Cyberscope

Audit Report

ETFSwap

April 2024

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

Commit 2dcaba4616309fa888140d295d113cc1733a13d6

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	IIVL	Inadequate Individual Vested Limit	Unresolved
Medium	DLCI	Dynamic Limit Calculation Inconsistency	Unresolved
Medium	IFU	Inefficient Functions Usages	Unresolved
Medium	MTV	Misleading Tax Variable	Unresolved
Minor / Informative	IRH	Inconsistent Removal Handling	Unresolved
Minor / Informative	IAC	Inefficient Amount Calculation	Unresolved
Minor / Informative	RCS	Redundant Comment Segments	Unresolved
Minor / Informative	RED	Redundant Enum Declarations	Unresolved
Minor / Informative	RSML	Redundant SafeMath Library	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L08	Tautology or Contradiction	Unresolved
Minor / Informative	L09	Dead Code Elimination	Unresolved
Minor / Informative	L13	Divide before Multiply Operation	Unresolved

Table of Contents

Review

Contract Name	ETFSwap
Repository	https://github.com/hamzabadshah1/etfswap
Commit	2dcaba4616309fa888140d295d113cc1733a13d6
Testing Deploy	https://testnet.bscscan.com/address/0x89ed55138069a210c00ffd9afd046571942d3bc3
Symbol	ETFS
Decimals	18
Total Supply	1,000,000,000
Badge Eligibility	Must Fix Criticals

Audit Updates

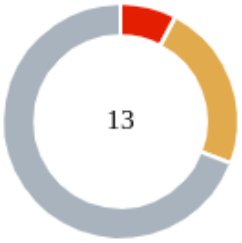
Initial Audit	27 Mar 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v1/audit.pdf
Corrected Phase 2	04 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v2/audit.pdf
Corrected Phase 3	05 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/etfswap/v3/audit.pdf
Corrected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

Corrected Phase 5	10 Apr 2024

Source Files

Filename	SHA256
contracts/ETFSwap.sol	d92a2c72419e787642fcd07f1af065b2e34183bdbdff76a02b0b0effa8368848
contracts/library/SafeMath.sol	731263b65452854c42f6f12bfdfb9deeea821c3a81aca05d2618bc2fdf10225a
contracts/interfaces/IERC20.sol	4f4f7372b9a27242232b3f03facfd771c480082a9f86f4b0a82103f3022212ae

Findings Breakdown



Critical	1
Medium	3
Minor / Informative	9

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	3	0	0	0
Minor / Informative	9	0	0	0

IIVL - Inadequate Individual Vested Limit

Criticality	Critical
Location	contracts/ETFSwap.sol#L313,337
Status	Unresolved

Description

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

```
require(
    (_type == OperationType.Team &&
        getTotalTeamVestedAmount() -
            individualTeamVestedLimit[_address] +
            difference <=
        totalAllocation) ||
        (_type == OperationType.Presale &&
            getTotalPresaleVestedAmount() -
                individualPresaleVestedLimit[_address] +
                difference <=
            totalAllocation),
    "Total allocated amount exceeded for members"
);
```

Example Scenario:

Consider a case where the `getTotalPresaleVestedAmount` returns 2800 tokens, the `totalAllocation` is 3000 tokens, and a specific address has an `individualPresaleVestedLimit` of 300 tokens. This setup indicates that the address could have minted up to 300 tokens.

When `setIndividualVestedLimit` is invoked for this address with a `limit` of 500, the calculated `difference` is 200 tokens (500 - 300 = 200).

The `require` check then evaluates as follows: 2800 - 300 + 200 <= 3000 resulting to 2700 <= 3000, which is `true`. However, this calculation fails to account for the 300 tokens already minted by the address. As such, the address is inaccurately considered eligible to claim up to the new limit of 500 tokens, disregarding the previously minted amount.

Recommendation

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

DLCI - Dynamic Limit Calculation Inconsistency

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

Description

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

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

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

Recommendation

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

IFU - Inefficient Functions Usages

Criticality	Medium
Location	contracts/ETFSwap.sol#L313,525,576,606
Status	Unresolved

Description

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

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

    function addToWhitelist(
        address _address,
        WhitelistType _type
    ) external onlyOwner {
        require(_address != address(0) && _address != owner, "Invalid
address");
        mapping(address => bool) storage whitelistToAdd = _type ==
            WhitelistType.Team
            ? teamWhitelist
            : presaleWhitelist;
        require(!whitelistToAdd[_address], "Address is already
whitelisted");
        whitelistToAdd[_address] = true;
        emit AddedToWhitelist(_address, _type);
    }

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

```
            getTotalTeamAllocation() +
                individualTeamVestedLimit[_teamAddress] <=
                TEAM_ALLOCATION,
            "Total team allocation limit reached"
        );
        if (!isInTeamAddresses(_teamAddress)) {
            teamAddresses.push(_teamAddress);
            if (_teamVestingStart[_teamAddress] == 0) {
                _teamVestingStart[_teamAddress] = block.timestamp;
                emit VestingStartInitialized(_teamAddress,
block.timestamp);
            }
        }
    }

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

Recommendation

It is recommended to merge these functions into a single, more comprehensive function capable of handling all related operations regarding the set or add to the whitelist functionality. This unified function could accept parameters to identify the operation type (e.g., setting vested limits, adding to whitelist, initializing vesting) and execute the corresponding logic based on these parameters. This approach would streamline the contract's functionality, reduce redundancy, and simplify the process of managing vested limits, whitelisting, and address initialization. By consolidating related functionalities, the

contract can achieve a more efficient, maintainable, and error-resistant implementation. This change would also potentially reduce the gas costs associated with deploying and interacting with the contract, as it minimizes the number of function calls required to perform related operations.

MTV - Misleading Tax Variable

Criticality	Medium
Location	contracts/ETFSwap.sol#L236
Status	Unresolved

Description

The contract uses the `buyTaxRate` variable, which implies that a tax is applied on buy transactions. However, the tax is not applied during the actual purchase of tokens but rather under specific conditions that do not necessarily correspond to a typical buy operation. Specifically, the tax is applied when tokens are transferred by the owner of the contract, as indicated by the condition if (`from == owner`). This condition suggests that the tax rate designated by `buyTaxRate` is only utilized in transactions initiated by the contract's owner, rather than being a general tax on all buy transactions. This misalignment between the variable name and its functional implementation can lead to confusion and misinterpretation of the contract's tax logic, potentially misleading stakeholders about the nature and circumstances under which taxes are applied.

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

Recommendation

It is recommended to revise the contract's tax implementation to more accurately reflect the intended tax logic. If the goal is to apply a buy tax on actual purchase transactions, the condition should be adjusted to check if the `from` address is the `liquidityPairAddress` , which would indicate a buy transaction from a liquidity pool. This approach would ensure that the tax is applied specifically to buy transactions, aligning the functionality with the variable name `buyTaxRate` . Implementing this change would enhance clarity and transparency in the contract's tax logic, ensuring that stakeholders have a correct understanding of when and how taxes are applied. Additionally, renaming the variable to more accurately describe its function could further reduce confusion and improve the contract's readability and maintainability.

IRH - Inconsistent Removal Handling

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L525,552,564
Status	Unresolved

Description

The contract is currently employing a single function, `addToWhitelist`, to add addresses to either a team whitelist or a presale whitelist based on the type specified. This function effectively manages additions by leveraging a condition to distinguish between the two whitelists. However, for removals, the contract diverges into two separate functions, the `removeFromTeamWhitelist` and `removeFromPresaleWhitelist`. This bifurcation introduces unnecessary complexity and potential inconsistencies in handling whitelist removals. Each removal function operates independently on its respective whitelist, thereby doubling the effort required to maintain and audit the contract's removal logic. Moreover, this approach increases the risk of bugs or inconsistencies, as any change in removal logic must be meticulously replicated across both functions.

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

    function removeFromTeamWhitelist(
        address[] calldata addresses
    ) external onlyOwner {
        for (uint256 i = 0; i < addresses.length; i++) {
            if (teamWhitelist[addresses[i]]) {
                teamWhitelist[addresses[i]] = false;
            }
        }
        emit RemovedFromWhitelist(addresses);
    }

    // Function to remove addresses from the presale whitelist
    function removeFromPresaleWhitelist(
        address[] calldata addresses
    ) external onlyOwner {
        for (uint256 i = 0; i < addresses.length; i++) {
            if (presaleWhitelist[addresses[i]]) {
                presaleWhitelist[addresses[i]] = false;
            }
        }
        emit RemovedFromWhitelist(addresses);
    }
```

Recommendation

It is recommended to streamline the removal process by consolidating the two existing removal functions into a single function. This unified function should accept an address array and a whitelist type, mirroring the design of the addToWhitelist function. By adopting this approach, the contract can ensure consistent handling of both additions and removals,

simplify its logic, and reduce the potential for errors. Furthermore, this modification will enhance the maintainability of the contract, as any future changes to the removal logic would only need to be implemented in one place.

IAC - Inefficient Amount Calculation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L360,370
Status	Unresolved

Description

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

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

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

Recommendation

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

RCS - Redundant Comment Segments

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L539
Status	Unresolved

Description

The contract contains multiple segments of code that are commented out. While commented code can serve as documentation or indicate future development plans, in this case, the commented-out code segments do not provide meaningful documentation or context. Instead, they introduce ambiguity regarding the intended features and current state of the contract. This could lead to confusion about the contract's capabilities and operational logic. The presence of such code may also suggest incomplete features or tentative updates that have not been fully implementated.

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

Recommendation

It is recommended to remove the segments of commented-out code from the contract. This will improve the clarity and readability of the contract's codebase, ensuring that it accurately reflects the implemented and active functionalities. Otherwise if certain

commented-out segments are placeholders for future development, they should be replaced with clear documentation outlining the intended features.

RED - Redundant Enum Declarations

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L95,100
Status	Unresolved

Description

The contract is employing two distinct enumerations, `WhitelistType` and `OperationType`, to represent the states of `Team` and `Presale`. While each enum is intended to define contextual states within different contract functionalities, their identical values indicate a redundancy in the contract's type definitions. This duplication could lead to confusion, increased complexity, and potential for errors in contract maintenance or expansion. When developers or auditors need to understand or modify the contract's logic, distinguishing between these enums without contextual overlap could unnecessarily complicate the process. Moreover, this redundancy may also complicate integration with external systems or contracts, which need to interact with these states uniformly.

```
enum WhitelistType {
    Team,
    Presale
}
enum OperationType {
    Team,
    Presale
}
```

Recommendation

It is recommended to consolidate the `WhitelistType` and `OperationType` enums into a single enumeration that represents both the whitelist and operation states. By unifying these enums, the contract can reduce redundancy, streamline code understanding, and simplify maintenance. This approach not only enhances code clarity and efficiency but also promotes better practices in smart contract design by avoiding unnecessary duplications. Furthermore, a single enum for both contexts can facilitate easier updates and

modifications to the contract's logic in the future, providing a more scalable and manageable framework for state representation.

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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L137,252,253,264,270,277,278,314,316,526,527,576,606
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _liquidityPairAddress
address _address
address[] storage _list
WhitelistType _type
OperationType _type
address _teamAddress
address _presaleAddress
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L323
Status	Unresolved

Description

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

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

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

Recommendation

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

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L276
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function isInAddresses(
        address _address,
        WhitelistType _type
    ) internal view returns (bool) {
        address[] storage addressesList = _type == WhitelistType.Team
            ? teamAddresses
            : presaleAddresses;
        for (uint256 i = 0; i < addressesList.length; i++) {
            if (addressesList[i] == _address) {
                return true;
            }
        }
        return false;
    }
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/ETFSwap.sol#L427,428
Status	Unresolved

Description

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

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

Recommendation

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

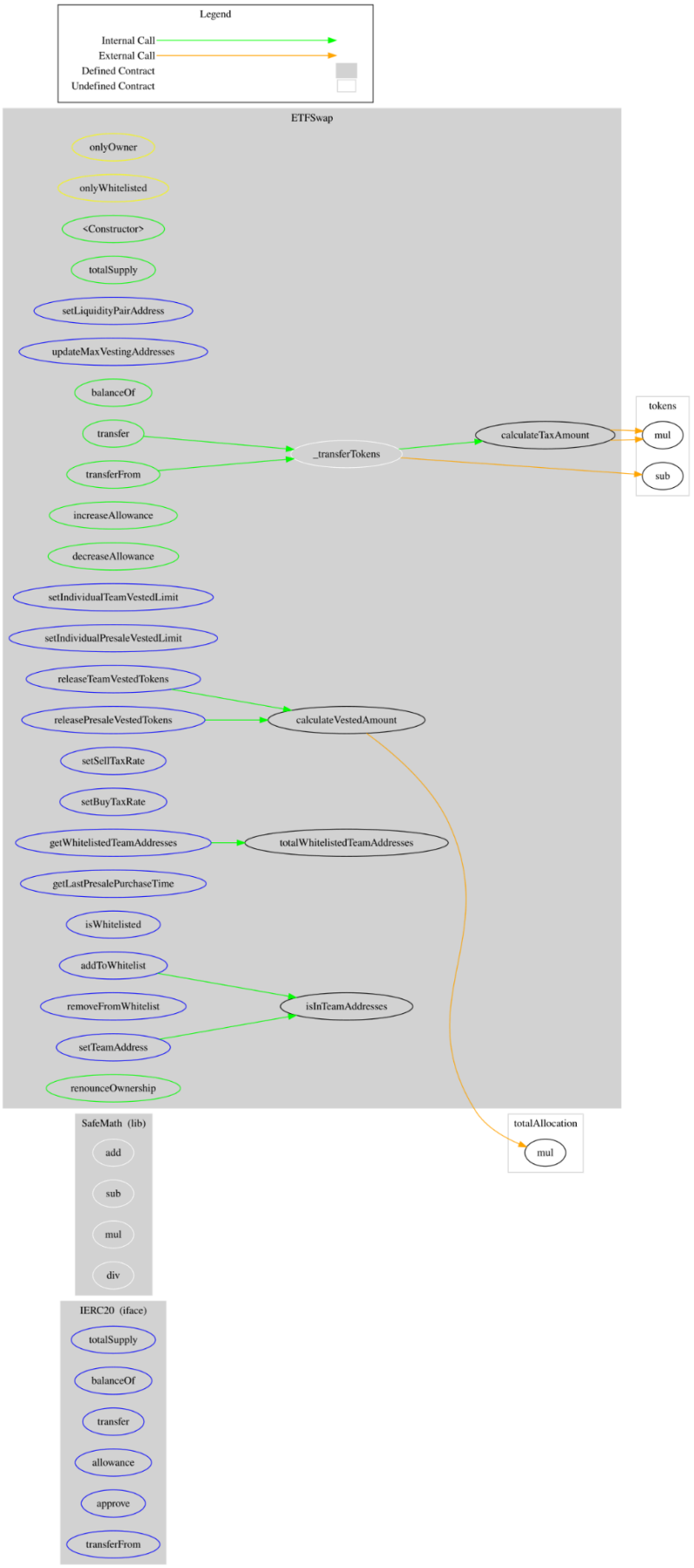
Functions Analysis

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

	getTotalPresaleVestedAmount	Private		
	releaseTeamVestedTokens	External	✓	onlyTeamAddresses
	releasePresaleVestedTokens	External	✓	onlyWhitelisted
	_releaseVestedTokens	Internal	✓	
	calculateVestedAmount	Private		
	setSellTaxRate	External	✓	onlyOwner
	setBuyTaxRate	External	✓	onlyOwner
	getWhitelistedTeamAddresses	External		-
	getWhitelistedPresaleAddresses	External		-
	totalWhitelistedTeamAddresses	Public		-
	totalWhitelistedPresaleAddresses	Public		-
	isTeamWhitelisted	External		-
	isPresaleWhitelisted	External		-
	addToWhitelist	External	✓	onlyOwner
	removeFromTeamWhitelist	External	✓	onlyOwner
	removeFromPresaleWhitelist	External	✓	onlyOwner
	setTeamAddress	External	✓	onlyOwner
	getTotalTeamAllocation	Private		
	setPresaleAddress	External	✓	onlyOwner
	getTotalPresaleAllocation	Private		
	renounceOwnership	Public	✓	onlyOwner

Inheritance Graph

Flow Graph


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

Summary

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

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io