Cyberscope

Audit Report

ETFSwap

April 2024

Repository	https://github.com/hamzabadshah1/etfswap
Commit	76fa7f7b0bb730ecb0db2302679b24de2f764f05
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
●	FAC	Flawed Allocation Check	Unresolved
●	CCC	Contradictory Condition Check	Unresolved
●	MTV	Misleading Tax Variable	Unresolved
●	OAC	Overlapping Access Control	Unresolved
●	UPS	Undefined Presale Start	Unresolved
●	UVL	Unused Vesting Limit	Unresolved
●	CR	Code Repetition	Unresolved
●	ETS	Excess Team Set	Unresolved
●	EIS	Excessively Integer Size	Unresolved
●	ILS	Inefficient Limit Setting	Unresolved
●	MEM	Misleading Error Messages	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MTEE	Missing Transfer Event Emission	Unresolved
●	RGF	Redundant Getter Function	Unresolved

●	RSML	Redundant SafeMath Library	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	RVV	Redundant Vesting Variables	Unresolved
●	UMV	Unused Mapping Variables	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

Table of Contents

Review

Contract Name	ETFSwap
Repository	https://github.com/hamzabadshah1/etfswap
Commit	76fa7f7b0bb730ecb0db2302679b24de2f764f05
Testing Deploy	https://testnet.bscscan.com/address/0x957715d93aa0ad21eed095e27efc0c35acca67d9
Symbol	ETFS
Decimals	18
Total Supply	1,000,000,000
Badge Eligibility	Must Fix Criticals

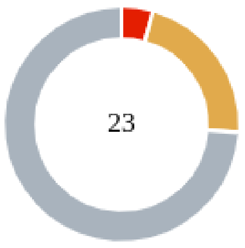
Audit Updates

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

Source Files

Filename	SHA256
contracts/ETFSwap.sol	cf1fdbe55d19a8c45c3691d1bccedf43d73811e9597725c83e393f5b89807b9a

Findings Breakdown



🟥	Critical	1
🟧	Medium	5
⬜	Minor / Informative	17

Severity	Unresolved	Acknowledged	Resolved	Other
🟥 Critical	1	0	0	0
🟧 Medium	5	0	0	0
⬜ Minor / Informative	17	0	0	0

FAC - Flawed Allocation Check

Criticality	Critical
Location	contracts/ETFSwap.sol#L321,362
Status	Unresolved

Description

The contract is designed to control the release of vested tokens to team members, ensuring that the total amount claimed by each member does not exceed their individual allocation or the overall team allocation (`TEAM_ALLOCATION`). The mechanism in place attempts to enforce this rule by tracking the total vested amount each team member has claimed in a mapping (`totalTeamVestedAmount[msg.sender]`). However, this implementation flaw only checks and updates the claimed amount on a per-user basis, without considering the cumulative total of all tokens vested to the entire team. This oversight allows each user to claim up to the total `TEAM_ALLOCATION` , significantly exceeding the intended limit when considering multiple team members. The checks and updates operations are performed individually for each sender, leading to a scenario where the collective amount vested to all team members could surpass the total allocated amount, undermining the contract's intention and possibly affecting the token's value and distribution fairness.

```
    function releaseTeamVestedTokens() external onlyWhitelisted {
        ...
        // Ensure the total vested amount for team members doesn't exceed
the allocated amount
        require(
            totalTeamVestedAmount[msg.sender] + vestedAmount <=
TEAM_ALLOCATION,
            "Total vested amount for team members exceeds allocated
amount"
        );

        // Ensure the individual vested limit for team members is not
exceeded
        require(
            totalTeamVestedAmount[msg.sender] + vestedAmount <=
                individualTeamVestedLimit[msg.sender],
            "Individual vested limit for team members exceeded"
        );

        ...

        // Update the total vested amount for the team member
        totalTeamVestedAmount[msg.sender] += vestedAmount;
    }

    // Function to release vested tokens for presale participants
    function releasePresaleVestedTokens() external {
        ...
        // Ensure the total vested amount for presale participants
doesn't exceed the allocated amount
        require(
            totalPresaleVestedAmount[msg.sender] + vestedAmount <=
                PRESALE_ALLOCATION,
            "Total vested amount for presale participants exceeds
allocated amount"
        );

        ...
        totalPresaleVestedAmount[msg.sender] += vestedAmount;
    }
```

Recommendation

It is recommended to introduce a global variable that tracks the cumulative total of all tokens vested to team members. This should be checked against the `TEAM_ALLOCATION` during each vesting operation to ensure the total amount distributed does not exceed the predefined allocation for the team. Additionally, the contract should maintain separate accounting for individual and total vesting amounts, updating both figures appropriately during each transaction. Implementing these changes will prevent any single member from disproportionately claiming tokens beyond their share and ensure the overall allocation limits are respected, maintaining the token distribution's integrity and fairness.

CCC - Contradictory Condition Check

Criticality	Medium
Location	contracts/ETFSwap.sol#L295,478,487
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 vesting start time for team members if the address being added to the whitelist is a team address and if its associated `_teamVestingStart` is zero. However, a logical discrepancy arises from this condition. The `isInTeamAddresses` check verifies whether an address belongs to the team addresses array, and any address added to `teamAddresses` have already a non-zero `_teamVestingStart` value. Consequently, the condition `isInTeamAddresses(_address) && _teamVestingStart[_address] == 0` is inherently contradictory, as an address recognized as a team address (thus passing the `isInTeamAddresses` check) would not have a `_teamVestingStart` 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.

```
    function isInTeamAddresses(address _address) private view returns
(bool) {
        for (uint256 i = 0; i < teamAddresses.length; i++) {
            if (teamAddresses[i] == _address) {
                return true;
            }
        }
        return false;
    }

    function addToWhitelist(address _address) external onlyOwner {
        require(_address != address(0) && _address != owner, "Invalid
address");

        // Check if the address is not already whitelisted
        if (!whitelist[_address]) {
            whitelist[_address] = true;

            // Initialize vesting start time for team members
            if (
                isInTeamAddresses(_address) &&
_teamVestingStart[_address] == 0
            ) {
                _teamVestingStart[_address] = block.timestamp;
            }

            emit AddedToWhitelist(_address);
        }
    }
```

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 `isInTeamAddresses` and `_teamVestingStart[_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.

MTV - Misleading Tax Variable

Criticality	Medium
Location	contracts/ETFSwap.sol#L288
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.

OAC - Overlapping Access Control

Criticality	Medium
Location	contracts/ETFSwap.sol#L328
Status	Unresolved

Description

The contract includes the `releaseTeamVestedTokens` function, which is safeguarded by the `onlyWhitelisted` modifier, restricting access exclusively to addresses on the whitelist. Furthermore, within this function, there's an additional requirement that the caller's (`msg.sender`) `_teamVestingStart` must be greater than zero, implicitly necessitating that the caller is not only whitelisted but also a recognized team member with initiated vesting. This dual-layered access control introduces redundancy by requiring that all team addresses must also be added to the whitelist to participate in token vesting. While these checks aim to reinforce security and ensure proper authorization, they complicate the process by mandating simultaneous management of two separate mappings for essentially the same group of users, which are team members eligible for token vesting. This approach increases the administrative overhead and complicates the logic, potentially leading to errors or oversights in maintaining the lists accurately and consistently.

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

Recommendation

It is recommended to simplify the access control mechanism within the `releaseTeamVestedTokens` function. If the intended functionality is for all team addresses to claim their vested tokens, then the `onlyWhitelisted` modifier may be unnecessary and could be removed. This change would streamline the process, relying solely on the check for a positive `_teamVestingStart` to confirm a caller's eligibility.

Such a revision would reduce complexity, lower the risk of administrative errors, and ensure a more direct and efficient method for team members to access their vested tokens. Removing the redundant whitelisting requirement would still preserve security while making the system more user-friendly and manageable.

UPS - Undefined Presale Start

Criticality	Medium
Location	contracts/ETFSwap.sol#L402
Status	Unresolved

Description

The contract contains the `releasePresaleVestedTokens` function, which is intended to allow presale participants to claim their allocated tokens in line with the `PRESALE_ALLOCATION`. The function contains a check to ensure that the `_presaleVestingStart` mapping for the caller (`msg.sender`) is set, indicating the start of their vesting period. This check is designed to prevent premature token claims and to enforce the vesting schedule. However, the contract lacks any mechanism to update or set values within the `_presaleVestingStart` mapping. This oversight renders it impossible for users to be eligible to claim their presale tokens, as the initial condition required for the vesting process to begin cannot be met. Consequently, this flaw directly impacts the functionality of the `releasePresaleVestedTokens` function and, by extension, the effectiveness of the presale token distribution strategy.

```
   function releasePresaleVestedTokens() external {
        require(
            _presaleVestingStart[msg.sender] > 0,
            "No vested tokens to release"
        );
        require(
            block.timestamp >=
                _presaleVestingStart[msg.sender] + RELEASE_INTERVAL,
            "Vesting period not ended yet"
        );

        ...

        // Update the total vested amount for the presale
participant
        totalPresaleVestedAmount[msg.sender] += vestedAmount;
    }
```

Recommendation

It is recommended to reconsider the implementation of the `releasePresaleVestedTokens` functionality. If the intended functionality is to allow presale participants to claim their allocated tokens, the contract must include a method to set the `_presaleVestingStart` for each participant. This could be achieved by introducing a new function accessible only by the contract owner or through an automated process triggered by participation in the presale. The function should allow the setting of the vesting start time for each participant, ensuring that users can meet the conditions required to claim their vested tokens. Alternatively, if managing individual vesting start times is not feasible or desired, consider revising or removing the dependency on the `releasePresaleVestedTokens` functionality. Adopting these changes will ensure the contract functions as intended and supports a fair and transparent distribution of presale tokens.

UVL - Unused Vesting Limit

Criticality	Medium
Location	contracts/ETFSwap.sol#L144,193
Status	Unresolved

Description

The contract is found to declare the `MAX_VESTING_ADDRESSES` variable with an initial value intended to limit the number of the `updateMaxVestingAddresses` function, designed to allow the contract owner to update this limit. However, the `MAX_VESTING_ADDRESSES` variable is not utilized in any operational logic or functions within the contract. This discrepancy indicates that both the variable's declaration and the corresponding update function are redundant, as they do not contribute to the contract's functionality or enforce any limits as initially intended.

```
   uint256 public MAX_VESTING_ADDRESSES = 100; // Example:
Limit to 100 addresses

    function updateMaxVestingAddresses(uint256 newValue)
external onlyOwner {
        require(newValue > 0, "New value must be greater than
zero");
        MAX_VESTING_ADDRESSES = newValue;
        emit MaxVestingAddressesUpdated(newValue);
    }
```

Recommendation

It is recommended to implement the corresponding functionality regarding the `MAX_VESTING_ADDRESSES` variable if there was an initial intention to limit the number of vesting addresses for a specific purpose within the contract's ecosystem. This implementation should ensure that the variable's value is actively checked and enforced in relevant contract operations. Alternatively, if there is no foreseeable use or requirement for this limiting functionality in the contract's future development, it would be prudent to remove both the variable and the update function. Doing so will simplify the contract's codebase, reduce unnecessary gas costs associated with these redundant operations, and enhance the contract's overall clarity and efficiency.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L321,362
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 `releaseTeamVestedTokens` function with the `releasePresaleVestedTokens` function share similar code segments.

```
   // Function to release vested tokens for team members
    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

        // Ensure the total vested amount for team members doesn't exceed
the allocated amount
        require(
            totalTeamVestedAmount[msg.sender] + vestedAmount <=
TEAM_ALLOCATION,
            "Total vested amount for team members exceeds allocated
amount"
        );

        // Ensure the individual vested limit for team members is not
exceeded
        require(
            totalTeamVestedAmount[msg.sender] + vestedAmount <=
                individualTeamVestedLimit[msg.sender],
            "Individual vested limit for team members exceeded"
        );

        // Deduct the vested amount from the owner's balance
        balances[owner] = balances[owner].sub(vestedAmount);

        // Transfer the vested amount to the team member
        balances[msg.sender] = balances[msg.sender].add(vestedAmount);
        emit Transfer(owner, msg.sender, vestedAmount);

        // Update the total vested amount for the team member
        totalTeamVestedAmount[msg.sender] += vestedAmount;
    }

    // Function to release vested tokens for presale participants
    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.

ETS - Excess Team Set

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L510
Status	Unresolved

Description

The contract contains the `setTeamAddress` function, empowering the owner to designate addresses as team addresses, thereby making them eligible to claim team allocation tokens. This functionality is crucial for distributing tokens among team members according to predefined allocation plans. However,the funtion permits the owner to continue adding team addresses without considering whether the total team allocation limit has been reached. This capability could lead to a situation where more addresses are designated as team members than the allocation can support, potentially diluting the token distribution and undermining the integrity of the allocation process. The absence of a check against the total team allocation before adding new team addresses is a fundamental flaw that can lead to practical and logistical challenges, including the over-commitment of tokens beyond the intended limit.

```
    function setTeamAddress(address _teamAddress) external onlyOwner {
        require(
            _teamAddress != address(0) && _teamAddress != owner,
            "Invalid address"
        );

        // Check if the address is not already in the teamAddresses
array
        if (!isInTeamAddresses(_teamAddress)) {
            teamAddresses.push(_teamAddress);

            // Initialize vesting start time for team member if not
already initialized
            if (_teamVestingStart[_teamAddress] == 0) {
                _teamVestingStart[_teamAddress] = block.timestamp;
                emit VestingStartInitialized(_teamAddress,
block.timestamp);
            }
        }
    }
```

Recommendation

It is recommended to implement a safeguard within the `setTeamAddress` function to prevent the addition of new team addresses once the total team allocation limit is reached. This could involve introducing an additional condition to check the cumulative allocated amount against the predefined total team allocation before allowing a new address to be added. Such a mechanism would ensure that the contract adheres to the allocation plan, preventing the possibility of exceeding the token distribution limits set for team members. Implementing this recommendation will enhance the contract's robustness, ensure fair and transparent token distribution, and uphold the integrity of the team allocation process.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L116
Status	Unresolved

Description

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

```
uint256 public sellTaxRate = 5; // Default sell tax rate: 5%
uint256 public buyTaxRate = 1; // Default buy tax rate: 1%
```

Recommendation

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

ILS - Inefficient Limit Setting

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L305,313,346,389
Status	Unresolved

Description

The contract is designed to enforce individual vested limits for team members and presale participants through `individualTeamVestedLimit` and `individualPresaleVestedLimit` mappings. These limits are crucial for maintaining equitable token distribution and preventing any single address from claiming an excessive share of the allocated tokens. However, the current implementation requires the contract owner to manually set these limits for each address via separate functions. This method not only introduces administrative overhead but also increases the risk of errors, as each limit must be individually managed and updated. Additionally, this approach can lead to significant inefficiencies in contract execution, particularly as the number of participants grows. The necessity to call separate functions for setting each address's limit can result in increased transaction costs and potential delays in managing the distribution process effectively.

```
require(
    totalTeamVestedAmount[msg.sender] + vestedAmount <=
        individualTeamVestedLimit[msg.sender],
    "Individual vested limit for team members exceeded"
);
...
require(
    totalPresaleVestedAmount[msg.sender] + vestedAmount <=
        individualPresaleVestedLimit[msg.sender],
    "Individual vested limit for presale participants exceeded"
);
...
function setIndividualTeamVestedLimit(
    address _address,
    uint256 limit
) external onlyOwner {
    individualTeamVestedLimit[_address] = limit;
}
...
function setIndividualPresaleVestedLimit(
    address _address,
    uint256 limit
) external onlyOwner {
    individualPresaleVestedLimit[_address] = limit;
}
```

Recommendation

It is recommended to streamline the process by integrating the limit setting mechanism directly into the functions used for adding team members and presale participant addresses within their respective allocations. This could involve extending these functions to accept the vested limit as an additional parameter, allowing the contract owner to set both the participant's address and their corresponding limit in a single transaction. Such a modification would not only reduce the complexity and potential for errors but also optimize contract efficiency by minimizing the number of transactions required for participant management. Implementing this change would significantly improve the administrative experience, reduce operational costs, and enhance the overall security and effectiveness of the token distribution process.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L419,429
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

Specifically the max tax can be set up to 25 %, not 100% as idicated by the error message.

```
require(
    newSellTaxRate <= 25,
    "Sell tax rate must be less than or equal to 100%"
)
...
require(
    newBuyTaxRate <= 25,
    "Buy tax rate must be less than or equal to 100%"
);
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L186,309,317
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
    function setLiquidityPairAddress(
        address _liquidityPairAddress
    ) external onlyOwner {
        liquidityPairAddress = _liquidityPairAddress;
    }

    function setIndividualTeamVestedLimit(
        address _address,
        uint256 limit
    ) external onlyOwner {
        individualTeamVestedLimit[_address] = limit;
    }

    function setIndividualPresaleVestedLimit(
        address _address,
        uint256 limit
    ) external onlyOwner {
        individualPresaleVestedLimit[_address] = limit;
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MTEE - Missing Transfer Event Emission

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

Description

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

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

Recommendation

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

RGF - Redundant Getter Function

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L95,466
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.

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#L186,78,497,510
Status	Unresolved

Description

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

```
    function setLiquidityPairAddress(
        address _liquidityPairAddress
    ) external onlyOwner {
        liquidityPairAddress = _liquidityPairAddress;
    }

    function addToWhitelist(address _address) external
onlyOwner {
        require(_address != address(0) && _address != owner,
"Invalid address");

        ...
        }
    }

    // Function to remove addresses from the whitelist
    function removeFromWhitelist(
        address[] calldata addresses
    ) external onlyOwner {
        for (uint256 i = 0; i < addresses.length; i++) {
            // Check if the address is currently whitelisted
            if (whitelist[addresses[i]]) {
                whitelist[addresses[i]] = false;
            }
        }
        emit RemovedFromWhitelist(addresses);
    }

    // Function to set a team address and initialize vesting
start time
    function setTeamAddress(address _teamAddress) external
onlyOwner {
        require(
            _teamAddress != address(0) && _teamAddress !=
owner,
            "Invalid address"
        );

        ...
    }
```

Recommendation

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

RVV - Redundant Vesting Variables

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L120
Status	Unresolved

Description

The contract initialize variables related to vesting periods, specifically `TEAM_VESTING_PERIOD` and `PRESALE_VESTING_PERIOD` , both set to a duration of 3 weeks. However these variables are not utilized in any of the contract's functions or logic. This lack of utilization renders the variables redundant, contributing neither to the functionality nor to the efficiency of the contract. The presence of such unused variables not only increases the contract's complexity without any operational benefit but also poses a risk of misunderstanding about the actual code functionality.

```
uint256 public constant TEAM_VESTING_PERIOD = 3 weeks;
uint256 public constant PRESALE_VESTING_PERIOD = 3 weeks;
```

Recommendation

It is recommended to reconsider the usage of these variables within the contract's logic. If there is no foreseeable use or requirement for these vesting period variables in future updates or functionalities, it would be prudent to remove them. Eliminating unused variables can streamline the contract's codebase, reduce potential confusion, and enhance the overall clarity and maintainability of the contract. Should these variables be intended for future use, it is advisable to document their intended purpose clearly to avoid any ambiguity.

UMV - Unused Mapping Variables

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L146,150
Status	Unresolved

Description

The contract is found to declare mappings, specifically aimed at tracking the total allocated tokens for team members (`totalTeamAllocation`) and presale participants (`totalPresaleAllocation`). However, these mappings are not utilized in any part of the contract's logic or operations. This lack of utilization indicates that the declared mappings are redundant within the current framework of the contract. The presence of such unused mappings not only unnecessarily increases the contract's complexity but also may lead to confusion regarding their purpose and potential impact on the contract's functionality.

```
// Counter to track the total allocated tokens for team members
uint256 public totalTeamAllocation;

// Counter to track the total allocated tokens for presale
participants
uint256 public totalPresaleAllocation;
```

Recommendation

It is recommended to assess the necessity of the declared mappings within the contract. If these mappings do not serve a specific purpose or are not intended for future use in the contract's logic, considering their removal could be beneficial. Removing unused mappings can help in simplifying the contract's codebase, reducing gas costs associated with contract deployment and interaction, and improving the overall readability and maintainability of the contract. Should there be plans to utilize these mappings in future updates or features, it is advisable to document their intended use clearly to ensure clarity and prevent any potential oversight.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L147,150
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
uint256 public totalTeamAllocation
uint256 public totalPresaleAllocation
```

Recommendation

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

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L144,187,295,306,314,478,510
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.

```
uint256 public MAX_VESTING_ADDRESSES = 100
address _liquidityPairAddress
address _address
address _teamAddress
```

Recommendation

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

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L410,411
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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L189
Status	Unresolved

Description

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

```
liquidityPairAddress = _liquidityPairAddress
```

Recommendation

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

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

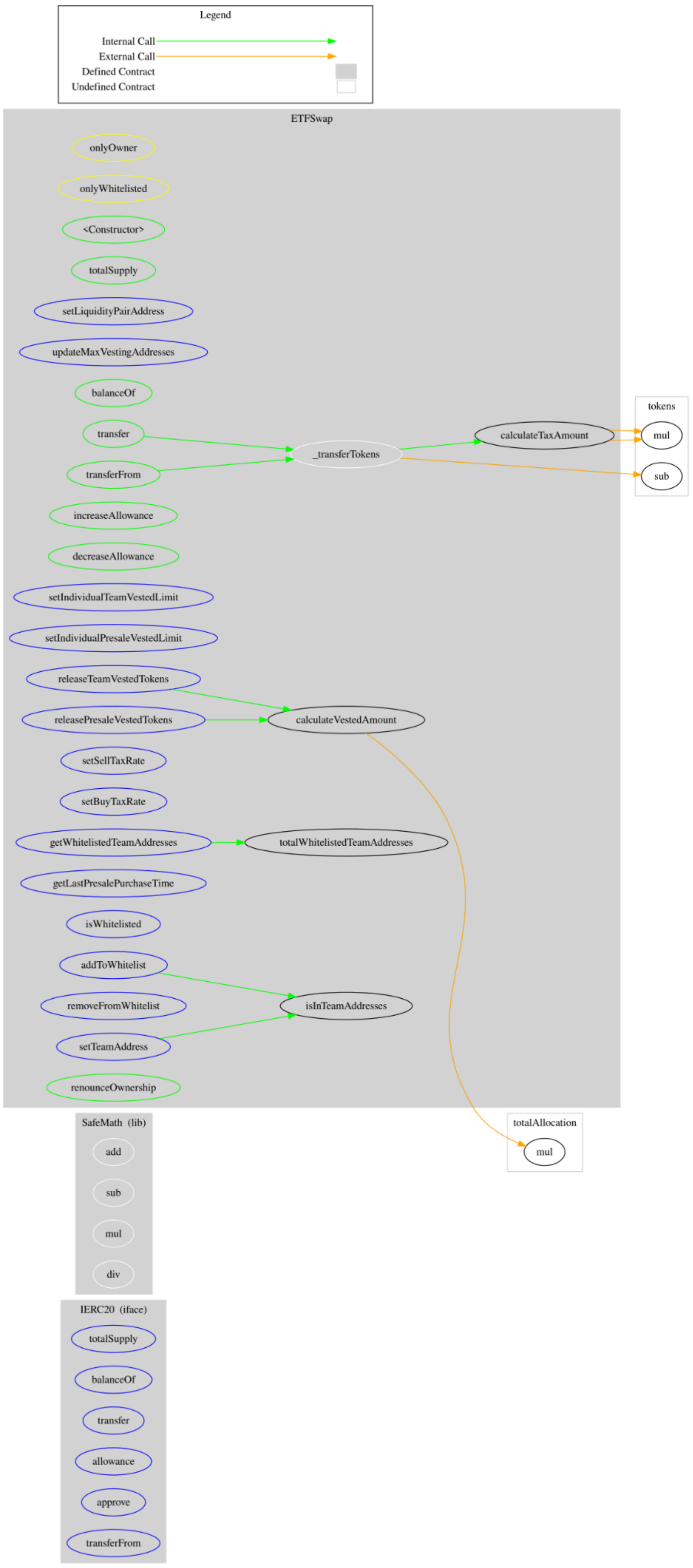
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
SafeMath	Library			
	add	Internal		
	sub	Internal		
	mul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

	balanceOf	Public		-
	_transferTokens	Internal	✓	
	transfer	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	calculateTaxAmount	Private		
	isInTeamAddresses	Private		
	setIndividualTeamVestedLimit	External	✓	onlyOwner
	setIndividualPresaleVestedLimit	External	✓	onlyOwner
	releaseTeamVestedTokens	External	✓	onlyWhitelisted
	releasePresaleVestedTokens	External	✓	-
	calculateVestedAmount	Private		
	setSellTaxRate	External	✓	onlyOwner
	setBuyTaxRate	External	✓	onlyOwner
	getWhitelistedTeamAddresses	External		-
	totalWhitelistedTeamAddresses	Public		-
	getLastPresalePurchaseTime	External		-
	isWhitelisted	External		-
	addToWhitelist	External	✓	onlyOwner
	removeFromWhitelist	External	✓	onlyOwner
	setTeamAddress	External	✓	onlyOwner
	renounceOwnership	Public	✓	onlyOwner

Inheritance Graph

IERC20
SafeMath
ETFSwap

Flow Graph


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

Summary

ETFSwap contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. ETFSwap is an interesting project that has a friendly and growing community. The Smart Contract analysis reported one critical error. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions. There is also a limit of max 15% 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