Cyberscope

Audit Report

TreasuryAi

December 2024

Network	BSC
Address	0x9eCEc886c1128C129Fc8BDDf1F4F938120733217
Audited by	© cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	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
●	CCR	Contract Centralization Risk	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	ITA	Ineffective Tracker Allocation	Unresolved
●	ITD	Inefficient Token Distribution	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	NCE	Non Compliant ERC20	Unresolved
●	PVC	Price Volatility Concern	Unresolved
●	RAC	Redundant Allocation Check	Unresolved
●	RC	Redundant Checks	Unresolved
●	RLF	Redundant Logic Functionality	Unresolved
●	RSD	Redundant Swap Duplication	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved

●	L09	Dead Code Elimination	Unresolved

Table of Contents

Analysis **1**
Diagnostics **2**
Table of Contents **4**
Risk Classification **6**
Review **7**
Audit Updates 7
Source Files 8
Findings Breakdown **9**
CCR - Contract Centralization Risk 10
Description 10
Recommendation 10
DDP - Decimal Division Precision 11
Description 11
Recommendation 11
ITA - Ineffective Tracker Allocation 12
Description 12
Recommendation 14
ITD - Inefficient Token Distribution 15
Description 15
Recommendation 16
MEE - Missing Events Emission 17
Description 17
Recommendation 17
NCE - Non Compliant ERC20 18
Description 18
Recommendation 18
PVC - Price Volatility Concern 19
Description 19
Recommendation 19
RAC - Redundant Allocation Check 20
Description 20
Recommendation 20
RC - Redundant Checks 21
Description 21
Recommendation 22
RLF - Redundant Logic Functionality 23
Description 23
Recommendation 24
RSD - Redundant Swap Duplication 25

Description 25
Recommendation 25
L02 - State Variables could be Declared Constant 26
Description 26
Recommendation 26
L04 - Conformance to Solidity Naming Conventions 27
Description 27
Recommendation 28
L05 - Unused State Variable 29
Description 29
Recommendation 29
L09 - Dead Code Elimination 30
Description 30
Recommendation 31
Functions Analysis 32
Inheritance Graph 35
Flow Graph 36
Summary 37
Disclaimer 38
About Cyberscope 39

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Contract Name	TAI
Compiler Version	v0.8.19+commit.7dd6d404
Optimization	200 runs
Explorer	https://bscscan.com/address/0x9ecec886c1128c129fc8bddf1f4f938120733217
Address	0x9ecec886c1128c129fc8bddf1f4f938120733217
Network	BSC
Symbol	TAI
Decimals	18
Total Supply	1,000,000,000
Badge Eligibility	Yes

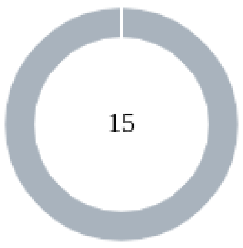
Audit Updates

Initial Audit	28 Nov 2024
Corrected Phase 2	03 Dec 2024
Corrected Phase 3	04 Dec 2024

Source Files

Filename	SHA256
TAI.sol	4b1c40f9707d49e2758d346fbca06c181f19d0385c52ddc4c1466c4db02b25df

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	15

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	15	0	0	0

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	TAI.sol#L317,333,338,342
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function setNoFeeWallet(address account, bool enabled) public
onlyOwner {}
function changeSwapThreshold(uint256 newSwapThreshold) external
onlyOwner {}
function excludeFromFees(address account, bool excluded) external
onlyOwner {}
function setExcludeFromBurn(address account, bool excluded)
external onlyOwner {}
function setExcludeFromBurnTrigger(address account, bool excluded)
external onlyOwner {}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	TAI.sol#L421
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 buyAmount = (swapAmount * buyAllocation) / 100;
uint256 sellAmount = (swapAmount * sellAllocation) / 100;
uint256 liquidityAmount = (swapAmount * liquidityAllocation) / 100;
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

ITA - Ineffective Tracker Allocation

Criticality	Minor / Informative
Location	TAI.sol#L, 495,569,504
Status	Unresolved

Description

The contract initializes the balance of the `trackerAddress` to facilitate a staged burning process for tokens. However, once tokens are sent to the `trackerAddress` , they are planned solely for burning and are effectively removed from circulation, even though they are not immediately burned. This creates a situation where these tokens, while not technically burned, cannot be considered part of the circulating supply. The presence of these tokens in a limbo state could lead to confusion or misrepresentation in supply metrics.

```
   function _sendTokensTotrackerAddress() internal {
        uint256 amountToSend = 900_000_000 * 10**18;
        require(balanceOf(msg.sender) >= amountToSend,
"Insufficient balance");
        balance[msg.sender] -= amountToSend;
        balance[trackerAddress] += amountToSend;
        emit Transfer(msg.sender, trackerAddress, amountToSend);
    }

...
    function burnTokens(uint256 amount, bool isBuy) internal {
        uint256 burnPercentage = isBuy ? buyBurnPercentage :
sellBurnPercentage;
        uint256 burnAmount = (amount * burnPercentage) / 100;

        if (_excludeFromBurnTrigger[msg.sender] ||
_excludeFromBurnTrigger[tx.origin]) {
            burnAmount = 0;
        }

        if (burnAmount > 0) {
            // Deduct burn amount from trackerAddress balance
            require(trackerAddress != address(0) &&
balance[trackerAddress] >= burnAmount, "Burn exceeds tracker
balance");
            balance[trackerAddress] -= burnAmount;

            // Decrease the total supply accurately
            _totalSupply -= burnAmount;

            // Send burned tokens to DEAD address
            balance[DEAD] += burnAmount;
            ...
            }
...

contract TrackerAddress {
    constructor() {}
}
```

Recommendation

It is recommended to either directly burn the allocated tokens instead of transferring them to the `trackerAddress` or explicitly account for and disclose these tokens as non-circulating supply in the contract documentation and any external communications. This ensures accurate supply calculations and improves clarity about the token’s distribution and usage.

ITD - Inefficient Token Distribution

Criticality	Minor / Informative
Location	TAI.sol#L260,586
Status	Unresolved

Description

The contract is found to initially allocate the entire token supply to the `msg.sender` before redistributing specific amounts to other addresses such as the tracker address and the contract's own address. This approach results in additional state updates and logical complexity. Instead of directly assigning the required amounts to their respective addresses during the initial distribution, the contract first increases the balance of `msg.sender` and subsequently deducts the amounts for redistribution. This unnecessarily complicates the logic and increases gas costs by performing redundant balance adjustments.

```
constructor() {
       ...
        balance[msg.sender] = _totalSupply;
        emit Transfer(address(0), msg.sender, _totalSupply);

        ...

        _createTrackerAddress();
        _sendTokensTotrackerAddress();
    }

    function _sendTokensTotrackerAddress() internal {
        uint256 amountToSend = 900_000_000 * 10**18;
        require(balanceOf(msg.sender) >= amountToSend,
"Insufficient balance");
        balance[msg.sender] -= amountToSend;
        balance[trackerAddress] += amountToSend;
        emit Transfer(msg.sender, trackerAddress, amountToSend);
    }
```

Recommendation

It is recommended to simplify the token distribution logic by directly transferring the designated amounts to the appropriate addresses during the initial allocation in the constructor. This optimization will reduce unnecessary state updates, simplify the contract's logic, and improve gas efficiency.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	TAI.sol#L318,345
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
   function setNoFeeWallet(address account, bool enabled) public
onlyOwner {
        require(account != address(0), "AI: Account is zero
address");
        _noFee[account] = enabled;
    }

    function setExcludeFromBurn(address account, bool excluded)
external onlyOwner {
        excludeFromBurn[account] = excluded;
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

NCE - Non Compliant ERC20

Criticality	Minor / Informative
Location	TAI.sol#L367
Status	Unresolved

Description

The contract implements an ERC20 token, but it does not fully comply with the ERC20 standard due to the restriction on zero-amount transfers. Specifically, the `_transfer` function includes the following check:

```
require(amount > 0, "Transfer amount must be greater than zero");
```

This restriction contradicts the ERC20 standard, which allows transfers of zero tokens. According to the ERC20 specification, a token contract **must** allow for zero-value transfers.

Recommendation

Remove the check to allow for zero-amount transfers, ensuring full compliance with the ERC20 standard.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	TAI.sol#L378
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variables `swapThreshold` and `feeCollected` set 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.

```
if (is_sell(from, to) && !inSwap && canSwap(from, to) &&
!isLiquidityAddition) {
    uint256 contractTokenBalance = balanceOf(address(this));
    if (contractTokenBalance >= swapThreshold) {
        uint256 swapAmount = feeCollected;
        if (swapAmount > 0) {
            distributeFees();
        }
    }
```

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.

RAC - Redundant Allocation Check

Criticality	Minor / Informative
Location	TAI.sol#L188,245
Status	Unresolved

Description

The contract is designed with fixed allocation values for buy, sell, and liquidity, which collectively sum to 100. This deterministic setup ensures that the `require` statement verifying the total allocations always evaluates to true, making it redundant. Consequently, the `require` statement serves no functional purpose and unnecessarily consumes gas during execution.

```
uint256 private buyAllocation = 40;
uint256 private sellAllocation = 40;
uint256 private liquidityAllocation = 20;

...
require(buyAllocation + sellAllocation + liquidityAllocation ==
100, "AI: Must equal to 100%");
```

Recommendation

It is recommended to remove the `require` statement validating that the total allocations equal 100, as this condition is inherently guaranteed by the fixed values assigned to the allocation variables. This optimization will reduce gas costs and improve the contract's efficiency.

RC - Redundant Checks

Criticality	Minor / Informative
Location	TAI.sol#L374,426,439,466
Status	Unresolved

Description

The contract is implementing redundant zero checks within internal functions such as `internalSwap` and `swapAndLiquify` . These checks are unnecessary because the calling functions already ensure that the input amounts are non-zero before invoking these internal functions. This results in additional, redundant computational steps that do not contribute to the functionality or security of the contract. Additionally, the `checkAndAdjustCriticalParameters` function includes redundant balance checks for the `trackerAddress` , even though the calling function already verifies the balance condition before invoking it. This duplication further increases gas costs and reduces code readability.

```
       if (buyAmount > 0 || sellAmount > 0) {
            internalSwap(buyAmount + sellAmount);
        }

        // Handle liquidity fees if allocation is non-zero
        if (liquidityAmount > 0) {
            swapAndLiquify(liquidityAmount);
        }
        ....

    function internalSwap(uint256 amount) internal {
        if (amount == 0) return;
        ...
        }

    function swapAndLiquify(uint256 amount) internal {
        if (amount == 0) return;
        ...
        }
    ...
        if (balance[trackerAddress] <= 1_000_000 * 10**18) {
            checkAndAdjustCriticalParameters();
        }
     function checkAndAdjustCriticalParameters() internal {
        if (balance[trackerAddress] <= 1_000_000 * 10**18 &&
!autoAdjusted["CriticalParameters"]) {
        ...
        }
```

Recommendation

It is recommended to remove the redundant zero checks from the internal functions and the duplicate balance checks from `checkAndAdjustCriticalParameters` to streamline the code, reduce gas costs, and enhance readability. Ensure that the validation logic remains robust and centralized within the calling functions to maintain security and prevent unintended behavior.

RLF - Redundant Logic Functionality

Criticality	Minor / Informative
Location	TAI.sol#L170,199,314,322,607
Status	Unresolved

Description

The contract contains functions and functionality that are redundant due to always-true logic or conditions that cannot be realistically false. For example, the `enableTrading` function becomes redundant once `isTradingEnabled` is set to true, as it cannot logically revert to false. Similarly, the `canSwap` function checks conditions that are either fixed or have no practical scenarios where they would evaluate differently. Moreover the `isNoFeeWallet` is not utilized within the code and mappings such as `isPresaleAddress` and `excludeFromBurn` are included but do not have clear or meaningful applications in the contract’s current logic.

```
   mapping(address => bool) private isPresaleAddress;
    mapping(address => bool) private excludeFromBurn;
    ...
    bool private canSwapFees = true;
    ...
    bool public isTradingEnabled = true;
    ...
    function enableTrading() external onlyOwner {
        require(!isTradingEnabled, "Trading already enabled");
        isTradingEnabled = true;
        emit _enableTrading();
    }
    ...
    function isNoFeeWallet(address account) external view returns
(bool) {
        return _noFee[account];
    }
    ...
    function canSwap(address ins, address out) internal view
returns (bool) {
        return canSwapFees && !isPresaleAddress[ins] &&
!isPresaleAddress[out];
    }
```

Recommendation

It is recommended to remove redundant functions, variables, and mappings that do not serve a practical use case or add unnecessary complexity to the contract. Simplifying the contract by eliminating these elements will improve readability, reduce deployment costs, and make the codebase easier to maintain.

RSD - Redundant Swap Duplication

Criticality	Minor / Informative
Location	TAI.sol#L426
Status	Unresolved

Description

The contract contains multiple swap methods that individually perform token swaps and transfer promotional amounts to specific addresses and features. This redundant duplication of code introduces unnecessary complexity and increases dramatically the gas consumption. By consolidating these operations into a single swap method, the contract can achieve better code readability, reduce gas costs, and improve overall efficiency.

```
if (buyAmount > 0 || sellAmount > 0) {
    internalSwap(buyAmount + sellAmount);
}

// Handle liquidity fees if allocation is non-zero
if (liquidityAmount > 0) {
    swapAndLiquify(liquidityAmount);
}
```

Recommendation

A more optimized approach could be adopted to perform the token swap operation once for the total amount of tokens and distribute the proportional amounts to the corresponding addresses, eliminating the need for separate swaps.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	TAI.sol#L184,192,193,194
Status	Unresolved

Description

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

```
uint256 public transferfee = 0
uint256 private buyAllocation = 40
uint256 private sellAllocation = 40
uint256 private liquidityAllocation = 20
```

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	TAI.sol#L77,197,198,200,216,217,218,219,603,607
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 WETH() external pure returns (address);
string constant private _name = "TreasuryAi"
string constant private _symbol = "TAI"
uint8 constant private _decimals = 18
event _enableTrading();
event _setPresaleAddress(address account, bool enabled);
event _changeThreshold(uint256 newThreshold);
...
function is_buy(address ins, address out) internal view returns
(bool) {
        return !isLpPair[out] && isLpPair[ins];
    }

function is_sell(address ins, address out) internal view returns
(bool) {
        return isLpPair[out] && !isLpPair[ins];
    }
```

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/stable/style-guide.html#naming-conventions.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	TAI.sol#L190,210,211
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
address payable immutable private devWallet =
payable(0x865942Cedb9AE5119264909E38302167c143DF6b)
uint256 private initialBuyBurnPercentage = buyBurnPercentage
uint256 private initialSellBurnPercentage = sellBurnPercentage
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	TAI.sol#L563,579,581
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 _burn(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: burn from the zero
address");

        _beforeTokenTransfer(account, address(0), amount);

        uint256 accountBalance = balanceOf(account);
...
        _totalSupply -= amount;

        emit Transfer(account, address(0), amount);
        emit Burn(account, amount);

        _afterTokenTransfer(account, address(0), amount);
    }

...
```

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.

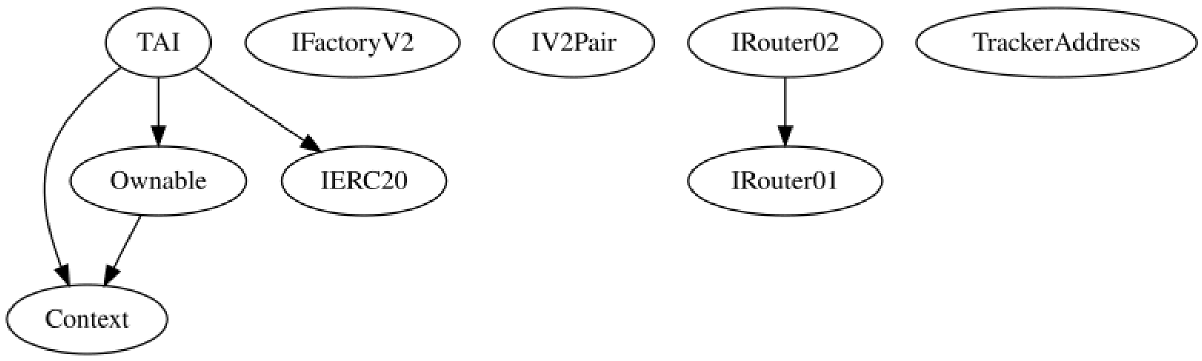
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
TAI	Implementation	Context, Ownable, IERC20		
		Public	✓	-
		External	Payable	-
	totalSupply	External		-
	decimals	External		-
	symbol	External		-
	name	External		-
	getOwner	External		-
	allowance	External		-
	balanceOf	Public		-
	transfer	Public	✓	-
	approve	External	✓	-
	transferFrom	External	✓	-
	isNoFeeWallet	External		-
	setNoFeeWallet	Public	✓	onlyOwner
	enableTrading	External	✓	onlyOwner
	changeSwapThreshold	External	✓	onlyOwner
	excludeFromFees	External	✓	onlyOwner

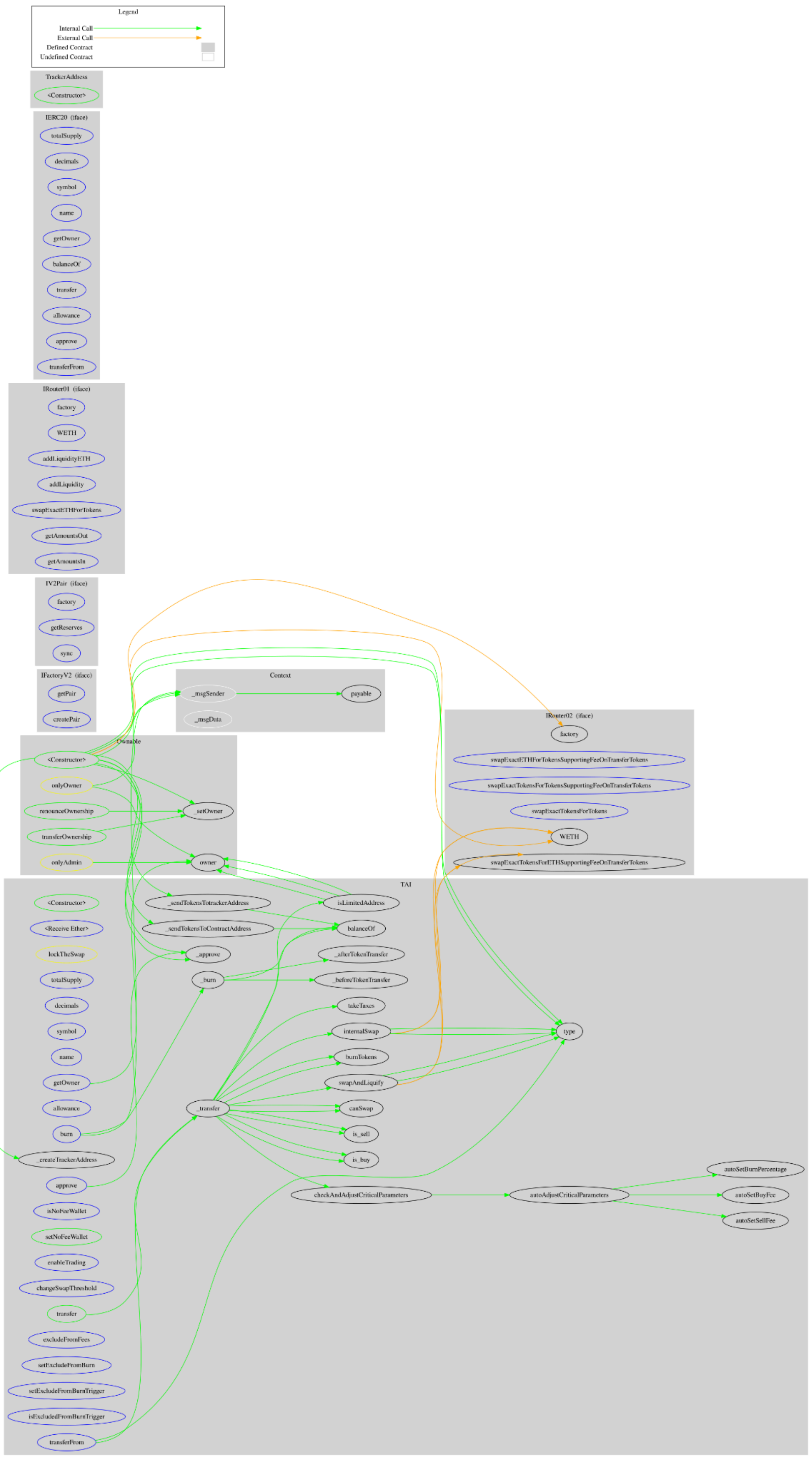
	setExcludeFromBurn	External	✓	onlyOwner
	setExcludeFromBurnTrigger	External	✓	onlyOwner
	isExcludedFromBurnTrigger	External		-
	_approve	Internal	✓	
	_transfer	Internal	✓	
	takeTaxes	Internal	✓	
	distributeFees	Internal	✓	lockTheSwap
	internalSwap	Internal	✓	
	swapAndLiquify	Internal	✓	
	burnTokens	Internal	✓	
	checkAndAdjustCriticalParameters	Internal	✓	
	autoAdjustCriticalParameters	Internal	✓	
	autoSetSellFee	Internal	✓	
	autoSetBuyFee	Internal	✓	
	autoSetBurnPercentage	Internal	✓	
	_burn	Internal	✓	
	_beforeTokenTransfer	Internal	✓	
	_afterTokenTransfer	Internal	✓	
	_createTrackerAddress	Internal	✓	
	_sendTokensTotrackerAddress	Internal	✓	
	isLimitedAddress	Internal		
	is_buy	Internal		
	is_sell	Internal		

	canSwap	Internal		
TrackerAddress	Implementation			
		Public	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
TrackerAddress
<Constructor>
IERC20 (iface)
totalSupply
decimals
symbol
name
getOwner
balanceOf
transfer
allowance
approve
transferFrom
IRouter01 (iface)
factory
WETH
addLiquidityETH
addLiquidity
swapExactETHForTokens
getAmountsOut
getAmountsIn
IV2Pair (iface)
factory
getReserves
sync
IFactoryV2 (iface)
getPair
createPair
Context
_msgSender
payable
_msgData
Ownable
<Constructor>
onlyOwner
renounceOwnership
transferOwnership
onlyAdmin
_setOwner
owner
IRouter02 (iface)
factory
swapExactETHForTokensSupportingFeeOnTransferTokens
swapExactTokensForTokensSupportingFeeOnTransferTokens
swapExactTokensForTokens
WETH
swapExactTokensForETHSupportingFeeOnTransferTokens
TAI
<Constructor>
<Receive Ether>
lockTheSwap
totalSupply
decimals
symbol
name
getOwner
allowance
burn
_createTrackerAddress
approve
isNoFeeWallet
setNoFeeWallet
enableTrading
changeSwapThreshold
transfer
excludeFromFees
setExcludeFromBurn
setExcludeFromBurnTrigger
isExcludedFromBurnTrigger
transferFrom
_sendTokensTotrackerAddress
_sendTokensToContractAddress
_approve
_burn
isLimitedAddress
balanceOf
_afterTokenTransfer
_beforeTokenTransfer
takeTaxes
internalSwap
burnTokens
swapAndLiquify
_transfer
canSwap
is_sell
is_buy
checkAndAdjustCriticalParameters
autoAdjustCriticalParameters
type
autoSetBurnPercentage
autoSetBuyFee
autoSetSellFee

Summary

TreasuryAi contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. TreasuryAi is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions. There is also a limit of max 3% fee.

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

cyberscope.io