Cyberscope

Audit Report

ETFSwap

March 2024

Repository	https://github.com/hamzabadshah1/etfswap
Commit	18dbea873c4a2fbbc23af27ca23e74bf96c51234
Audited by	© cyberscope

Table of Contents

Review

Contract Name	ETFSwap
Repository	https://github.com/hamzabadshah1/etfswap
Commit	18dbea873c4a2fbbc23af27ca23e74bf96c51234
Testing Deploy	https://testnet.bscscan.com/address/0x9cade57f36433496041f337ddcdb40ffed0b5b88
Symbol	ETFS
Decimals	18
Total Supply	1,000,000,000

Audit Updates

Initial Audit	28 Mar 2024

Source Files

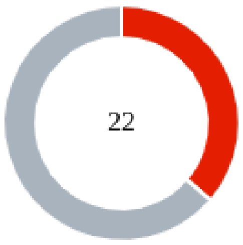
Filename	SHA256
contracts/ETFSwap.sol	2a1da86ba8b6730db9efdc87685e4d77a1ca9608f79e9ae47d6b8c09d08825c3

Overview

Contract Readability Comment

The audit scope is to check for security vulnerabilities, validate the business logic and propose potential optimizations. The contract is missing the fundamental principles of a Solidity smart contract regarding gas consumption, code readability, and data structures including the handling of balances and the total supply values. According to the previously mentioned issues, the contract cannot be assumed that it is in a production-ready state. Given these issues, it is not advisable to assume that the contract is in a production-ready state. The development team is strongly encouraged to re-evaluate the business logic and Solidity guidelines to ensure that the contract adheres to established best practices and security measures. It is recommended that the team review the contract's gas consumption and optimize it accordingly to minimize costs and improve the contract's efficiency. The code's readability should also be improved by simplifying function definitions and using descriptive variable names, as this will enhance the contract's auditability and maintenance.

Findings Breakdown



●	Critical	8
●	Medium	0
●	Minor / Informative	14

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	8	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	14	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	IAH	Incomplete Allowance Handling	Unresolved
🔴	ELFM	Exceeds Fees Limit	Unresolved
🔴	ISA	Inaccurate Supply Adjustment	Unresolved
🔴	MTL	Misapplied Tax Logic	Unresolved
🔴	TSD	Total Supply Diversion	Unresolved
🔴	UVC	Unrestricted Vesting Claims	Unresolved
🔴	UVP	Unrestricted Vesting Participation	Unresolved
🔴	WVM	Whitelist Vesting Mismatch	Unresolved
⚪	CR	Code Repetition	Unresolved
⚪	IDI	Immutable Declaration Improvement	Unresolved
⚪	MMN	Misleading Method Naming	Unresolved
⚪	MVI	Misplaced Vesting Invocation	Unresolved
⚪	RED	Redudant Event Declaration	Unresolved
⚪	RGF	Redundant Getter Function	Unresolved

●	RMF	Redundant Mint Functionality	Unresolved
●	RRS	Redundant Require Statement	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L19	Stable Compiler Version	Unresolved

IAH - Incomplete Allowance Handling

Criticality	Critical
Location	contracts/ETFSwap.sol#L74,173
Status	Unresolved

Description

The contract is designed to incorporate an ERC-20 token standard using the `transferFrom` function, which is intended to enable token transfers on behalf of another address, contingent upon a sufficient allowance being set. This functionality relies on the `allowed` mapping to track the allowances set by token holders for various spenders. However, a critical oversight in the contract's design is the absence of any function to set or modify the allowances within the `allowed` mapping. Without the ability to increase, set, or decrease the allowance amounts, the execution of the `transferFrom` function becomes impractical. This omission effectively nullifies a fundamental component of the ERC-20 standard, preventing token holders from granting permission to third parties to spend tokens on their behalf, thus severely limiting the token's utility and interoperability within the broader Ethereum ecosystem.

```
mapping(address => mapping(address => uint256)) allowed;

  function transferFrom(
        address from,
        address to,
        uint256 tokens
    ) public returns (bool success) {
        require(to != address(0), "Invalid address");
        require(balances[from] >= tokens, "Insufficient
balance");
        require(allowed[from][msg.sender] >= tokens, "Allowance
exceeded")
...
}
```

Recommendation

It is recommended to implement functions regarding the increase and decrease functionality of the allowance, `increaseAllowance` , and `decreaseAllowance` functions found in standard ERC-20 implementations. These functions are essential for managing allowances in a secure and flexible manner, allowing token holders to control how many tokens can be spent by third parties. Adding these functionalities will not only align the contract with common ERC-20 standards, enhancing its compatibility and utility within the ecosystem but also ensure that the contract's intended allowance and transferFrom mechanisms function as expected. This adjustment will significantly improve the contract's usability and security, enabling a more robust and flexible token economy.

ELFM - Exceeds Fees Limit

Criticality	Critical
Location	contracts/ETFSwap.sol#L294,304
Status	Unresolved

Description

The contract owner has the authority to increase over the allowed limit of 25%. The owner may take advantage of it by calling the `setSellTaxRate` or `setBuyTaxRate` function with a high percentage value.

```
   function setSellTaxRate(uint256 newSellTaxRate) external
onlyOwner {
        require(
            newSellTaxRate <= 100,
            "Sell tax rate must be less than or equal to 100%"
        );
        sellTaxRate = newSellTaxRate;
    }

    function setBuyTaxRate(uint256 newBuyTaxRate) external
onlyOwner {
        require(
            newBuyTaxRate <= 100,
            "Buy tax rate must be less than or equal to 100%"
        );
        buyTaxRate = newBuyTaxRate;
    }
```

Recommendation

The contract could embody a check for the maximum acceptable value. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

ISA - Inaccurate Supply Adjustment

Criticality	Critical
Location	contracts/ETFSwap.sol#L249,269
Status	Unresolved

Description

The contract implements `releaseTeamVestedTokens` and `releasePresaleVestedTokens` functions to allow vested tokens to be claimed by eligible addresses. However, a significant oversight in these functions is that while they add the vested amount to the balance of `msg.sender`, they do not correspondingly deduct this amount from any other account, including the `owner`. This approach results in the tokens being effectively created at the moment of vesting claim, rather than being transferred from an existing supply. Consequently, this mechanism inflates the token's total supply beyond the intended limit, diverging from the standard practice of maintaining a constant total supply by ensuring that tokens are transferred from one account to another, rather than being newly minted upon vesting claims.

```
    function releaseTeamVestedTokens() external onlyWhitelisted
{
        ...
        balances[msg.sender] =
balances[msg.sender].add(vestedAmount);
        emit Transfer(owner, msg.sender, vestedAmount);
    }

    function releasePresaleVestedTokens() external {
        ...
        balances[msg.sender] =
balances[msg.sender].add(vestedAmount);
        emit Transfer(owner, msg.sender, vestedAmount);
        lastPresalePurchaseTime[msg.sender] = block.timestamp;
    }
```

Recommendation

It is recommended to reconsider the mechanism of release for vested tokens. Specifically, since these operations represent a transfer of ownership to the claimant, it is crucial to ensure that a corresponding amount is deducted from the sender's balance to preserve the total supply's integrity. This could involve maintaining a separate vesting pool account from which vested tokens are deducted or directly deducting the vested amount from the owner's balance, assuming the owner's account initially holds the tokens allocated for vesting. Implementing such a mechanism will ensure that the total supply remains constant and that the vesting process accurately reflects the transfer of tokens from one party to another, adhering to the principles of token conservation and supply integrity.

MTL - Misapplied Tax Logic

Criticality	Critical
Location	contracts/ETFSwap.sol#L191
Status	Unresolved

Description

The contract utilizes the `calculateTaxAmount` function to determine the tax amount on transactions, applying different tax rates based on the transaction's nature. However, the logic used to differentiate between sell and buy transactions is flawed. The contract applies the `sellTaxRate` if the recipient (to) address is either the zero address or the contract's owner, and the `buyTaxRate` in all other scenarios. This approach inaccurately represents the nature of buy and sell transactions, as the determination does not accurately reflect the transaction's direction (buying or selling). Given that both `transfer` and `transferFrom` functions include a requirement that `to != address(0)` , the `sellTaxRate` is effectively applied only when the recipient is the contract's owner, and the `buyTaxRate` is applied in all other cases. This misalignment does not accurately capture the intended tax implications of sell and buy transactions, potentially leading to incorrect tax calculations and unintended economic incentives or penalties within the token ecosystem.

```
    function calculateTaxAmount(
        address from,
        address to,
        uint256 tokens
    ) private view returns (uint256) {
        if (from != owner) {
            if (to == address(0) || to == owner) {
                return tokens.mul(sellTaxRate).div(100);
            } else {
                return tokens.mul(buyTaxRate).div(100);
            }
        }
        return 0;
    }
```

Recommendation

It is recommended to reconsider the approach under which the tax is applied to accurately reflect sell and buy transactions. If distinguishing between sell and buy taxes is desired, the contract should incorporate logic to identify transactions involving a liquidity pair address or a similar mechanism that accurately differentiates between selling to and buying from the market. This could involve checking if the from or to address is a recognized liquidity pair address, thereby providing a more accurate basis for applying the correct tax rate. Adjusting the tax application logic will ensure that the tax rates are applied as intended, reflecting the actual economic actions of selling and buying, and aligning the contract's functionality with its economic goals.

TSD - Total Supply Diversion

Criticality	Critical
Location	contracts/ETFSwap.sol#L126,135
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 `TEAM_ALLOCATION` amount is not added to the `msg.sender` balances.

```
   uint256 public constant TOTAL_SUPPLY =
        1_000_000_000 * (10 ** uint256(decimals));

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;


    function totalSupply() public pure returns (uint256) {
        return TOTAL_SUPPLY;
    }
```

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.

UVC - Unrestricted Vesting Claims

Criticality	Critical
Location	contracts/ETFSwap.sol#L234,244,253,264,274
Status	Unresolved

Description

The contract incorporates `releaseTeamVestedTokens` and `releasePresaleVestedTokens` functions to enable authorized addresses to claim their tokens under a specified vesting period. However, a critical oversight in the implementation is the absence of checks to prevent repetitive claims by the same address beyond their total eligible vested amount. The functions reset the vesting start time (`_teamVestingStart` or `_presaleVestingStart`) to the current timestamp after each claim, allowing addresses to repeatedly claim vested tokens every time a vesting cycle passes, without verifying if the total vested amount has been fully claimed. This loophole could lead to the unintended release of tokens beyond the original allocation, undermining the vesting schedule's integrity and potentially affecting the token's distribution and value.

```
   function releaseTeamVestedTokens() external onlyWhitelisted
{
        ...
        uint256 vestedAmount = calculateVestedAmount(
            _teamVestingStart[msg.sender],
            TEAM_ALLOCATION
        );
        ...
    }

    function releasePresaleVestedTokens() external {
        ...

        uint256 vestedAmount = calculateVestedAmount(
            _presaleVestingStart[msg.sender],
            PRESALE_ALLOCATION
        );
        ...
    }

    function calculateVestedAmount(
        uint256 vestingStart,
        uint256 totalAllocation
    ) private view returns (uint256) {
        uint256 elapsedTime = block.timestamp - vestingStart;
        uint256 vestingPeriods = elapsedTime /
RELEASE_INTERVAL;
        uint256 vestedAmount =
totalAllocation.mul(vestingPeriods).div(5); // 20% released
every 3 weeks
        return vestedAmount;
    }
```

Recommendation

It is recommended to reconsider the circumstances under which the vesting functions are called and introduce limits to prevent or halt the vesting mechanism once an address has claimed the total amount eligible. This could involve tracking the total amount of tokens each address has claimed through vesting and comparing it against their total allocated amount before allowing further claims. Implementing a cap on the total vested amount claimable by each address will ensure that the vesting process adheres to its intended limitations, preserving the token's economic model and ensuring fairness among participants. Additionally, refining the logic to accurately reflect the vesting period's conclusion and prevent further claims once the allocated amount is fully vested will strengthen the contract's security and its adherence to the predefined vesting schedule.

UVP - Unrestricted Vesting Participation

Criticality	Critical
Location	contracts/ETFSwap.sol#L206
Status	Unresolved

Description

The contract is designed to allow the owner to initiate vesting for addresses through the `handleVesting` function, which is triggered under specific conditions. This function checks if the `from` address is the owner and if the `to` address is neither the zero address nor the owner. If these conditions are met, and the transferred tokens match either the `TEAM_ALLOCATION` or `PRESALE_ALLOCATION` , it starts the vesting process for the to address by setting a vesting start timestamp. However, there is no implemented mechanism to limit the number of addresses the owner can initiate vesting for, nor is there a cap on the total amount that can be allocated through this mechanism. This oversight allows the owner to potentially distribute the `TEAM_ALLOCATION` or `PRESALE_ALLOCATION` tokens to an unlimited number of addresses, which could lead to dilution of the token's value or unfair distribution practices, especially since the total supply of tokens is constant.

```
   function handleVesting(address from, address to, uint256
tokens) private {
        if (from == owner) {
            if (to != address(0) && to != owner) {
                if (tokens == TEAM_ALLOCATION &&
_teamVestingStart[to] == 0) {
                    _teamVestingStart[to] = block.timestamp;
                    if (!isInTeamAddresses(to)) {
                        teamAddresses.push(to);
                    }
                }
                if (
                    tokens == PRESALE_ALLOCATION &&
                    _presaleVestingStart[to] == 0
                ) {
                    _presaleVestingStart[to] = block.timestamp;
                }
            }
        }
```

Recommendation

It is recommended to introduce a hard cap on the total amount of tokens that can be allocated through the vesting mechanism and to limit the number of addresses that can participate in the vesting. This could be achieved by implementing a counter that tracks the total allocated tokens for both `TEAM_ALLOCATION` and `PRESALE_ALLOCATION` and ensuring that this total does not exceed a predefined limit. Additionally, introducing checks to limit the number of new addresses that can be added to the vesting process could prevent potential abuse of this mechanism. These measures will ensure a fair and transparent distribution process, aligning with the principle of equitable token economics and protecting the interests of all stakeholders involved in the contract.

WVM - Whitelist Vesting Mismatch

Criticality	Critical
Location	contracts/ETFSwap.sol#L234
Status	Unresolved

Description

The contract contains the `releaseTeamVestedTokens` function, which is designed to allow team members to claim their vested tokens. This function is protected by an `onlyWhitelisted` modifier, presumably to restrict access to a predefined group of users. However, there is a significant discrepancy in the function's logic since it requires that the caller's (`msg.sender`) `_teamVestingStart` timestamp be greater than zero, indicating that tokens have been allocated for vesting to this address. This requirement inherently suggests that the function is intended for use by team members whose vesting has already commenced, as indicated by their inclusion in the `teamAddresses` through the `handleVesting` function. This setup presents a contradiction since the `onlyWhitelisted` modifier may allow users who are not part of the `teamAddresses` to attempt to call this function, while the requirement for a non-zero `_teamVestingStart` implies that only users with commenced vesting should have access. Furthermore, the mechanism for setting `_teamVestingStart` is exclusively tied to the `handleVesting` function, which does not cover the addition of new team members to `teamAddresses` directly through the `setTeamAddresses` function, leading to potential inconsistencies in who can successfully call `releaseTeamVestedTokens` .

```
    function releaseTeamVestedTokens() external onlyWhitelisted
{
        require(
            _teamVestingStart[msg.sender] > 0,
            "No vested tokens to release"
        );
        ...
        }

     function setTeamAddresses(
        address[] calldata _teamAddresses
    ) external onlyOwner {
        for (uint256 i = 0; i < _teamAddresses.length; i++) {
            if (_teamAddresses[i] != address(0) &&
_teamAddresses[i] != owner) {
                teamAddresses.push(_teamAddresses[i]);
            }
        }
    }
```

Recommendation

It is recommended to reconsider the conditions under which the `releaseTeamVestedTokens` function can be called and the mechanisms for setting vesting start times. To ensure consistency and clarity, the contract should align the `whitelisting` process with the vesting initiation process. This could involve ensuring that all whitelisted addresses are also properly initialized in the vesting logic, possibly by setting their `_teamVestingStart` at the time they are added to the `whitelist` or `teamAddresses`. Alternatively, revising the `onlyWhitelisted` modifier to specifically check for a valid vesting start time or directly integrating the vesting start logic into the process of adding a user to the `whitelist` or `teamAddresses` could resolve the discrepancy. This approach would streamline the process, ensuring that only eligible team members can release their vested tokens and that all such members are appropriately accounted for in the contract's logic.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L145,166
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.

Specifically the `transfer` function share similar code segments with the `transferFrom` function and the `releaseTeamVestedTokens` function with the `releasePresaleVestedTokens` function.

```
   function transfer(
        address to,
        uint256 tokens
    ) public returns (bool success) {
        require(to != address(0), "Invalid address");
        require(balances[msg.sender] >= tokens, "Insufficient
balance");

        uint256 taxAmount = calculateTaxAmount(msg.sender, to,
tokens);
        uint256 transferAmount = tokens.sub(taxAmount);

        balances[msg.sender] =
balances[msg.sender].sub(tokens);
        balances[to] = balances[to].add(transferAmount);
        balances[owner] = balances[owner].add(taxAmount);

        handleVesting(msg.sender, to, tokens);

        emit Transfer(msg.sender, to, transferAmount);
        emit Transfer(msg.sender, owner, taxAmount);
        return true;
    }

    function transferFrom(
        address from,
        address to,
        uint256 tokens
    ) public returns (bool success) {
        ..
    }
    function releaseTeamVestedTokens() external onlyWhitelisted {
        require(
            _teamVestingStart[msg.sender] > 0,
            "No vested tokens to release"
        );
        require(
            block.timestamp >= _teamVestingStart[msg.sender] +
RELEASE_INTERVAL,
            "Vesting period not ended yet"
        );

        uint256 vestedAmount = calculateVestedAmount(
            _teamVestingStart[msg.sender],
            TEAM_ALLOCATION
        );
        _teamVestingStart[msg.sender] = block.timestamp; // Reset
vesting start
        balances[msg.sender] =
balances[msg.sender].add(vestedAmount);
        emit Transfer(owner, msg.sender, vestedAmount);
    }

    function releasePresaleVestedTokens() external {
        ...
```

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.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L122
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.

```
owner
```

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.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L324,336
Status	Unresolved

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

The contract includes the functions `getWhitelist` and `totalWhitelisted` that, by their names, imply they return information about all whitelisted addresses and the total number of such addresses, respectively. However, in practice, these functions operate exclusively on the `teamAddresses` array, checking for addresses within this array that are marked as `whitelisted`. This implementation can lead to confusion as the names suggest a broader functionality than is actually provided. Specifically, these functions do not account for or return all `whitelisted` addresses but rather focus on a subset of addresses that are both part of `teamAddresses` and marked as `whitelisted`. This discrepancy between the function names and their actual functionality can mislead users and developers, leading to incorrect assumptions about the contract's behavior and potentially hindering the effective management and utilization of the whitelist feature.

```
    function getWhitelist() external view returns (address[] memory)
{
        address[] memory whitelistAddresses = new
address[](totalWhitelisted());
        uint256 count = 0;
        for (uint256 i = 0; i < teamAddresses.length; i++) {
            if (whitelist[teamAddresses[i]]) {
                whitelistAddresses[count] = teamAddresses[i];
                count++;
            }
        }
        return whitelistAddresses;
    }

    function totalWhitelisted() public view returns (uint256) {
        uint256 count = 0;
        for (uint256 i = 0; i < teamAddresses.length; i++) {
            if (whitelist[teamAddresses[i]]) {
                count++;
            }
        }
        return count;
    }
```

Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. The team is advised to keep in mind the readability of the code. It is recommended to clarify the functionality of these functions to accurately reflect their operations and limitations. This could involve renaming `getWhitelist` and `totalWhitelisted` to more accurately describe that they are related specifically to team addresses that are whitelisted. Alternatively, if the intention was indeed to provide functionality for querying all whitelisted addresses, not just those within `teamAddresses`, the contract should be adjusted to include this broader scope in the implementation of these functions. Ensuring that function names and their implementations are aligned will improve the contract's clarity and usability, facilitating better interaction and integration with other components and users of the contract.

MVI - Misplaced Vesting Invocation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L159,184,206
Status	Unresolved

Description

The contract is structured to invoke the `handleVesting` function during every execution of the `transfer` and `transferFrom` functions. This design choice is problematic because the `handleVesting` function is tailored to manage specific vesting conditions, for team members or presale participants, and is intended to be triggered only under certain conditions, ideally by the `owner`. The current implementation, which indiscriminately calls `handleVesting` with every transfer operation, not only imposes unnecessary computational overhead on all transactions but also potentially exposes the vesting process to unintended manipulation or errors, as it relies on the `from` address being the owner for its logic. This approach conflates the general token transfer functionality with the specialized vesting logic, leading to inefficiencies and a lack of clarity in the contract's operations.

```
   function transfer(
        address to,
        uint256 tokens
    ) public returns (bool success) {
        ...
        handleVesting(msg.sender, to, tokens);
        ...
        return true;
    }

    function transferFrom(
        address from,
        address to,
        uint256 tokens
    ) public returns (bool success) {
        ...
        handleVesting(from, to, tokens);
        ...
        return true;
    }

    function handleVesting(address from, address to, uint256 tokens)
private {
        if (from == owner) {
            if (to != address(0) && to != owner) {
                if (tokens == TEAM_ALLOCATION && _teamVestingStart[to]
== 0) {
                    _teamVestingStart[to] = block.timestamp;
                    if (!isInTeamAddresses(to)) {
                        teamAddresses.push(to);
                    }
                }
                if (
                    tokens == PRESALE_ALLOCATION &&
                    _presaleVestingStart[to] == 0
                ) {
                    _presaleVestingStart[to] = block.timestamp;
                }
            }
        }
    }
```

Recommendation

It is recommended to separate the vesting logic from the general transfer operations by implementing a separate, owner-restricted function for handling vesting-related tasks. This function should be explicitly called under the appropriate conditions, rather than being automatically invoked within every transfer. Additionally, consider introducing mechanisms for validating vesting conditions and ensuring that vesting can only be initiated or modified under well-defined circumstances. By segregating the vesting functionality from standard token transfers, the contract can achieve greater efficiency, security, and clarity, ensuring that vesting operations are conducted as intended without interfering with or being inadvertently triggered by unrelated token transactions.

RED - Redudant Event Declaration

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L368
Status	Unresolved

Description

The contract uses events that are not emitted within the contract's functions. As a result, these declared events are redundant and serve no purpose within the contract's current implementation.

```
    event Approval(
        address indexed owner,
        address indexed spender,
        uint256 value
    );
```

Recommendation

To optimize contract performance and efficiency, it is advisable to regularly review and refactor the codebase, removing the unused event declarations. This proactive approach not only streamlines the contract, reducing deployment and execution costs but also enhances readability and maintainability.

RGF - Redundant Getter Function

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L76,347
Status	Unresolved

Description

The contract includes a public mapping `lastPresalePurchaseTime` that tracks the timestamp of the last presale purchase for each address. Given that this mapping is declared as `public`, Solidity automatically generates a getter function for it. This means that any external caller can directly query the mapping with an address to retrieve the corresponding last presale purchase time. However, the contract also explicitly defines the function `getLastPresalePurchaseTime`, which essentially replicates the functionality of the auto-generated getter for the `lastPresalePurchaseTime` mapping. This explicit function is redundant, as it does not provide any additional logic or access control beyond what the auto-generated getter already offers. The presence of this redundant function could lead to confusion and increase the contract's complexity without offering any tangible benefits.

```
    mapping(address => uint256) public lastPresalePurchaseTime;

     function getLastPresalePurchaseTime(
         address account
     ) external view returns (uint256) {
         return lastPresalePurchaseTime[account];
     }
```

Recommendation

It is recommended to remove the explicitly defined `getLastPresalePurchaseTime` function from the contract. This simplification will reduce the contract's size and complexity, making it easier to understand and audit. Users and other contracts can directly access the public `lastPresalePurchaseTime` mapping to retrieve the last presale purchase time for any address, leveraging the getter function automatically generated by Solidity for public state variables. Eliminating the redundant function will also clarify the contract's interface and reduce the potential for confusion among developers and users interacting with the contract.

RMF - Redundant Mint Functionality

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L135,284
Status	Unresolved

Description

The contract contains a `mint` function intended to allow the owner to issue new tokens, aiming to provide flexibility in managing the token supply. However, this functionality is effectively nullified by a logical flaw since the `totalSupply` function is hard-coded to always return the `TOTAL_SUPPLY` constant, making it impossible for the mint condition to be met. This is because the requirement for minting new tokens to have a total supply less than `TOTAL_SUPPLY` cannot be satisfied under any circumstances. As a result, the `mint` function, despite its presence, cannot be executed successfully, rendering it redundant and potentially misleading regarding the contract's actual capabilities.

```
    function totalSupply() public pure returns (uint256) {
        return TOTAL_SUPPLY;
    }
    function mint(address account, uint256 amount) external
onlyOwner {
        require(account != address(0), "Invalid address");
        require(
            totalSupply().add(amount) <= TOTAL_SUPPLY,
            "Total supply exceeded"
        );
        balances[account] = balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }
```

Recommendation

It is recommended to reconsider the `mint` functionality within the contract. If the intention behind the `mint` function is to allow for a flexible supply mechanism, then the implementation of the `totalSupply` function and the `TOTAL_SUPPLY` constant should be adjusted to reflect a dynamic total supply that can increase with minting operations. Alternatively, if a fixed supply model is desired, it may be prudent to remove the `mint` function altogether to avoid confusion and ensure clarity regarding the contract's capabilities. This approach will help align the contract's functionality with its intended operational model and stakeholder expectations.

RRS - Redundant Require Statement

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L38
Status	Unresolved

Description

The contract utilizes a `require` statement within the `add` function aiming to prevent overflow errors. This function is designed based on the SafeMath library's principles. In Solidity version 0.8.0 and later, arithmetic operations revert on overflow and underflow, making the overflow check within the function redundant. This redundancy could lead to extra gas costs and increased complexity without providing additional security.

```
function add(uint256 a, uint256 b) internal pure returns
(uint256) {
      uint256 c = a + b;
      require(c >= a, "SafeMath: addition overflow");
      return c;
}
```

Recommendation

It is recommended to remove the `require` statement from the add function since the contract is using a Solidity pragma version equal to or greater than 0.8.0. By doing so, the contract will leverage the built-in overflow and underflow checks provided by the Solidity language itself, simplifying the code and reducing gas consumption. This change will uphold the contract's integrity in handling arithmetic operations while optimizing for efficiency and cost-effectiveness.

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.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L310,316.356
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 addToWhitelist(address[] calldata addresses) external
onlyOwner {
        for (uint256 i = 0; i < addresses.length; i++) {
            whitelist[addresses[i]] = true;
        }
    }

    function removeFromWhitelist(
        address[] calldata addresses
    ) external onlyOwner {
        for (uint256 i = 0; i < addresses.length; i++) {
            whitelist[addresses[i]] = false;
        }
    }

    function setTeamAddresses(
        address[] calldata _teamAddresses
    ) external onlyOwner {
        for (uint256 i = 0; i < _teamAddresses.length; i++) {
            if (_teamAddresses[i] != address(0) &&
_teamAddresses[i] != owner) {
                teamAddresses.push(_teamAddresses[i]);
            }
        }
    }
```

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#L225,357
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 _address
address[] calldata _teamAddresses
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L299,307,312,320,361
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
sellTaxRate = newSellTaxRate
buyTaxRate = newBuyTaxRate
whitelist[addresses[i]] = true;
whitelist[addresses[i]] = false;
teamAddresses.push(_teamAddresses[i]);
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L279,280
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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L2
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

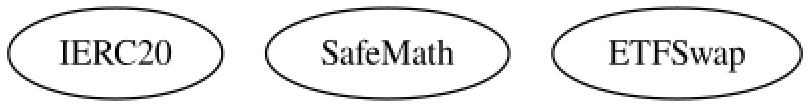
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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

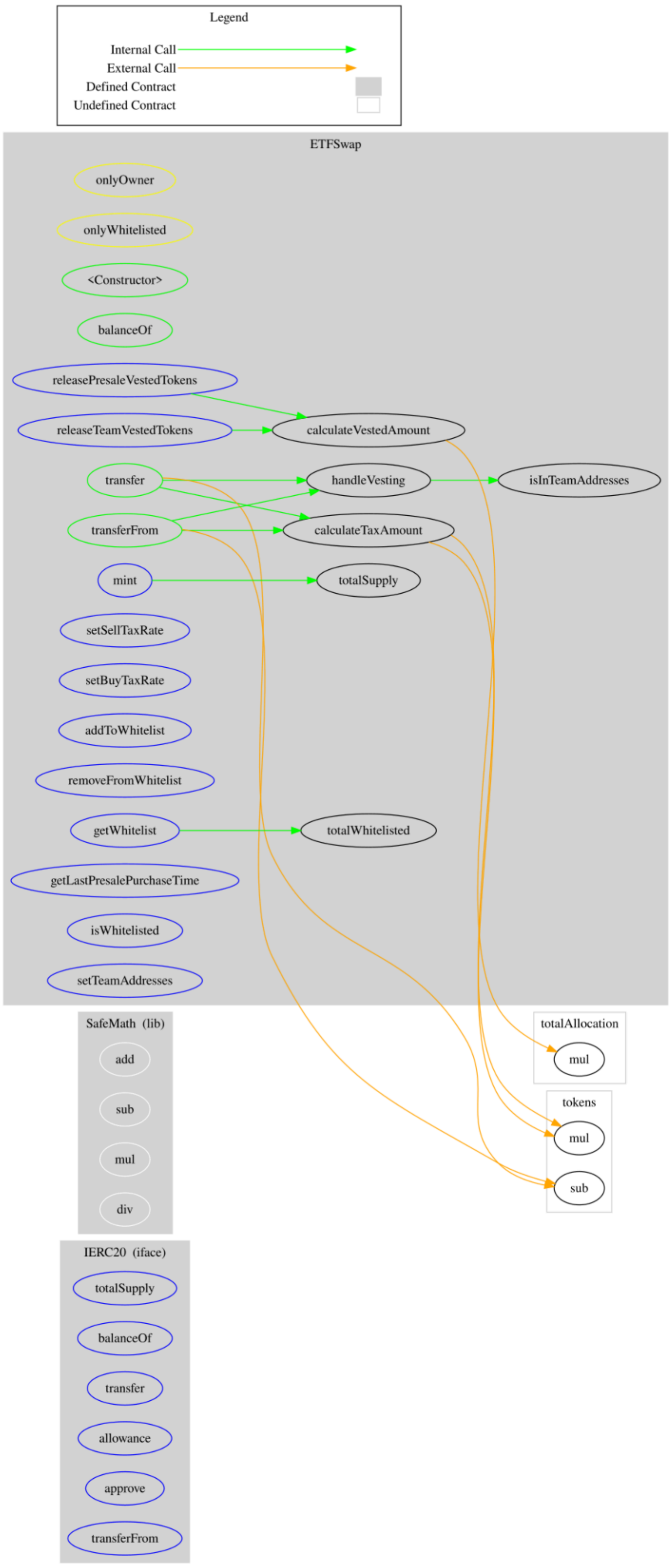
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
SafeMath	Library			
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
ETFSwap	Implementation			
		Public	✓	-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-

	transferFrom	Public	✓	-
	calculateTaxAmount	Private		
	handleVesting	Private	✓	
	isInTeamAddresses	Private		
	releaseTeamVestedTokens	External	✓	onlyWhitelisted
	releasePresaleVestedTokens	External	✓	-
	calculateVestedAmount	Private		
	mint	External	✓	onlyOwner
	setSellTaxRate	External	✓	onlyOwner
	setBuyTaxRate	External	✓	onlyOwner
	addToWhitelist	External	✓	onlyOwner
	removeFromWhitelist	External	✓	onlyOwner
	getWhitelist	External		-
	totalWhitelisted	Public		-
	getLastPresalePurchaseTime	External		-
	isWhitelisted	External		-
	setTeamAddresses	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
ETFSwap
onlyOwner
onlyWhitelisted
<Constructor>
balanceOf
releasePresaleVestedTokens
releaseTeamVestedTokens
calculateVestedAmount
transfer
handleVesting
isInTeamAddresses
transferFrom
calculateTaxAmount
mint
totalSupply
setSellTaxRate
setBuyTaxRate
addToWhitelist
removeFromWhitelist
getWhitelist
totalWhitelisted
getLastPresalePurchaseTime
isWhitelisted
setTeamAddresses
SafeMath (lib)
add
sub
mul
div
totalAllocation
mul
tokens
mul
sub
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.

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