BAIL security

BAILSEC.IO

EMAIL : OFFICE@BAILSEC.IO

TWITTER : @BAILSECURITY

TELEGRAM : @HELLOATBAILSEC

FINAL REPORT:

OnlyCocksCrypto

January 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:

Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	OnlyCocksCrypto
Website	onlycockscrypto.club
Type	ERC20 Token
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/OnlyCocksCrypto/OnlyCocksCrypto/blob/7f6121ff6b859efb1e9f81a5b9321f1c29ac550e/cox_avalanche.sol
Resolution 1	https://github.com/OnlyCocksCrypto/OnlyCocksCrypto/blob/f99c4508f5fc8ad54a896a90d559201fc513a8ce/cox_avalanche.sol
Resolution 2	https://github.com/OnlyCocksCrypto/OnlyCocksCrypto/tree/edb17c5a4c96770c72547072676725b5f98e0734
Resolution 3	https://github.com/OnlyCocksCrypto/OnlyCocksCrypto/blob/5351f96f86815d02c5eca91b5cbf8e3ed36a93ca/cox_avalanche.sol

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	0			
Medium	4	1	1	2
Low	5	1	1	3
Informational	5	1		3
Configurational	0			
Governance	1			1
Quality assurance	1			1
Total	16	3	2	10

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Configurational	Issues which may arise due to different configurational settings
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior
Quality assurance	Aggregated minor issues, ensuring a high quality codebase.

3. Detection

cox_avalanche

The cox contract is a customized ERC20 token which applies a transfer-tax on different operations. Additionally to the transfer-tax, the following limitations occur:

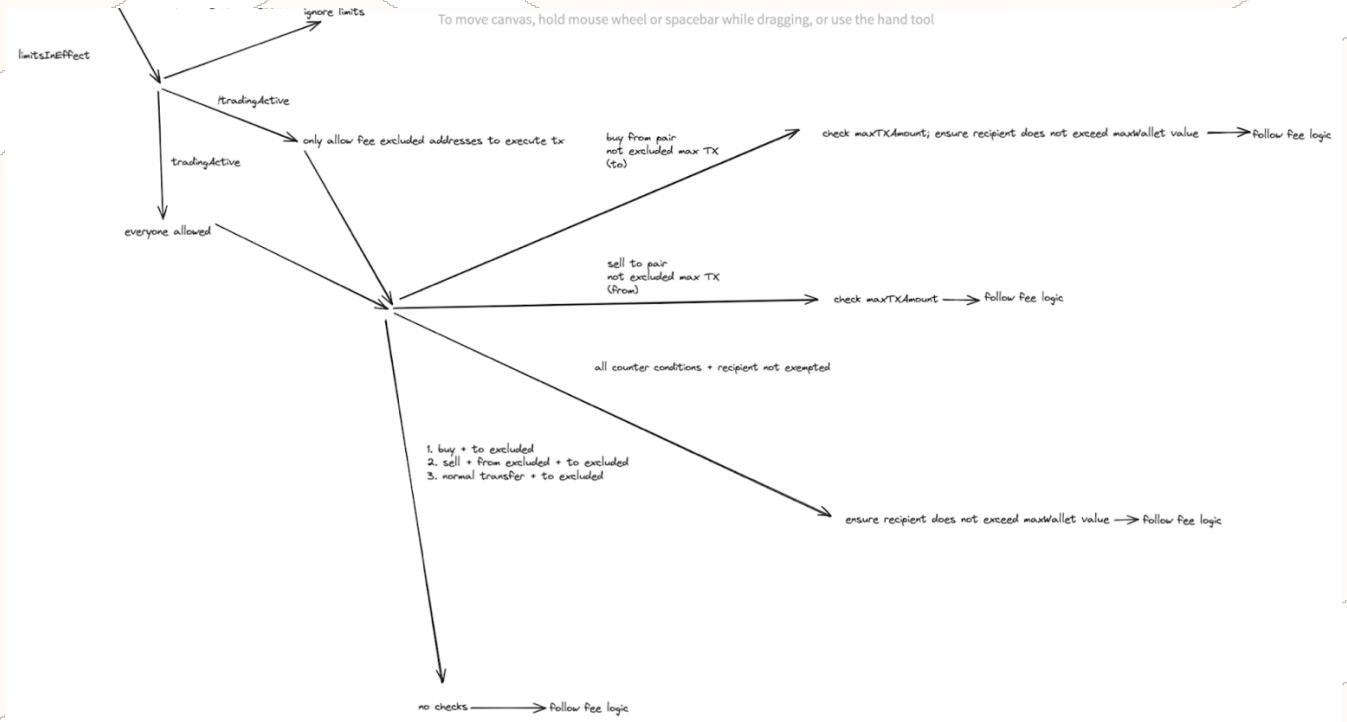
1. Maximum amount per transaction: Certain operations limit the maximum transfer amount to a percentage from the total supply. This percentage parameter can be freely set by the contract owner within the updateMaxTxnAmount and can not be set below 0.5%.

2. Maximum wallet check: The contract incorporates a check which ensures that non-excluded wallets cannot exceed the maximum amount per wallet, which is initially set to 1% and can be changed by the contract owner to the upside.

3. Trading active: This variable is initially set to false and can be set to true by the owner. If false, only fee exempted addresses can transfer/receive tokens.

3. Limits in effect: This variable is initially set to true and can be set to false by the owner. If true, it checks the aforementioned three conditions. If false, no conditions are checked and transfers are freely executable. The contract owner has the freedom to set this to false, however, it cannot be turned on again once it is set to false.

A visualization of the aforementioned checks if limits are in effect can be found below:



Summarized, the following scenarios apply:

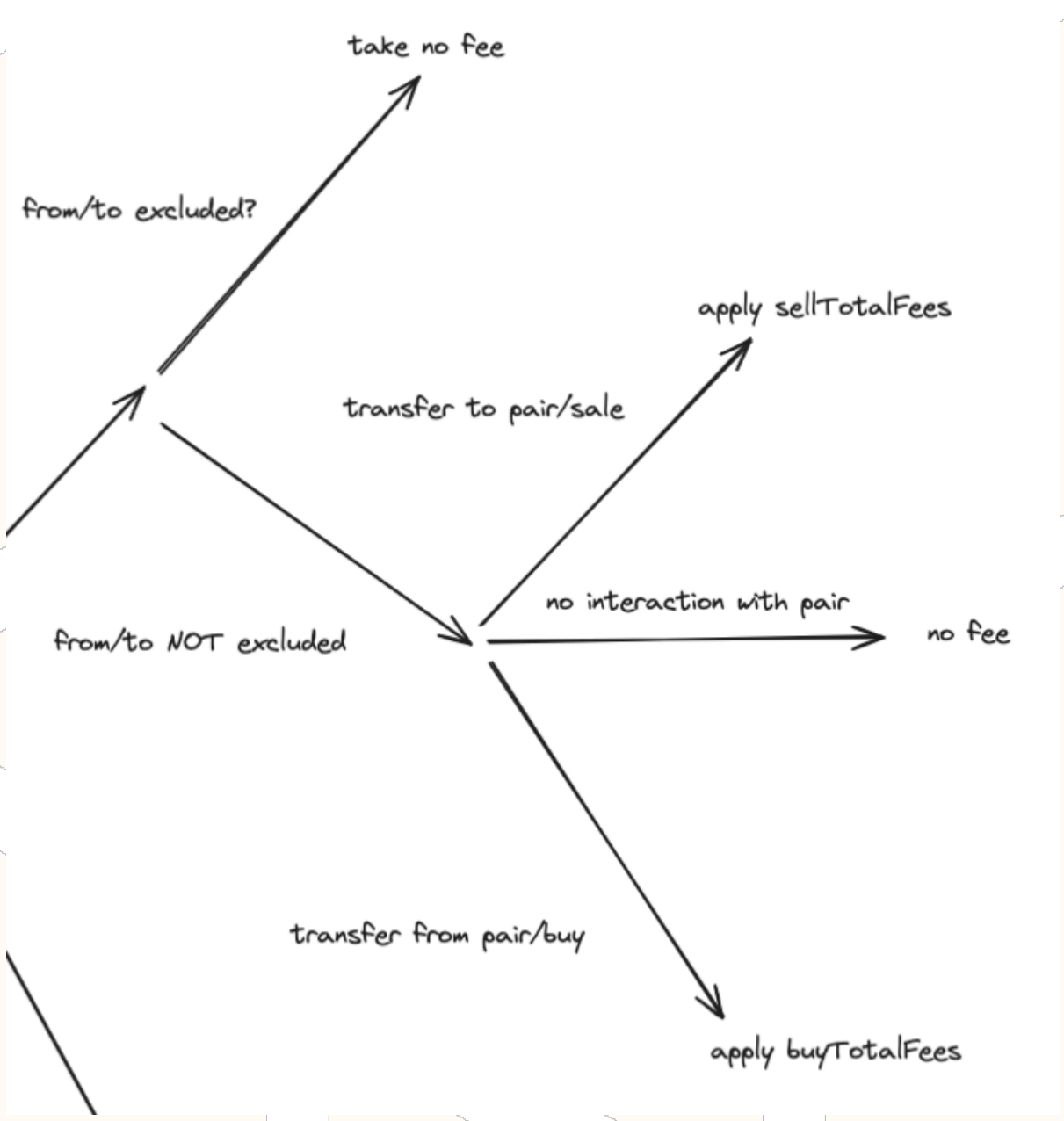
a) A buy transaction from an un-exempted user is subject to a max wallet check and a max transfer check.
b) A sell transaction from an un-exempted user is subject to a max transfer check.
c) Any transfer, including these to a pair and a non-exempt recipient address is subject to a max wallet check.

In Addition to the aforementioned limitations, the contract has a pre-migration phase, which is active after deployment and prevents normal trading activities. Specific addresses, determined by the contract owner, can execute transfers even in this special period.

To counter any potential sniping or malicious behavior, the contract incorporates a blacklist mechanism which allows the owner to blacklist arbitrary addresses. To enhance decentralization, the blacklist mechanism can be turned off and then never turned on again, ensuring safe trading for users.

On transfers to and from specific pair addresses (again, settable by the owner), a fee is applied, with the special case of a fee exemption if the from or to address is exempted from the fee. For buy transactions (from pair), the buyTotalFees variable applies, on sell transactions (to pair), the sellTotalFees variable applies. For both scenarios, the fee can only be set up to 5%.

This can be illustrated as follows:



Last but not least it has to be noted that the token has a totalSupply of 88 999 888 tokens, all minted to the deployer upon deployment.

Resolution 3:

In the third resolution round, the tradingActive limitation has been removed, which now grants the contract owner less centralization privileges over transfers.

Issue	Governance Privileges
Severity	Governance
Description	The codebase has limited governance privileges such as blacklisting addresses in the blacklisting period, setting fees for certain transfers and setting limits in the limit period. However, all these features can be disabled irreversibly by the contract owner, ensuring a safe and decentralized trading experience.
Recommendations	As a user, we highly recommend inspecting related state variables before investing. From a code-perspective, there is nothing we have to add.
Comments / Resolution	Acknowledged.

Issue	Non-exclusive conditions can result in full DoS
Severity	Medium

Description	The logic within the limitations check is as follows:
	
	`//when buy`
	`if (`
	`automatedMarketMakerPairs[from] &&`
	`!_isExcludedMaxTransactionAmount[to]`
	`) {`
	`require(`
	`amount <= maxTransactionAmount,`
	`"Buy transfer amount exceeds the maxTransactionAmount."`
	`);`
	`require(`
	`amount + balanceOf(to) <= maxWallet,`

```
                "Max wallet exceeded"
                );
        }
        //when sell
        else if (
                automatedMarketMakerPairs[to] &&
                !_isExcludedMaxTransactionAmount[from]
        ) {
                require(
                amount <= maxTransactionAmount,
                "Sell transfer amount exceeds the
maxTransactionAmount."
                );
        } else if (!_isExcludedMaxTransactionAmount[to]) {
                require(
                amount + balanceOf(to) <= maxWallet,
                "Max wallet exceeded"
                );
        }
```

There are three general conditions, when buying tokens, when selling tokens and (supposedly) when not interacting with an AMM.
Each condition is checked on its own, the issue here lies within the fact that, if one condition is not met, an attempt for the other conditions is being executed. This will result in issues for the following example condition:

a) Alice sells tokens and is exempted from the max tx amount

-> This does not fall in the 2nd condition, since Alice is exempted, however, it will naturally fall under the third condition. In that scenario, since the pair is not automatically whitelisted, it will completely DoS sells, since the pair will likely exceed the max tx amount.

Recommendations

First, the root-cause of this issue lies within the fact that during development, not sufficient care was taken of the different conditions, especially, it was not checked what happens with a

transaction if a previous condition does not hold, e.g. it was not checked that certain transactions might fall under the third condition, while this is not desired.

Second, all AMM pairs should be naturally whitelisted (due to the predeterministic nature of UniswapV2 pairs, this can be directly done in the constructor with prominent tokens such as USDT/AVAX/USDC), but should also be done within the _setAutomatedMarketMakerPair function, as otherwise swaps can be DoS'ed not only with single swaps but also with multi-hop swaps, which sent tokens directly from one pair to another pair:

```
function _swap(uint[] memory amounts, address[] memory path, address _to) internal virtual {
for (uint i; i < path.length - 1; i++) {
(address input, address output) = (path[i], path[i + 1]);
(address token0,) = UniswapV2Library.sortTokens(input, output);
uint amountOut = amounts[i + 1];
(uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOut) : (amountOut, uint(0));
address to = i < path.length - 2 ? UniswapV2Library.pairFor(factory, output, path[i + 2]) : _to;
IUniswapV2Pair(UniswapV2Library.pairFor(factory, input, output)).swap(
amount0Out, amount1Out, to, new bytes(0)
);
}
}
```

Remember, in UniswapV2, a pair can become the direct recipient of a transfer from another pair, for multi-swaps.

Third, it is mandatory for the developer to present corresponding tests for the different conditions.

Buy Transactions (Transfer from AMM to User)

From (AMM) Excluded, To ExcludedFrom

(AMM) Excluded, To Not Excluded

From (AMM) Not Excluded, To Excluded

From (AMM) Not Excluded, To Not Excluded

Sell Transactions (Transfer from User to AMM)

From Excluded, To (AMM) Excluded

From Excluded, To (AMM) Not Excluded

From Not Excluded, To (AMM) Excluded

From Not Excluded, To (AMM) Not Excluded

Normal Transfers (Neither Buy nor Sell)

Both Excluded

From Excluded, To Not Excluded

From Not Excluded, To Excluded

Both Not Excluded

AMM to AMM (Multi-swaps)

Both Excluded

From Excluded, To Not Excluded
From Not Excluded, To Excluded

Both Not Excluded

Comments / Resolution	Acknowledged, pairs are not whitelisted from maxWallet upon deployment. This will result in a DoS for swaps at some point. However, the owner can still whitelist it after deployment.

Issue	Sybil attacks can circumvent all maximum checks
Severity	Medium
Description	The maximum transfer check during a sell transaction was implemented in an effort to prevent large single sales. However, this can be circumvented as follows: Since standard transfers from address <-> address do not take any transfer-tax, users can simply transfer tokens to different addresses or contracts and then sell all tokens in the same block. This can be done with a smart contract that calls several other smart contracts to sell tokens in the same transaction.
Recommendations	Consider if it makes sense to implement transfers between normal wallets to counter this scenario or if it is an accepted risk (this risk is often present with different projects, so it might get acknowledged as well).
Comments / Resolution	Acknowledged.

Issue	Lack of accounted fee and recipient address usage indicates unfinalized development cycle
Severity	Medium

Description	The contract has three different variables which indicate how much tokens are stored for each entity due to the applied fee:
	
	L 40:
	
	uint256 public tokensForRevShare;
	
	L 41:
	
	uint256 public tokensForLiquidity;
	
	L 42:
	
	uint256 public tokensForTeam;
	
	These variables are increased whenever a fee is taken:
	
	if (automatedMarketMakerPairs[to] && sellTotalFees > 0) {
	fees = amount.mul(sellTotalFees).div(100);
	tokensForLiquidity += (fees * sellLiquidityFee) / sellTotalFees;
	tokensForTeam += (fees * sellTeamFee) / sellTotalFees;
	tokensForRevShare += (fees * sellRevShareFee) / sellTotalFees;
	}
	// on buy
	else if (automatedMarketMakerPairs[from] && buyTotalFees > 0) {
	fees = amount.mul(buyTotalFees).div(100);
	tokensForLiquidity += (fees * buyLiquidityFee) / buyTotalFees;
	tokensForTeam += (fees * buyTeamFee) / buyTotalFees;
	tokensForRevShare += (fees * buyRevShareFee) / buyTotalFees;
	}
	
	Additionally, the following addresses are stored:

	L 14: address public revShareWallet; L 15: address public teamWallet; Indicating that these addresses are corresponding to the accounted fee amounts. However, both the addresses and the fees are not used for any purpose at all. The only way how fees are taken out is simply by the owner calling: *withdrawStuckCOX* and *withdrawStuckToken* This indicates that the contract has not finalized its development cycle, as in ideal scenario, these tokens can be claimed by the corresponding addresses up to their accounted share.
Recommendations	While any stuck tokens can obviously still be claimed, such an issue will inherently damage the brand name. If anyone inspects the contract and recognizes such an issue, they will inherently call out the *OnlyCocksCrypto* development team for deploying unfinished/bad code. The fees should only be claimable by the corresponding addresses and upon claim the fee storage variables should be reset to zero.
Comments / Resolution	Resolved, this logic has been removed.

Issue	swapping variable has absolutely no impact
Severity	Medium

Description	The swapping variable is used within the following condition checks and is also set to false if the latter condition is given:
	
	L 1036:
	
	if (limitsInEffect) {
	if (
	from != owner() &&
	to != owner() &&
	to != address(0) &&
	to != address(0xdead) &&
	!swapping
	)
	
	L 1087:
	
	if (
	canSwap &&
	swapEnabled &&
	!swapping &&
	!automatedMarketMakerPairs[from] &&
	!_isExcludedFromFees[from] &&
	!_isExcludedFromFees[to]
	) {
	swapping = false;
	}
	Moreover, it is then used to determine if a fee should be taken from the transfer:
	
	L 1098:
	
	bool takeFee = !swapping;
	
	This whole practice is completely redundant, as the swapping variable will always be false, no matter what state transitions have

	happened. Therefore, the takeFee variable will always be true, indicating that a fee should be taken in any scenario. We assume, that the contract intends to implement an auto liquidity feature, due to the comments, such as: L 952: // only use to disable contract sales if absolutely necessary (emergency use only) However, the logic for such a feature is completely non-existent.
Recommendations	Consider re-thinking about the idea of this smart contract and potentially adjusting it to reflect the desired functionalities.
Comments / Resolution	Partially resolved, this logic has been removed partially. The swapping variable is still checked within the following if-clause: if (from != owner() && to != owner() && to != address(0) && to != address(0xdead) && !swapping) However, it has no impact since this variable is naturally always false.

Issue	canSwap is redundant
Severity	Low

Description	The canSwap variable within the _transfer function is determined as follows:
	
	L 1085:
	
	bool canSwap = contractTokenBalance >= swapTokensAtAmount;
	
	Afterwards, it is included within the condition check for the swapping variable change:
	
	if (
	canSwap &&
	swapEnabled &&
	!swapping &&
	!automatedMarketMakerPairs[from] &&
	!_isExcludedFromFees[from] &&
	!_isExcludedFromFees[to]
	) {
	swapping = false;
	}
	
	bool takeFee = !swapping;
	
	However, due to the issue with the swapping variable, the canSwap variable becomes redundant completely, even if the contract balance is below the swapTokensAtAmount threshold, a fee will be taken.
	
	Additionally, the following comment was found:
	
	L 911:
	
	“// change the minimum amount of tokens to sell from fees”
	
	Which indicates that accumulated tokens within the token contract

	should probably be swapped. However, the contract does not expose such a functionality. This issue is also present for the swapEnabled variable: Whether this is true or false, does not have any impact.
Recommendations	Consider re-evaluating the idea behind this logic and applying corresponsive changes.
Comments / Resolution	Resolved, this has been removed.

Issue	newNum safeguard is flawed for updateMaxTxnAmount and updateMaxWalletAmount
Severity	Low
Description	Both aforementioned function are using an unconventional approach for calculations, such as: newNum >= ((totalSupply() * 5) / 1000) / 1e18, Generally, the calculation should always be totalSupply() * minPercentage / denominator In the calculation used within the code, it would result in the safeguard becoming zero for a totalSupply of 100e18: 100e18 * 5 / 1000 / 1e18 = 0.5 -> 0 (solidity rounds down) While we acknowledge that the totalSupply is hardcoded and will not be 100e18 in the production, we still are the opinion that the code should be bug-free.

Recommendations	Consider using the correct approach for these calculations, as in the example. The correct denominator (10_000) should be used.
Comments / Resolution	Acknowledged.

Issue	swapTokensAtAmount logic is redundant
Severity	Low

Description	As already mentioned in issues below, the whole if clause starting from line 1087:
	
	bool canSwap = contractTokenBalance >= swapTokensAtAmount;
	
	if (
	canSwap &&
	swapEnabled &&
	!swapping &&
	!automatedMarketMakerPairs[from] &&
	!_isExcludedFromFees[from] &&
	!_isExcludedFromFees[to]
	) {
	swapping = false;
	}
	
	has no impact, therefore, also the swapTokensAtAmount variable is redundant.

Recommendations	Consider the broader idea behind this logic and re-evaluate the codebase.
Comments / Resolution	Partially resolved, the logic for this state variable has been removed. However, the variable itself is still existent. This does not expose any issues.

Issue	Transfer within withdrawStuckCOX will revert for certain recipients
Severity	Low
Description	Within the withdrawStuckCOX function, AVAX is transferred as follows: payable(msg.sender).transfer(address(this).balance); This will not work for addresses that execute fallback logic on AVAX receipts as it only forwards 2300 gas.
Recommendations	Consider using .call instead of .transfer.
Comments / Resolution	Acknowledged.

Issue	Inconsistency in denominator usage for arithmetic operations
Severity	Low
Description	Throughout the codebase, a lot of arithmetic operations are being executed such as: *swapTokensAtAmount = (totalSupply * 5) / 10000; // 0.05%* *fees = amount.mul(buyTotalFees).div(100);* as examples. Whenever arithmetic operations are being executed, they should be done in BPS, using a denominator of 10_000 and a corresponding multiplier. However, this was not done correctly throughout the codebase, resulting in inconsistencies. While such an inconsistency is not only a bad practice, it can also

	result in rounding issues if smaller denominators are being used. This issue is present throughout the whole codebase and will also limit the granularity in functions such as updating buy and sell fees, as it would not allow adding floating point percentages.
Recommendations	Consider being consistent and using a denominator of 10_000 (100%) and the corresponding multipliers (1 = 0.01%; 10 = 0.1%; 100 = 1%).
Comments / Resolution	Acknowledged.

Issue	Use of safeMath is redundant
Severity	Informational
Description	The contract is compiled using Solidity version 0.8.20. Within its codebase, it employs the SafeMath library for arithmetic operations. In Solidity versions prior to 0.8.0, arithmetic operations were susceptible to overflows and underflows, which could lead to significant vulnerabilities. The SafeMath library was commonly used to safeguard against these issues by providing arithmetic operations that include built-in checks for overflows and underflows. However, starting from Solidity 0.8.0, these safety checks have been integrated directly into the language itself. All arithmetic operations in Solidity 0.8.0 and later versions automatically check for overflows and underflows, reverting the transaction if any such occurrence is detected. This change renders the use of the SafeMath library obsolete and unnecessary for contracts compiled with Solidity 0.8.0 or higher. The continued use of SafeMath in a contract written in Solidity 0.8.20 introduces redundant code, which increases gas costs

	unnecessarily and can complicate code readability and maintainability.
Recommendations	Consider removing SafeMath and replacing the SafeMath operations with the standard arithmetic symbols.
Comments / Resolution	Acknowledged.

Issue	Redundant logic for tradingActive is still present
Severity	**Informational**

Description	n the third resolution round, the logic for limiting the trading to only excluded addresses was removed:
	
	`if (!tradingActive) {`
	`require(`
	`_isExcludedFromFees[from] || _isExcludedFromFees[to],`
	`"Trading is not active."`
	`);`
	`}`
	
	However, the state variable tradingActive and its corresponding setter logic is still present:
	
	L 24:
	
	`bool public tradingActive = false;`
	
	L 90:
	
	`function enableTrading() external onlyOwner {`
	`tradingActive = true;`
	`swapEnabled = true;`
	`preMigrationPhase = false;`
	`}`

Recommendations	Consider removing all logic which is related to the tradingActive state variable.
Comments / Resolution	

Issue	AbiEncoderV2 is inherently present for solidity versions >= 0.8.0
Severity	Informational
Description	The contract is utilizing Solidity version 0.8.20. Within its codebase, it explicitly declares the use of AbiEncoderV2 using the pragma directive pragma AbiEncoderV2. Starting from Solidity version 0.8.0, the AbiEncoderV2 has been the default ABI encoder. This version of the encoder supports complex data types and offers enhanced safety and gas efficiency. Since the contract is written in Solidity 0.8.20, it inherently uses AbiEncoderV2 by default. The explicit declaration of pragma AbiEncoderV2; in a contract using Solidity 0.8.20 is redundant. This redundancy does not impact the functionality or the security of the contract. However, it may lead to confusion or misunderstanding about the necessity of this declaration for future developers or auditors who might work on or review this code.
Recommendations	Consider removing the AbiEncoderV2 declaration.
Comments / Resolution	Acknowledged.

Issue	Contract is flattened
Severity	Informational
Description	Currently, the contract is flattened with some older versions of OpenZeppelin, while this is not an issue per-se, we alway recommend properly importing the OpenZeppelin files instead of flattening the contract.
Recommendations	Consider simply importing OpenZeppelin's files instead of flattening the contract.
Comments / Resolution	Resolved.

Issue	Dust accumulation for fee storage variables
Severity	Informational

Description	Within the _transfer function, fees are assigned as follows:
	
	fees = amount.mul(sellTotalFees).div(100);
	tokensForLiquidity += (fees * sellLiquidityFee) / sellTotalFees;
	tokensForTeam += (fees * sellTeamFee) / sellTotalFees;
	tokensForRevShare += (fees * sellRevShareFee) /sellTotalFees;
	
	First of all, as already mentioned the multiplier and denominator should be adjusted. Secondly, the third calculation can be just done using the leftover value from fees - tokensForLiquidity - tokensForTeam. Using this logic would prevent any potential dust which accumulates due to solidity's rounding mechanism.
	
	This issue is just a theoretical issue as the storage variables are not used for transfer purposes.

Recommendations	Consider simply calculating the leftover amount to prevent any dust.
Comments / Resolution	Acknowledged.

Issue	Quality Assurance: Minor Issues
Severity	**Quality assurance**
Description	The contract contains one or more minor issues, in an effort to keep the report size reasonable, we will enumerate these issues below: address public constant deadAddress = address(0xdead); This declaration is unused. —---------- event UpdateLBRouter(address indexed newAddress, address indexed oldAddress); This event is unused. —---------- event SwapAndLiquify(uint256 tokensSwapped, uint256 ethReceived, uint256 tokensIntoLiquidity); This event is unused.

	—----------
	if (fees > 0) {
	super._transfer(from, address(this), fees);
	}
	amount -= fees;
	
	The amount decrease can happen within the if-clause.
	
	—----------

Recommendations	Consider fixing these minor issues.
Comments / Resolution	Acknowledged.