Cyberscope

Audit Report

ETFSwap

April 2024

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

Commit 0xaef811cabc8251dda180bcd5dff1710e1198c355

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	IVL	Inaccurate Vesting Limits	Unresolved
Medium	DTAC	Duplicated Total Allocation Calculation	Unresolved
Medium	DLCI	Dynamic Limit Calculation Inconsistency	Unresolved
Medium	IFU	Inefficient Functions Usages	Unresolved
Minor / Informative	IAC	Inefficient Amount Calculation	Unresolved
Minor / Informative	MCM	Misleading Comment Messages	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L08	Tautology or Contradiction	Unresolved

Table of Contents

Review

Contract Name	ETFSwap
Repository	https://github.com/hamzabadshah1/etfswap
Commit	567fed8444a7134cb84da4985b5d0838a6bd7fc5
Testing Deploy	https://testnet.bscscan.com/address/0xaef811cabc8251dda180bcd5dff1710e1198c355
Symbol	ETFS
Decimals	18
Total Supply	1,000,000,000
Badge Eligibility	Must Fix Criticals

Audit Updates

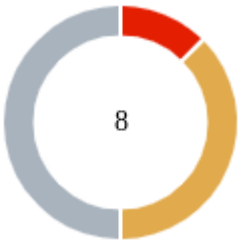
Initial Audit	27 Mar 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v1/audit.pdf
Corrected Phase 2	04 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v2/audit.pdf
Corrected Phase 3	05 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v3/audit.pdf
Corrected Phase 4	08 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v4/audit.pdf

Corrected Phase 5	10 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v5/audit.pdf
Corrected Phase 6	16 Apr 2024

Source Files

Filename	SHA256
contracts/ETFSwap.sol	1dc6b96f600c2168235f77407fbeb3116f1d54ab284230034c34dc45049e9560

Findings Breakdown



Critical	1
Medium	3
Minor / Informative	4

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	3	0	0	0
Minor / Informative	4	0	0	0

IVL - Inaccurate Vesting Limits

Criticality	Critical
Location	contracts/ETFSwap.sol#L322,383
Status	Unresolved

Description

The contract sets the individual Vested Limit to a new value based on the `difference` calculated for each address. However, if an address has already vested some amount (`totalVested`), the sum of this amount and any newly vested amount (totalVested + vestedAmount) could exceed the newly set `difference` (`individualLimit`). This is because the set of the individual vested limit does not account for previously vested amounts during the `setIndividualVestedLimit` function. As a result, addresses that have already claimed tokens will find themselves unable to claim additional tokens allowed under their total allocation, as the new limits incorrectly cap their ability to vest further tokens based on an incomplete calculation of their previously vested amounts.

```
if (_type == OperationType.Team) {
    individualTeamVestedLimit[_address] = difference;
    emit IndividualTeamVestedLimitSet(_address, difference);
} else {
    individualPresaleVestedLimit[_address] = difference;
    emit IndividualPresaleVestedLimitSet(_address, difference);
...
uint256 vestedAmount = calculateVestedAmount(vestingStart, allocation);
vestingStart = block.timestamp;
require(
    totalVested + vestedAmount <= individualLimit,
    "Individual vested limit exceeded"
);
```

Recommendation

It is recommended to reconsider the tokenomics, the circumstances under which new token limits are set, and the calculations involved in such settings. The contract should adequately handle scenarios where addresses have previously claimed any vested amount.

This involves adjusting the calculation and checks to consider these already vested amounts when setting new limits. Ensuring that the new limits are set correctly and account for past transactions will prevent issues where users are prevented from vesting additional tokens they are entitled to under their total allocation. This adjustment will improve the fairness and functionality of the vesting process, aligning it more closely with the intended economic model and user expectations.

DTAC - Duplicated Total Allocation Calculation

Criticality	Medium
Location	contracts/ETFSwap.sol#L315,332,341
Status	Unresolved

Description

The contract is using a `require` statement to ensure that the total vested amount plus a new limit does not exceed the total allocation (`totalAllocation`). However, there is a critical flaw in how the total vested amounts are computed. Both the `getTotalTeamVestedAmount` and `getTotalPresaleVestedAmount` functions accumulate the already vested amounts for all respective addresses. Additionally, `newLimit` includes the already vested amount of the address for which the new limit is being set. This leads to a situation where the total vested amount for any address is effectively considered twice in the calculation. Consequently, this double counting may cause the contract to prematurely reach the `totalAllocation` limit, which in reality has not been fully utilized, potentially restricting further valid allocations.

```
        (_type == OperationType.Team &&
            getTotalTeamVestedAmount() + newLimit <= totalAllocation) ||
            (_type == OperationType.Presale &&
                getTotalPresaleVestedAmount() + newLimit <=
                totalAllocation),
        "Total allocated amount exceeded for 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 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 reconsider the way the total vested amount and the new limit calculations are performed to prevent double counting. Specifically, modifying the calculation logic to exclude already counted amounts in `newLimit` or adjusting the total calculation method could correct this error. By ensuring that each vested amount is only counted once, the contract will more accurately reflect the true utilization of allocated funds, preventing unnecessary halts on allocations and maintaining the intended functionality and fairness of the vesting process. This correction will improve the robustness and trustworthiness of the contract's financial management systems.

DLCI - Dynamic Limit Calculation Inconsistency

Criticality	Medium
Location	contracts/ETFSwap.sol#L292,301
Status	Unresolved

Description

The contract contains the `calculateMaxIndividualTeamLimit` and `calculateMaxIndividualPresaleLimit` functions, designed to determine the maximum allocation limit per individual for team and presale members, respectively. These functions calculate the limit based on the total allocation divided by the current number of addresses in the team or presale list. However, since the length of these address arrays can change dynamically as new members are added or removed, the functions can return varying maximum limits over time. This variability contradicts the intended functionality of providing a fixed maximum limit for individual allocations. Consequently, this dynamic calculation method may lead to inconsistencies in allocation limits and could potentially impact the fair distribution of allocations among team or presale members.

```
   function calculateMaxIndividualTeamLimit() internal view returns
(uint256) {
        require(
            teamAddresses.length > 0,
            "Number of team members must be greater than zero"
        );
        return TEAM_ALLOCATION / teamAddresses.length;
    }

    function calculateMaxIndividualPresaleLimit()
        internal
        view
        returns (uint256)
    {
        require(
            presaleAddresses.length > 0,
            "Number of team members must be greater than zero"
        );
        return PRESALE_ALLOCATION / presaleAddresses.length;
    }
```

Recommendation

It is recommended to establish a fixed maximum individual limit that does not rely on the fluctuating number of team or presale members. One approach could involve setting a predefined constant for the maximum limit, independent of the array lengths. Alternatively, if a dynamic approach is necessary, implementing a mechanism to adjust individual limits only at specific intervals or under controlled conditions could mitigate the issue. This change would ensure a consistent and transparent allocation process, aligning with the principle of fairness and predictability in the distribution of allocations.

IFU - Inefficient Functions Usages

Criticality	Medium
Location	contracts/ETFSwap.sol#L290,497,531,561
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 `setIndividualVestedLimit`, `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 setIndividualVestedLimit(
        address _address,
        uint256 limit,
        OperationType _type
    ) external onlyOwner {
        uint256 claimedAmount = (_type == OperationType.Team)
            ? totalTeamVestedAmount[_address]
            : totalPresaleVestedAmount[_address];
        uint256 difference = limit - claimedAmount;
        uint256 newLimit = limit + claimedAmount;
        require(
            difference >= 0,
            "New limit cannot be less than the previously claimed
amount"
        );
        uint256 maxIndividualLimit = (_type == OperationType.Team)
            ? calculateMaxIndividualTeamLimit()
            : calculateMaxIndividualPresaleLimit();
        require(
            newLimit <= maxIndividualLimit,
            "New limit exceeds the maximum allowed individual limit"
        );
        uint256 totalAllocation = (_type == OperationType.Team)
            ? TEAM_ALLOCATION
            : PRESALE_ALLOCATION;
        ...
    }

    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;
        require(!whitelistToAdd[_address], "Address is already
whitelisted");
        whitelistToAdd[_address] = true;
        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 regarding the set or add to the whitelist functionality. 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.

IAC - Inefficient Amount Calculation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L332,341
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.

MCM - Misleading Comment Messages

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L229
Status	Unresolved

Description

The contract is using misleading comment messages. These comment messages do not accurately reflect the actual implementation, making it difficult to understand the source code. As a result, the users will not comprehend the source code's actual implementation.

```
if (from == liquidityPairAddress) {
    // Apply buy tax rate if the tokens are being transferred by the
owner
    return (tokens * buyTaxRate) / (100);
```

Recommendation

The team is advised to carefully review the comment in order to reflect the actual implementation. To improve code readability, the team should use more specific and descriptive comment messages.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L129,244,245,256,262,291,293,498,499,514,531,561
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
OperationType _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#L300
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.

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		
	calculateMaxIndividualTeamLimit	Internal		
	calculateMaxIndividualPresaleLimit	Internal		
	setIndividualVestedLimit	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
	removeFromWhitelist	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
onlyTeamAddresses
<Constructor>
totalSupply
setLiquidityPairAddress
balanceOf
transfer
transferFrom
_transferTokens
calculateTaxAmount
increaseAllowance
decreaseAllowance
setIndividualVestedLimit
getTotalPresaleVestedAmount
getTotalTeamVestedAmount
calculateMaxIndividualPresaleLimit
calculateMaxIndividualTeamLimit
releaseTeamVestedTokens
releasePresaleVestedTokens
_releaseVestedTokens
calculateVestedAmount
setSellTaxRate
setBuyTaxRate
getWhitelistedTeamAddresses
totalWhitelistedTeamAddresses
getWhitelistedPresaleAddresses
totalWhitelistedPresaleAddresses
isTeamWhitelisted
isPresaleWhitelisted
addToWhitelist
removeFromWhitelist
getTotalTeamAllocation
setTeamAddress
isInTeamAddresses
setPresaleAddress
getTotalPresaleAllocation
isInAddressList
renounceOwnership
isInPresaleAddresses

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