of liquidity to a Uniswap V2 pool by approving token transfers and interacting with the UniswapV2Router's `addLiquidityETH` method. The function is marked as private, indicating that it can only be called from within the contract itself. Despite its presence and the potential utility it offers for liquidity management, the `addLiquidity` function is never invoked in any part of the contract's code. This omission results in the liquidity addition functionality being effectively dormant and unutilized within the contract's operational framework.

```
    function addLiquidity(uint256 tokenAmount, uint256
ethAmount) private {
        // approve token transfer to cover all possible
scenarios
        _approve(address(this), address(uniswapV2Router),
tokenAmount);

        // add the liquidity
        uniswapV2Router.addLiquidityETH{value: ethAmount}(
            address(this),
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            deadAddress,
            block.timestamp
        );
    }
```

Recommendation

It is recommended to reconsider the contract's code implementation concerning the `addLiquidity` function. If the intended functionality of the contract includes the capability to add liquidity to a Uniswap V2 pool, it is crucial to ensure that the `addLiquidity` function is appropriately integrated and utilized within the contract's workflow. This integration may involve adding calls to addLiquidity at relevant points in the contract or developing additional functions that leverage this capability for liquidity management purposes. Conversely, if the addition of liquidity is not a required feature for the contract's intended operations, it may be prudent to remove the `addLiquidity` function from the codebase. This action would streamline the contract, eliminating unnecessary code and reducing potential confusion regarding the contract's capabilities and intended use cases.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	JLT.sol#L725
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.

```
uniswapV2Pair
```

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.

IFH - Inefficient Fee Handling

Criticality	Minor / Informative
Location	JLT.sol#L1020,1040
Status	Unresolved

Description

The contract is currently implementing fee management logic within the `_tokenTransfer` function through two separate `if` conditions to determine whether to call `removeAllFee` or `restoreAllFee`. The first set of conditions checks if either the sender or recipient is excluded from fees, and if so, it calls `removeAllFee`. Similarly, it checks if neither the sender nor the recipient is the UniswapV2Pair, and if this condition is met, it again calls `removeAllFee`. This process is then repeated in a similar manner for `restoreAllFee`. This approach results in redundant code and unnecessary complexity, as each condition is evaluated independently despite their actions being related to the fee management process. This redundancy could lead to potential inefficiencies in gas usage and makes the code harder to read and maintain.

```
    function _tokenTransfer(address sender, address recipient,
uint256 amount) private
    {
        if(_isExcludedFromFee[sender] ||
_isExcludedFromFee[recipient])
        {
           removeAllFee();
        }
        if(sender != uniswapV2Pair && recipient != uniswapV2Pair){
            removeAllFee();
        }

        ...

        if(_isExcludedFromFee[sender] ||
_isExcludedFromFee[recipient])
        {
            restoreAllFee();
        }
        if(sender != uniswapV2Pair && recipient != uniswapV2Pair){
            restoreAllFee();
        }
    }
```

Recommendation

It is recommended to consolidate the fee management logic within the `_tokenTransfer` function by combining the conditions for removing and restoring fees into single `if` statements. This can be achieved by utilizing logical OR (||) operators to merge conditions that trigger the same action. For instance, the conditions for calling `removeAllFee` can be combined into one if statement using the logical OR operator to check if either the sender or recipient is excluded from fees or if neither party is the UniswapV2Pair. The same approach can be applied to the conditions for `restoreAllFee`. This consolidation will simplify the logic, making the code more efficient and easier to understand. Additionally, it will reduce the gas cost associated with executing these conditional checks by minimizing the number of operations performed. Simplifying the fee management process in this manner will enhance the contract's overall efficiency and maintainability.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	JLT.sol#L984
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) private {
        // generate the uniswap pair path of token -> weth
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();

        _approve(address(this), address(uniswapV2Router),
tokenAmount);

        // make the swap

uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0, // accept any amount of ETH
            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.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	JLT.sol#L979
Status	Unresolved

Description

The contract sends funds to a `treasuryWallet` as part of the transfer flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
treasuryWallet.transfer(address(this).balance);
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	JLT.sol#L955,1128
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `numTokensSellToAddToLiquidity` sets a threshold where the contract will trigger the swap functionality. If the variable is set to a big number, then the contract will swap a huge amount of tokens for ETH.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

```
 function _transfer(
        address from,
        address to,
        uint256 amount
    ) private {
        ...
        uint256 contractTokenBalance =
balanceOf(address(this));
        bool overMinTokenBalance = contractTokenBalance >=
numTokensSellToAddToLiquidity;
        if (
            overMinTokenBalance &&
            !inSwapAndLiquify &&
            to == uniswapV2Pair &&
            swapAndLiquifyEnabled &&
            _treasuryFee > 0
        ) {
            //send bnb to marketing wallet
            swapAndLiquify(contractTokenBalance);
        }

        ...
        }

    function setNumTokensSellToAddToLiquidity(uint256
newAmount) external onlyOwner() {
        numTokensSellToAddToLiquidity = newAmount;
    }
```

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

RES - Redundant Event Statement

Criticality	Minor / Informative
Location	JLT.sol#L706
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The `MinTokensBeforeSwapUpdated` event statement is not used in the contract's implemantation.

```
event MinTokensBeforeSwapUpdated(uint256 minTokensBeforeSwap);
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it. It is recommend removing the unused event statement from the contract.

RLF - Redundant Liquidity Functionality

Criticality	Minor / Informative
Location	JLT.sol#L893,1073
Status	Unresolved

Description

The contract contain the `_takeLiquidity` and `takeTreasury`. However both functions are set the `_rOwned` variable, and moreover the `_liquidityFee` is permanently set to zero, with no available method to modify or update this value post-deployment. This permanent zero setting effectively nullifies the intended functionality of the `_liquidityFee`, rendering any mechanisms that rely on this fee for liquidity provision or adjustment inoperative. This flaw not only compromises the flexibility and adaptability of the contract in responding to liquidity needs but also raises concerns about the contract's overall economic design.

```
    function _takeLiquidity(uint256 tLiquidity) private {
         uint256 currentRate =  _getRate();
         uint256 rLiquidity = tLiquidity.mul(currentRate);
         _rOwned[address(this)] = _rOwned[address(this)].add(rLiquidity);
         if(_isExcluded[address(this)])
             _tOwned[address(this)] =
_tOwned[address(this)].add(tLiquidity);
    }

    function takeTreasury(address sender, uint256 tTransferAmount,
uint256 rTransferAmount, uint256 tAmount) private
    returns (uint256, uint256) {
         if(_treasuryFee==0) {  return(tTransferAmount, rTransferAmount);
}
         uint256 tTreasury = tAmount.div(100).mul(_treasuryFee);
         uint256 rTreasury = tTreasury.mul(_getRate());
         rTransferAmount = rTransferAmount.sub(rTreasury);
         tTransferAmount = tTransferAmount.sub(tTreasury);
         _rOwned[address(this)] = _rOwned[address(this)].add(rTreasury);
         emit Transfer(sender, address(this), tTreasury);
         return(tTransferAmount, rTransferAmount);
    }
```

Recommendation

It is recommended to implement a functionality that enables the modification of the `_liquidityFee` value, thereby reinstating its intended purpose and ensuring the contract's capability to effectively manage liquidity. This enhancement could be achieved by introducing a setter function, granting the contract owner or an authorized governance body the authority to modify the `_liquidityFee` in accordance with the evolving economic conditions and liquidity demands of the contract. Conversely, if the `_liquidityFee` is unnecessary for the contract's operations, it is advisable to consider its removal to streamline the contract's functionality and eliminate any redundant features.

RRS - Redundant Require Statement

Criticality	Minor / Informative
Location	JLT.sol#L184
Status	Unresolved

Description

The contract utilizes a `require` statement within the `add` function aiming to prevent overflow errors. This function is designed based on the SafeMath library's principles. In Solidity version 0.8.0 and later, arithmetic operations revert on overflow and underflow, making the overflow check within the function redundant. This redundancy could lead to extra gas costs and increased complexity without providing additional security.

```
function add(uint256 a, uint256 b) internal pure returns
(uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");
        return c;
}
```

Recommendation

It is recommended to remove the `require` statement from the add function since the contract is using a Solidity pragma version equal to or greater than 0.8.0. By doing so, the contract will leverage the built-in overflow and underflow checks provided by the Solidity language itself, simplifying the code and reducing gas consumption. This change will uphold the contract's integrity in handling arithmetic operations while optimizing for efficiency and cost-effectiveness.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	JLT.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	JLT.sol#L1109,1113
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 includeInFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = false;
    }

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

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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	JLT.sol#L679,683,684,685,695
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 private _tTotal = 10000000000 * (10**18)
string private _name = "JLT"
string private _symbol = "JLT"
uint8 private _decimals = 18
address public deadAddress =
0x000000000000000000000000000000000000dEaD
```

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	JLT.sol#L488,489,505,526,687,693,697,901,907,1132
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);
uint256 public _taxFee = 1
uint256 public _burnFee = 1
uint256 public _treasuryFee = 1
uint256 _amount
bool _enabled
```

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	JLT.sol#L1123,1129
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.

```
_taxFee = taxFee;
_treasuryFee = treasuryFee;
_burnFee = burnFee;
numTokensSellToAddToLiquidity = newAmount
```

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	JLT.sol#L335,362,388,398,413,423,428,794,798,1002
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) {
        // According to EIP-1052, 0x0 is the value returned for
not-yet created accounts
        // and
0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a
470 is returned
        // for accounts without code, i.e. `keccak256('')`
        bytes32 codehash;
        bytes32 accountHash =
0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a
470;
        // solhint-disable-next-line no-inline-assembly
        assembly { codehash := extcodehash(account) }
        return (codehash != accountHash && codehash != 0x0);
    }

...
```

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	JLT.sol#L1064,1076
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 tBurn = tAmount.div(100).mul(_burnFee)
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	JLT.sol#L1118
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.

```
treasuryWallet = newWallet
```

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	JLT.sol#L342,441
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 { codehash := extcodehash(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	JLT.sol#L3
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 ^0.8.9;
```

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.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner

	_transferOwnership	Internal	✓	
SafeMath	Library			
	add	Internal		
	sub	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	div	Internal		
	mod	Internal		
	mod	Internal		
Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	_functionCallWithValue	Private	✓	
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	✓	-

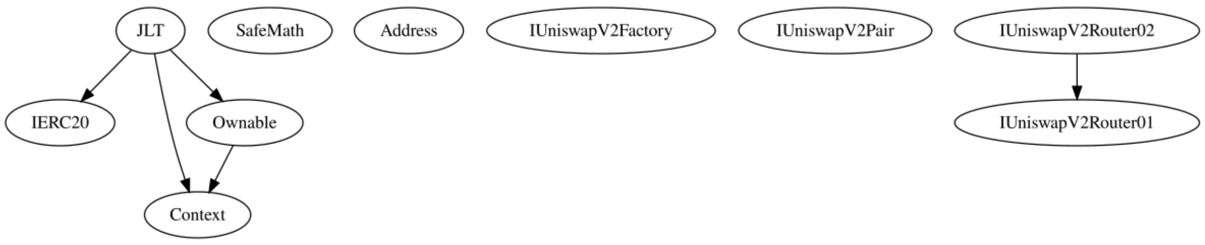
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-
	quote	External		-
	getAmountOut	External		-
	getAmountIn	External		-
	getAmountsOut	External		-
	getAmountsIn	External		-
IUniswapV2Router02	Interface	IUniswapV2Router01		
	removeLiquidityETHSupportingFeeOnTransferTokens	External	✓	-
	removeLiquidityETHWithPermitSupportingFeeOnTransferTokens	External	✓	-
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External	✓	-
	swapExactETHForTokensSupportingFeeOnTransferTokens	External	Payable	-
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-
JLT	Implementation	Context, IERC20, Ownable		
		Public	✓	-
	name	Public		-

	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	isExcludedFromReward	Public		-
	totalFees	Private		
	reflectionFromToken	Private		
	tokenFromReflection	Private		
	excludeFromReward	Public	✓	onlyOwner
	includeInReward	External	✓	onlyOwner
	_transferBothExcluded	Private	✓	
	_reflectFee	Private	✓	
	_getValues	Private		
	_getTValues	Private		
	_getRValues	Private		
	_getRate	Private		
	_getCurrentSupply	Private		

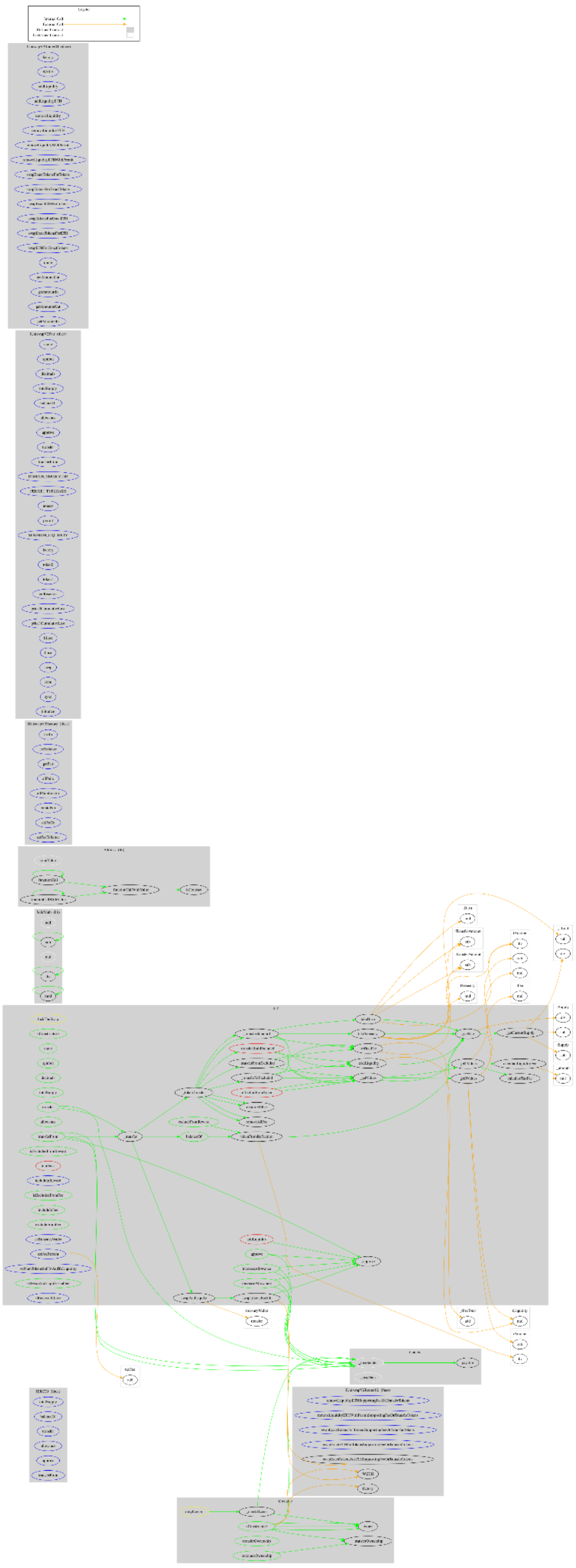
	_takeLiquidity	Private	✓	
	calculateLiquidityFee	Private		
	calculateTaxFee	Private		
	removeAllFee	Private	✓	
	restoreAllFee	Private	✓	
	_approve	Private	✓	
	_transfer	Private	✓	
	swapAndLiquify	Private	✓	lockTheSwap
	swapTokensForEth	Private	✓	
	addLiquidity	Private	✓	
	_tokenTransfer	Private	✓	
	_transferStandard	Private	✓	
	takeBurn	Private	✓	
	takeTreasury	Private	✓	
	_transferToExcluded	Private	✓	
	_transferFromExcluded	Private	✓	
	isExcludedFromFee	Public		-
	includeInFee	Public	✓	onlyOwner
	excludeFromFee	Public	✓	onlyOwner
	setTreasuryWallet	External	✓	onlyOwner
	setFeePercent	External	✓	onlyOwner
	setNumTokensSellToAddToLiquidity	External	✓	onlyOwner
	setSwapAndLiquifyEnabled	Public	✓	onlyOwner

		External	Payable	-

Inheritance Graph



Flow Graph



Summary

JLT 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 20% fees.

Disclaimer

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