Audit Report

Eda Token

February 2024

Network	BSC
Address	0x6Fd2233fc16474688517F2a79472465fae00a9e8
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
●	LOPS	Lack of Pair Synchronization	Unresolved
●	MSM	Missing Swap Mutex	Unresolved
●	TSD	Total Supply Diversion	Unresolved
●	TFD	Transfer Functions Distinction	Unresolved
●	CR	Code Repetition	Unresolved
●	DSM	Data Structure Misuse	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	IMUIC	Inefficient Maximum Unsigned Integer Calculation	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	RFC	Redundant Fee Calculation	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved

●	L09	Dead Code Elimination	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

Table of Contents

Review

Contract Name	EdaToken
Compiler Version	v0.8.10+commit.fc410830
Optimization	200 runs
Explorer	https://bscscan.com/address/0x6fd2233fc16474688517f2a79472465fae00a9e8
Address	0x6fd2233fc16474688517f2a79472465fae00a9e8
Network	BSC
Symbol	EDA
Decimals	18
Total Supply	100,000,000,000,000
Badge Eligibility	Must Fix Criticals

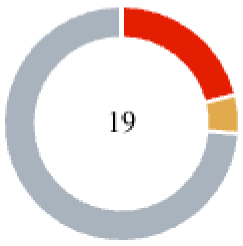
Audit Updates

Initial Audit	20 Feb 2024

Source Files

Filename	SHA256
EdaToken.sol	825db1a10efea6714d766a91a3f374a48b2d2f482ac0d93c8ec16c6983cebb47

Findings Breakdown



🟥	Critical	4
🟧	Medium	1
⬜	Minor / Informative	14

Severity		Unresolved	Acknowledged	Resolved	Other
🟥	Critical	4	0	0	0
🟧	Medium	1	0	0	0
⬜	Minor / Informative	14	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	EdaToken.sol#L1151,1191
Status	Unresolved

Description

The contract enforces transaction limits to regulate the maximum transfer amount and maximum wallet holding of its token, set at 0.1% and 0.5% of the total supply, respectively. While these checks are designed to prevent excessive concentration of tokens in a single address and ensure fair distribution, they pose a significant risk to liquidity pools. Given the nature of liquidity pools to accumulate large quantities of tokens to facilitate trading, these restrictions could block buy and sell transactions involving the pool. Specifically, if a liquidity pool's balance exceeds 0.5% of the total token supply, any further attempts to transfer tokens to the pool (e.g., selling tokens) could be reverted by the contract, disrupting normal trading activities.

```
if (_msgSender() != owner()) {
  require(amount <= ((totalSupply() * 1) / 10) / 100, 'eda: maximum
transfer allowed is 0.1% of total supply');
}
if (_msgSender() != owner()) {
  require(
    amount.add(balanceOf(recipient)) <= ((totalSupply() * 1) / 2) / 100,
    'eda: maxiumum wallet holding for all addresses is 0.5% of total
supply'
  );
}
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so transactions involving liquidity pools are not reverted due to balance restrictions. By doing so, the contract can better accommodate the operational needs of liquidity pools while maintaining its intended security and distribution controls, thereby supporting a stable and liquid market for its tokens.

LOPS - Lack of Pair Synchronization

Criticality	Critical
Location	EdaToken.sol#L1184
Status	Unresolved

Description

In the contract's transfer flow, a significant operation involves transferring funds to the dead address from the active PancakeSwap pair, which is intended to remove liquidity permanently or reduce available supply, thereby potentially affecting the token's price. However, after executing this operation, the contract does not perform a synchronization of the PancakeSwap pair. This omission can lead to discrepancies between the actual token balances and the internal reserves tracked by the PancakeSwap pair contract, potentially impacting the accuracy of liquidity and price calculations within the decentralized exchange (DEX).

```
_transfer(_pancakeV2Pair, _dead, (_amounts[0] * 90) / 100);
```

Recommendation

To ensure the integrity of the liquidity pool and accurate price determination on the DEX, it is crucial to synchronize the pair's reserves after directly manipulating token balances that affect the liquidity pool. Implementing this synchronization allows the DEX to accurately reflect the current state of reserves, ensuring that trading operations are based on the latest and most accurate information. After completing the transfer of funds to the dead address, the team could call the synchronization function of the PancakeSwap pair. This can typically be achieved by invoking the `sync` function on the pair contract, which updates the reserves to match the current balances of the tokens in the liquidity pool.

MSM - Missing Swap Mutex

Criticality	Critical
Location	EdaToken.sol#L1164
Status	Unresolved

Description

Within the token transfer logic, the contract includes a mechanism for swapping its own tokens for ETH using the `swapTokensForExactETH` function provided by the IPancakeRouter01 interface. This operation is initiated when the contract's token balance exceeds a certain threshold relative to the total supply. Following the swap, the contract attempts to add liquidity and then transfers a portion of the received tokens to a designated address. The absence of a reentrancy guard (mutex) in this transaction sequence exposes the contract to potential reentrancy attacks or infinite loops since the contract will call recursively the transfer function.

```
if (balanceOf(address(this)) >= ((totalSupply() * 1) / 100) / 100) {
  uint256 _slashed = balanceOf(address(this)).div(2);
  IPancakeRouter01 pRouter = IPancakeRouter01(_router);
  address[] memory path = new address[](2);
  path[0] = address(this);
  path[1] = pRouter.WETH();
  uint256[] memory _amounts = pRouter.getAmountsOut(_slashed, path);
  uint256 reserveETH = IERC20(path[1]).balanceOf(_pancakeV2Pair);
  ...
}
```

Recommendation

The team is advised to safeguard the swapping process and ensure the integrity of the token swap and liquidity addition operations, by implementing a reentrancy guard. By implementing a reentrancy guard and adhering to best practices for smart contract security, the contract can significantly mitigate the risk of reentrancy attacks and infinite loops in transactions, thereby protecting its operations and the assets it manages.

TSD - Total Supply Diversion

Criticality	Critical
Location	EdaToken.sol#L1245,1260
Status	Unresolved

Description

The total supply of a token is the total number of tokens that have been created, while the balances of individual accounts represent the number of tokens that an account owns. The total supply and the balances of individual accounts are two separate concepts that are managed by different variables in a smart contract. These two entities should be equal to each other.

In the contract, the amount that is added to the total supply does not equal the amount that is added to the balances. As a result, the sum of balances is diverse from the total supply.

Specifically, the contract transfers 20% of the fee amount to the contract holders and 10% of the fee amount to the active liquidity pools, whose balance is greater than zero. However, these conditions impact the number of tokens transferred. Even if one holder has zero balance or 1 liquidity pool is inactive the total supply will diverge from the sum of balances.

```
function _distributeToHolders(uint256 fee) private {
  if (fee > 0) {
    uint256 percentage = (fee * 20) / 100;
    uint256 division = percentage / _holders.length;

    for (uint256 i; i < _holders.length; i++) {
      if (balanceOf(_holders[i]) > 0) {
        _transfer(_msgSender(), _holders[i], division);
      }
    }
  }
}
function _accumulateForLPs(uint256 fee) private {
  if (fee > 0) {
    uint256 percentage = (fee * 10) / 100;
    uint256 _division = percentage / _lps.length;
    for (uint256 i = 0; i < _lps.length; i++) {
      if (_lpsActive[_lps[i]]) {
        _transfer(_msgSender(), _lps[i], _division);
      }
    }
  }
}
```

Recommendation

The total supply and the balance variables are separate and independent from each other. The total supply represents the total number of tokens that have been created, while the balance mapping stores the number of tokens that each account owns. The sum of balances should always equal the total supply.

TFD - Transfer Functions Distinction

Criticality	Medium
Location	EdaToken.sol#L269,1142
Status	Unresolved

Description

The `transfer` and `transferFrom` functions of an ERC20 token are used to transfer tokens from one user to another. The `transfer` function is augmented with additional features such as a fee mechanism, token swapping, and restrictions on the maximum transfer amount and wallet holdings as a percentage of the total supply. Conversely, the `transferFrom` function adheres to the standard ERC20 transfer mechanism, focusing solely on transferring the specified amount from the sender to the recipient, with allowance checks to ensure the transaction does not exceed the sender's permitted amount.

This discrepancy might introduce complexity and potential confusion for users and external contracts interacting with the token, as the effects and outcomes of transferring tokens depend on the method used. For example, a `transfer` operation might result in fees, redistribution of tokens, and liquidity provision, while a `transferFrom` operation would bypass these mechanisms entirely.

```
function transfer(address recipient, uint256 amount) public override
returns (bool) {
  uint256 _percentage = (amount * 10) / 100;
  uint256 _rAmount = amount.sub(_percentage);
  uint256 _splitAmount = amount.sub(_rAmount);

  if (!_isExcluded[_msgSender()]) {
    ...
    _transfer(_msgSender(), recipient, _rAmount);
    ...
  } else {
    ...
    _transfer(_msgSender(), recipient, amount);
  }
  return true;
}

function transferFrom(
  address sender,
  address recipient,
  uint256 amount
) public virtual override returns (bool) {
  _transfer(sender, recipient, amount);

  uint256 currentAllowance = _allowances[sender][_msgSender()];
  require(currentAllowance >= amount, 'ERC20: transfer amount exceeds
allowance');
  unchecked {
    _approve(sender, _msgSender(), currentAllowance - amount);
  }

  return true;
}
```

Recommendation

To address this inconsistency and ensure a uniform experience for all token transfers, the team is advised to align the functionalities of `transfer` and `transferFrom` functions by either incorporating the additional mechanisms (fee, token swapping, etc.) into both functions or simplifying the transfer function to match the standard behavior observed in transferFrom. The choice should be based on the token's intended use case and economic model. By aligning the functionalities of both functions, the contract can provide a consistent and secure experience for token transfers, aligning with user expectations and the principles of the ERC20 standard, while still supporting the token's unique economic model and features.

CR - Code Repetition

Criticality	Minor / Informative
Location	EdaToken.sol#L1148,1188,1247,1262
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
if (_msgSender() != owner()) {
  require(_rAmount <= ((totalSupply() * 1) / 10) / 100, 'eda: maximum
transfer allowed is 0.1% of total supply');
}
if (_msgSender() != owner()) {
  require(
    _rAmount.add(balanceOf(recipient)) <= ((totalSupply() * 1) / 2) / 100,
    'eda: maxiumum wallet holding for all addresses is 0.5% of total
supply'
  );
}
...
uint256 percentage = (fee * 20) / 100;
uint256 division = percentage / _holders.length;
uint256 percentage = (fee * 10) / 100;
uint256 _division = percentage / _lps.length;
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

DSM - Data Structure Misuse

Criticality	Minor / Informative
Location	EdaToken.sol#L1057
Status	Unresolved

Description

The contract uses the variable `_holders` as an array. The business logic of the contract does not require iterating this structure sequentially. Thus, unnecessary loops are produced that increase the required gas.

```
address[] _holders;
```

Recommendation

The contract could use a data structure that provides instant access. For instance, a Set or a Map would fit better to the business logic of the contract. This way the time complexity will be reduced from o(n) to o(1).

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	EdaToken.sol#L1158,1159,1160,1161,1162
Status	Unresolved

Description

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

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

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

```
_distributeToHolders(_splitAmount);
_accumulateForLPs(_splitAmount);
_transfer(_msgSender(), _marketing, (_splitAmount * 20) / 100);
_transfer(_msgSender(), address(this), (_splitAmount * 10) / 100);
_transfer(_msgSender(), _dead, (_splitAmount * 40) / 100);
```

Recommendation

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

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	EdaToken.sol#L1079,1080,1084
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
_router
_pancakeV2Pair
_marketing
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

IMUIC - Inefficient Maximum Unsigned Integer Calculation

Criticality	Minor / Informative
Location	EdaToken.sol#L1222,1236,1238
Status	Unresolved

Description

The contract utilizes a convoluted method to determine the maximum unsigned integer value by first casting -1 to an int256, and then to a uint256 within the `_indexOf` function. This approach indirectly achieves the goal of initializing `j` with the maximum possible uint256 value to signify an unsuccessful search when an address does not match any entry in the `_addresses` array. However, this method is not straightforward and may lead to confusion or inefficiencies in code understanding and execution. Additionally, the contract does not break from the loop once the address is found within the array, hence the contract will keep iterating until it reaches the end of the array.

```
function _indexOf(address[] memory _addresses, address a) private pure
returns (uint256 j) {
  j = uint256(int256(-1));
  for (uint256 i; i < _addresses.length; i++) {
    if (_addresses[i] == a) j = i;
  }
}
function _beforeTokenTransfer(
  address from,
  address to,
  uint256 amount
) internal override {
  if (_indexOf(_holders, from) == uint256(int256(-1)))
_holders.push(from);

  if (_indexOf(_holders, to) == uint256(int256(-1))) _holders.push(to);

  emit BeforeTransferHash(keccak256(abi.encodePacked(from, to, amount)));
}
```

Recommendation

For clarity and efficiency, the team is recommended to directly use Solidity's built-in constant for the maximum value of a uint256. This can be achieved by initializing `j` with `type(uint256).max`, which clearly conveys the intent and eliminates the need for type casting. Additionally, introducing a break statement after finding the address can optimize the loop by terminating it early, further improving the function's efficiency. This modification simplifies the code, making it more readable and slightly more gas efficient by directly utilizing Solidity's language features for maximum value representation and optimizing loop execution.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	EdaToken.sol#L1127,1136,1212,1218
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.

```
_isExcluded[_ex] = true;
_isExcluded[_inc] = false;
_lpsActive[lp] = true;
_lpsActive[lp] = false;
```

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.

RFC - Redundant Fee Calculation

Criticality	Minor / Informative
Location	EdaToken.sol#L1145
Status	Unresolved

Description

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

As part of the transfer flow, there is an inefficiency in how the fee amount and the recipient's transfer amount are calculated. The contract initially calculates a fee by taking 10% of the transfer amount (`(amount * 10) / 100`), and then determines the reduced amount (`_rAmount`) to be transferred to the recipient by subtracting this fee from the total amount. Subsequently, it redundantly recalculates the fee amount by subtracting `_rAmount` from the original amount, which is essentially the same value calculated initially. This redundancy leads to unnecessary computations, potentially increasing gas costs for users executing this function.

```
uint256 _percentage = (amount * 10) / 100;
uint256 _rAmount = amount.sub(_percentage);
uint256 _splitAmount = amount.sub(_rAmount);
```

Recommendation

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

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	EdaToken.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	EdaToken.sol#L1062
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 _dead = 0x000000000000000000000000000000000000dEaD
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	EdaToken.sol#L826,828,854,900,1098,1099,1100,1114,1124,1133
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
function DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint256);
function WETH() external pure returns (address);
address _token
address _recipient
uint256 _amount
address payable _recipient
address _ex
address _inc
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	EdaToken.sol#L394,450
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function _burn(address account, uint256 amount) internal virtual {
    require(account != address(0), 'ERC20: burn from the zero address');

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

    uint256 accountBalance = _balances[account];
...
    }
    _totalSupply -= amount;

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

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

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	EdaToken.sol#L1079,1084
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
_router = router
_marketing = marketing_
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	EdaToken.sol#L1,81,105,127,477,549,774,798,895,1050
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity >=0.5.0;
pragma solidity >=0.6.2;
pragma solidity ^0.8.0;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	EdaToken.sol#L1,81,105,127,477,549,1050
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
IERC20Metadata	Interface	IERC20		
	name	External		-
	symbol	External		-
	decimals	External		-
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		

ERC20	Implementation	Context, IERC20, IERC20Metadata		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	_transfer	Internal	✓	
	_mint	Internal	✓	
	_burn	Internal	✓	
	_approve	Internal	✓	
	_beforeTokenTransfer	Internal	✓	
	_afterTokenTransfer	Internal	✓	
Ownable	Implementation	Context		
		Public	✓	-

	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
SafeMath	Library			
	tryAdd	Internal		
	trySub	Internal		
	tryMul	Internal		
	tryDiv	Internal		
	tryMod	Internal		
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	mod	Internal		
	sub	Internal		
	div	Internal		
	mod	Internal		
IPancakeFactory	Interface			
	feeTo	External		-
	feeToSetter	External		-

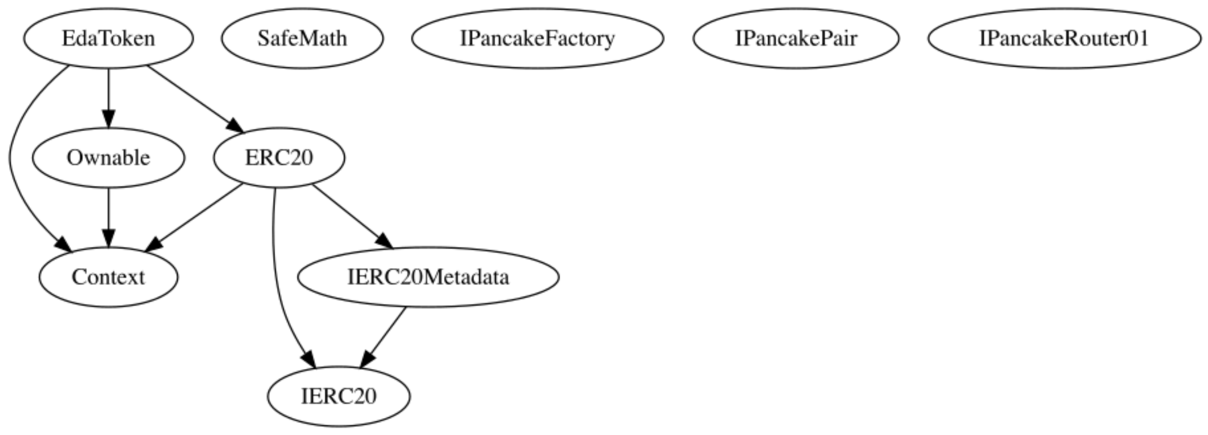
	getPair	External		-
	allPairs	External		-
	allPairsLength	External		-
	createPair	External	✓	-
	setFeeTo	External	✓	-
	setFeeToSetter	External	✓	-
IPancakePair	Interface			
	name	External		-
	symbol	External		-
	decimals	External		-
	totalSupply	External		-
	balanceOf	External		-
	allowance	External		-
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
	DOMAIN_SEPARATOR	External		-
	PERMIT_TYPEHASH	External		-
	nonces	External		-
	permit	External	✓	-
	MINIMUM_LIQUIDITY	External		-
	factory	External		-

	token0	External		-
	token1	External		-
	getReserves	External		-
	price0CumulativeLast	External		-
	price1CumulativeLast	External		-
	kLast	External		-
	mint	External	✓	-
	burn	External	✓	-
	swap	External	✓	-
	skim	External	✓	-
	sync	External	✓	-
	initialize	External	✓	-
IPancakeRouter01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
	removeLiquidity	External	✓	-
	removeLiquidityETH	External	✓	-
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-

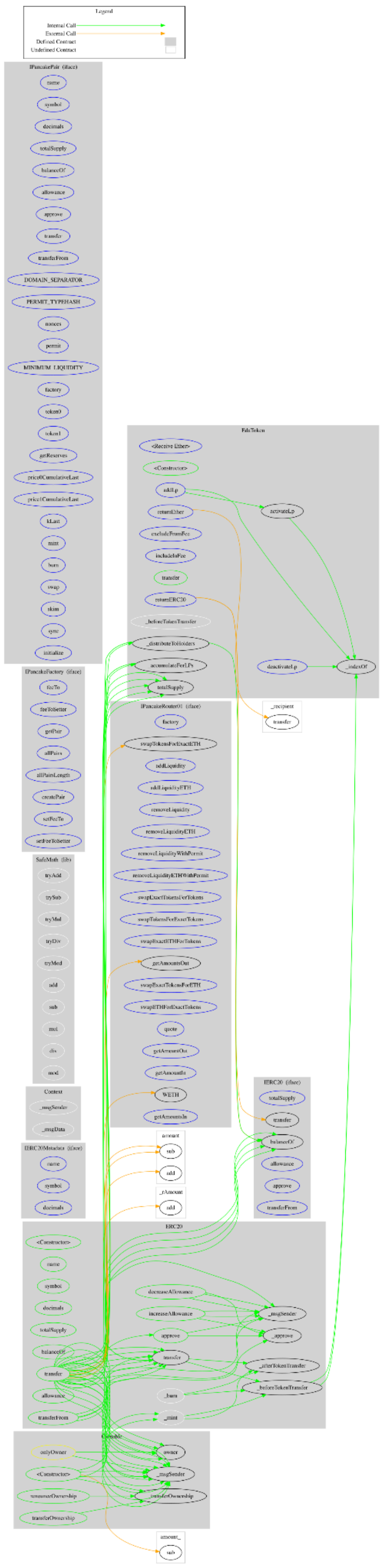
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-
	quote	External		-
	getAmountOut	External		-
	getAmountIn	External		-
	getAmountsOut	External		-
	getAmountsIn	External		-
EdaToken	Implementation	Context, ERC20, Ownable		
		External	Payable	-
		Public	✓	Ownable ERC20
	returnERC20	External	✓	onlyOwner
	returnEther	External	✓	onlyOwner
	excludeFromFee	External	✓	onlyOwner
	includeInFee	External	✓	onlyOwner
	transfer	Public	✓	-
	addLp	External	✓	onlyOwner
	activateLp	Public	✓	onlyOwner
	deactivateLp	External	✓	onlyOwner
	_indexOf	Private		

	_beforeTokenTransfer	Internal	✓	
	_distributeToHolders	Private	✓	
	_accumulateForLPs	Private	✓	

Inheritance Graph



Flow Graph



Summary

Eda Token contract implements a token mechanism. This audit investigates security issues, business logic concerns, and potential improvements. Eda Token is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler errors and one critical issue. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions. There is also a fixed 10% fee when interacting with the `transfer` function.

Disclaimer

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io