Cyberscope

Audit Report

XMY

Aug 2023

Network	ARBITRUM
Address	0xC7ca0aBfeFBAC7128d0533312532F6Ad8E07ded3
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://arbiscan.io/address/0xc7ca0abfefbac7128d0533312532f6ad8e07ded3

Audit Updates

Initial Audit	29 Jul 2023

Source Files

Filename	SHA256
XMY.sol	3d3569e63786da8e5f3fd4c338182a5a2fba84fa313124c6530b6ba2de3c0616

Overview

The XMY is a token contract that incorporates a presale functionality, among other features. The XMY token follows the ERC20 standard, which means it provides basic functionalities such as transferring tokens, querying balances, and managing allowances. The contract defines the token's name, symbol, and decimals, ensuring compatibility with various wallets and exchanges.

TokenDistributor Functionality

This contract is designed to distribute tokens. It includes the function `claimToken` which allows the owner to claim a specified amount of a token and transfer it to a designated address.

Presale Mechanism

Participation

The `presale` function allows users to participate in the presale by sending value to the contract. Participants are able to specify an invitor (referral) and the number of pieces they want to buy. The contract ensures that the presale hasn't ended and that there are sufficient pieces available for purchase. If a participant hasn't previously associated with an invitor, the contract allows them to bind one during this transaction. Once this binding is successful, the contract records the participant's purchase amount and processes potential rewards for the invitor. The participant is then added to a list of presale participants, which the contract subsequently sorts. If for any reason the binding process doesn't succeed, the contract ensures the sent Ether is returned to the participant.

Rewards

If the participant is successfully bound to an invitor, they are rewarded with tokens based on the number of pieces they bought. The `processPreSaleReward` function calculates rewards for the invitor based on the total pieces bought by their referrals. For every 5 pieces bought by their referrals, the invitor gets a reward.

Sorting Participants

After a successful purchase, the list of presale participants (`preList`) is sorted based on the number of pieces they bought using the sortPreList function. This function uses an insertion sort algorithm, placing participants who bought more pieces earlier in the list, in order to use this list in the Ending Presale functionality.

Ending Presale

The `endPresale` function can be called by the contract owner to mark the end of the presale. It records the end time, calculates the total tokens sold during the presale, and identifies the top 30 participants. Any unsold tokens are burned (sent to the zero address).

Liquidity Provider Management

Adding Liquidity Providers

The `_addLpProvider` function adds an address to the list of liquidity providers (lpProviders).

Checking Referrals and Liquidity

The `checkLowerCount` function calculates the number of an account's referrals who hold at least a certain amount of tokens and the number of those referrals who provide at least a certain amount of liquidity.

Determining Mining Level

The `checkMineLv` function determines the mining level of an account based on the account's liquidity provision and the activity of their referrals.

Mining Rewards for Liquidity Providers

Processing Mining Rewards

The `processMineLP` function processes mining rewards for liquidity providers. It calculates rewards based on the liquidity they provide and distributes rewards to them and their inviters.

Updating Mining Cycle

The `updateMineCycle` function updates the mining cycle, determining the amount of rewards to be distributed in the current cycle.

Distributing Invitor Rewards

The `procesInvitorReward` function calculates and distributes rewards to inviters based on the mining rewards of their invitees. It has a tiered reward system based on the inviter's level. The `procesUpInvitorReward` function further distributes rewards up the inviter chain, ensuring that higher-level inviters also get a share of the rewards.

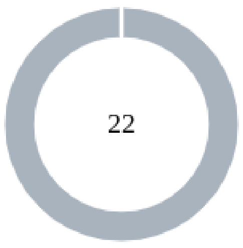
Claiming Mine Rewards

The `getMineReward` function allows users to claim their mining rewards. A portion of the reward is sent to a technical fund.

Claiming Invitor Rewards

The `getInvitorReward` function allows users to claim their inviter rewards. A portion of the reward is burned.

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	22

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	22	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CSAO	Contract State Access Optimization	Unresolved
●	UACA	Unsafe Address Comparison Assumption	Unresolved
●	ZD	Zero Division	Unresolved
●	VO	Variable Optimization	Unresolved
●	MRPC	Missing Reward Prevalidation Check	Unresolved
●	FSA	Fixed Swap Address	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	MRM	Missing Revert Messages	Unresolved
●	TUU	Time Units Usage	Unresolved
●	CR	Code Repetition	Unresolved
●	IDI	Immutable Declaration Improvement	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
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

CSAO - Contract State Access Optimization

Criticality	Minor / Informative
Location	XMY.sol#L671
Status	Unresolved

Description

The contract is utilizing a loop to calculate the `presaleSold` variable by incrementing it with the `totalPresaleAmount` in each iteration. This approach leads to repeated state access and modification of the `presaleSold` variable within the loop, which can be inefficient, especially when dealing with a large number of iterations.

```
for (uint256 i = 0; i <preList.length;i++) {
            uint256 totalPresaleAmount = presaleAmount[preList[i]] +
presaleReward[preList[i]];
            leftPresale[preList[i]] = totalPresaleAmount;
            presaleSold += totalPresaleAmount;
```

Recommendation

It is recommended to optimize the contract by calculating the value of `totalPresaleAmount` within the loop but updating the `presaleSold` variable only once at the end of the loop. This can be achieved by using a temporary variable to accumulate the total presale amount and then assigning it to `presaleSold` after the loop. By making this change, the contract can reduce the number of state accesses, potentially leading to lower gas costs and more efficient execution.

UACA - Unsafe Address Comparison Assumption

Criticality	Minor / Informative
Location	XMY.sol#L382,398
Status	Unresolved

Description

The contract is utilizing the `_isAddLiquidity` and `_isRemoveLiquidity` functions to determine whether to add or remove liquidity. These functions rely on comparing the `tokenOther` address with the contract's own address to decide which reserve value to use (`r0` or `r1`). This approach is unsafe, as it makes an assumption about the ordering of addresses that may not hold true in all cases. Basing the decision to add or remove liquidity on this comparison can lead to incorrect behavior, potentially resulting in unintended liquidity adjustments.

```
   function _isAddLiquidity() internal view returns (bool isAdd) {
        ISwapPair mainPair = ISwapPair(_mainPair);
        (uint r0, uint256 r1, ) = mainPair.getReserves();

        address tokenOther = weth;
        uint256 r;
        if (tokenOther < address(this)) {
            r = r0;
        } else {
            r = r1;
        }

        uint bal = IERC20(tokenOther).balanceOf(address(mainPair));
        isAdd = bal > r;
    }

    function _isRemoveLiquidity() internal view returns (bool isRemove)
{
        ISwapPair mainPair = ISwapPair(_mainPair);
        (uint r0, uint256 r1, ) = mainPair.getReserves();

        address tokenOther = weth;
        uint256 r;
        if (tokenOther < address(this)) {
            r = r0;
        } else {
            r = r1;
        }

        uint bal = IERC20(tokenOther).balanceOf(address(mainPair));
        isRemove = r >= bal;
    }
```

Recommendation

It is recommended to replace the address comparison logic with a more robust mechanism that accurately reflects the intended Add or Remove Liquidity functionality of the contract.

ZD - Zero Division

Criticality	Minor / Informative
Location	XMY.sol#L479
Status	Unresolved

Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

Specifically, the expression `balanceOf(_mainPair) - 1` in the calculation of the `liquidity` variable can result in a zero denominator if the balance of `_mainPair` less than 1.

```
uint256 liquidity = (amount * ISwapPair(_mainPair).totalSupply() + 1) /
(balanceOf(_mainPair) - 1);
```

Recommendation

It is recommended to add checks before performing the division to ensure that the denominator is never zero. It is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero, or should not allow the execution of the corresponding statements.

VO - Variable Optimization

Criticality	Minor / Informative
Location	XMY.sol#L173
Status	Unresolved

Description

The contract is utilizing `uint256` for various state variables, which in some cases may be over-allocating storage space. While `uint256` is the most gas-efficient for arithmetic operations due to EVM optimization, it's not always the most storage-efficient choice. Specifically, the variables `_decimals` , `mineRate` , `preSaleRate` , `_buyLPFee` , `sell_burnFee` , `addLiquidityFee` , `removeLiquidityFee` , and `_progressMineLPBlockDebt` are all set to uint256 but their values suggest they could fit into smaller data types, such as `uint8` or `uint16` .

```
uint256 private _decimals = 18;
uint256 public mineRate = 60;
uint256 public preSaleRate = 20;
uint256 public _buyLPFee = 200;
uint256 public sell_burnFee = 800;
uint256 public addLiquidityFee = 250;
uint256 public removeLiquidityFee = 250;
uint256 public _progressMineLPBlockDebt = 50;
```

Recommendation

It is recommended to refine the data types of the aforementioned variables to more appropriately sized types. For instance:

- `_decimals` , `mineRate` , `preSaleRate` , `_buyLPFee` , `sell_burnFee` , `addLiquidityFee` , `removeLiquidityFee` and `_progressMineLPBlockDebt` can be set to `uint8` as their values are within the range of 0 to 255.

By adjusting these variables to their appropriate types, the contract can achieve better storage efficiency and potentially reduce gas costs for operations involving these variables.

MRPC - Missing Reward Prevalidation Check

Criticality	Minor / Informative
Location	XMY.sol#L1021,1066
Status	Unresolved

Description

The contract attempts to refund a user without verifying if the contract has enough balance to refund the user or if. The functions `getMineReward` and `getInvitorReward` in the contract are designed to distribute `mining` and `invitor` rewards to users. These functions calculate the amount of reward a user is entitled to and then attempt to transfer this amount using the `TokenDistributor(sender).claimToken` method. However, there's a lack of prevalidation checks to ensure that the contract has a sufficient balance of tokens to distribute these rewards. Without these checks, there's a risk that users might not receive their expected rewards, leading to potential disputes or loss of trust in the contract's operations.

```
   function getMineReward()external{
        uint256 totalMineReward = mineReward[msg.sender];
        require(totalMineReward > 0);
        address sender = address(mineRewardDistributor);
        uint256 techAmount = totalMineReward * 3/100;
        mineReward[msg.sender] = 0;
        TokenDistributor(sender).claimToken(address(this),
fundAddress, techAmount);
        TokenDistributor(sender).claimToken(address(this),
msg.sender, totalMineReward - techAmount);
    }

    function getInvitorReward()external{
        ...
        if(availableInvitorReward >0){
            uint256 techAmount = availableInvitorReward * 10 /
100;
            invitorReward[msg.sender] = 0;
            TokenDistributor(sender).claimToken(address(this),
deadAddress, techAmount);
            TokenDistributor(sender).claimToken(address(this),
msg.sender, availableInvitorReward - techAmount);
        }
    }
```

Recommendation

It is recommended to add pre-validation checks with require statements when attempting to transfer rewards to users. The checks should verify that the contract has enough balance to reward the user. If this condition not met, the contract should revert with a clear error message indicating the reason for the revert.

FSA - Fixed Swap Address

Criticality	Minor / Informative
Location	XMY.sol#L263
Status	Unresolved

Description

The swap address is assigned once and it can not be changed. It is a common practice in decentralized exchanges to create new swap versions. A contract that cannot change the swap address may not be able to catch up to the upgrade. As a result, the contract will not be able to migrate to a new liquidity pool pair or decentralized exchange.

```
consttuctor
        {
        ...
        _swapRouter = ISwapRouter(routerAddress);
        IERC20(weth).approve(address(_swapRouter), MAX);

        _allowances[address(this)][address(_swapRouter)] = MAX;

        ISwapFactory swapFactory =
ISwapFactory(_swapRouter.factory());
        _mainPair = swapFactory.createPair(address(this),
weth);
        ...
        }
```

Recommendation

The team is advised to add the ability to change the pair and router address in order to cover potential liquidity pool migrations. It would be better to support multiple pair addresses so the token will be able to have the same behavior in all the decentralized liquidity pairs.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	XMY.sol#L717,724,764,1015,1027,1074
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 setFundAddress(address addr) external onlyOwner {
        fundAddress = addr;
        _feeWhiteList[addr] = true;
        _addLpProvider(addr);
    }

    function setFeeWhiteList(
        address[] calldata addr,
        bool enable
    ) public onlyOwner {
    ...
    }

    function _addLpProvider(address adr) private {
        ...
    }

    function getInvitorReward()external{
    ...
    }

    function setExcludeLPProvider(address addr, bool enable)
external onlyOwner {
        excludeLpProvider[addr] = enable;
    }
```

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.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	XMY.sol#L745,754
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The `fundAddress` owner may take advantage of it by calling the `claimTokens` and `claimContractToken` functions.

```
  function claimToken(
        address token,
        uint256 amount,
        address to
    ) external  {
        require(fundAddress == msg.sender);
        IERC20(token).transfer(to, amount);
    }

    function claimContractToken(address contractAddress,
address token, uint256 amount) external {
        require(fundAddress == msg.sender);
        TokenDistributor(contractAddress).claimToken(token,
fundAddress, amount);
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions. Some suggestions are:

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.
- Renouncing the ownership will eliminate the threats but it is non-reversible.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	XMY.sol#L737,1074
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.

The contract updates the router address even if its current state is the same as the one passed as an argument. As a result, the contract performs redundant storage writes.

```
    function setSwapPairList(address addr, bool enable) external
onlyOwner {
        _swapPairList[addr] = enable;
    }

    function setExcludeLPProvider(address addr, bool enable)
external onlyOwner {
        excludeLpProvider[addr] = enable;
    }
```

Recommendation

The team is advised to take these segments into consideration 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.

MRM - Missing Revert Messages

Criticality	Minor / Informative
Location	XMY.sol#L415,456,458,483,692,750,755
Status	Unresolved

Description

The contract is missing error messages. These missing error messages are 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.

```
require(balanceOf(from) >= amount);
require(block.timestamp > startPreTradeTime);
require(block.timestamp > startTradeTime);
require(_userLPAmount[to] >= liquidity);
require(leftAmount > 0 && block.timestamp > endPresaleTime );
require(fundAddress == msg.sender);
require(fundAddress == msg.sender);
```

Recommendation

The team is advised to carefully review the source code in order to address these issues. To accelerate the debugging process and mitigate these issues, the team should use more specific and descriptive error messages.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	XMY.sol#L685
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
startPreTradeTime = launchTime - 3600;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

CR - Code Repetition

Criticality	Minor / Informative
Location	XMY.sol#L382,398
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 functions `_isAddLiquidity` and `_isRemoveLiquidity` share similar code segments.

```
    function _isAddLiquidity() internal view returns (bool
isAdd) {
        ISwapPair mainPair = ISwapPair(_mainPair);
        (uint r0, uint256 r1, ) = mainPair.getReserves();

        address tokenOther = weth;
        uint256 r;
        if (tokenOther < address(this)) {
            r = r0;
        } else {
            r = r1;
        }

        uint bal =
IERC20(tokenOther).balanceOf(address(mainPair));
        isAdd = bal > r;
    }

    function _isRemoveLiquidity() internal view returns (bool
isRemove) {
        ...
        isRemove = r >= bal;
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	XMY.sol#L263,269,273
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
_swapRouter
_mainPair
mineRewardDistributor
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	XMY.sol#L167,168,169,175,176,177,179,180,188,190,192,193,200,241,243,247
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.

```
string private _name = "XMY"
string private _symbol = "XMY"
uint256 private _decimals = 18
uint256 public mineRate = 60
uint256 public preSaleRate = 20
address public routerAddress =
address(0x1b02dA8Cb0d097eB8D57A175b88c7D8b47997506)
address public weth =
address(0x82aF49447D8a07e3bd95BD0d56f35241523fBab1)
address public deadAddress =
address(0x000000000000000000000000000000000000dEaD)
uint256 public _buyLPFee = 200
uint256 public sell_burnFee = 800
uint256 public addLiquidityFee = 250
uint256 public removeLiquidityFee = 250
uint256 public gapTime = 180 days
uint256 public _progressMineLPBlockDebt = 50

...
```

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	XMY.sol#L48,103,138,172,178,181,188,190,195,196,197,214,215,216,239,240,241,242,243,255,257
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.

```
function WETH() external pure returns (address);
address internal _owner
address public _owner
mapping(address => bool) public _feeWhiteList
ISwapRouter public _swapRouter
mapping(address => bool) public _swapPairList
uint256 public _buyLPFee = 200
uint256 public sell_burnFee = 800
mapping(address => address) public _inviter
mapping(address => address[]) public _binders
mapping(address => mapping(address => bool)) public
_maybeInvitor
mapping(address => uint256) public _userLPAmount
address public _lastMaybeAddLPAddress
uint256 public _lastMaybeAddLPAmount

...
```

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	XMY.sol#L684
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.

```
   function setLaunchTime(uint256 launchTime) external onlyOwner {
        require(0 == startTradeTime && endPresaleTime !=0);
        startTradeTime = launchTime;
        startPreTradeTime = launchTime - 3600;
    }
```

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	XMY.sol#L594
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(totalPieces - piece >=0)
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	XMY.sol#L651,653,890,891
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 rewardPieces = sumPieces / 5;
cycleAmount = _balances[address(mineRewardDistributor)] * 3 / 10;
cycleMineAmount = cycleAmount * 6 /10;
```

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	XMY.sol#L438,439,440,441,522,779,905,974,1029
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.

```
bool takeFee
bool isSell
bool isRemove
bool isAdd
uint256 feeAmount
uint256 i
uint256 availableInvitorReward;
```

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	XMY.sol#L718
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.

```
fundAddress = addr;
```

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.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	XMY.sol#L573,768
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {size := extcodesize(invitor)}
            if (size > 0) {
                return false ;
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	XMY.sol#L144,751
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.

```
IERC20(token).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**
ArbSys	Interface			
	arbBlockNumber	External		-
IERC20	Interface			
	decimals	External		-
	symbol	External		-
	name	External		-
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
ISwapRouter	Interface			
	factory	External		-
	WETH	External		-
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External	✓	-

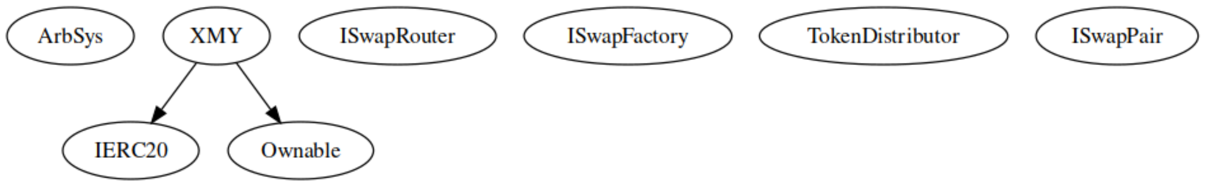
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
ISwapFactory	Interface			
	createPair	External	✓	-
	getPair	External		-
Ownable	Implementation			
		Public	✓	-
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
TokenDistributor	Implementation			
		Public	✓	-
	claimToken	External	✓	-
ISwapPair	Interface			
	getReserves	External		-
	token0	External		-
	balanceOf	External		-
	totalSupply	External		-

XMY	Implementation	IERC20, Ownable		
		Public	✓	-
	symbol	External		-
	name	External		-
	decimals	External		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	_approve	Private	✓	
	_basicTransfer	Internal	✓	
	_isAddLiquidity	Internal		
	_isRemoveLiquidity	Internal		
	_transfer	Private	✓	
	_tokenTransfer	Private	✓	
	_takeTransfer	Private	✓	
	_bindInvitor	Private	✓	
	getBinderLength	External		-
	presale	External	Payable	-
	sortPreList	Private	✓	

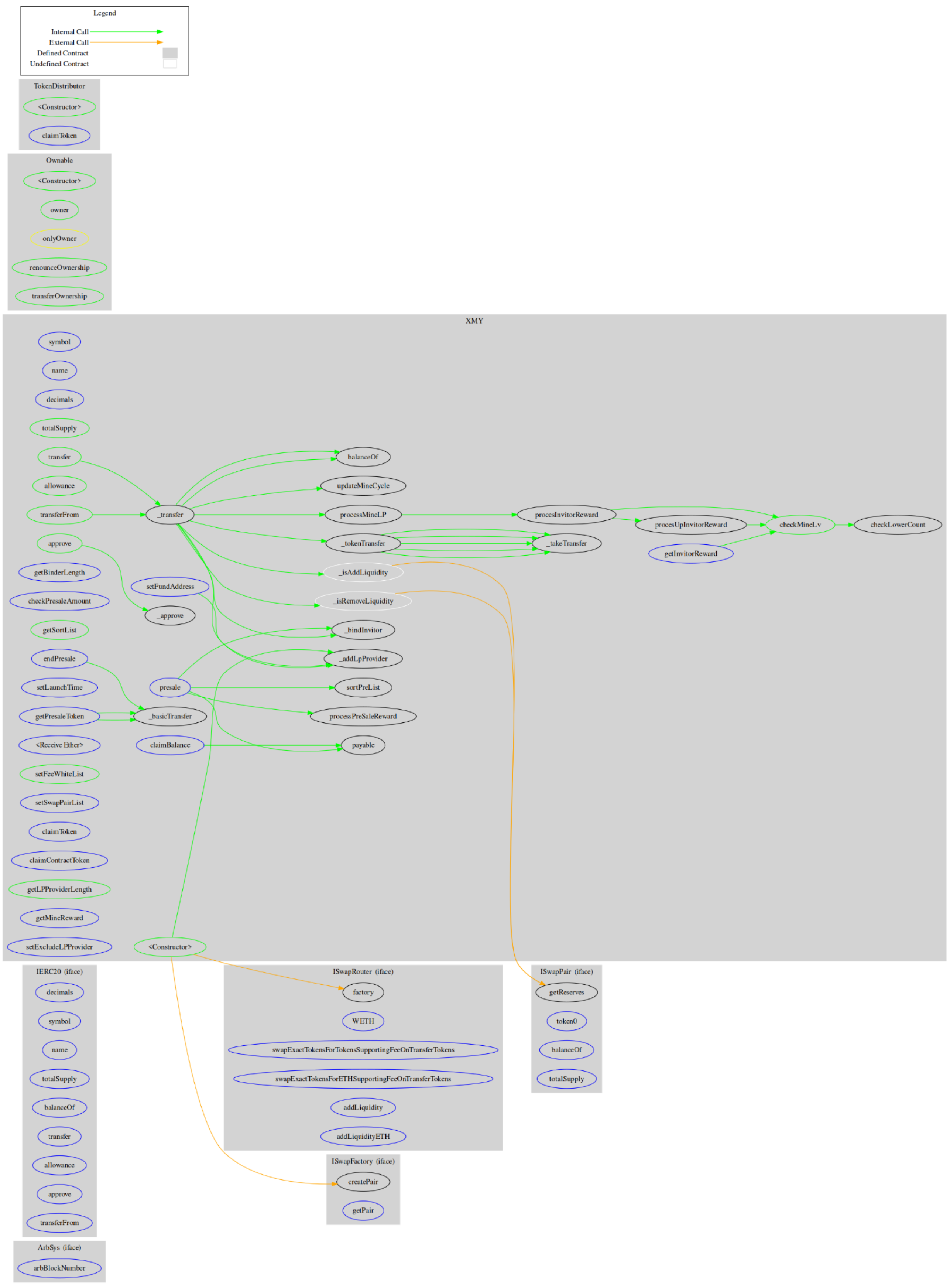
	processPreSaleReward	Internal	✓	
	checkPresaleAmount	External		-
	getSortList	Public		-
	endPresale	External	✓	onlyOwner
	setLaunchTime	External	✓	onlyOwner
	getPresaleToken	External	✓	-
		External	Payable	-
	setFundAddress	External	✓	onlyOwner
	setFeeWhiteList	Public	✓	onlyOwner
	setSwapPairList	External	✓	onlyOwner
	claimBalance	External	✓	onlyOwner
	claimToken	External	✓	-
	claimContractToken	External	✓	-
	getLPProviderLength	Public		-
	_addLpProvider	Private	✓	
	checkLowerCount	Private		
	checkMineLv	Public		-
	processMineLP	Private	✓	
	updateMineCycle	Private	✓	
	procesInvitorReward	Private	✓	
	procesUpInvitorReward	Private	✓	
	getMineReward	External	✓	-
	getInvitorReward	External	✓	-

	setExcludeLPProvider	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
TokenDistributor
<Constructor>
claimToken
Ownable
<Constructor>
owner
onlyOwner
renounceOwnership
transferOwnership
XMY
symbol
name
decimals
totalSupply
transfer
allowance
transferFrom
approve
getBinderLength
checkPresaleAmount
getSortList
endPresale
setLaunchTime
getPresaleToken
<Receive Ether>
setFeeWhiteList
setSwapPairList
claimToken
claimContractToken
getLPProviderLength
getMineReward
setExcludeLPProvider
_transfer
setFundAddress
_approve
presale
_basicTransfer
claimBalance
<Constructor>
balanceOf
updateMineCycle
processMineLP
_tokenTransfer
_isAddLiquidity
_isRemoveLiquidity
_bindInvitor
_addLpProvider
sortPreList
processPreSaleReward
payable
procesInvitorReward
_takeTransfer
procesUpInvitorReward
getInvitorReward
checkMineLv
checkLowerCount
IERC20 (iface)
decimals
symbol
name
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
ISwapRouter (iface)
factory
WETH
swapExactTokensForTokensSupportingFeeOnTransferTokens
swapExactTokensForETHSupportingFeeOnTransferTokens
addLiquidity
addLiquidityETH
ISwapFactory (iface)
createPair
getPair
ISwapPair (iface)
getReserves
token0
balanceOf
totalSupply
ArbSys (iface)
arbBlockNumber

Summary

The XMY contract is a multifaceted Ethereum-based token system that seamlessly integrates presale functionalities, liquidity management, and a tiered reward distribution mechanism. At its core, it offers users the ability to participate in presales, provide liquidity, and earn rewards through mining and referrals. The contract is designed with a robust fee structure, imposing a maximum fee of 8% on certain transactions, complemented by an additional 2.5% fee when users add or remove liquidity. The audit revealed that the contract is free from critical issues, and the administrative functions vested in the owner do not pose risks to user transactions.

Disclaimer

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