Cyberscope

Audit Report
ETFSwap

April 2024

Repository	https://github.com/hamzabadshah1/etfswap
Commit	564c7df2a60856313d9b6885628c9eb822049fa8
Audited by	© cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

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

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	IVL	Inadequate Vested Limit	Unresolved
🟠	IFC	Inefficient Function Complexity	Unresolved
🟠	MTV	Misleading Tax Variable	Unresolved
⚪	EIS	Excessively Integer Size	Unresolved
⚪	IAC	Inefficient Amount Calculation	Unresolved
⚪	MTEE	Missing Transfer Event Emission	Unresolved
⚪	RLC	Redundant Logic Checks	Unresolved
⚪	RSML	Redundant SafeMath Library	Unresolved
⚪	RSW	Redundant Storage Writes	Unresolved
⚪	L04	Conformance to Solidity Naming Conventions	Unresolved
⚪	L13	Divide before Multiply Operation	Unresolved
⚪	L18	Multiple Pragma Directives	Unresolved
⚪	L19	Stable Compiler Version	Unresolved

Table of Contents

Review

Contract Name	ETFSwap
Repository	https://github.com/hamzabadshah1/etfswap
Commit	76fa7f7b0bb730ecb0db2302679b24de2f764f05
Testing Deploy	https://testnet.bscscan.com/address/0x1ef60793caaa1ae689ca24906c565b26ca377f78
Symbol	ETFS
Decimals	18
Total Supply	1,000,000,000
Badge Eligibility	Must Fix Criticals

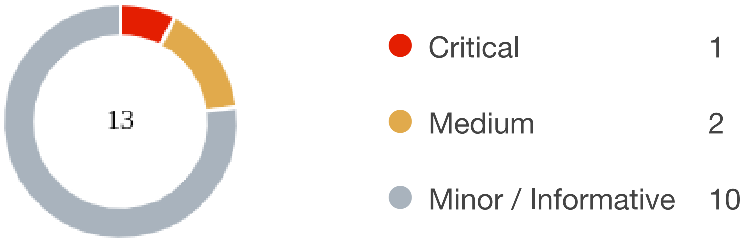
Audit Updates

Initial Audit	27 Mar 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v1/audit.pdf
Corrected Phase 2	04 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v2/audit.pdf
Corrected Phase 3	05 Apr 2024

Source Files

Filename	SHA256
contracts/ETFSwap.sol	0375ef314de8d183efb4c0ebe312346c0cbc0042cfde1c4a7949500b91fa3774

Findings Breakdown



Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	1	0	0	0
●	Medium	2	0	0	0
●	Minor / Informative	10	0	0	0

IVL - Inadequate Vested Limit

Criticality	Critical
Location	contracts/ETFSwap.sol#L250,271
Status	Unresolved

Description

The contract contains logic to set individual vested limits for team members and presale participants. However, it does not account for cases where an address has already vested the total allowed amount under its previous limit. The current implementation deducts the old limit and adds the new limit to calculate the total vested amount against the `TEAM_ALLOCATION` or `PRESALE_ALLOCATION`. This approach can inadvertently allow the new limit to exceed the total intended allocation if not properly checked against already vested amounts. Essentially, the contract assumes the total vested amount can be adjusted by simply changing the limit, without verifying whether the funds under the previous limit have been fully utilized or not. This could lead to scenarios where the new limits bypass the total allocation constraints, potentially resulting in over-allocation.

```
   // Function to set individual vested limit for team members
    function setIndividualTeamVestedLimit(
        address _address,
        uint256 limit
    ) external onlyOwner {
        require(
            getTotalTeamVestedAmount() -
                individualTeamVestedLimit[_address] +
                limit <=
                TEAM_ALLOCATION,
            "Total allocated amount exceeded for team members"
        );
        individualTeamVestedLimit[_address] = limit;
        emit IndividualTeamVestedLimitSet(_address, limit);
    }

    // Function to set individual vested limit for presale
participants
    function setIndividualPresaleVestedLimit(
        address _address,
        uint256 limit
    ) external onlyOwner {
        require(
            getTotalPresaleVestedAmount() -
                individualPresaleVestedLimit[_address] +
                limit <=
                PRESALE_ALLOCATION,
            "Total allocated amount exceeded for presale
participants"
        );
        individualPresaleVestedLimit[_address] = limit;
        emit IndividualPresaleVestedLimitSet(_address, limit);
    }
```

Recommendation

It is recommended to refactor the limit-setting functionality to include checks against amounts already vested by an address. This adjustment ensures that changing an individual's vested limit does not inadvertently allow for the total allocated amount to be exceeded. Specifically, before updating an individual's limit, verify whether the new limit respects the total allocation when considering the amount already vested to that address. If the intent is to permit limit adjustments for already vested amounts, the contract must enforce strict validations to prevent over-allocation. Implementing such safeguards will maintain the integrity of the allocation process and ensure adherence to predefined limits.

IFC - Inefficient Function Complexity

Criticality	Medium
Location	contracts/ETFSwap.sol#L250,267,448,462,501,531
Status	Unresolved

Description

The contract incorporates multiple functions to manage the presale and team whitelist members, including setting individual vested limits, adding addresses to whitelists, and specifying team or presale addresses. This approach introduces additional complexity and redundancy, as each category (team and presale) has separate but functionally similar methods for managing whitelist status, vesting limits, and vesting start times. This redundancy not only makes the contract more cumbersome to interact with but also increases the potential for errors and inconsistencies in functionality. By having distinct paths for essentially parallel operations, the contract's maintainability and readability are adversely affected, complicating future updates.

```
   // Function to set individual vested limit for team members
    function setIndividualTeamVestedLimit(
        address _address,
        uint256 limit
    ) external onlyOwner {
        require(
            getTotalTeamVestedAmount() -
                individualTeamVestedLimit[_address] +
                limit <=
                TEAM_ALLOCATION,
            "Total allocated amount exceeded for team members"
        );
        individualTeamVestedLimit[_address] = limit;
        emit IndividualTeamVestedLimitSet(_address, limit);
    }

    // Function to set individual vested limit for presale participants
    function setIndividualPresaleVestedLimit(
        address _address,
        uint256 limit
    ) external onlyOwner {
        require(
            getTotalPresaleVestedAmount() -
                individualPresaleVestedLimit[_address] +
                limit <=
                PRESALE_ALLOCATION,
            "Total allocated amount exceeded for presale participants"
        );
        individualPresaleVestedLimit[_address] = limit;
        emit IndividualPresaleVestedLimitSet(_address, limit);
    }


    function addToTeamWhitelist(address _address) external onlyOwner {
        require(_address != address(0) && _address != owner, "Invalid address");
        if (!teamWhitelist[_address]) {
            teamWhitelist[_address] = true;
            if (
                isInTeamAddresses(_address) && _teamVestingStart[_address] == 0
            ) {
                _teamVestingStart[_address] = block.timestamp;
            }
            emit AddedToWhitelist(_address);
        }
    }
    // Function to add an address to the presale whitelist and
initialize vesting start time
    function addToPresaleWhitelist(address _address) external
onlyOwner {
        require(_address != address(0) && _address != owner,
"Invalid address");
        if (!presaleWhitelist[_address]) {
            presaleWhitelist[_address] = true;
            if (
                isInPresaleAddresses(_address) &&
                _presaleVestingStart[_address] == 0
            ) {
                _presaleVestingStart[_address] =
block.timestamp;
            }
            emit AddedToWhitelist(_address);
        }
    }

    function setTeamAddress(address _teamAddress) external
onlyOwner {
        require(
            _teamAddress != address(0) && _teamAddress !=
owner,
            "Invalid address"
        );
        require(
            getTotalTeamAllocation() +
                individualTeamVestedLimit[_teamAddress] <=
                TEAM_ALLOCATION,
            "Total team allocation limit reached"
        );
        if (!isInTeamAddresses(_teamAddress)) {
            teamAddresses.push(_teamAddress);
            if (_teamVestingStart[_teamAddress] == 0) {
                _teamVestingStart[_teamAddress] =
block.timestamp;
                emit VestingStartInitialized(_teamAddress,
block.timestamp);
            }
        }
    }

    function setPresaleAddress(address _presaleAddress)
external onlyOwner {
        require(
            _presaleAddress != address(0) && _presaleAddress !=
owner,
            "Invalid address"
        );
        require(
            getTotalPresaleAllocation() +
                individualPresaleVestedLimit[_presaleAddress]
<=
                PRESALE_ALLOCATION,
            "Total team allocation limit reached"
        );
        if (!isInPresaleAddresses(_presaleAddress)) {
            presaleAddresses.push(_presaleAddress);
            if (_presaleVestingStart[_presaleAddress] == 0) {
                _presaleVestingStart[_presaleAddress] =
block.timestamp;
                emit VestingStartInitialized(_presaleAddress,
block.timestamp);
            }
        }
    }
```

Recommendation

It is recommended to refactor the code to consolidate the similar functionalities into unified functions. This consolidation can be achieved by parameterizing the functions to accept a type argument (e.g., team or presale) and thereby execute the corresponding logic based on the type. Such an approach would significantly reduce the codebase's complexity, streamline interactions, and enhance the clarity of the contract's operations. Implementing these changes would improve the contract's efficiency, reduce deployment and transaction costs, and lower the risk of bugs or logical inconsistencies arising from redundant code.

MTV - Misleading Tax Variable

Criticality	Medium
Location	contracts/ETFSwap.sol#L218
Status	Unresolved

Description

The contract is intended to impose a tax rate on buy transactions through the `buyTaxRate` variable. This naming suggests that the tax should only apply to buy operations. However, the contract's logic does not differentiate between buy transactions and other types of transfers, leading to the application of the `buyTaxRate` to all transfer transactions. This approach not only deviates from the expected behavior implied by the variable's name but also introduces a misleading interpretation of the contract's functionality. The current implementation, as highlighted by the code snippet, indiscriminately applies the `buyTaxRate` to any tokens being transferred, without distinguishing if the transaction is a buy or a different form of transfer. This inconsistency can lead to confusion and potentially unintended financial implications for users interacting with the contract.

```
// Apply buy tax rate if the tokens are being transferred to
another address
return tokens.mul(buyTaxRate).div(100);
```

Recommendation

It is recommended to rename the variable to accurately reflect its functionality. If the tax rate is intended to apply universally to all transfers, a more generic name should be considered. This change would eliminate ambiguity and align the variable's name with its actual application within the contract. Furthermore, if distinguishing between different transaction types is desired for future implementation, additional logic should be incorporated to accurately apply taxes based on the nature of each transaction. This approach will enhance clarity, improve user understanding, and ensure the contract's operations are transparent and aligned with its intended design.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L46,47
Status	Unresolved

Description

The contract is using a bigger unsigned integer data type that the maximum size that is required. By using an unsigned integer data type larger than necessary, the smart contract consumes more storage space and requires additional computational resources for calculations and operations involving these variables. This can result in higher transaction costs, longer execution times, and potential scalability bottlenecks.

```
uint256 public sellTaxRate;
uint256 public buyTaxRate;
```

Recommendation

To address the inefficiency associated with using an oversized unsigned integer data type, it is recommended to accurately determine the required size based on the range of values the variable needs to represent.

IAC - Inefficient Amount Calculation

Criticality	Minor / Informative
Status	Unresolved

Description

The contract utilizes functions `getTotalTeamVestedAmount` and `getTotalPresaleVestedAmount` to calculate the total vested amounts for team members and presale participants, respectively. These functions iterate over arrays of addresses (`teamAddresses` and `presaleAddresses`) to sum up the vested amounts stored in corresponding mappings. This iterative approach, while straightforward, is inefficient and could lead to increased gas costs during execution, especially as the number of addresses grows. More importantly, this method is called repeatedly in contexts where maintaining up-to-date totals is crusial, further compounding the inefficiency.

```
   // Function to calculate the total vested amount for all
team members
    function getTotalTeamVestedAmount() private view returns
(uint256) {
        uint256 totalAmount = 0;
        for (uint256 i = 0; i < teamAddresses.length; i++) {
            totalAmount +=
totalTeamVestedAmount[teamAddresses[i]];
        }
        return totalAmount;
    }

    // Function to calculate the total vested amount for all
presale participants
    function getTotalPresaleVestedAmount() private view returns
(uint256) {
        uint256 totalAmount = 0;
        for (uint256 i = 0; i < presaleAddresses.length; i++) {
            totalAmount +=
totalPresaleVestedAmount[presaleAddresses[i]];
        }
        return totalAmount;
    }
```

Recommendation

It is recommended to optimize the contract's efficiency by maintaining running totals of the vested amounts for both team members and presale participants as global state variables. Instead of recalculating these totals via iteration each time they are needed, the contract should update the totals dynamically whenever a vested amount is added or modified. This strategy involves adjusting the global totals in tandem with any change to an individual's vested amount—both during initial assignment and any subsequent updates. Implementing this change will not only reduce gas costs by eliminating the need for iterative calculations but also simplify the logic related to managing vested amounts. Furthermore, this approach ensures that the totals are always current and readily available for any checks or operations requiring up-to-date information, enhancing the contract's performance and reliability.

MTEE - Missing Transfer Event Emission

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L169
Status	Unresolved

Description

The contract is a missing transfer event emission when fees are transferred to the contract address as part of the transfer process. This omission can lead to a lack of visibility into fee transactions and hinder the ability of decentralized applications (DApps) like blockchain explorers to accurately track and analyze these transactions.

```
// Mint initial allocations
balances[msg.sender] += PRESALE_ALLOCATION;
balances[msg.sender] += ECOSYSTEM_ALLOCATION;
balances[msg.sender] += LIQUIDITY_ALLOCATION;
balances[msg.sender] += CASHBACK_ALLOCATION;
balances[msg.sender] += PARTNERS_ALLOCATION;
balances[msg.sender] += COMMUNITY_REWARDS_ALLOCATION;
balances[msg.sender] += MM_ALLOCATION;
balances[msg.sender] += TEAM_ALLOCATION;
```

Recommendation

To address this issue, it is recommended to emit a transfer event after transferring the taxed amount to the contract address. The event should include relevant information such as the sender, recipient (contract address), and the amount transferred.

RLC - Redundant Logic Checks

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L453,467
Status	Unresolved

Description

The contract contains functions `isTeamWhitelisted` and `isPresaleWhitelisted` designed to check if an address is whitelisted for team and presale participation, respectively. However, the implementation of these checks are redundant because the corresponding mappings (`teamWhitelist` and `presaleWhitelist`) are updated to true before these functions can effectively perform any meaningful validation. Specifically, the logic within `addToTeamWhitelist` and `addToPresaleWhitelist` sets an address as whitelisted prior to any operations that might rely on the state of being whitelisted, such as initializing vesting start times. This approach diminishes the utility of the `isInTeamAddresses` and `isInPresaleAddresses` checks, as the mappings already reflect the whitelisted status, rendering separate checks for whitelisting status potentially superfluous.

```
   // Function to add an address to the whitelist and
initialize vesting start time
    function addToTeamWhitelist(address _address) external
onlyOwner {
        require(_address != address(0) && _address != owner,
"Invalid address");
        if (!teamWhitelist[_address]) {
            teamWhitelist[_address] = true;
            if (
                isInTeamAddresses(_address) &&
_teamVestingStart[_address] == 0
            ) {
           ...
    }

    // Function to add an address to the presale whitelist and
initialize vesting start time
    function addToPresaleWhitelist(address _address) external
onlyOwner {
        require(_address != address(0) && _address != owner,
"Invalid address");
        if (!presaleWhitelist[_address]) {
            presaleWhitelist[_address] = true;
            if (
                isInPresaleAddresses(_address) &&
                _presaleVestingStart[_address] == 0
            ...
    }
```

Recommendation

It is recommended to revise the implementation to ensure that the checks for an address's whitelisted status are meaningful and contribute to the contract's logic. This could involve reevaluating the use of `isInTeamAddresses` and `isInPresaleAddresses` checks to ensure they precede any changes to an address's whitelisted status, thereby preserving their intended functionality. If these functions are intended to perform additional validations or set conditions before an address can be whitelisted, their logic should be clearly defined and executed before modifying the whitelisting mappings. Alternatively, if the current whitelisting checks are deemed unnecessary, consider streamlining the contract by removing or repurposing them to avoid confusion and reduce complexity. Ensuring that each part of the contract's code has a clear and meaningful purpose will enhance its readability, maintainability, and overall security.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol @openzeppelin/contracts/utils/math/SafeMath.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 in cases where the explanatory error message is not used.

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

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. 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.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	,contracts/ETFSwap.sol#L117,
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
   function setLiquidityPairAddress(
        address _liquidityPairAddress
    ) external onlyOwner {
        require(
            _liquidityPairAddress != address(0),
            "Liquidity Pair can't be a null address"
        );
        liquidityPairAddress = _liquidityPairAddress;
        emit LiquidityPairAddressSet(
            _liquidityPairAddress,
            msg.sender,
            block.timestamp
        );
    }

    // Function to add an address to the whitelist and
initialize vesting start time
    function addToTeamWhitelist(address _address) external
onlyOwner {
        require(_address != address(0) && _address != owner,
"Invalid address");
        if (!teamWhitelist[_address]) {
            teamWhitelist[_address] = true;
            if (
                isInTeamAddresses(_address) &&
_teamVestingStart[_address] == 0
            ) {
                _teamVestingStart[_address] = block.timestamp;
            }
            emit AddedToWhitelist(_address);
        }
    }

    // Function to add an address to the presale whitelist and
initialize vesting start time
    function addToPresaleWhitelist(address _address) external
onlyOwner {
        require(_address != address(0) && _address != owner,
"Invalid address");
        if (!presaleWhitelist[_address]) {
            presaleWhitelist[_address] = true;
            if (
                isInPresaleAddresses(_address) &&
                _presaleVestingStart[_address] == 0
            ) {
                _presaleVestingStart[_address] =
block.timestamp;
            }
            emit AddedToWhitelist(_address);
        }
    }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L118,227,228,239,245,252,268,448,462,501,531
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.

```
address _liquidityPairAddress
address _address
address[] storage _list
address _teamAddress
address _presaleAddress
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L350,351
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 vestingPeriods = elapsedTime / RELEASE_INTERVAL
uint256 vestedAmount =
totalAllocation.mul(vestingPeriods).div(5)
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L2 @openzeppelin/contracts/utils/math/SafeMath.sol#L4 @openzeppelin/contracts/token/ERC20/IERC20.sol#L4
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 ^0.8.0;
pragma solidity ^0.8.19;
```

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	contracts/ETFSwap.sol#L2 @openzeppelin/contracts/utils/math/SafeMath.sol#L4 @openzeppelin/contracts/token/ERC20/IERC20.sol#L4
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 ^0.8.19;
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.

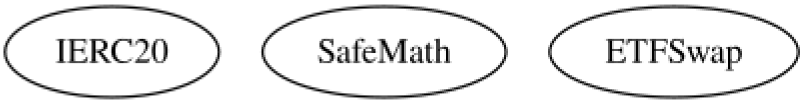
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ETFSwap	Implementation			
		Public	✓	-
	totalSupply	Public		-
	setLiquidityPairAddress	External	✓	onlyOwner
	balanceOf	Public		-
	_transferTokens	Internal	✓	
	transfer	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	calculateTaxAmount	Private		
	isInAddressList	Private		
	isInTeamAddresses	Private		
	isInPresaleAddresses	Private		
	setIndividualTeamVestedLimit	External	✓	onlyOwner
	setIndividualPresaleVestedLimit	External	✓	onlyOwner
	getTotalTeamVestedAmount	Private		
	getTotalPresaleVestedAmount	Private		

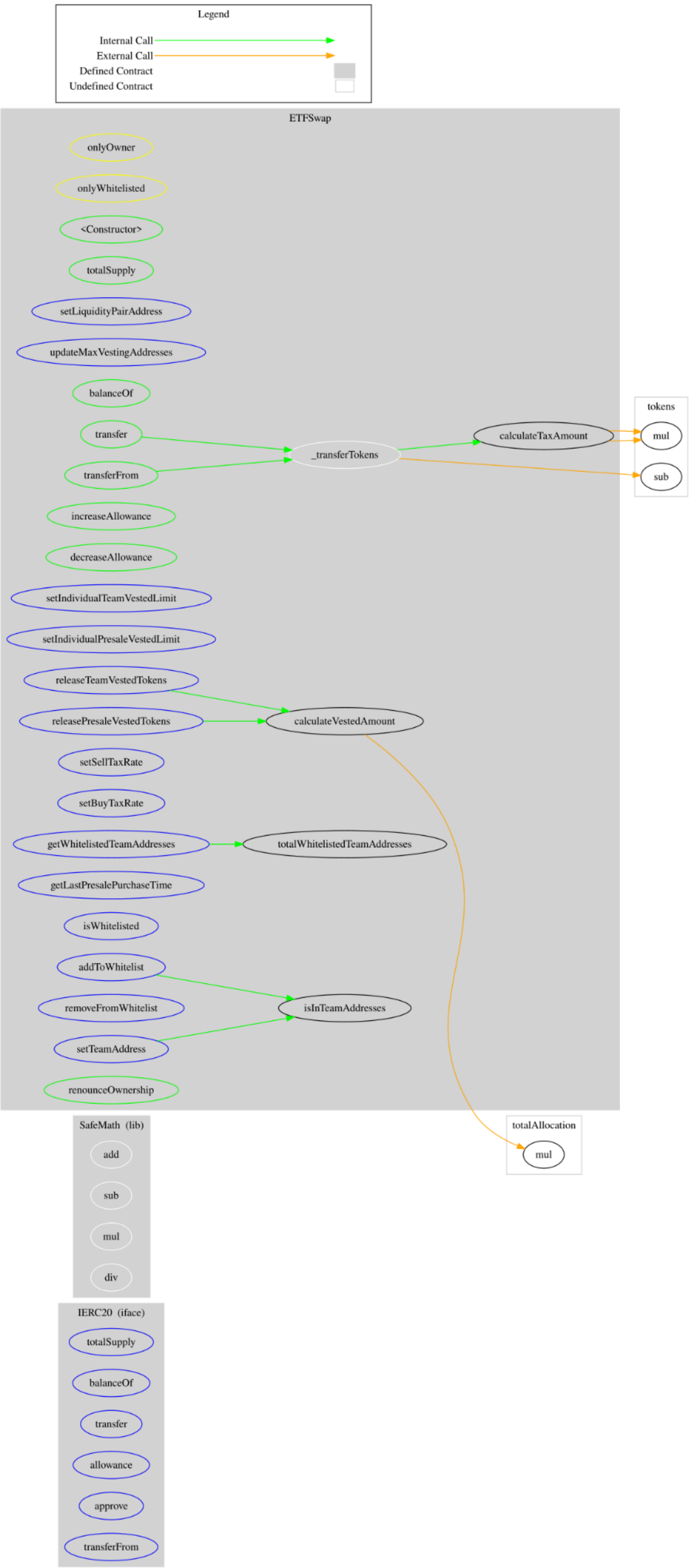
	releaseTeamVestedTokens	External	✓	onlyTeamAddresses
	releasePresaleVestedTokens	External	✓	onlyWhitelisted
	_releaseVestedTokens	Internal	✓	
	calculateVestedAmount	Private		
	setSellTaxRate	External	✓	onlyOwner
	setBuyTaxRate	External	✓	onlyOwner
	getWhitelistedTeamAddresses	External		-
	getWhitelistedPresaleAddresses	External		-
	totalWhitelistedTeamAddresses	Public		-
	totalWhitelistedPresaleAddresses	Public		-
	isTeamWhitelisted	External		-
	isPresaleWhitelisted	External		-
	addToTeamWhitelist	External	✓	onlyOwner
	addToPresaleWhitelist	External	✓	onlyOwner
	removeFromTeamWhitelist	External	✓	onlyOwner
	removeFromPresaleWhitelist	External	✓	onlyOwner
	setTeamAddress	External	✓	onlyOwner
	getTotalTeamAllocation	Private		
	setPresaleAddress	External	✓	onlyOwner
	getTotalPresaleAllocation	Private		
	renounceOwnership	Public	✓	onlyOwner
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		
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
ETFSwap
onlyOwner
onlyWhitelisted
<Constructor>
totalSupply
setLiquidityPairAddress
updateMaxVestingAddresses
balanceOf
transfer
transferFrom
_transferTokens
calculateTaxAmount
tokens
mul
sub
increaseAllowance
decreaseAllowance
setIndividualTeamVestedLimit
setIndividualPresaleVestedLimit
releaseTeamVestedTokens
releasePresaleVestedTokens
calculateVestedAmount
setSellTaxRate
setBuyTaxRate
getWhitelistedTeamAddresses
totalWhitelistedTeamAddresses
getLastPresalePurchaseTime
isWhitelisted
addToWhitelist
removeFromWhitelist
isInTeamAddresses
setTeamAddress
renounceOwnership
SafeMath (lib)
add
sub
mul
div
totalAllocation
mul
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom

Summary

ETFSwap contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. ETFSwap is an interesting project that has a friendly and growing community. The Smart Contract analysis reported one critical error. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions. There is also a limit of max 25% fees.

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

https://www.cyberscope.io