Cyberscope

Audit Report
AuditAI

September 2024

Network	ETH
Address	0x66af1DbAa0B9E5eD76ebe1DB7A44ce2C81995c8D
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
●	PRE	Potential Reentrance Exploit	Unresolved
●	GLE	Gas Limit Exhaustion	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	ITO	Inconsistent Transfer Override	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MU	Modifiers Usage	Unresolved
●	NWES	Nonconformity with ERC-20 Standard	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	RDI	Reward Distribution Inconsistency	Unresolved
●	RED	Redundant Event Declaration	Unresolved
●	RRA	Redundant Repeated Approvals	Unresolved
●	RSU	Redundant State Updates	Unresolved
●	UTPD	Unverified Third Party Dependencies	Unresolved

●	ZARA	Zero Address Rewards Allocation	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved

Table of Contents

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	AuditAI
Compiler Version	v0.8.20+commit.a1b79de6
Optimization	200 runs
Explorer	https://etherscan.io/address/0x66af1dbaa0b9e5ed76ebe1db7a44ce2c81995c8d
Address	0x66af1dbaa0b9e5ed76ebe1db7a44ce2c81995c8d
Network	ETH
Symbol	AUDAI
Decimals	18
Total Supply	100,000,000
Badge Eligibility	Must Fix Criticals

Audit Updates

Initial Audit	05 Sep 2024

Source Files

Filename	SHA256
auditAIToken.sol	644f56355e1a9c95853697bfe0d6d84ef498ac58cfe52b571d63f706363226fd

Overview

The AUDITAI contract implements a token deployed in the ETH network.

Functionality:

- Token.
- Fee mechanism.
- Distributes fees to address with more than 100,000 tokens.

Key Functions:

- _transfer(address from, address to, uint256 amount) internal override
- claim( ) public
- checkHolders(address from, address to) internal
- distribution( ) internal

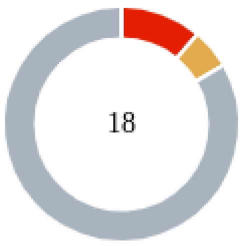
Roles:

- Owner:

The contract's owner can interact with the following functions:

1. function renounceOwnership() external onlyOwner
2. function transferOwnership(address newOwner) external onlyOwner
3. function excludeFromTax(address _address, bool _isExclude) external onlyOwner
4. function excludeFromRewards(address _address, bool _isExclude) public onlyOwner
5. function openTrading() external onlyOwner
6. function setStakingAddress(address _address) external onlyOwner

Findings Breakdown



●	Critical	2
●	Medium	1
●	Minor / Informative	15

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	2	0	0	0
●	Medium	1	0	0	0
●	Minor / Informative	15	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	auditAIToken.sol#L233
Status	Unresolved

Description

The transactions are initially disabled for all users excluding the authorized addresses. The owner can enable the transactions for all users. Once the transactions are enable the owner will not be able to disable them again.

```
function openTrading() external  onlyOwner {
        require(!tradingOpen,"trading is already open");
        tradingOpen = true;
}
```

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. Some suggestions are:

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

PRE - Potential Reentrance Exploit

Criticality	Critical
Location	auditAIToken.sol#L136,196
Status	Unresolved

Description

The contract makes an external call to transfer funds to recipients using the payable call method. The recipient could be a malicious contract that has an untrusted code in its fallback function that makes a recursive call back to the original contract. The re-entrance exploit could be used by a malicious user to drain the contract's funds or to perform unauthorized actions. This could happen because the original contract does not update the state before sending funds.

```
function claim() public {
        uint amount = holdersList[msg.sender].amountToClaim;
        require( amount > 0, "Nothing to claim");
         (bool success, ) = payable(msg.sender).call{value:
amount}("");
        require(success, "developer transfer failed.");
        holdersList[msg.sender].amountToClaim = 0;
        holdersList[msg.sender].lastClaimedTimestamp =
block.timestamp;
        totalAvailableToClaim -= amount;
}
```

```
function distribution() internal {
        uint balanceThis = address(this).balance;
        uint amountToDistribute = balanceThis -
totalAvailableToClaim;
        require(amountToDistribute > 0, "Contract balance is
zero");
        uint toSentMarketing = (amountToDistribute * 40) / 100;
        uint toSentDeveloper = (amountToDistribute * 20) / 100;
        uint holdersReward = amountToDistribute -
toSentDeveloper - toSentMarketing;

        (bool success, ) = developer.call{value:
toSentDeveloper}("");
        require(success, "developer transfer failed.");
        (bool success2, ) = marketing.call{value:
toSentMarketing}("");
        require(success2, "marketing transfer failed.");

        ...
    }
```

Recommendation

The team is advised to prevent the potential re-entrance exploit as part of the solidity best practices. Some suggestions are:

- Add lockers/mutexes in the method scope. It is important to note that mutexes do not prevent cross-function reentrancy attacks.
- Do Not allow contract addresses to receive funds.
- Adhere to the Checks-Effects-Interactions Pattern: Initially, perform all necessary checks, including verifying the caller, validating arguments, ensuring adequate Ether, and confirming token balances. Subsequently, make the required state changes within the contract. Interact with external contracts only after all state updates are complete. This sequence minimises the risk of reentrancy attacks and unintended consequences from external contract interactions. For more information please read the official documentation.
https://docs.soliditylang.org/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern

GLE - Gas Limit Exhaustion

Criticality	Medium
Location	auditAIToken.sol#L196
Status	Unresolved

Description

The contract executes operations that may result in transactions exceeding the gas limit. Specifically, the `distribution()` function iterates over a set of indices and with each iteration updates the contract's state. As the number of indices increases, the gas consumption grows proportionally. Since the `distribution()` function is invoked whenever the contract accrues more than 1 ETH, transactions may fail as the iteration count increases.

```
function distribution() internal {
    ...
    if(indexToHolder[holderLastIndex] != address(0)){
        totalAvailableToClaim += holdersReward;
        for (uint i = 0; i <= holderLastIndex; i++) {
            uint amount = balanceOf(indexToHolder[i]) +
            stakersAmount[indexToHolder[i]];
            uint reward = (holdersReward * amount /
            totalHoldedTokens);
            holdersList[indexToHolder[i]].amountToClaim +=
reward;
        }
    }
}
```

Recommendation

The team should consider revising the implementation of the `distribution()` function to prevent exceeding the gas limit. One possible solution would be to establish a gas threshold, upon reaching which the function would terminate and record the current index. In subsequent calls, the function would resume from the recorded index. The interaction between any modification and other functions in the contract, as well as any reciprocal effects, should be thoroughly reviewed.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	auditAIToken.sol#L221,227,233,238
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 excludeFromTax(address _address, bool _isExclude)
external onlyOwner {}

function excludeFromRewards(address _address, bool _isExclude)
public onlyOwner {}

function openTrading() external  onlyOwner {}

function setStakingAddress(address _address) 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.

ITO - Inconsistent Transfer Override

Criticality	Minor / Informative
Location	ERC20.sol#L171
Status	Unresolved

Description

The smart contract inherits the ERC20.sol contract from the OpenZeppelin library version `v5.x.x` , which introduces notable changes in the transfer mechanism. Specifically, the new version includes the `_update(address from, address to, uint256 value) internal virtual` function, which handles the transfer of a value amount of tokens between `from` and `to` , or mints and burns tokens if either address is the zero address. OpenZeppelin advises that all customizations for transfers, mints, and burns should now be done by overriding the `_update` function which is defined as virtual.

In previous versions, the `_transfer()` , `_mint()` , and `_burn()` functions were all defined as internal virtual, allowing developers to directly override them for customization. This is not possible in version `v5.x.x` .

```
* NOTE: This function is not virtual, {_update} should be
overridden instead.
     */
function _transfer(address from, address to, uint256 value)
internal {
        if (from == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        if (to == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(from, to, value);
}
```

Nevertheless, the contracts of this project have modified the `_transfer()` function to become virtual and directly overridden, bypassing the `_update` mechanism. This approach breaks the consistency of the code and contradicts OpenZeppelin's best practices.

```
* NOTE: This function is not virtual, {_update} should be
overridden instead.
     */
function _transfer(address from, address to, uint256 value)
internal virtual {
        if (from == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        if (to == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(from, to, value);
}
```

Recommendation

The team is advised to follow the best practices outlined in the OpenZeppelin implementation to maintain code consistency and avoid potential issues arising from such discrepancies. By overriding the `_update` function, as recommended, customizations to transfers, mints, and burns can be implemented without breaking the intended structure of the code.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	auditAIToken.sol#L136
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 claim() public {
            ...
     }
```

Recommendation

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

MU - Modifiers Usage

Criticality	Minor / Informative
Location	auditAIToken.sol#L222,228,239
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(_address != address(0), "address 0");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

NWES - Nonconformity with ERC-20 Standard

Criticality	Minor / Informative
Location	auditAIToken.sol#L85
Status	Unresolved

Description

The contract is not fully conforming with the ERC20 standard. According to the standard, transfers of 0 values must be treated as normal transfers and fire the Transfer event. However the contract implements a conditional check that prohibits transfers of 0 values. This discrepancy can lead to inconsistencies and incompatibilities with other contracts.

```
function _transfer(address from, address to, uint256 amount) internal
override {
        require(amount > 0, "Transfer amount must be greater than
zero");
        ...
}
```

Recommendation

The incorrect implementation of the ERC20 standard could potentially lead to inconsistencies when interacting with the contract, as other contracts or applications that expect the ERC20 interface may not behave as expected. The team is advised to review and revise the implementation of the transfer mechanism to ensure full compliance with the ERC20 standard. https://eips.ethereum.org/EIPS/eip-20.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	auditAIToken.sol#L147
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)  internal
lockTheSwap{
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = uniswapRouter.WETH();

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


uniswapRouter.swapExactTokensForETHSupportingFeeOnTransferToken
s(
            tokenAmount,
            0,
            path,
            feeCollector,
            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	auditAIToken.sol#L196
Status	Unresolved

Description

The contract sends funds to a `developer` and a `marketing` wallet as part of the transfer flow. These addresses can either be wallet addresses or contracts. If either of the addresses belongs to a contract then it may revert from incoming payments. As a result, the error will propagate to the token's contract and revert the transfer.

```
function distribution() internal {
    ...
    (bool success, ) = developer.call{value:
toSentDeveloper}("");
    require(success, "developer transfer failed.");
    (bool success2, ) = marketing.call{value:
toSentMarketing}("");
    require(success2, "marketing transfer failed.");
    ...
}
```

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.

RDI - Reward Distribution Inconsistency

Criticality	Minor / Informative
Location	auditAIToken.sol#L32
Status	Unresolved

Description

The contract implements the `excludeFromRewards` function, which is intended to exclude specific addresses, such as the owner and the contract itself, from receiving rewards. However, the contract does not adequately account for the exclusion of critical addresses, such as the `uniswapRouter`, `uniswapPair`, and `universalRouter`. This oversight may result in a significant portion of rewards being inadvertently assigned to these addresses, thereby rendering them inaccessible.

```
function excludeFromRewards(address _address, bool _isExclude) public
onlyOwner {
        require(_address != address(0), "address 0");
        excludedFromRewards[_address] = _isExclude;
        emit UpdateExcludedFromRewards(_address, _isExclude);
}
```

Recommendation

The team is advised to exclude critical addresses from receiving rewards through the `excludeFromRewards` function. Implementing this measure will ensure that the contract behaves as intended and aligns with the expected reward distribution mechanism.

RED - Redundant Event Declaration

Criticality	Minor / Informative
Location	auditAIToken.sol#L61,62,63,64,67,68,69
Status	Unresolved

Description

The contract uses events that are not emitted within the contract's functions. As a result, these declared events are redundant and serve no purpose within the contract's current implementation.

```
event FeePaid(address indexed payer, uint256 amount);
event FeePaidFrom(address indexed from, address indexed to,
uint256 amount);
event TokensTransfered(address indexed buyer, uint256 amount);
event TokensTransferedFrom(address indexed from, address
indexed to, uint256 amount);
event BuyTaxUpdated(uint newBuyFee);
event SellTaxUpdated(uint newSellFee);
event FeeWalletUpdated(address indexed newTreasury);
```

Recommendation

To optimize contract performance and efficiency, it is advisable to regularly review and refactor the codebase, removing the unused event declarations. This proactive approach not only streamlines the contract, reducing deployment and execution costs but also enhances readability and maintainability.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	auditAIToken.sol#L152
Status	Unresolved

Description

The contract is designed to `approve` token transfers during the contract's operation by calling the _approve function before specific operations. This approach results in additional gas costs since the approval process is repeated for every operation execution, leading to inefficiencies and increased transaction expenses.

```
_approve(address(this), address(uniswapRouter), tokenAmount);
```

Recommendation

Since the approved address is a trusted third-party source, it is recommended to optimize the contract by approving the maximum amount of tokens once in the initial set of the variable, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

RSU - Redundant State Updates

Criticality	Minor / Informative
Location	auditAIToken.sol#L163
Status	Unresolved

Description

The contract contains functions that redundantly update the state with the same variables, increasing complexity, gas consumption, and reducing readability. Specifically, the `checkHolders()` function updates records for users whose balances exceed a fixed threshold and is invoked after every transfer operation. In the case that a user already exceeds the threshold, the function redundantly deletes and rewrite their records with unchanged values for the `_amountToClaim` and `_lastClaimedTimestamp` in `holdersList[holders[i]]` mapping. This redundancy raises the gas fees for transfers involving users whose status has not changed.

```
function checkHolders(address from, address to) internal {
        uint hundredThousand = 100000e18;
        address[] memory holders = new address[](2);
        holders[0] = from;
        holders[1] = to;

        address[] memory stakers = new address[](2);
        stakers[0] = from;
        stakers[1] = to;

        for(uint i=0; i<2; i++){
            uint _amountToClaim =
holdersList[holders[i]].amountToClaim;
            uint _lastClaimedTimestamp =
holdersList[holders[i]].lastClaimedTimestamp;
            if(holdersList[holders[i]].index != 0){
                totalHoldedTokens -= stakersAmount[stakers[i]];
                totalHoldedTokens -= balanceOf(holders[i]);
            }
            delete
indexToHolder[holdersList[holders[i]].index];
            delete holdersList[holders[i]];
            if(balanceOf(holders[i]) +
stakersAmount[stakers[i]] >= hundredThousand &&
!excludedFromRewards[holders[i]]){
                holderLastIndex++;
                holdersList[holders[i]] = Holder({
                    index: holderLastIndex,
                    amountToClaim: _amountToClaim,
                    lastClaimedTimestamp: _lastClaimedTimestamp
                });
                indexToHolder[holderLastIndex] = holders[i];
                uint totalHolded = balanceOf(holders[i]) +
stakersAmount[stakers[i]];
                totalHoldedTokens += totalHolded;
            }
        }
    }
```

Recommendation

The team should consider revising the `checkHolders()` function and the transfer mechanism to ensure optimal gas consumption for all users.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	auditAIToken.sol#L244
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions. In this case the staking contract has the authority to modify the state for the staked amount of any user potentially affecting functions such as the reward distribution.

```
function updateStakedBalance(address user, uint amount)
external {
        require(msg.sender == staking, "Only staking contract
can update balance!");
        stakersAmount[user] = amount;
}
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

ZARA - Zero Address Rewards Allocation

Criticality	Minor / Informative
Location	auditAIToken.sol#L196
Status	Unresolved

Description

The contract's `distribution()` function is missing a check to ensure that rewards are not allocated to the zero address. Specifically the function loops over a number of indices in the range of `[0, holderLastIndex]`. Due to the design of the checkHolders function it is possible that some indices return a mapping `indexToHolder[i] = address(0)`, especially for indices whose previously pointed addresses are no longer considered holders. In such cases the contract assigns rewards to the zero address proportional to the burned amounts in that address. This may not be the intended distribution and may yield inconsistencies.

```
function checkHolders(address from, address to) internal {
        ...
        delete indexToHolder[holdersList[holders[i]].index];
        ...
```

```
function distribution() internal {
    uint balanceThis = address(this).balance;
    uint amountToDistribute = balanceThis -
totalAvailableToClaim;
    require(amountToDistribute > 0, "Contract balance is
zero");
    uint toSentMarketing = (amountToDistribute * 40) / 100;
    uint toSentDeveloper = (amountToDistribute * 20) / 100;
    uint holdersReward = amountToDistribute - toSentDeveloper -
         toSentMarketing;

    (bool success, ) = developer.call{value:
toSentDeveloper}("");
    require(success, "developer transfer failed.");
    (bool success2, ) = marketing.call{value:
toSentMarketing}("");
    require(success2, "marketing transfer failed.");

    if(indexToHolder[holderLastIndex] != address(0)){
        totalAvailableToClaim += holdersReward;
        for (uint i = 0; i <= holderLastIndex; i++) {
            uint amount = balanceOf(indexToHolder[i]) +
            stakersAmoun[indexToHolder[i]];
            uint reward = (holdersReward * amount /
            totalHoldedTokens);
            holdersList[indexToHolder[i]].amountToClaim +=
reward;
        }
    }
}
```

Recommendation

The is advised to implement a conditional check against the zero address to ensure rewards are properly handled.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	auditAIToken.sol#L17,18,41
Status	Unresolved

Description

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

```
address private universalRouter =
0x3fC91A3afd70395Cd496C647d5a6CC9D4B2b7FAD
address private uniswapFeeCollector =
0x000000fee13a103A10D593b9AE06b3e05F2E7E1c
uint256 private sThreshold = 3000000000000000000000
```

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	auditAIToken.sol#L53,221,227,238
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.

```
mapping(address => bool) public _isExcludedFromFee
bool _isExclude
address _address
```

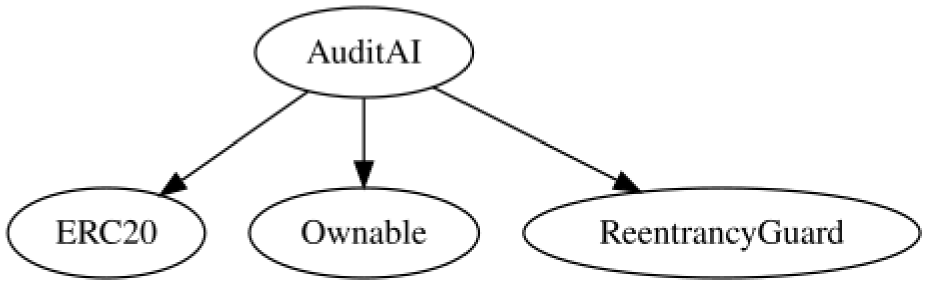
Recommendation

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.

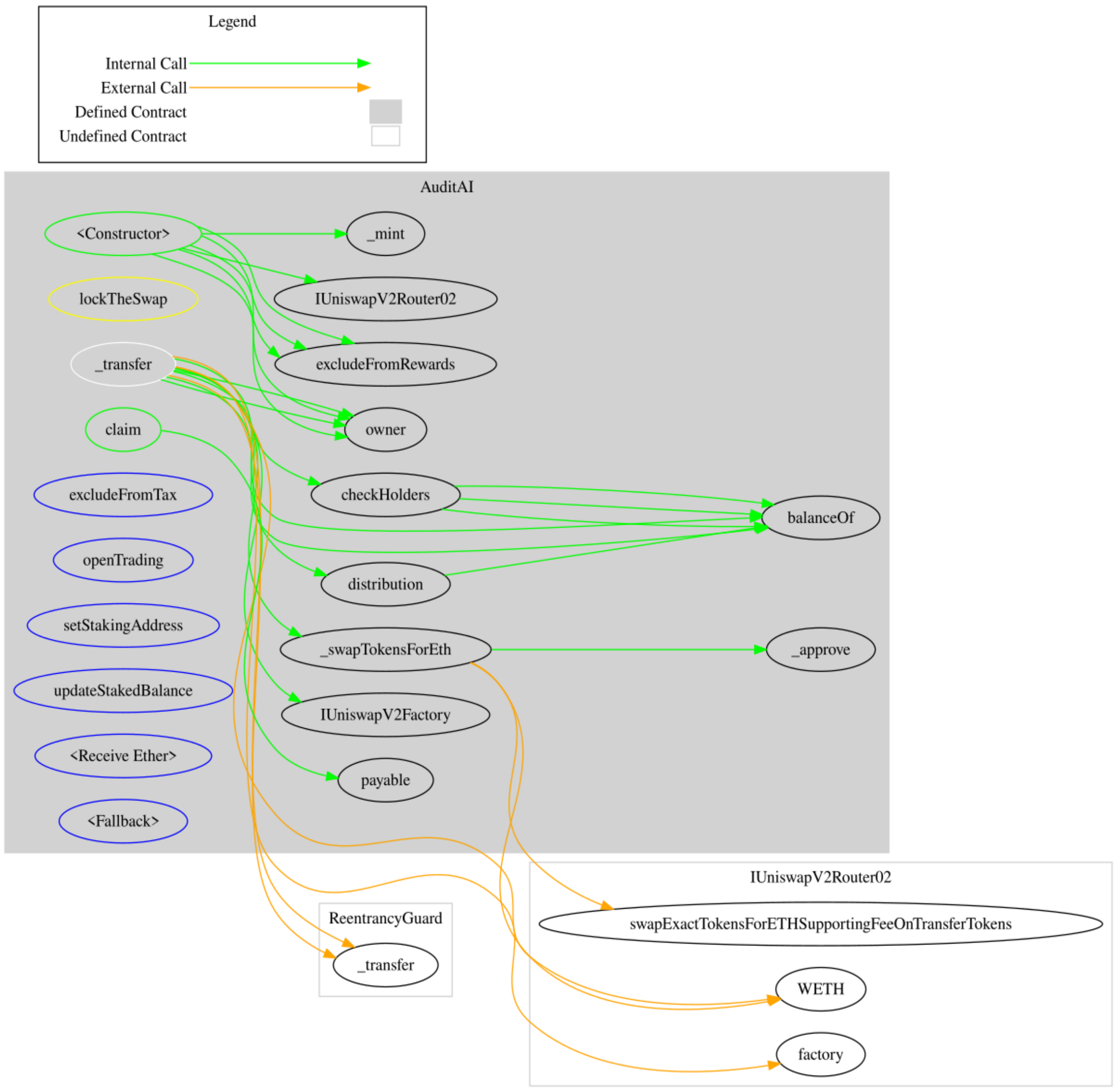
Functions Analysis

Contract	Type	Bases		
Function Name	**Visibility**	**Mutability**	**Modifiers**	
AuditAI	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	ERC20 Ownable
	_transfer	Internal	✓	
	claim	Public	✓	-
	_swapTokensForEth	Internal	✓	lockTheSwap
	checkHolders	Internal	✓	
	distribution	Internal	✓	
	excludeFromTax	External	✓	onlyOwner
	excludeFromRewards	Public	✓	onlyOwner
	openTrading	External	✓	onlyOwner
	setStakingAddress	External	✓	onlyOwner
	updateStakedBalance	External	✓	-
		External	Payable	-
		External	Payable	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
AuditAI
<Constructor>
_mint
lockTheSwap
IUniswapV2Router02
_transfer
excludeFromRewards
claim
owner
excludeFromTax
checkHolders
balanceOf
openTrading
distribution
setStakingAddress
_swapTokensForEth
_approve
updateStakedBalance
IUniswapV2Factory
<Receive Ether>
payable
<Fallback>
IUniswapV2Router02
swapExactTokensForETHSupportingFeeOnTransferTokens
ReentrancyGuard
_transfer
WETH
factory

Summary

The audited contract implements a token mechanism. This audit investigated security issues, business logic concerns and potential improvements. The Smart Contract analysis reported possible compiler errors due to the nature of the ITO finding and a number of critical and medium severity issues. In particular, the contract may be vulnerable to re-entrancy attacks, which could jeopardise the tokens accumulated in the contract and the intended behaviour. Other issues of moderate severity concern the consistent and robust execution of the code.

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

cyberscope.io