Cyberscope

Audit Report

Moonlabs

March 2023

SHA256 c021ea3c57bfbba72ee8097b0f50c50b75041ef3ec8fcecbcedca640b783a490
f3ef41718d2bd2a2fd45234063921d24a71bafafa0387eeab0b6b1c41484311b
feb646c4c147c0c44e0115a3055e12d9f1491e9d7d86d47923c23fe1a0bece73
b2cea2f09dfd04c62fea3649a435324f71aa9920ca9ff4dc72d6df531733fad8
7cef1147938ea185aa39c1c9807e8e8373896f8cf290901bf7f6ef8c1032bde1

Audited by © cyberscope

Table of Contents

Review

Testing Deploy

Contract Name	Explorer
MoonLabsLiquidityLocker	https://testnet.bscscan.com/address/0x8A4A17Cb72f118c1eee262F5D1144678A48A3Fab
MoonLabsReferral	https://testnet.bscscan.com/address/0x78a9af1C8FB53d9999A8bB6e66A3E2436C25db7A
MoonLabsTokenLocker	https://testnet.bscscan.com/address/0x945149ba2F7569c1B1eAe068003f4b3747F3F07A
MoonLabsVesting	https://testnet.bscscan.com/address/0x47408f44B874c55A28A97Fc0802dD8fAdBC876B1
MoonLabsWhitelist	https://testnet.bscscan.com/address/0x2b97895c6Ac40429f04c57140771F7D743bEAd04

Audit Updates

Initial Audit	09 Mar 2023

Source Files

Filename	SHA256
IDEXRouter.sol	bff5d69c904ec69fff1c7a5e7c4c19088ddd c4f1342ea13f83317b0fe1136f3f
MoonLabsLiquidityLocker.sol	c021ea3c57bfbba72ee8097b0f50c50b75 041ef3ec8fcecbcedca640b783a490
MoonLabsReferral.sol	f3ef41718d2bd2a2fd45234063921d24a7 1bafafa0387eeab0b6b1c41484311b
MoonLabsTokenLocker.sol	feb646c4c147c0c44e0115a3055e12d9f1 491e9d7d86d47923c23fe1a0bece73
MoonLabsVesting.sol	b2cea2f09dfd04c62fea3649a435324f71a a9920ca9ff4dc72d6df531733fad8
MoonLabsWhitelist.sol	7cef1147938ea185aa39c1c9807e8e837 3896f8cf290901bf7f6ef8c1032bde1

Introduction

The Moonlab ecosystem consists of five contracts. Two utility contracts and three locker contracts.

MoonLabsWhitelist

The MoonLabsWhitelist contract is used for creating whitelists for Moon Labs products. Whitelisting a token allows users to waive all fees on related Moon Labs products.

Roles

The contract roles consist of the owner role.

The `owner` is responsible for:

- Adding or removing a whitelisted address.
- Claim all contract deposited balance.
- Claim all `usdContract` token balance.

The users have the authority to:

- Purchase a whitelist with or without a discount code.
- Check the whitelisted address.

MoonLabsReferral

The MoonLabsReferral smart contract is used for creating and managing referral codes. It allows users to create referral codes for customers to use while purchasing Moon Labs products.

Roles

The contract roles consist of the owner role.

The `owner` role is responsible for:

- Delete referral code.
- Add reserved codes.
- Assign or remove reserved codes.
- Add or remove addMoonLabsContract.
- Claim all contract deposited balance.

The user has the authority to:

- Check active codes.
- Get the address to the referral code.
- Get referral code to address.
- Add rewards earned for a code.
- Create, delete, transfer, and reserve referral codes.

MoonLabsVesting

The MoonLabsVesting Contract allows token owners to create ERC20 token locks. The Contract supports creating multiple vesting instances, choosing between linear and standard Locks, and transferring Locks to other addresses. The Contract also includes a feature to buy back and burn the native token, as well as a referral code system and whitelist function. Lock creators cannot modify locks once they have been created, and withdraw owners cannot extend or change lock details.

Roles

The contract roles consist of the owner role.

The `owner` role is responsible for:

- Claim all contract deposited balance.
- Configure contract parameters like address, prices, and thresholds.

The user has the authority to:

- Create one or multiple vesting instances for a single token with fees or without fees or with a discount code.
- Transfer vesting instances ownership.
- Get nonces from address.
- Get the address from nonce.
- Get the lock tokens from the address.
- View claimable tokens.
- Withdraw unlocked tokens.

MoonLabsTokenLocker

The MoonLabsTokenLocker contract is designed to allow users to create locks for ERC20 tokens. Lock creators can extend, transfer, add to, and split locks, but cannot unlock tokens prematurely. Users can create lock instances for the same token and choose either a linear or standard lock. The Contract also includes a feature to buy back and burn the native token, as well as a referral code system and whitelist function.

Roles

The contract roles consist of the owner role.

The `owner` role is responsible for:

- Claim all contract deposited balance.
- Configure contract parameters like address, prices, and thresholds.

The user has the authority to:

- Create one or multiple vesting instances for a single token with fees or without fees or with a discount code.
- Transfer vesting instances ownership.
- Get nonces from address.
- Get the address from nonce.
- Get the lock tokens from the address.
- View claimable tokens.
- Withdraw unlocked tokens.
- Change withdrawal address.
- Relock or add tokens to an existing lock with or without fees.
- Divide a lock into multiple locks with or without fees.

MoonLabsLiquidityLocker

The MoonLabsLiquidityLocker contract is responsible for creating liquidity locks for Uniswap-based AMM tokens. The main purpose of the contract is to allow users to create locks for selected wallets with the option to choose between standard or linear lock types. The lock type is determined by the start date, with the default being a standard lock. The locked tokens remain locked until their respective unlock date without any exceptions, and lock owners are not allowed to unlock them prematurely. The Contract also includes a feature to buy back and burn the native token, as well as a referral code system and whitelist function.

Roles

The contract roles consist of the owner role.

The `owner` role is responsible for:

- Claim all contract deposited balance.
- Configure contract parameters like importance, address, prices, and thresholds.

The user has the authority to:

- Create one or multiple vesting instances for a single token with fees or without fees or with a discount code.
- Transfer vesting instances ownership.
- Get nonces from address.
- Get the address from nonce.
- Get the lock tokens from the address.
- View claimable tokens.
- Withdraw unlocked tokens.
- Change withdrawal address.
- Relock or add tokens to an existing lock with or without fees.
- Divide a lock into multiple locks with or without fees.

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	UCF	Unimplemented Contract Function	Unresolved
Critical	MWL	Misleading Whitelisted Logic	Unresolved
Critical	DVI	Discount Value Inconsistency	Unresolved
Critical	UCV	Uninitialized Contract Variables	Unresolved
Minor / Informative	DSO	Data Structure Optimization	Unresolved
Minor / Informative	CO	Code Optimization	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	MSC	Missing Sanity Checks	Unresolved
Minor / Informative	MPV	Missing Percentages Validation	Unresolved
Minor / Informative	PRA	Potential Re-entrance Attack	Unresolved
Minor / Informative	CRO	Code Readability Optimizations	Unresolved
Minor / Informative	TPP	Token Pair Prevalidation	Unresolved

●	AAO	Accumulated Amount Overflow	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

UCF - Unimplemented Contract Function

Criticality	Critical
Location	MoonLabsLiquidityLocker.sol#L672
Status	Unresolved

Description

The contract has an unimplemented function that has been defined in the code. Specifically, the function `getClaimableTokens` has been declared but is missing the necessary code to execute its intended functionality. As a result, users will not be able to claim unlocked tokens.

```
/**
    * @notice Retrieve unlocked tokens for a lock instance
    * @param _nonce ID of desired lock instance
    * @return Number of unlocked tokens
    */
  function getClaimableTokens(uint64 _nonce) public view returns (uint)
{}
```

Recommendation

It is recommended to implement the missing function. Leaving unimplemented code in the application increases the risk of unintended behavior and could potentially create security vulnerabilities.

MWL - Misleading Whitelisted Logic

Criticality	Critical
Location	MoonLabsLiquidityLocker.sol#L376
Status	Unresolved

Description

The contract performs an incorrect if statement about the whitelisted user, which could lead to unintended behavior and potential security vulnerabilities.

The if statement checks if a user is whitelisted. But, as the comment states the contract expects not whitelisted addresses. As a result, whitelisted users will pay fees.

```
/// Check if the token is not whitelisted
if (whitelistContract.getIsWhitelisted(tokenAddress)) {
```

Recommendation

It is recommended to update the if statement to ensure that the correct conditions are checked. For instance,

```
if (!whitelistContract.getIsWhitelisted(tokenAddress)) {..}
```

DVI - Discount Value Inconsistency

Criticality	Critical
Location	MoonLabsWhitelist.sol#L
Status	Unresolved

Description

The contract applies a discount on transfer transactions based on the number of tokens being transferred. However, the require statement for the transfer and the transfer itself is inconsistent and result in an incorrect discount being applied to the transaction.

```
require(usdContract.balanceOf(msg.sender) >= costUSD - (costUSD *
codeDiscount) / 100, "Insignificant balance");
usdContract.transferFrom(msg.sender, address(this), (costUSD *
codeDiscount) / 100);
```

Recommendation

It is recommended to apply the same discount calculation logic in all cases. That means that the discount calculation logic should be consistent in every statement.

UCV - Uninitialized Contract Variables

Criticality	Critical
Location	MoonLabsWhitelist.sol#L58,59,90,93
Status	Unresolved

Description

The contract utilizes a variable in a transaction flow that is not initialized. The uninitialized variable can result in unexpected behavior, including the failure of the transaction or the loss of funds.

The variables `codeDiscount` and `codeCommission` are used to calculate the discount value for the transfer function, but they are not initialized. This can result in incorrect calculations or the transfer of incorrect amounts.

```
uint32 public codeDiscount;
require(usdContract.balanceOf(msg.sender) >= costUSD - (costUSD *
codeDiscount) / 100, "Insignificant balance");
usdContract.transferFrom(msg.sender, address(this), (costUSD *
codeDiscount) / 100);

uint32 public codeCommission;
distributeCommission(code, (costUSD * codeCommission) / 100);
```

Recommendation

It is recommended to initialize or calculate all the variables used in transaction flows. In this case, the variables `codeDiscount` and `codeCommission` should be initialized to a default value or set to the appropriate value before being used in any calculations.

DSO - Data Structure Optimization

Criticality	Minor / Informative
Location	MoonLabsReferral.sol#L48
Status	Unresolved

Description

The contract uses the valuable `reservedCodes` as an array. The business logic of the contract does not require to iterate this structure sequentially. Thus, unnecessary loops are produced that increase the required gas. Additionally, the stored values could be optimized to consume less memory.

```
string[] private reservedCodes;
```

Recommendation

The contract could use a data structure that provides instant access. For instance, a Set or a Map would fit better to the business logic of the contract. This way the time complexity will be reduced from o(n) to o(1).

The stored values could be stored as hashed values directly `keccak256(abi.encodePacked(code))`. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

CO - Code Optimization

Criticality	Minor / Informative
Location	MoonLabsReferral.sol
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

There are code statements that could be optimized and redundant calculations.

The `keccak256(abi.encodePacked(""))` statement is utilized in multiple code segments.

The `depositAmount` and `totalDeposit` are the same. Hence the `MathUpgradeable.mulDiv` calculation is redundant.

```
keccak256(abi.encodePacked(""))

MathUpgradeable.mulDiv(amountSent, depositAmount, totalDeposit)
```

Recommendation

The team is advised to take into consideration these segments and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

- The statement `keccak256(abi.encodePacked(""))` could be defined as constant since it is utilized in multiple code segments.
- Redundant code statements could be removed.

CR - Code Repetition

Criticality	Minor / Informative
Location	MoonLabsTokenLocker.sol#133,173,216,259 MoonLabsVesting.sol#L119,158,200,242 MoonLabsLiquidityLocker.sol#L124,153,185,220,313,355,383,428
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.

```
uint totalDeposit;
for (uint32 i; i < lock.length; i++) {
    totalDeposit += lock[i].depositAmount;
}

uint64 _nonce = nonce;
/// Create a vesting instance for every struct in the lock array
for (uint64 i; i < lock.length; i++) {
    _nonce++;
    createVestingInstance(tokenAddress, lock[i], _nonce, amountSent,
totalDeposit);
}
nonce = _nonce;

require((depositAmount) <=
IERC20Upgradeable(tokenAddress).balanceOf(msg.sender), "Token
balance");

uint previousBal =
IERC20Upgradeable(tokenAddress).balanceOf(address(this));
/// Transfer tokens from sender to contract
transferTokensFrom(tokenAddress, msg.sender, depositAmount);
uint amountSent =
IERC20Upgradeable(tokenAddress).balanceOf(address(this)) - previousBal;
```

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 to reduce the complexity and size of the contract.

- The contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.
- The contract could utilize a safeTransfer library for common functionality that is widely used on the Moonlabs ecosystem.

MSC - Missing Sanity Checks

Criticality	Minor / Informative
Location	MoonLabsVesting.sol#L309,486 MoonLabsTokenLocker.sol#L353,475,527,766 MoonLabsLiquidityLocker.sol#L228
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

The variables `newOwner`, `lock.ownerAddress`, `lock.withdrawAddress` and the `initializer` addresses are not properly sanitized.

```
function initialize(...){...}

function transferVestingOwnership(uint64 _nonce, address newOwner)
external {...}
function createVestingInstance(address tokenAddress, LockParams
calldata lock, uint64 _nonce, uint amountSent, uint totalDeposit)
private {...}

function setLockWithdrawAddress(uint64 _nonce, address _address)
external {...}
function splitLockETH(address to, address withdrawAddress, uint64
_nonce, uint amount) external payable {...}
function splitLockPercent(address to, address withdrawAddress, uint64
_nonce, uint amount) external {...}

function transferLockOwnership(uint64 _nonce, address newOwner)
external {...}
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

- The `initializer` addresses shouldn't be set to zero addresses.

- The `newOwner` address shouldn't be set to zero address.
- The `lock.withdrawAddress` address shouldn't be set to zero address.
- The `lock.ownerAddress` address shouldn't be set to zero address.

MPV - Missing Percentages Validation

Criticality	Minor / Informative
Location	MoonLabsVesting.sol#L388,396 MoonLabsTokenLocker.sol#L641,649 MoonLabsLiquidityLocker.sol#L560,568
Status	Unresolved

Description

The contract is processing percentage variables that have not been properly sanitized and checked that they form the proper shape. These percentages may produce vulnerability issues.

The percentage `codeDiscount` and `codeCommission` are not properly sanitized.

```
function setCodeDiscount(uint8 _codeDiscount) external onlyOwner {
    codeDiscount = _codeDiscount;
}

function setCodeCommission(uint8 _codeCommission) external onlyOwner {
    codeCommission = _codeCommission;
}
```

Recommendation

The team is advised to properly check the percentages according to the required specifications.

- The `codeCommission` percentage should not be set to values greater than 100.
- The `codeDiscount` percentage should not be set to values greater than 100.

PRA - Potential Re-entrance Attack

Criticality	Minor / Informative
Location	MoonLabsTokenLocker.sol#L827 MoonLabsVesting.sol#L556 MoonLabsLiquidityLocker.sol#L717
Status	Unresolved

Description

The contract contains a function that can be vulnerable to reentrancy attacks. Specifically, the `distributeCommission` function allows a user to withdraw commissions from the contract balance, but the function does not prevent reentrancy attacks.

An attacker could potentially call the `createLockWithCodeEth` function multiple times that internally calls `distributeCommission` before the function completes, causing the contract to transfer `commission` repeatedly.

```
function distributeCommission(string memory code, uint commission)
private {
    /// Get referral code owner
    address payable to =
payable(referralContract.getAddressByCode(code));
    /// Send ether to code owner
    (bool sent, ) = to.call{ value: commission }("");
    require(sent, "Failed to send Ether");
    /// Log rewards in the referral contract
    referralContract.addRewardsEarned(code, commission);
}
```

Recommendation

To prevent reentrancy attacks, the contract should use a lock solution such as the Mutex or ReentrancyGuard pattern. These patterns use a lock variable to prevent multiple calls to the function while it is still being executed.

CRO - Code Readability Optimizations

Criticality	Minor / Informative
Location	MoonLabsLiquidityLocker.sol MoonLabsReferral.sol MoonLabsVesting.sol MoonLabsTokenLocker.sol
Status	Unresolved

Description

The code in the contract is difficult to read and understand. The code repetition and the long code segments make it hard to follow the logic and understand the purpose of the code. This could increase the risk of introducing errors during maintenance or modification of the code.

```
require(referralContract.checkIfActive(code), "Invalid code");

require(msg.value == (ethLockPrice * lock.length - (((ethLockPrice *
codeDiscount) / 100) * lock.length)), "Incorrect price");
```

Recommendation

To improve code readability and reduce the risk of errors, the contract should include modifiers for common code segments and break long code segments into smaller segments or variables to make them more manageable. This will help other developers to better understand the code and maintain it more easily.

TPP - Token Pair Prevalidation

Criticality	Minor / Informative
Location	MoonLabsLiquidityLocker.sol#L698 MoonLabsVesting.sol#L569 MoonLabsTokenLocker.sol#L807
Status	Unresolved

Description

The contract allows users to initiate swap transactions between two tokens without first checking if a token pair exists. This could result in the loss of funds if the contract is unable to find a liquidity pool for the specified token pair.

```
function handleBurns() private {
    /// Check if the threshold is met
    uint _burnMeter = burnMeter;
    if (burnMeter >= burnThreshold) {
      /// Buy tokenToBurn via Uniswap router and send to the dead
address
      address[] memory path = new address[](2);
      path[0] = routerContract.WETH();
      path[1] = address(tokenToBurn);

routerContract.swapExactETHForTokensSupportingFeeOnTransferTokens{
value: _burnMeter }(0, path,
0x000000000000000000000000000000000000dEaD, block.timestamp);
      _burnMeter = 0;
      burnMeter = _burnMeter;
    }
}
```

Recommendation

It is recommended to pre-validate that a token pair exists before allowing users to initiate swap transactions. A valid pair address should have token0, token1, factory

AAO - Accumulated Amount Overflow

Criticality	Minor / Informative
Location	MoonLabsLiquidityLocker.sol MoonLabsTokenLocker.sol MoonLabsVesting.sol
Status	Unresolved

Description

The contract is using the `noonce` variable to accumulate values. The contract could lead to an overflow when the total value of a variable exceeds the maximum value that can be stored in that variable's data type. This can happen when an accumulated value is updated repeatedly over time, and the value grows beyond the maximum value that can be represented by the data type.

```
uint64 public nonce; /// Unique lock identifier
```

Recommendation

The team is advised to carefully investigate the usage of the variables that accumulate value. A suggestion is to add checks to the code to ensure that the value of a variable does not exceed the maximum value that can be stored in its data type.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	MoonLabsWhitelist.sol#L149 MoonLabsLiquidityLocker.sol#L691 MoonLabsTokenLocker.sol#L801 MoonLabsVesting.sol#L520
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
IERC20Upgradeable(tokenAddress).transfer(to, amount);

usdContract.transfer(referralContract.getAddressByCode(code),
commission);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	MoonLabsWhitelist.sol#L35,38,39
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.

```
uint public nonce
uint32 public codeDiscount
uint32 public codeCommission
```

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	MoonLabsWhitelist.sol#L51,65,82,91,116 MoonLabsVesting.sol#L32,33,345,374,413,421,429,437,445,453,461,469,477,486,519,537,568,627,693 MoonLabsTokenLocker.sol#L32,33,375,402,430,461,520,587,656,732,740,748,756,764,772,780,788,796,804,812,821,830,839,883,906,936,1032,1077 MoonLabsReferral.sol#L173,181,247,334 MoonLabsLiquidityLocker.sol#L31,32,295,321,353,355,410,412,471,535,605,613,621,629,637,645,653,661,669,677,685,694,703,712,745
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 _tokenToBurn
address _feeCollector
uint64 _nonce
address _routerAddress
address _referralAddress
uint _burnThreshold
uint _ethLockPrice
uint8 _codeDiscount
uint8 _codeCommission
uint8 _burnPercent
uint8 _percentLockPrice
uint _ethSplitPrice
uint _ethRelockPrice

...
```

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	MoonLabsVesting.sol#L438,479 MoonLabsTokenLocker.sol#L757,814 MoonLabsLiquidityLocker.sol#L630
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.

```
burnThreshold = _burnThreshold
burnPercent = _burnPercent
```

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.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	MoonLabsVesting.sol#L487 MoonLabsTokenLocker.sol#L822,831,840 MoonLabsLiquidityLocker.sol#L695,704,713
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(_percentLockPrice <= 10000, "Max percent")
require(_percentSplitPrice <= 10000, "Max percent")
require(_percentRelockPrice <= 10000, "Max percent")
```

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.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	MoonLabsReferral.sol#L49,57,91,136,154,282
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require(checkIfActive(code) == false, "Code in use")
require(checkIfReserved(_code) == false, "Code reserved")
require(checkIfActive(code) == true, "Code not in use")
require(checkIfReserved(_code) == false, "Code is reserved")
require(checkIfReserved(_code) == true, "Code not reserved")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	MoonLabsVesting.sol#L332 MoonLabsTokenLocker.sol#L362
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.

```
distributeCommission(
      code,
      (((ethLockPrice * codeCommission) / 100) * lock.length)
    )

distributeCommission(
      code,
      (((_ethLockPrice * codeCommission) / 100) * lock.length)
    )
```

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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	MoonLabsVesting.sol#L121,139,171,197,234,251,298,315,382,630,642,699 MoonLabsTokenLocker.sol#L147,165,198,224,262,279,328,345,408,436,1037,1049,1061,1083
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint32 i
uint64 i
uint64 timeElapsed
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	MoonLabsWhitelist.sol#L29 MoonLabsVesting.sol#L40,414 MoonLabsTokenLocker.sol#L40,733 MoonLabsLiquidityLocker.sol#L39,606
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.

```
usdAddress = _usdAddress
feeCollector = _feeCollector
```

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	MoonLabsWhitelist.sol#L8 MoonLabsVesting.sol#L10 MoonLabsTokenLocker.sol#L10 MoonLabsReferral.sol#L11 MoonLabsLiquidityLocker.sol#L9
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 ^0.8.17;
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	MoonLabsWhitelist.sol#L55,71,108,129 MoonLabsVesting.sol#L606,620 MoonLabsTokenLocker.sol#L976,990 MoonLabsLiquidityLocker.sol#L810,824
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
usdContract.transferFrom(msg.sender, address(this), costUSD)
usdContract.transferFrom(msg.sender, address(this), (costUSD *
codeDiscount) / 100)
usdContract.transferFrom(address(this), msg.sender,
usdContract.balanceOf(address(this)))
usdContract.transfer(referralContract.getAddressByCode(code),
commission)
IERC20Upgradeable(tokenAddress).transferFrom(from, address(this),
amount)
IERC20Upgradeable(tokenAddress).transfer(to, amount)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IMoonLabsReferral	Interface			
	checkIfActive	External		-
	getAddressByCode	External		-
	addRewardsEarned	External	✓	-
IMoonLabsWhitelist	Interface			
	getIsWhitelisted	External		-
MoonLabsLiquidityLocker	Implementation	OwnableUpgradeable		
	initialize	Public	✓	initializer
	createLockWhitelist	External	✓	-
	createLockPercent	External	✓	-
	createLockEth	External	Payable	-
	createLockWithCodeEth	External	Payable	-
	withdrawUnlockedTokens	External	✓	-
	transferLockOwnership	External	✓	-
	relockETH	External	Payable	-
	relockPercent	External	✓	-
	splitLockETH	External	Payable	-
	splitLockPercent	External	✓	-
	claimETH	External	✓	onlyOwner
	setFeeCollector	External	✓	onlyOwner
	setRouter	External	✓	onlyOwner

	setReferralContract	External	✓	onlyOwner
	setBurnThreshold	External	✓	onlyOwner
	setLockPrice	External	✓	onlyOwner
	setSplitPrice	External	✓	onlyOwner
	setRelockPrice	External	✓	onlyOwner
	setCodeDiscount	External	✓	onlyOwner
	setCodeCommission	External	✓	onlyOwner
	setTokenToBurn	External	✓	onlyOwner
	setBurnPercent	External	✓	onlyOwner
	setPercentLockPrice	External	✓	onlyOwner
	setPercentSplitPrice	External	✓	onlyOwner
	setPercentRelockPrice	External	✓	onlyOwner
	getNonceFromOwnerAddress	External		-
	getNonceFromTokenAddress	External		-
	getLock	External		-
	createLockInstance	Private	✓	
	getClaimableTokens	Public		-
	transferTokensFrom	Private	✓	
	transferTokensTo	Private	✓	
	handleBurns	Private	✓	
	distributeCommission	Private	✓	
	deleteLockInstance	Private	✓	
IMoonLabsReferral	Interface			
	checkIfActive	External		-
	getCodeByAddress	External		-
	getAddressByCode	External		-
	addRewardsEarned	External	✓	-
	addRewardsEarnedUSD	External	✓	-

MoonLabsReferral	Implementation	IMoonLabsReferral, Ownable		
	createCode	External	✓	-
	deleteCode	External	✓	-
	deleteCodeOwner	External	✓	onlyOwner
	setCodeAddress	External	✓	-
	addReservedCodes	External	✓	onlyOwner
	assignReservedCode	External	✓	onlyOwner
	addMoonLabsContract	External	✓	onlyOwner
	removeMoonLabsContract	External	✓	onlyOwner
	addRewardsEarned	External	✓	-
	addRewardsEarnedUSD	External	✓	-
	getRewardsEarned	External		-
	getRewardsEarnedUSD	External		-
	getCodeByAddress	External		-
	getAddressByCode	External		-
	claimETH	External	✓	onlyOwner
	removeReservedCode	Public	✓	onlyOwner
	checkIfActive	Public		-
	checkIfReserved	Public		-
	upper	Private		
	_upper	Private		
IMoonLabsReferral	Interface			
	checkIfActive	External		-
	getAddressByCode	External		-
	addRewardsEarned	External	✓	-

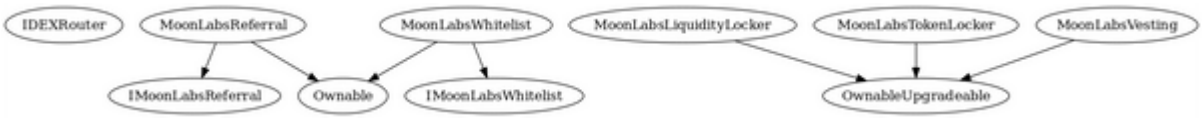
IMoonLabsWhitelist	Interface			
	getIsWhitelisted	External		-
MoonLabsTokenLocker	Implementation	OwnableUpgradeable		
	initialize	Public	✓	initializer
	createLockWhitelist	External	✓	-
	createLockPercent	External	✓	-
	createLockEth	External	Payable	-
	createLockWithCodeEth	External	Payable	-
	withdrawUnlockedTokens	External	✓	-
	transferLockOwnership	External	✓	-
	setLockWithdrawAddress	External	✓	-
	relockETH	External	Payable	-
	relockPercent	External	✓	-
	splitLockETH	External	Payable	-
	splitLockPercent	External	✓	-
	claimETH	External	✓	onlyOwner
	setFeeCollector	External	✓	onlyOwner
	setRouter	External	✓	onlyOwner
	setReferralContract	External	✓	onlyOwner
	setBurnThreshold	External	✓	onlyOwner
	setLockPrice	External	✓	onlyOwner
	setSplitPrice	External	✓	onlyOwner
	setRelockPrice	External	✓	onlyOwner
	setCodeDiscount	External	✓	onlyOwner
	setCodeCommission	External	✓	onlyOwner
	setTokenToBurn	External	✓	onlyOwner
	setBurnPercent	External	✓	onlyOwner
	setPercentLockPrice	External	✓	onlyOwner

	setPercentSplitPrice	External	✓	onlyOwner
	setPercentRelockPrice	External	✓	onlyOwner
	getNonceFromOwnerAddress	External		-
	getNonceFromWithdrawAddress	External		-
	getNonceFromTokenAddress	External		-
	getLock	External		-
	getClaimableTokens	Public		-
	createLockInstance	Private	✓	
	transferTokensFrom	Private	✓	
	transferTokensTo	Private	✓	
	handleBurns	Private	✓	
	distributeCommission	Private	✓	
	deleteLockInstance	Private	✓	
	calculateLinearWithdraw	Private		
IMoonLabsReferral	Interface			
	checkIfActive	External		-
	getAddressByCode	External		-
	addRewardsEarned	External	✓	-
IMoonLabsWhitelist	Interface			
	getIsWhitelisted	External		-
MoonLabsVesting	Implementation	OwnableUpgradeable		
	initialize	Public	✓	initializer
	createLockWhitelist	External	✓	-
	createLockPercent	External	✓	-
	createLockEth	External	Payable	-

	createLockWithCodeEth	External	Payable	-
	withdrawUnlockedTokens	External	✓	-
	transferVestingOwnership	External	✓	-
	claimETH	External	✓	onlyOwner
	setFeeCollector	External	✓	onlyOwner
	setRouter	External	✓	onlyOwner
	setReferralContract	External	✓	onlyOwner
	setBurnThreshold	External	✓	onlyOwner
	setLockPrice	External	✓	onlyOwner
	setCodeDiscount	External	✓	onlyOwner
	setCodeCommission	External	✓	onlyOwner
	setTokenToBurn	External	✓	onlyOwner
	setBurnPercent	External	✓	onlyOwner
	setPercentLockPrice	External	✓	onlyOwner
	getNonceFromWithdrawAddress	External		-
	getNonceFromTokenAddress	External		-
	getInstance	External		-
	getClaimableTokens	Public		-
	createVestingInstance	Private	✓	
	transferTokensFrom	Private	✓	
	transferTokensTo	Private	✓	
	deleteVestingInstance	Private	✓	
	distributeCommission	Private	✓	
	handleBurns	Private	✓	
	calculateLinearWithdraw	Private		
IMoonLabsReferral	Interface			
	checkIfActive	External		-
	getAddressByCode	External		-

	addRewardsEarnedUSD	External	✓	-
IMoonLabsWhitelist	Interface			
	getIsWhitelisted	External		-
MoonLabsWhitelist	Implementation	IMoonLabsWhitelist, Ownable		
		Public	✓	-
	purchaseWhitelist	External	✓	-
	purchaseWhitelistWithCode	External	✓	-
	ownerWhitelistAdd	External	✓	onlyOwner
	ownerWhitelistRemove	External	✓	onlyOwner
	claimETH	External	✓	onlyOwner
	claimUSD	External	✓	onlyOwner
	getIsWhitelisted	Public		-
	distributeCommission	Private	✓	

Inheritance Graph



Flow Graph

Summary

Moonlabs contract implements a utility, financial, and locker 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