Cyberscope

Audit Report

ETFSwap

April 2024

Repository https://github.com/hamzabadshah1/etfswap

Commit c6ade8d8d1df24f51de5d1480b3ab50548ff3747

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
Critical	IIVL	Inadequate Individual Vested Limit	Unresolved
Medium	CCC	Contradictory Condition Check	Unresolved
Medium	IFU	Inefficient Functions Usages	Unresolved
Medium	MTV	Misleading Tax Variable	Unresolved
Minor / Informative	IAC	Inefficient Amount Calculation	Unresolved
Minor / Informative	RSML	Redundant SafeMath Library	Unresolved
Minor / Informative	RTE	Redundant Transfer Events	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L08	Tautology or Contradiction	Unresolved
Minor / Informative	L13	Divide before Multiply Operation	Unresolved

Table of Contents

Review

Contract Name	ETFSwap
Repository	https://github.com/hamzabadshah1/etfswap
Commit	c6ade8d8d1df24f51de5d1480b3ab50548ff3747
Testing Deploy	https://testnet.bscscan.com/address/0x6df566a8968824efbe2632ab42395b244527d6ce
Symbol	ETFS
Decimals	18
Total Supply	1,000,000,000
Badge Eligibility	Must Fix Criticals

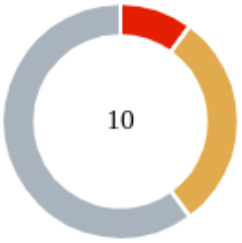
Audit Updates

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 https://github.com/cyberscope-io/audits/blob/main/etfswap/v3/audit.pdf
Corrected Phase 4	08 Apr 2024

Source Files

Filename	SHA256
contracts/ETFSwap.sol	7a3402e5d1342cc2197ece898f1a3bfb917fe44f4f063bd8c94e0d65c1ef6733

Findings Breakdown



Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	3	0	0	0
Minor / Informative	6	0	0	0

IIVL - Inadequate Individual Vested Limit

Criticality	Critical
Location	contracts/ETFSwap.sol#L303,325
Status	Unresolved

Description

The contract is implementing a require check within the `setIndividualTeamVestedLimit` and `setIndividualPresaleVestedLimit` functions to ensure that the new vested limit set for an address does not exceed the predetermined presale allocation. However, this check incorrectly deducts the previously set individual vested limit from the total presale vested amount calculation. This approach fails to account for the actual amount already minted and claimed by the address, potentially allowing for the setting of a new limit that, when combined with the already minted amount, exceeds the total presale allocation. This oversight in the require check logic could lead to scenarios where the total presale allocation is inadvertently exceeded, undermining the contract's intention to strictly enforce allocation limits.

```
require(
    getTotalTeamVestedAmount() -
        individualTeamVestedLimit[_address] +
        difference <=
        TEAM_ALLOCATION,
    "Total allocated amount exceeded for team members"
)
...
require(
    getTotalPresaleVestedAmount() -
        individualPresaleVestedLimit[_address] +
        difference <=
        PRESALE_ALLOCATION,
    "Total allocated amount exceeded for presale participants"
);
```

Recommendation

It is recommended to reconsider the implementation of the `require` check to accurately reflect the contract's intention of preventing the setting of a vested limit that, when combined with already minted amounts, exceeds the presale allocation. Specifically, the calculation should not subtract the `individualTeamVestedLimit` or `individualPresaleVestedLimit` when determining if the new limit is within the presale allocation bounds. Instead, the contract should directly compare the sum of the `claimedAmount` and the new difference to the allocation, ensuring that the total amount (the one already set plus newly set limit) does not exceed the presale allocation. This adjustment will ensure that the contract accurately tracks and enforces the total allocation limits, taking into account both previously minted amounts and new limits being set, thereby preventing over-allocation.

CCC - Contradictory Condition Check

Criticality	Medium
Location	contracts/ETFSwap.sol#L517
Status	Unresolved

Description

The contract contains the `addToWhitelist` function designed to add addresses to the whitelist. Within this function, there is a conditional statement intended to initialize the `vestingStart` time for team members if the address being added to the whitelist is in `isInAddresses` function and if its associated `vestingStart` is zero. However, a logical discrepancy arises from this condition. The `isInAddresses` function verifies whether an address belongs to the team or presale addresses array, and any address added to `teamAddresses` or `presaleAddresses` have already a non-zero `vestingStart` value. Consequently, the condition `isInAddresses(_address, _type) && vestingStart[_address] == 0` is inherently contradictory, as an address recognized as in `isInAddresses` would not have a `vestingStart` of zero. This contradiction means the intended functionality to initialize the vesting start time for newly whitelisted team addresses is effectively nullified, as the condition will always evaluate to false.

```
if (isInAddresses(_address, _type) && vestingStart[_address] ==
0) {
    vestingStart[_address] = block.timestamp;
}
```

Recommendation

It is recommended to streamline the `addToWhitelist` function's logic to accurately reflect the contract's intended operations. Specifically, re-evaluate and possibly remove the contradictory check involving `isInAddresses` and `vestingStartg[_address] == 0`. Instead, consider ensuring that the vesting start time for team members is set at an appropriate stage, such as when an address is first designated as a team address before

any whitelisting operations. This adjustment will resolve the logical inconsistency and ensure the functionality works as expected.

IFU - Inefficient Functions Usages

Criticality	Medium
Location	contracts/ETFSwap.sol#L293,315,502,548,578
Status	Unresolved

Description

The contract is currently utilizing separate functions to manage related functionalities, specifically for setting individual vested limits, adding addresses to whitelists, and setting specific addresses for team and presale allocations. These functions include `setIndividualTeamVestedLimit`, `setIndividualPresaleVestedLimit`, `addToWhitelist`, `setTeamAddress`, and `setPresaleAddress`. Each of these functions performs operations that are closely related, such as setting limits for vested amounts, adding addresses to different whitelists, and initializing vesting starts for team and presale addresses. The separation of these functionalities into multiple functions leads to increased complexity and redundancy within the contract's codebase. This not only makes the contract more difficult to maintain but also increases the potential for errors and inconsistencies in how these operations are executed.

```
    function setIndividualTeamVestedLimit(
        address _address,
        uint256 limit
    ) external onlyOwner {
        uint256 claimedAmount =
totalTeamVestedAmount[_address];
        uint256 difference = limit - claimedAmount;
        require(
            difference >= 0,
            "New limit cannot be less than the previously
claimed amount"
        );
        require(
            getTotalTeamVestedAmount() -
                individualTeamVestedLimit[_address] +
                difference <=
                TEAM_ALLOCATION,
            "Total allocated amount exceeded for team members"
        );
        individualTeamVestedLimit[_address] = difference;
        emit IndividualTeamVestedLimitSet(_address,
difference);
    }

    function setIndividualPresaleVestedLimit(
        address _address,
        uint256 limit
    ) external onlyOwner {
        uint256 claimedAmount =
totalPresaleVestedAmount[_address];
        uint256 difference = limit - claimedAmount;
        require(
            difference >= 0,
            "New limit cannot be less than the previously
claimed amount"
        );
        require(
            getTotalPresaleVestedAmount() -
                individualPresaleVestedLimit[_address] +
                difference <=
                PRESALE_ALLOCATION,
            "Total allocated amount exceeded for presale
participants"
        );
        individualPresaleVestedLimit[_address] = difference;
        emit IndividualPresaleVestedLimitSet(_address,
difference);
    }

    function addToWhitelist(
```

```
        address _address,
        WhitelistType _type
    ) external onlyOwner {
        require(_address != address(0) && _address != owner,
"Invalid address");
        mapping(address => bool) storage whitelistToAdd = _type
==
            WhitelistType.Team
            ? teamWhitelist
            : presaleWhitelist;
        mapping(address => uint256) storage vestingStart =
_type ==
            WhitelistType.Team
            ? _teamVestingStart
            : _presaleVestingStart;
        require(!whitelistToAdd[_address], "Address is already
whitelisted");
        whitelistToAdd[_address] = true;
        if (isInAddresses(_address, _type) &&
vestingStart[_address] == 0) {
            vestingStart[_address] = block.timestamp;
        }
        emit AddedToWhitelist(_address, _type);
    }

    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 merge these functions into a single, more comprehensive function capable of handling all related operations. This unified function could accept parameters to identify the operation type (e.g., setting vested limits, adding to whitelist, initializing vesting) and execute the corresponding logic based on these parameters. This approach would streamline the contract's functionality, reduce redundancy, and simplify the process of managing vested limits, whitelisting, and address initialization. By consolidating related functionalities, the contract can achieve a more efficient, maintainable, and error-resistant implementation. This change would also potentially reduce the gas costs associated with deploying and interacting with the contract, as it minimizes the number of function calls required to perform related operations.

MTV - Misleading Tax Variable

Criticality	Medium
Location	contracts/ETFSwap.sol#L243
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.

IAC - Inefficient Amount Calculation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L337,346
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.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	contracts/ETFSwap.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.

RTE - Redundant Transfer Events

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L106
Status	Unresolved

Description

The contract is currently designed to emit a separate `Transfer` event for each allocation of tokens to the same address. This approach is observed in the process of minting tokens, where allocations for presale, ecosystem, liquidity, cashback, partners, community rewards, market making (MM), and team are individually transferred to the contract creator's address. Each of these transfers triggers its own `Transfer` event, despite all tokens being sent to the same recipient. This method leads to an unnecessary increase in the number of events emitted by the contract, which could potentially clutter the event log and slightly increase the cost of contract execution due to the gas costs associated with emitting events.

```
balances[msg.sender] += PRESALE_ALLOCATION;
emit Transfer(address(0), msg.sender, PRESALE_ALLOCATION)
balances[msg.sender] += ECOSYSTEM_ALLOCATION;
emit Transfer(address(0), msg.sender, ECOSYSTEM_ALLOCATION)
balances[msg.sender] += LIQUIDITY_ALLOCATION;
emit Transfer(address(0), msg.sender, LIQUIDITY_ALLOCATION)
balances[msg.sender] += CASHBACK_ALLOCATION;
emit Transfer(address(0), msg.sender, CASHBACK_ALLOCATION)
balances[msg.sender] += PARTNERS_ALLOCATION;
emit Transfer(address(0), msg.sender, PARTNERS_ALLOCATION)
balances[msg.sender] += COMMUNITY_REWARDS_ALLOCATION;
emit Transfer(address(0), msg.sender,
COMMUNITY_REWARDS_ALLOCATION)
balances[msg.sender] += MM_ALLOCATION;
emit Transfer(address(0), msg.sender, MM_ALLOCATION)
balances[msg.sender] += TEAM_ALLOCATION;
emit Transfer(address(0), msg.sender, TEAM_ALLOCATION);
```

Recommendation

It is recommended to consolidate these individual `Transfer` events into a single event that emits the total amount of tokens transferred to the address. This can be achieved by

summing up all allocations and emitting one Transfer event with the total amount. This approach will simplify the event log, making it easier to track the total supply transfer, and could also marginally reduce the gas costs associated with contract execution. The code modification would involve calculating the total allocation by adding all individual allocations together and then emitting a single `Transfer` event to reflect the total amount transferred. This change would enhance the efficiency and clarity of the contract's token distribution process.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L138,253,254,265,271,278,279,294,316,503,504,548,578
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
WhitelistType _type
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.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L299,321
Status	Unresolved

Description

A tautology is a logical statement that is always true, regardless of the values of its variables. A contradiction is a logical statement that is always false, regardless of the values of its variables.

Using tautologies or contradictions can lead to unintended behavior and can make the code harder to understand and maintain. It is generally considered good practice to avoid tautologies and contradictions in the code.

```
require(
            difference >= 0,
            "New limit cannot be less than the previously
claimed amount"
        )
```

Recommendation

The team is advised to carefully consider the logical conditions is using in the code and ensure that it is well-defined and make sense in the context of the smart contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L404,405
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.

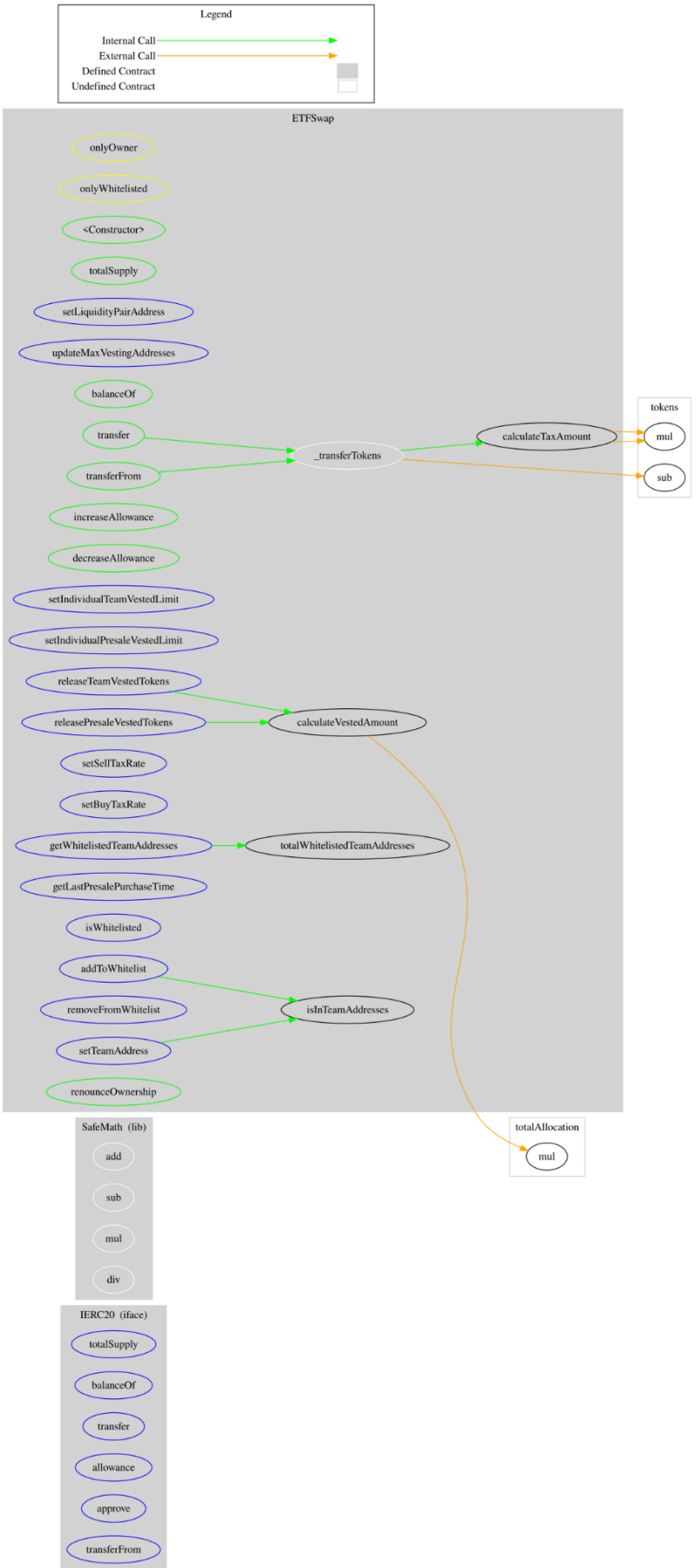
Functions Analysis

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		
	isInAddresses	Internal		
	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		-
	addToWhitelist	External	✓	onlyOwner
	removeFromTeamWhitelist	External	✓	onlyOwner
	removeFromPresaleWhitelist	External	✓	onlyOwner
	setTeamAddress	External	✓	onlyOwner
	getTotalTeamAllocation	Private		
	setPresaleAddress	External	✓	onlyOwner
	getTotalPresaleAllocation	Private		
	renounceOwnership	Public	✓	onlyOwner

Inheritance Graph

Flow Graph


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