Cyberscope

Audit Report

AuditAI

September 2024

SHA256

0ec9691de33d9efd9adb30ff9f86d3245e32e23a5f6ce07db21ff2b800969b6b

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	GLE	Gas Limit Exhaustion	Unresolved
Minor	CCR	Contract Centralization Risk	Unresolved
Minor	EAFR	Excluded Addresses From Rewards	Unresolved
Minor	MU	Modifiers Usage	Unresolved
Minor	NWES	Nonconformity with ERC-20 Standard	Unresolved
Minor	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
Minor	PTRP	Potential Transfer Revert Propagation	Unresolved
Minor	RRA	Redundant Repeated Approvals	Unresolved
Minor	RSU	Redundant State Updates	Unresolved
Minor	RDI	Reward Distribution Inconsistency	Unresolved
Minor	UTPD	Unverified Third Party Dependencies	Unresolved
Minor	ZARA	Zero Address Rewards Allocation	Unresolved
Minor	L02	State Variables could be Declared Constant	Unresolved
Minor	L04	Conformance to Solidity Naming Conventions	Unresolved

Table of Contents

Analysis 1
Diagnostics 2
Table of Contents 3
Risk Classification 5
Review 6
Audit Updates 6
Source Files 6
Overview 7
Findings Breakdown 8
ST - Stops Transactions 8
Description 9
Recommendation 9
GLE - Gas Limit Exhaustion 10
Description 10
Recommendation 10
Team Update 11
CCR - Contract Centralization Risk 12
Description 12
Recommendation 12
EAFR - Excluded Addresses From Rewards 13
Description 13
Recommendation 14
MU - Modifiers Usage 15
Description 15
Recommendation 15
NWES - Nonconformity with ERC-20 Standard 16
Description 16
Recommendation 16
PLPI - Potential Liquidity Provision Inadequacy 17
Description 17
Recommendation 18
PTRP - Potential Transfer Revert Propagation 19
Description 19
Recommendation 19
RRA - Redundant Repeated Approvals 20
Description 20
Recommendation 20
RSU - Redundant State Updates 21
Description 21

Recommendation 22
RDI - Reward Distribution Inconsistency 23
Description 23
Recommendation 23
UTPD - Unverified Third Party Dependencies 24
Description 24
Recommendation 24
ZARA - Zero Address Rewards Allocation 25
Description 25
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
Functions Analysis **29**
Inheritance Graph **30**
Flow Graph **31**
Summary **32**
Disclaimer **33**
About Cyberscope **34**

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

Badge Eligibility	Yes

Audit Updates

Initial Audit	05 Sep 2024 https://github.com/cyberscope-io/audits/blob/main/audai/v1/audit.pdf
Corrected Phase 2	24 Sep 2024
Testing Deploy	https://sepolia.etherscan.io/address/0xca7b4c6cde8457ee7ea7904060dd008c2c6d8194

Source Files

Filename	SHA256
auditAIToken.sol	0ec9691de33d9efd9adb30ff9f86d3245e32e23a5f6ce07db21ff2b800969b6b

Overview

The AUDITAI contract implements a token mechanism.

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

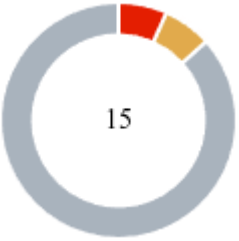
Roles:

- 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	1
Medium	1
Minor / Informative	13

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	1	0	0	0
Minor / Informative	13	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	AUDAI token.sol#L234
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.

GLE - Gas Limit Exhaustion

Criticality	Medium
Location	AUDAI token.sol#L192
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 indexes 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 nonReentrant{
...
    for (uint i = holderMinIndex; i <= holderLastIndex; i++) {
        if(indexToHolder[i] != address(0) && tempMinIndex ==
holderMinIndex){
            tempMinIndex = i;
        }
        uint amount = balanceOf(indexToHolder[i]) +
stakersAmount[indexToHolder[i]];
        uint reward = (holdersReward * amount /
totalHoldedTokens);
        holdersList[indexToHolder[i]].amountToClaim += reward;
    }
    holderMinIndex = tempMinIndex;
}
}
```

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.

Team Update

As noted in finding EAFR, the current implementation of the `distribution` function excludes lower-indexed addresses from receiving rewards. A suggested implementation of the distribution mechanism addresses this by processing eligible non-zero addresses until a predefined gas limit is reached. The function then records the last processed index, allowing subsequent calls to continue execution from that point. Once the final index has been processed, the function should iterate from the beginning of the list to ensure all eligible, non-zero, addresses are consistently accounted for.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	AUDAI token.sol#L222,228,234,239
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.

EAFR - Excluded Addresses From Rewards

Criticality	Minor / Informative
Location	AUDAI token.sol#L192
Status	Unresolved

Description

The contract implements a reward distribution function that allocates rewards to addresses identified as holders with a total balance exceeding 100,000 tokens. Eligible addresses are tracked in the `indexToHolder` mapping, and the function iterates over this mapping within the index range `[holderMinIndex, holderLastIndex]`.

Assuming a set of eligible addresses `[addr0, addr1, addr2, ..., addrk]` where all are non-zero, the first call to the function processes the rewards for all addresses and sets `holderMinIndex=1`. On a subsequent call with the same arguments, the iteration begins at index 1, therefore excluding `addr0` from receiving rewards. This process continues with `holderMinIndex` being incremented with each call (e.g., set to 2 on the next run), excluding `addr1` thereafter. As a result, repetitive calls of the function exclude the lower-indices from receiving rewards, eventually leaving all but the last addresses excluded.

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

Recommendation

The team is advised to revise the implementation of the distribution function to ensure all eligible addresses receive proper rewards.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	AUDAI token.sol#L223,229,240
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	AUDAI token.sol#L80
Status	Unresolved

Description

The contract is not fully conforming to the ERC20 Standard. Specifically, 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 between the contract's implementation and the ERC20 standard may 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 problems 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	AUDAI token.sol#L143
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.swapExactTokensForETHSupportingFeeOnTransferTokens
(
            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	AUDAI token.sol#L200
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.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	AUDAI token.sol#L148
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	AUDAI token.sol#L159
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.

RDI - Reward Distribution Inconsistency

Criticality	Minor / Informative
Location	AUDAI token.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's address, from receiving rewards. However, the function does not adequately account for the exclusion of critical addresses, including 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.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	AUDAI token.sol#L245
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	AUDAI token.sol#L192
Status	Unresolved

Description

The contract's `distribution` function lacks a check to prevent rewards from being assigned to the zero address. The function iterates over a range of indices `[holderMinIndex, holderLastIndex]` to distribute rewards. As noted in finding EAFR, the function increases `holderMinIndex` with each call, eventually excluding lower-indexed addresses from future reward allocations. However, when the function processes a zero-value address, it updates `holderMinIndex` to the index of the first non-zero address after the zero-value, treating the zero address as excluded. Despite this, zero-value indices beyond the updated `holderMinIndex` will still receive rewards in future iterations. Furthermore, the zero address might have already received rewards from previous iterations, contributing to unintended distribution errors.

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

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

Recommendation

The team 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	AUDAI token.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	AUDAI token.sol#L54,222,228,239
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
```

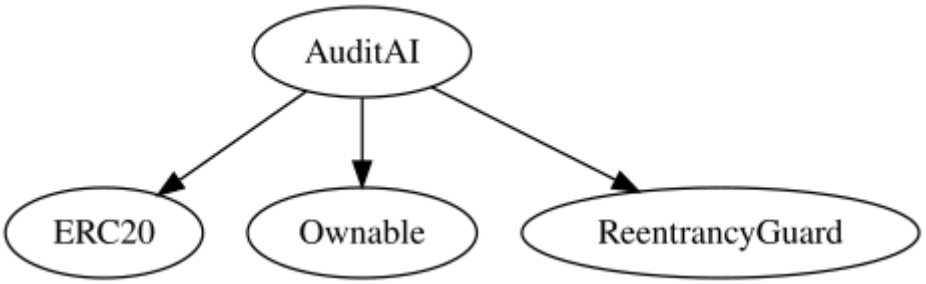
Recommendation

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.

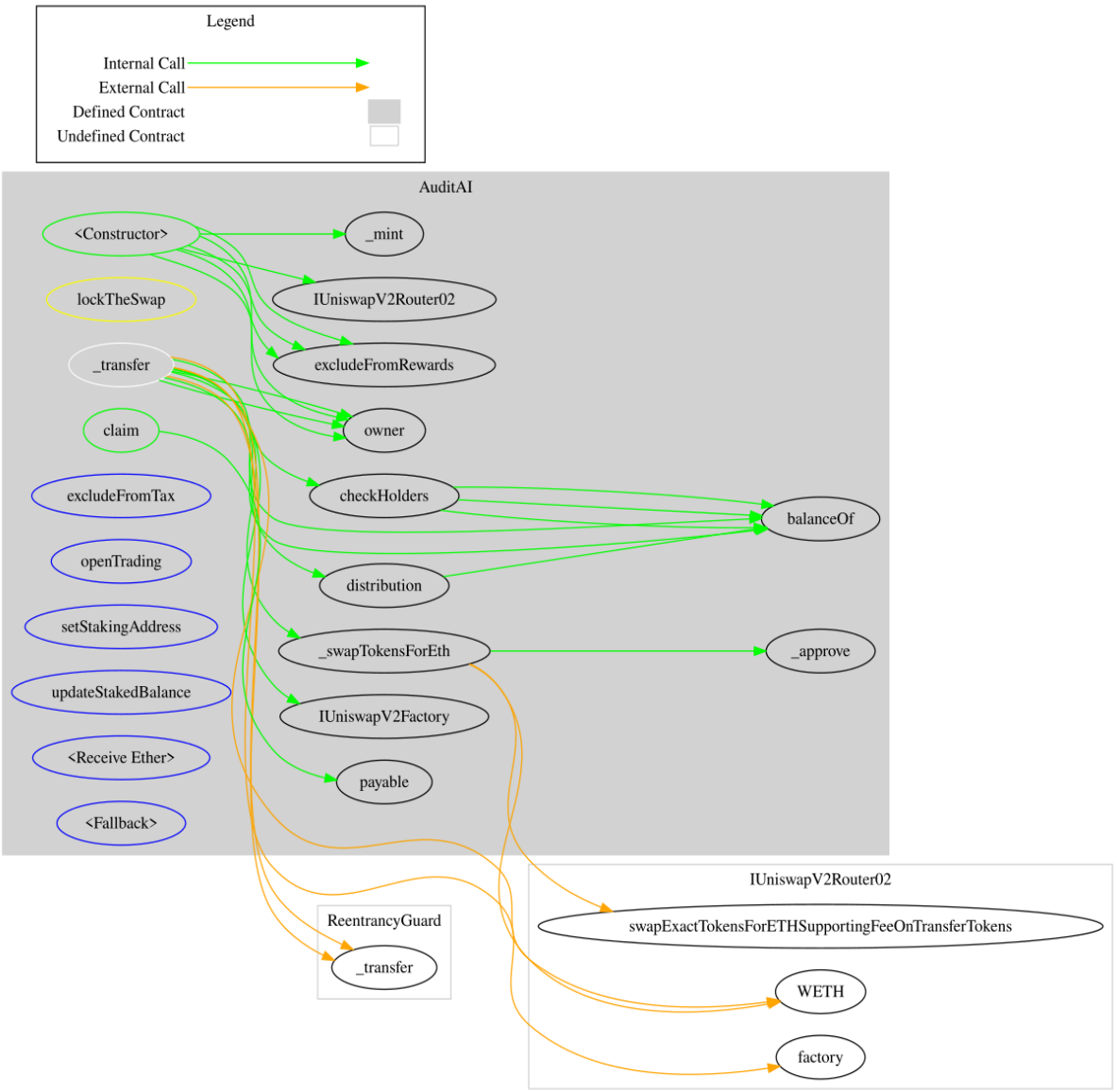
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
AuditAI	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	ERC20 Ownable
	_transfer	Internal	✓	
	claim	Public	✓	nonReentrant
	_swapTokensForEth	Internal	✓	lockTheSwap
	checkHolders	Internal	✓	
	distribution	Internal	✓	nonReentrant
	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
IUniswapV2Router02
lockTheSwap
excludeFromRewards
_transfer
owner
claim
checkHolders
balanceOf
excludeFromTax
openTrading
distribution
setStakingAddress
_swapTokensForEth
_approve
updateStakedBalance
IUniswapV2Factory
<Receive Ether>
payable
<Fallback>
ReentrancyGuard
_transfer
IUniswapV2Router02
swapExactTokensForETHSupportingFeeOnTransferTokens
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 a critical and a medium severity issue. 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